Assurance Case Generation Using Model-based Engineering and Formal Verification

Fang Yan

Doctor of Philosophy

University of York

Computer Science

July 2023

Abstract

Assurance cases (ACs) have been a well-established means to provide compelling justifications for ensuring confidence in critical system properties. The development of ACs should follow an incremental and iterative approach, allowing them to evolve continuously and uphold their assurance validity as the system evolves. However, the generation and maintenance of ACs are labour-intensive tasks that are prone to human errors.

Model-based Engineering (MBE) has been exploited to generate model-based ACs. One of the key advantages of MBE is the traceability automatically established between system artifacts and AC models, enabling the automated creation and updates of ACs in response to system modifications. However, there are existing limitations in the current solutions, specifically in providing adequate support for generating ACs from heterogeneous notations. On the other hand, the generation of evidence remains a critical aspect of the AC process. Formal verification offers a rigorous means of providing evidence for ACs and has the potential for automation. However, there is a research gap of the insufficient automation and coordination of various formal verification techniques.

This thesis proposes an framework for automatically deriving model-based ACs from system artifacts using MBE and formal verification techniques including model checking and theorem proving. There are three main contributions in this research: a method for generating the AC structure compliant with Structured Assurance Case Metamodel from artefacts of different formats using MBE techniques; a method for refining the lower-level structure from formal syntax of robotic design models; and a method for generating AC evidence from formal semantics of robotic models by coordinating formal verification techniques including FDR and PRISM model checking, and Isabelle/HOL theorem proving. A methodology is provided to guide the joint use of these methods to produce an AC.

The framework is evaluated using four robotic systems of different natures, showing its generality across different domains. Several challenges identified in the research literature have been addressed to enhance the efficiency and effectiveness of AC generation and AC evolution within the MBE framework. Our framework provides engineers with an efficient solution for the derivation and evolution of model-based ACs. Also, the integration of automated formal verification processes significantly reduces the reliance on formal expertise, thereby promoting the practical adoption of formal verification in the AC process.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Hypothesis, research questions and contributions 4

1.3 Thesis structure 5

2 Background and Literature Review **7**

2.1 Assurance cases 7

2.1.1 Assurance case notations 8

2.1.2 SACM metamodel 9

2.2 Model-based assurance case generation 11

2.2.1 AC Patterns 11

2.2.2 Data management and pattern instantiation using MBE 13

2.2.3 Model query 15

2.3 Assurance case process using formal verification 17

2.3.1 AC evidence generation by application of formal verification 17

2.3.2 Formalization of system design models 19

2.3.3 Formalization of requirements 20

2.4 RoboChart introduction 22

2.4.1 RoboChart notation 22

2.4.2 RoboChart tool support 24

2.5 Summary 25

3 AC Generation Framework **26**

4 AC Generation **34**

4.1 Overview 34

4.1.1 Epsilon model management 36
4.2 AC generation by pattern instantiation 38
4.2.1 System artefact unification 38
4.2.2 AC pattern design 41
4.2.3 Pattern instantiation 45
4.3 AC generation by model query and integration 49
4.3.1 An abstract running example 49
4.3.2 Model query method 50
4.3.3 Integration 52
4.4 AC graphical presentation 53
4.5 Case study and evaluation 56
4.5.1 Implementation 56
4.5.2 Case study 56
4.5.3 Evaluation 60
4.6 Summary 63

5 AC Evidence Generation by Model Checking 65

5.1 Overview and background 65
5.1.1 CSP and its RoboChart assertion language 68
5.1.2 PRISM and its RoboChart assertion language 69
5.1.3 Evidence generation method 71
5.2 Claim structuring 72
5.2.1 Claim classification 72
5.2.2 Kapture templates for requirements 73
5.2.3 Requirement template design 75
5.2.4 Claim structuring 84
5.3 Assertion generation 85
5.3.1 Assertion template design 85
5.3.2 Assertion template instantiation 94
5.3.3 Assertion generation integration 95
5.4 Evidence model generation and integration 97

5.5 Case study and evaluation . . . 99
5.5.1 Implementation . . . 100
5.5.2 Case study: mail delivery robot . . . 100
5.5.3 Case study: High Voltage Controller . . . 109
5.5.4 Evaluation . . . 118
5.6 Summary . . . 121

6 AC Evidence Generation by Theorem Proving 124

6.1 Overview and background . . . 124
6.1.1 Z Machine . . . 125
6.1.2 Methodology . . . 128
6.1.3 A running example . . . 129
6.2 Semantic domain . . . 130
6.2.1 Observational variables . . . 130
6.2.2 Healthiness conditions . . . 131
6.3 Model transformation rules . . . 132
6.3.1 Types . . . 132
6.3.2 RoboChart context . . . 133
6.3.3 Nodes . . . 134
6.3.4 RoboChart transition behaviour . . . 134
6.3.5 RoboChart state machine . . . 136
6.4 Verification method . . . 137
6.4.1 Structural invariants . . . 137
6.4.2 AC claim formalization . . . 138
6.4.3 Invariant proving . . . 139
6.5 Implementation . . . 141
6.5.1 EGL template design . . . 142
6.5.2 Transformation from RoboChart to Z Machine . . . 144
6.6 Case studies and evaluation . . . 145
6.6.1 System description . . . 145
6.6.2 Z Machine model . . . 146
6.6.3 Requirement formalization and verification . . . 147

6.6.4 Evaluation 148
6.7 Summary 150

7 Conclusion **152**
7.1 Summary of contributions 152
7.2 Future Work 155
7.2.1 Automation of claim formalization using NLP 155
7.2.2 Extension of formal semantics for RoboChart 155
7.2.3 Incremental system development and verification supported by bidirectional communication between environments 156
7.2.4 Integration of other verification techniques into AC process 156

A Algorithms Referenced from Chapter 4 **157**

B Algorithms Referenced from Chapter 5 **166**

C Algorithms Referenced from Chapter 6 **169**

D Templates for Z Machine Notation **181**

E Z Machine notations for LRE state machine **183**

List of Figures

2.1 Main elements of GSN. 8
2.2 An example safety case in GSN notation [23]. 9
2.3 SACM Assurance Case Packages Class Diagram [27]. 9
2.4 Metamodel of an asserted Claim [28]. 10
2.5 Metamodel of argument package [27]. 10
2.6 RoboChart model of LRE [50]. 23

3.1 AC generation methodology. 27
3.2 An example AC module generated by pattern instantiation with system artefacts. . . 28
3.3 AC module of Figure 3.2 with G4 refined by model query. 30
3.4 AC module of Figure 3.3 with solutions S03 and S04 instantiated with formal evidence. 32

4.1 AC generation framework. 35
4.2 Excerpt of LRE hazard table spreadsheet. 40
4.3 Metamodel of a hazard table. 40
4.4 Excerpt EMF models of LRE hazard table. 41
4.5 An generic AC pattern in GSN notation. 42
4.6 AC pattern instantiated with hazard table. 44
4.7 ETL transformation for an argument package. 46
4.8 ETL transformation from a cause to a claim 48
4.9 AC derived from LRE hazard. 48
4.10 Hazard table for an abstract safety property of a state machine. 49
4.11 AC fragment for the abstract example. 50
4.12 Hazard table for an abstract safety property of a state machine. 50
4.13 Model query process for SM1. 51

4.14 A procedure to integrate the AC structures generated through pattern instantiation and model query. . . . 52

4.15 *AssertedInference* with related elements of AC model. . . . 53

4.16 *AssertedEvidence* with related elements of AC model. . . . 53

4.17 The grouping of an AC module in GSN notation. . . . 54

4.18 RoboChart model of LRE [50]. . . . 57

4.19 EMF model of AUV Hazard analysis results for Hazard 1. . . . 58

4.20 LRE AC structure generated by pattern instantiation. . . . 58

4.21 Lower-structure of LRE AC by model query. . . . 61

5.1 An illustration of the chapter's purpose with an abstract example. . . . 66

5.2 The process of evidence generation for RoboChart by model checking. . . . 67

5.3 The method of evidence generation by model checking. . . . 71

5.4 A Kapture screenshot of the definition of *guard_event* and *required_event* in R1. . . . 84

5.5 R1 in Kapture. . . . 85

5.6 An example of a FDR model-checking report in RoboTool. . . . 98

5.7 An example of a PRISM model-checking report in RoboTool. . . . 98

5.8 Map of the workplace [84] . . . 101

5.9 Hazard analysis results for hazard cause of running out of power. . . . 101

5.10 AC in GSN notation for the mail-delivering robot. . . . 102

5.11 Module and Robotic platform in RoboChart model for delivery robot [55]. . . . 103

5.12 Controller in RoboChart model for delivery robot [55]. . . . 103

5.13 Battery state machine [55]. . . . 104

5.14 Task state machine [55]. . . . 104

5.15 Movement state machine [55]. . . . 105

5.16 SR1_1_1 exported from Kapture. . . . 106

5.17 SR1_1_2 exported from Kapture. . . . 106

5.18 SR1_1_3 exported from Kapture. . . . 107

5.19 Probability assertion for SR1_1_1. . . . 107

5.20 Reward assertion for SR1_1_2. . . . 108

5.21 Temporal assertion for SR1_1_3. . . . 108

5.22 Model checking result for SR1_1_1. 108
5.23 AC module for mail delivery robot updated with formal evidence. 109
5.24 Block diagram of one part of the paint robot, containing the HVC [85]. 110
5.25 Finite state diagram of the HVC controller [85]. 110
5.26 HVC RoboChart module [85]. 111
5.27 HVC RoboChart state machine [85]. 111
5.28 HVC hazard analysis EMF models. 112
5.29 Preliminary AC of HVC. 113
5.30 HVC Claim for SR1_1_1. 114
5.31 HVC Claim for SR1_2_1. 114
5.32 HVC Claim for SR1_1_1 validation. 115
5.33 HVC assertion for SR1_1_1 verification. 115
5.34 HVC assertion for SR1_2_1 verification. 116
5.35 HVC assertion for SR1_1_1 validation. 116
5.36 Updated AC of HVC using FDR verification results 117
5.37 SR1_3_1_timed in Kapture template. 118
5.38 Assertion for SR1_3_1_timed. 119

6.1 Pipeline for generating AC evidence for RoboChart state machine in Isabelle/HOL. . 128
6.2 RoboChart module of Chemical detector [54]. 129
6.3 RoboChart state machine model of Gas Analysis [54]. 130
6.4 Gas Analysis example for mapping **Rule** 2. 133
6.5 Examples for mapping Rule 8, 8.1, 8.2, 8.3. 135
6.6 Screenshot of applying `zpog_full` for proving `InitialToReading()` preserving R1. 140
6.7 Screenshot of applying *sledgehammer* for proving `InitialToReading()` preserving R1. 140
6.8 Implementation environment of AC evidence generation by theorem proving. 142
6.9 RoboChart state machine of LRE adapted from [50]. 146

A.1 A template of hazard table spreadsheet. 157

List of Tables

3.1 Case study arrangement for three phases 33

5.1 Classification of AC claims 72
5.2 Mapping between templates of the assertions and the requirements 86
5.3 Status of tool support for the steps in AC process 100

6.1 RoboChart element, rules, and mapping 132

To my family, thank you for your support.

Acknowledgements

I would like to express my heartfelt gratitude to my supervisors, Dr. Simon Foster and Prof. Ibrahim Habli, for their unwavering support and guidance throughout my four-year research journey. Despite the challenges posed by the pandemic, they dedicated their time and effort to mentor and guide me. Their expertise and insight significantly helped me forming the direction and outcomes of my academic research, laying a solid technical foundation for my work and making this work a stronger piece than it could have been.

I am deeply grateful to my colleagues in the RoboStar group for their generous assistance. In particular, I would like to extend my special thanks to Dr. Pedro Ribeiro for teaching me the CSP semantics of RoboChart and the RoboChart verification in FDR, Dr. Randall Ye for his guidance on RoboChart verification in PRISM, and Dr. Alvaro Miyazawa for patiently addressing all my queries regarding RoboTool and RoboChart. Their patience, responsiveness, and willingness to help, along with their high level of tolerance for my naive questions, have been truly invaluable.

Without their help, I wouldn't have completed my contribution on model checking in this thesis.

I would like to express my appreciation that Colin O'Halloran and Nick Tudor from D-RisQ have kindly provided me with access to the Kapture tool, which has played a crucial role in my research methodology.

I would also like to acknowledge Dr. Steve Dai for his assistance in resolving various Overleaf errors that I encountered during my thesis writing up.

Last but not least, I would like to extend my sincere gratitude to Dr. Xinwei Fang and his family for their support during challenging times in my personal life.

The contributions of these individuals have been instrumental in shaping my research journey, and I am truly grateful for their presence in my academic and personal life.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for a degree or other qualification at this University or elsewhere.

Parts of the research described in this thesis have previously published in:

- Yan, Fang, Simon Foster, and Ibrahim Habli. "Safety case generation by model-based engineering: state of the art and a proposal." In The Eleventh International Conference on Performance, Safety and Robustness in Complex Systems and Applications, proceedings, pp. 4-7. International Academy, Research, and Industry Association, 2021.
- Yan, Fang. "Generation and verification of executable assurance case by model-based engineering." In 2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 323-326. IEEE, 2021.
- Yan, Fang, Simon David Foster, Ibrahim Habli, and Ran Wei. "Model-Based Generation of Hazard-Driven Arguments and Formal Verification Evidence for Assurance Cases." In 10th International Conference on Model-Driven Engineering and Software Development, pp. 252-263. SciTePress, 2022.
- Yan, Fang, Simon David Foster, and Ibrahim Habli. "Automated Compositional Verification for Robotic State Machines using Isabelle/HOL." In 27th International Conference on Engineering of Complex Computer Systems. IEEE, 2023.

Chapter 1

Introduction

This chapter presents the background motivation for the research in Section 1.1, providing an overview of the reasons behind the study. In Section 1.2, the research questions that the study aims to answer are presented in detail, along with how these questions will lead to a substantial original contribution to knowledge and understanding. Additionally, the overall structure of this thesis is given in Section 1.3, providing readers with a clear understanding of the organization of the document.

1.1 Motivation

Robotics and Autonomous System (RAS) are defined as technological systems that can sense, analyse, interact with and manipulate their physical environment [1]. They are playing an increasingly important role across a wide range of areas and draw on a variety of related disciplines, including cyber-physical systems, self-adaptive systems, Open Adaptive System (OAS), systems of systems [2–5], and more. The integration of artificial intelligence (AI) has become a prominent feature in RAS systems, which have gained considerable attention from both academic and industrial circles.

In light of the high level of automation, human interaction, and complex environmental coupling involved in RAS systems, it is imperative to ensure that these systems satisfy essential properties before and throughout their operational lifespan. These properties may include security, safety, and reliability, among others, which are of utmost significance for high-risk fields like automotive, nuclear energy, and healthcare. The assurance process comprises identifying system properties as objectives and utilizing system design strategies, as well as verification and validation results and other artefacts, to substantiate the argument that the final product satisfies those objectives.

An Assurance Case (AC) is a document that systematically presents the outcome of the assurance process. The AC is developed incrementally and iteratively throughout the system development process and serves as a means of communication among various stakeholders. Furthermore, safety standards such as ISO26262 [6] for the automotive sector and MOD Def Stan 00-55 [7] for safety-critical software in defense equipment require the use of an AC.

A formal definition of Assurance Case is as below according to [8].

> "A collection of auditable claims, arguments, and evidence created to support the contention that a defined system/service will satisfy its assurance requirements."

ACs are produced through a process that progresses alongside system development. This process takes various system development artefacts, such as hazard analysis results, system architecture, verification methods and results, among others, as inputs to construct ACs. The AC aims to demonstrate a specific property of the system and is often referred to as a safety case when developed for safety properties.

Using a safety case as an example, hazard analysis and risk assessments can generate a set of safety requirements that a system must fulfil. The AC process then identifies safety claims based on these requirements, breaks them down into sub-claims using system architecture, and explains how the claims will be satisfied by the system with reference to design artefacts including the system specification and the design models. Finally, the AC provides evidence to support the claims based on the system's verification and validation results, such as model checking results and test reports.

These system artefacts are produced at different phases of the development lifecycle. Thus, the AC should be developed incrementally, according to the availability of the system artefacts. Moreover, since system development is an iterative process, ACs must evolve after the initial construction by updating the input data from the system development process and ensuring the validity of assurance. Without this evolution, ACs quickly lose their value.

For example, during system development, if there is a design modification, the validity of the verification results may be affected, and thus the claims supported by this verification may no longer be satisfied. If a new function is introduced to the system, a new hazard might be added based on the hazard analysis results, and a new AC module will be required for hazard elimination. It is common for a modification in one system artefact to trigger a series of cascading changes in the AC elements, because there exists chains of deductive reasoning and the artefact provenance among system artefacts. Specifically, a coherent AC requires us to keep a large number of artefacts synchronised. To maintain the validity of AC claims, we need to update the modified artefacts in the ACs and the relationship between these artefacts.

After deployment, the reality gap [9] between the system models and the physical product presented in RAS systems developed using MDE techniques, necessitates the system modifications. Moreover, the inherent uncertainty [10] in RAS systems during the design time often leads to adaptations in various aspects, including functions, safety constraints, environment assumptions, and more. These factors, the reality gap and uncertainty, may contribute to the continuous co-evolution of the ACs throughout the entire lifecycle of RAS systems.

Based on the above, one of the challenges during the system design phase is the repeated workload of AC generation that results from system development iterations. Once in operation, RAS systems may undergo frequent updates, which require a corresponding evolution in the AC as well. Thus, the AC process can be labour-intensive and vulnerable to human error. A possible solution to this problem is to make ACs "executable" in nature, which means that machines can read, process, and output the ACs. This allows for the automation of AC updates.

A lot of research has been conducted on the automation of the AC process. Two techniques that are often used in the research are Model-Based Engineering (MBE) and formal verification.

MBE is a well-adopted technique for system development because it allows for a more rigorous engineering approach, where we can prototype design elements and encapsulate the system requirements in a unified model. With its success in system design [11, 12], MBE has expanded into other areas, including the AC process [13–15] which generates the model-based ACs from design artefacts. The "model-based ACs" here are referred to the ACs that are presented in the form of models. AC process automated by MBE can reduce manual workload while improving the accuracy and efficiency of the process. The foundation for automation is the traceability established between AC models and system models. This traceability enables automatic updates of AC elements driven by system modifications. The automatic establishment of this traceability can be realized by MBE techniques such as model queries.

Compared to non-model-based ACs, such as ACs in graphical or textual format, model-based ACs can leverage MBE techniques and tools for automated generation. MBE provides rich and flexible model management capabilities including syntax validation, model simulation, the transformation of models to text and other models, code generation, and more.

Formal verification methods have been explored in the literature for evidence generation where model-checking methods are mostly discussed [16–18]. To close the loop of the AC process, providing concrete evidence to support claims is crucial. Different techniques, such as simulation, testing, review, and formal verification, can be employed to provide the required evidence. Among these techniques, formal verification, which includes model checking and theorem proving, can offer mathematically rigorous verification results and is equipped with automated verification capabilities. Therefore, it is beneficial to automate the generation of AC evidence using formal verification and further realize the automation of the generation process of ACs. The precondition for using formal verification in the AC process is that the metamodels (i.e., the abstract formal syntax) of the system models are supplemented with formal semantics. For example, the RoboChart modeling language [19], which is tailored from UML for RAS systems, is supported by the formal semantics of CSP and PRISM. Although various verification techniques are involved in AC generation, this work focuses on formal verification.

Previous research [14,20,21] has investigated the application of MBE techniques in the model-based AC process, with support of formal verification. My survey work [22] reveals that the application of MBE in the model-based AC process is diverse in terms of the MBE techniques used, the process phases in which they are applied, the employed formal verification methods, and the degree of process automation. Although established methods for model-based AC exist, it is concluded that a fully automated approach for the AC process is not available, and there are gaps in research. Specifically,

- The current support for generating ACs from various modelling notations is limited. System design models, classified as structured artefacts, can be used to derive AC models automatically [21] through model querying, which is a different method than pattern instantiation. However, the available solution using this method is specific to certain modelling languages and environments. To address this limitation, a more generalised solution that incorporates modelling notations of different kinds is necessary.
- The current AC process lacks integration with formal verification and does not address the application of formal verification in systems developed with MDE techniques. The common approach is to instantiate AC evidence using the results obtained from formal verification activities conducted on the system [17, 18]. However, formal verification is not an integral part of the AC process, resulting in a lack of seamlessness in the automation process. Some re-

search has focused on integrating formal verification in the AC process [16, 20] by deriving formal evidence directly from the argument claim. However, both the works that integrate formal verification and those that do not primarily focus on generating formal evidence from formal design models. It is important to note that in engineering practice, systems are typically developed using Model-Driven Engineering (MDE) techniques, and the system modelling languages utilized may or may not have formal semantics. Therefore, to effectively leverage formal verification in the AC process, a solution that focuses on the systems developed using MDE techniques, is needed.

- Since formal verification is not commonly integrated as part of the AC process, the automation of formal evidence generation is often overlooked in AC process research, where the formalization of claims and system models remains a manual process. This creates a gap in the automation loop of the AC process when formal verification is integrated into it. Furthermore, the practical application of formal verification in AC engineering practice is impeded by the reliance on manual processes and the need for formal expertise. Without automation, the efficiency of the formal verification process are limited, making it challenging to apply formal verification techniques in real-world AC engineering scenarios.

The system design may require modification after deployment, which is referred to as "offline" modification, and is usually separate from the operation. In this research work, we consider the AC process during both the system development phase before deployment and the offline modification phase after deployment.

For RAS systems incorporating AI components, it is technically feasible to update the system 'online', meaning the system can be modified while it is operating. However, a fully automated AC process for online updates is currently infeasible due to limitations in available techniques for online hazard analysis, online validation and verification, which may require the human inference to certain extent [3]. Therefore, updating ACs to support the online evolution of RAS systems is not addressed in this work.

Here, we present the initial steps towards developing an automated tool-chain for AC generation and evolution. A framework including a methodology and the tool implementation is proposed for engineering an automated AC process that can cover both the development and offline modification phases. This framework involves deriving and updating AC from system artefacts and robotic design models, and generating AC evidence through formal verification.

1.2 Hypothesis, research questions and contributions

The assurance case process requires human experts to interpret results and make informed decisions about system properties. Although the process cannot be fully replaced by tools, software tools can facilitate the automation of certain aspects of the assurance case development process.

The research hypothesis is as follows:

- The generation process of ACs can be substantially automated through the use of MBE techniques and various formal verification techniques.

The research objective is as follow:

A methodology for the derivation and evolution of AC from system development artefacts and robotic design models automated using MBE and formal verification.

Hence, **the research questions** are proposed as follows:

1. How can we generate assurance cases from the system engineering artefacts?

We will assess the existing methods for model-based AC generation and identify their limitations, explore the solution to generate and integrate the AC models generated from different sources. In this context, the system artefacts refer to the system development documents such as the specification, verification and validation plan and results, the hazard analysis, the assurance plan, system models, etc.

A contribution of this work is an AC pattern-based method that utilizes MBE techniques to generate the AC from artefacts of different formats.

2. How can we elaborate on the AC structure using robotic design models?

We will assess the existing methods for generating assurance cases from system design models, identify the limitations of established solutions, and propose a novel solution to address these gaps.

A contribution of this work is a model-query-based method for generating **the lower-level structure** of ACs from the formal syntax of robotic design models. In this thesis, an AC module consists of two parts: the "upper-level structure" that encompasses the properties to be argued, and the "lower-level structure" that relates to the system specification, design decisions (in the form of system models), and down to the verification that are used to support the upper-level properties. Therefore, when creating AC from the system design models, the resulting structure of the ACs falls into the lower-level category. The upper-level structure will be generated using the pattern-based method.

3. How can we generate AC evidence from the formal semantics of the design models by coordinating multiple formal verification methods?

We will define a systematic process for generating AC evidence from AC claims, select appropriate formal verification tools to automate the process and identify a set of claims that can be verified using formal verification techniques.

A contribution is a method for generating **the AC evidence** from formal semantics of robotic models by model checking and theorem proving. This contribution may greatly facilitate the practical application of formal verification in engineering practice.

1.3 Thesis structure

In the thesis, there are seven chapters.

Chapter 1 is the introduction, which describes the motivation, research questions and contributions.

Chapter 2 is the background and context for this research. It first introduces the notations of AC and the modeling language RoboChart as the background information of the research. Then, we look at current research in this field of study and evaluate it with regard to the applicability to the questions raised in Section 1.2.

Chapter 3 outlines the methodology of the AC generation, and depicts the research structure. An abstract running example is used to show how the complete framework works. This example will be explained along with the introduction of each phase.

Chapter 4 presents the AC generation method, answering the first two research questions. This chapter presents in detail the approach to generating SACM-compliant AC by employing a combination of pattern-based and model-query-based methods. Section 4.1 gives an overview of the method structure. The method of pattern instantiation is presented in Section 4.2 in details including four steps. The model query method for AC structure generation is discussed in Section 4.3. Section 4.4 discusses the generation of GSN graphics through the transformation from AC models to a graphical presentation in dot file format. Section 4.5 briefly introduces the implementation and then provides a LRE case study followed by the evaluation of the method proposed in this chapter. Section 4.6 summarizes the chapter and conducts a comparison with related work.

Chapter 5 addresses the third research question and presents an extension of the method described in Chapter 4 by generating formal evidence directly from the claims for RoboChart models through model checking. Section 5.1 introduces the RoboChart assertion language, which is used in the approach, and the structure of the approach, which consists of four steps: claim structuring, assertion generation, model checking, and evidence generation. In the rest of this chapter, Section 5.2 discusses the activities in Step 1, Section 5.3 discusses the activities in Step 2. Step 3 is automated by tools and no further details are given in this chapter. Step 4 is described in Section 5.4. In Section 5.5, the implementation is briefly introduced, two case studies (a mail delivery robot and a high voltage controller) are presented to demonstrate the application of evidence model generation through FDR and PRISM model checking. Section 5.5.4 evaluates the methods in this chapter. Section 5.6 concludes this chapter.

Chapter 6 also addresses the third research question and makes a contribution to the automatic generation of AC evidence through the verification of design models in the Isabelle/HOL theorem prover. Section 6.1 presents the background and the objective of this chapter, then introduce the Z Machine notation, which is used as basis to give RoboChart a formal semantics. The structure of the research method is presented in Section 6.1.2 including seven activities. The first activity involves providing RoboChart with a formal semantics by extending the Z Machine's semantic domain, and is discussed in Section 6.2. The second activity involves defining the mapping rules from RoboChart elements to Z Machine notations and is discussed in Section 6.3. The evidence model generation through theorem proving is discussed in Sections 6.4. The implementation of the transformation from RoboChart to Z Machine is provided in Section 6.5. Section 6.6 is the case study and the evaluation. The chapter is summerized in Section 6.7.

We summarise the contributions and propose seven directions for the future work in Chapter 7.

Chapter 2

Background and Literature Review

In this chapter, we investigate and critically evaluate the existing research in this area of knowledge, specifically with regard to the questions posed in Section 1.2.

There are three main pillars in the research: AC, MBE, and formal verification. In Section 2.1, we will delve into the concept of AC and explore its various notations. Section 2.2 will focus on the application of MBE in the AC process. Section 2.3 will investigate the utilization of formal verification in the context of AC. In Section 2.4, an introduction of the RoboChart modelling language is provided because RoboChart is the main system modelling language we used in this work.

We have published a peer-reviewed research paper entitled "Safety Case Generation by Model-based Engineering: State of the Art and a Proposal" [22] at the Eleventh International Conference on Performance, Safety and Robustness in Complex Systems and Applications (PESARO 2021). The paper contains a review of the literature and background material included in Section 2.2, as well as some content from Section 2.3.

2.1 Assurance cases

In this section, we will examine the notations used to represent ACs, explore the available support for the application of MBE in the AC process, and investigate the practical process of AC generation. By answering these questions, we will be able to identify the suitable AC notations to be used in my work.

An assurance case is a structured argument with a hierarchical structure that breaks down the top-level properties into multiple sub-levels. These properties may include safety, security, reliability, performance, among others, with safety being the most discussed property and often presented in the form of a safety case, which is a special type of assurance case.

Assurance of these properties is crucial for system operation, especially in high-risk industries such as automotive and healthcare. Although assurance cases may not be mandatory in all safety-related industries, they are widely used as a valuable tool for communication among different stakeholders and are often required by safety standards such as ISO26262 [6].

2.1.1 Assurance case notations

ACs can be represented in various formats such as text, graphics, and machine-readable models. The structured graphical forms are the most widely used notations for ACs, including Goal Structuring Notation (GSN) [23] and Claims-Arguments-Evidence (CAE) [24]. These notations use various shapes to graphically represent AC elements and their relationships in a structured manner. This facilitates the understanding of different information required for arguing the system properties by the system stakeholders.

CAE was developed by Bloomfield et al. [24] in 1998 as a notation for structuring safety cases. The main elements of this notation are claims, arguments, evidence, justification, context, assumptions, and counterarguments.

GSN was developed by Kelly [25] in 1998 at the University of York. It is a graphical argument notation that explicitly documents the elements and structure of an argument and its relationship to evidence. In GSN, claims are documented as goals, and items of evidence are cited in solutions. In addition to goals and solutions, the other main elements of GSN are strategy, context, justification, and assumptions. Two types of linkages between elements are also used: SupportedBy and InContextOf, as shown in Figure 2.1.

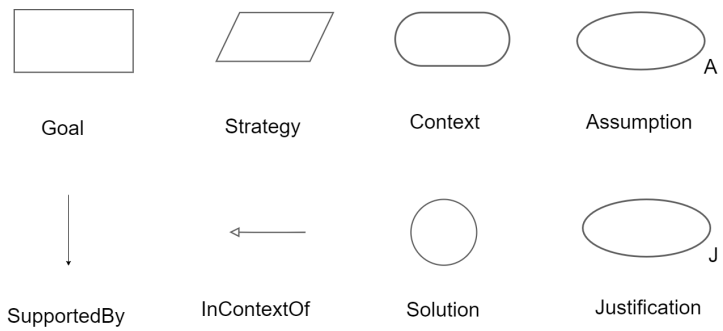


Figure 2.1: Main elements of GSN.

Although CAE and GSN share similar notation methods [26], GSN is the dominant notation used in engineering and safety-critical industries. Figure 2.2 shows an example of an abstract safety case in GSN. G1 is the top-level goal, which is decomposed into two lower-level goals, G2 and G3. Furthermore, G2 is decomposed through strategy S1 into G4, G5, and G6, each of which is supported by the evidence provided in the corresponding solutions. This demonstrates how the argument is structured. For the goal to be satisfied, the context, such as C1, assumptions, such as A1, and justification, such as J1, need to be valid to support the argument.

However, GSN was not originally supported by metamodels. Therefore, some works [14, 20] have proposed AC metamodels for GSN. Meanwhile, the Object Management Group (OMG) released the Structured Assurance Case Meta-Model (SACM) [27], which aims to improve standardization and interoperability. Compared to other proposed AC metamodels, SACM provides unique features, such as fine-grained modularity, controlled terminology, and traceability from argument to evidence artifacts [28]. SACM can support a variety of notations, including GSN and CAE.

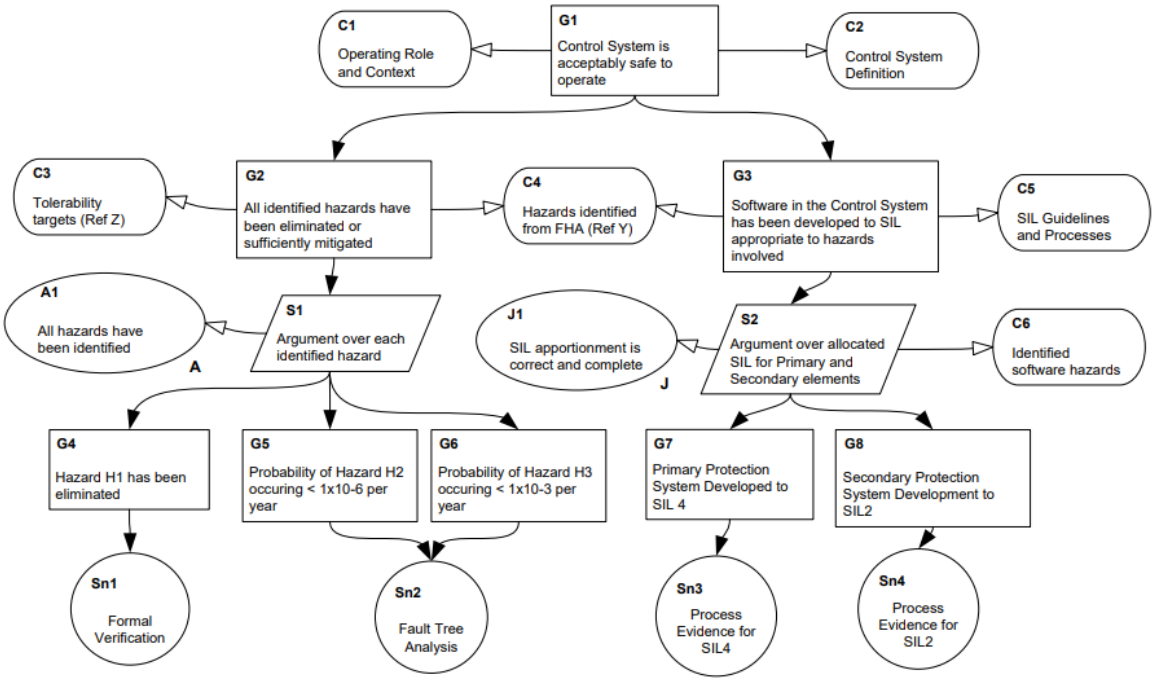


Figure 2.2: An example safety case in GSN notation [23].

2.1.2 SACM metamodel

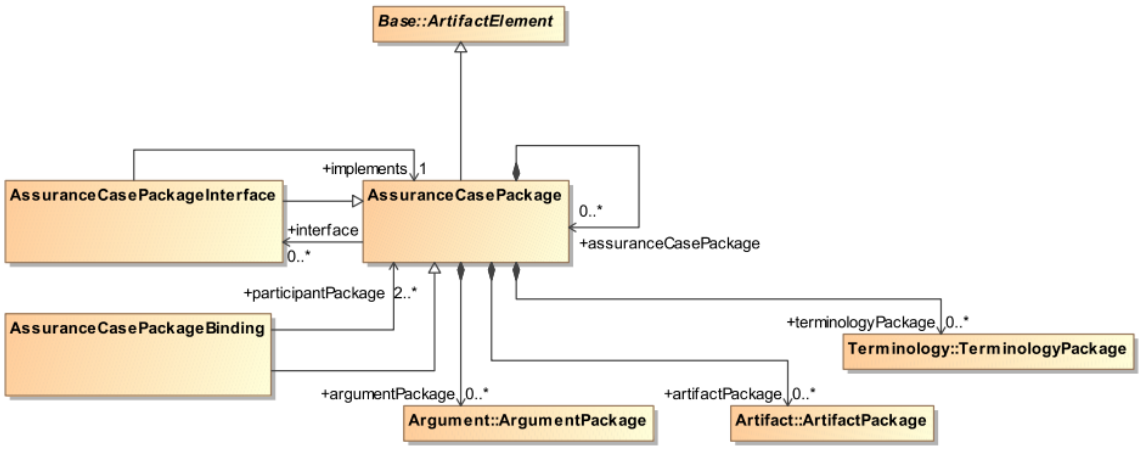


Figure 2.3: SACM Assurance Case Packages Class Diagram [27].

This section provides a brief introduction to the SACM metamodels. The metamodel consists of five packages. The Base package defines the fundamental elements of SACM, such as element names and descriptions. The Argumentation package consists of claims, evidence citations, and inferential links among them. The Artifact package captures the concepts used in providing evidence for the arguments made for system properties and represents the evidence and context files referenced. The Terminology package captures the concepts used in expressing the claims regarding system properties, such as expressions and argumentation components. Finally, the AssuranceCase package contains the Argumentation, Terminology, and Artifact packages.

The class diagram of the Assurance Case Packages is presented in Figure 2.3, which is the main component for building the argument of system properties. AssuranceCase packages consist of argument packages, artefact packages, and terminology packages.

Figure 2.4 depicts the metamodel of an asserted claim, which can have a name and a description, captured by LangString and Description respectively. This is equivalent to a goal in GSN. A Claim is asserted if it is supported by other Claims.

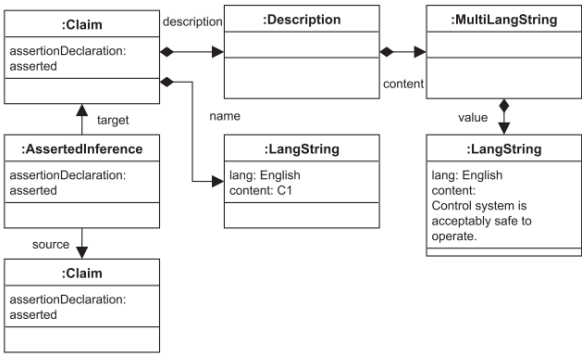


Figure 2.4: Metamodel of an asserted Claim [28].

Since SACM supports GSN notation, there is a mapping between SACM elements and GSN elements. Basically, an Argumentation package functions the same as a GSN module, and its metamodel is shown in Figure 2.5.

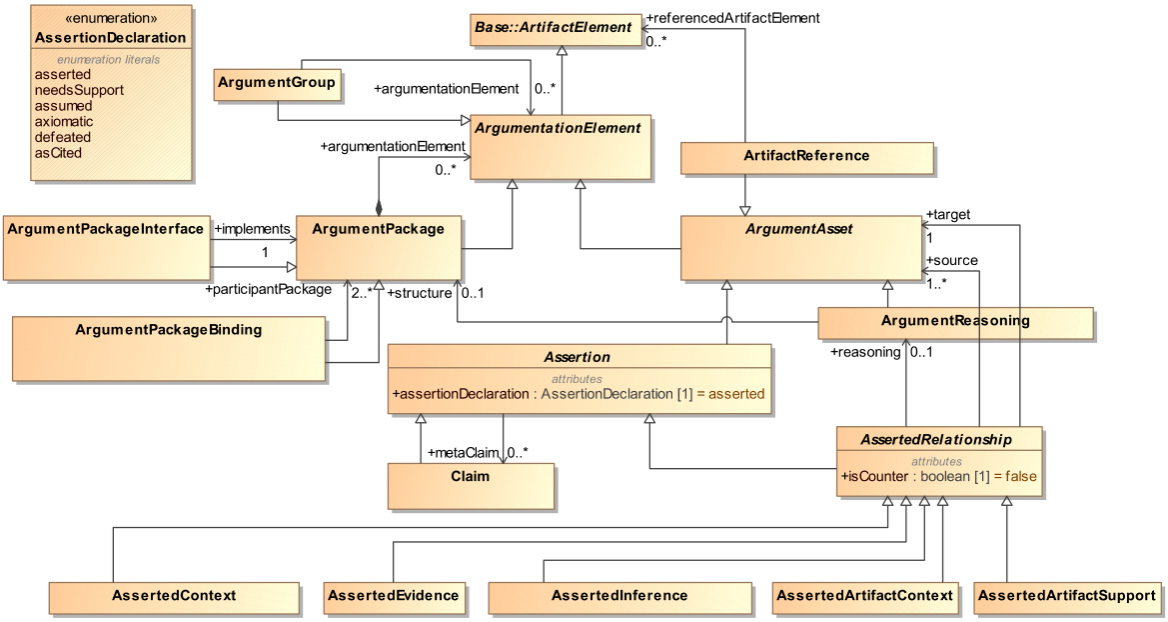


Figure 2.5: Metamodel of argument package [27].

The ArgumentAsset groups the argument elements, including Claims, ArtifactReferences, ArgumentReasoning, and AssertedRelationships. Claims are equivalent to the goals in GSN, while ArtifactReference is a reference to the artefact that defines evidence, equivalent to the GSN solution. ArgumentReasoning is the strategy to infer lower-level claims and is equivalent to the GSN strategy.

AssertedRelationships are the relationships between different assets. According to Figure 2.5, each AssertedRelationship can have one source and one target ArgumentAsset, which can be a claim, a context, a justification, an assumption, a strategy, or evidence, and one or zero ArgumentReasoning (i.e., the strategy). An AssertedRelationship can be an AssertedInference representing the inference links between the lower-level claims that are the source of the link to the higher-level claims, which are the target and equivalent to the GSN 'SupportedBy'. AssertedContext builds links from an artefact that defines context to claims and is equivalent to the GSN 'inContextOf'. AssertedEvidence builds links from evidence to claims that the evidence supports and is equivalent to the GSN 'SupportedBy'. For instance, a top-level claim such as 'the system operates safely' is supported by lower-level claims such as 'the control unit meets safety requirements' and 'the operators follow the operation manual' through an AssertedInference. Furthermore, the first lower-level claim is supported by evidence such as 'control unit test report' and 'formal verification results' through an AssertedInference.

Observations

1. GSN is a well-established and widely-used graphical notation in both industry and academy.

2. SACM offers a strong foundation for model-based engineering to manage ACs and more expressiveness than GSN, with possible strong tool-chain support in the future.

3. We will use the SACM metamodel for constructing AC and GSN for graphical presentation.

2.2 Model-based assurance case generation

In this section, we will investigate techniques used for the automation of model-based AC generation, specifically the methods of pattern instantiation and model query, along with their advantages and disadvantages.

From a practical perspective, it is beneficial to create AC using predefined patterns within an organization to reduce repetitive tasks. A typical AC process based on predefined patterns includes three main steps: AC pattern design, instantiation data management, and pattern instantiation. The pattern design will be discussed in Section 2.2.1, while instantiation data and pattern instantiation will be discussed in Section 2.2.2. Another method of AC generation using model query will be investigated in Section 2.2.3.

2.2.1 AC Patterns

The concept of AC patterns was introduced by Kelly [29]. It is an abstract structure containing placeholders that can be instantiated by concrete argument elements. For example, hazards with placeholders $\{\text{Hazard}\}$ in patterns need to be replaced by specific hazard content. The patterns capture repeatedly used structures of successful arguments in an abstract way [30]. Several studies have proposed patterns for various applications, such as [16, 30]. The patterns vary depending on the properties to be argued and the types of systems.

The instantiation data refers to the system artefacts required as inputs for AC development and should

be organized in formats that the instantiating tool can read. These artefacts, along with their relationships with the elements of the pattern and their interrelationships among themselves, should be identified. With the pattern and instantiation data available, instantiation involves replacing the AC placeholder elements with concrete values from the instantiable data. This can be done manually or automatically depending on the data format and tool support. If automatic instantiation is desired, machine-readable instantiation data is required. Similarly, the data processing can be manual or automatic through MBE.

After instantiation, the system artefacts may evolve, and the AC should be updated accordingly. When the system configuration is stable, the corresponding AC module may be ready and complete for certification review.

Several projects have proposed AC patterns for various applications. Different notations have been used for patterns, but the dominating notation is GSN [31].

Luo et al. [32] categorize GSN safety case patterns into two groups: the safety argument pattern and the confidence argument pattern. The safety argument pattern can be further divided into three types: product safety argument pattern, which argues the safety properties of the product; process safety argument pattern, which focuses on the quality of the development process; and compliance safety argument pattern, which demonstrates compliance with target standards or regulations.

Denney and Pai [20] propose a two-level pattern for arguing about system properties. The first level of the pattern builds the structure from hazards to safety requirements, while the second level builds the structure from safety requirements down to solutions. The deliberate separation of the structure pattern is intended to facilitate automatic instantiation implemented in the AdvoCATE tool.

Calinescu et al. [17] present a novel assurance argument pattern specific for self-adaptive systems, expressed in the GSN standard. This pattern incorporates all system configurations and provides justification for how the system requirements are fulfilled for each configuration.

Prokhorova et al. [16] have categorized safety requirements into eight types and provided the corresponding AC pattern for each type based on their formal verification methods.

Hawkins et al. [33] have defined a comprehensive AC pattern for Distributed Multiple Independent Levels of Security/Safety (D-MILS) systems which share common architectural features and will be assessed using a common verification framework. Thus, the patterns are specific to D-MILS systems.

Ayoub et al. [34] have designed a pattern that is suitable for constructing safety cases for medical infusion pumps that are developed using a model-based approach.

Šljivo et al. [35] have designed a high-level AC pattern for the assurance of architectural patterns. They first creates a set of architectural design patterns, each addressing a specific technical issue. For example, the security gateway pattern addresses the problem of an attacker using an entry point to inject malicious content into the internal unit. They then use these architectural design patterns to generate the system design, which was further used to generate the AC fragments. The AC pattern used mainly considers whether the design pattern can solve the corresponding issues, thus lacks of the generalization.

Observation

1. There is already an established and comprehensive set of patterns available for reference. However, it should be noted that many of the existing patterns are designed for specific development environments or applications. To enable the widespread adoption of model-based AC generation across various domains, the need for a generic AC pattern becomes evident.

2.2.2 Data management and pattern instantiation using MBE

Pattern instantiation is the mainstream method of AC generation, which can be done manually or automatically or using a hybrid approach. Much research effort has been put into the automation of pattern instantiation, which will be discussed in this section.

For pattern instantiation, the system artefacts must be identified, and then the mapping with the AC patterns must be built. Then, the automated instantiation of AC patterns can be carried out based on this mapping information by model-based engineering.

Denney and Pai [20] provide a solution and the tool AdvoCATE for the automatic generation of ACs using the pattern instantiation method. The tool also supports model management, including AC query and review, which are convenient for AC management. They built the data relationship in two phases, matching the two levels of the AC pattern as previously mentioned in Section 2.2.1 using data tables. The data table records the traceable relationship between system artefacts such as hazards, hazard causes, and safety requirements, and the relationship between the system artefacts and the element of the AC pattern, by labeling the artefact data as a goal, context, solution, etc.

The first table stores the relationships between the components of the higher structure of the AC pattern that covers claims for hazards down to safety requirements and the elements in the hazard analysis table, including hazard, hazard cause, hazard mode, mitigation, and requirement. The second table stores the relationships between the components of the lower structure of the AC pattern that covers claims for safety requirements down to solutions and the elements in the requirement table, including requirement, implementation, verification, etc. This two-level structure of AC that matches two data tables is referenced in my method and used in the description of the research framework in Chapter 3.

The table columns contain the identifiers of the AC pattern components that will be instantiated, e.g., goal G1; while the rows store the specific values of the instantiation data, for instance, HR1 represents a hazard with the identification number of 1. The instantiation process is automated by reading the data table, which contains the mapping between the system values and the AC components. The system artefacts' formats are not restricted since the mapping relationship is manually collected from the system artefacts.

Basir et al. [36] have done related work prior to [20]. They generate safety cases for the Simulink-generated code in the form of GSN. When the code generators are not qualified to be trustworthy, the extra guarantee that the output of the tool is correct and safe is required. So the AC is used to demonstrate that the Simulink-generated code satisfies its requirements. The safety cases are built hierarchically based on the system architecture model recovered from code analysis. The first hierarchy is from the safety requirements of the system level to the component level. The second hierarchy is from component-level safety requirements to source code. A tool is designed by exploiting the XML transformation using the tools AutoCert [37] (for analysis) and ASCE [38] (for presentation)

to automate the full construction of the component-level safety cases and the partial construction at the system level. The integration of the system level and component level is still to be further developed. The work addresses a specific property that the Simulink-generated code satisfies the requirement and the AC structure constructed is only applicable to this property.

Agrawal et al. [39] generated a safety case in the form of a safety tree, which is derived from an artefact tree. They first identify the traceability among all the system artefacts related to the mitigation of system hazards, such as safety requirements, design solutions, analytical models, source code, assumptions, diverse tests, reviews, and simulations. The traceability is defined in a Traceability Information Model (TIM). The TIM table can be generated manually or automatically, utilizing the traceability support in common requirement management tools like DOORS. An artefact tree is generated for a hazard by recursively following the trace links defined in the TIM to retrieve the artefacts automatically. Then, a safety case is generated by inserting the strategies between the elements of the artefact tree. However, this method does not address the system artefacts that are not managed in those engineering tools and therefore cannot be retrieved automatically. The safety cases are constructed according to the TIM tables instead of a predefined pattern and are not compliant with SACM metamodels. The method also supports the identification of changes in safety cases when the artefact tree is updated, and a set of predefined recommendations for change impact analysis is provided.

Lin et al. [40] proposed a framework for automatic generation of assurance case using a safety pattern. The proposed approach establishes data relationships by modelling the system development process using UML class diagrams. As the system development process includes both development activities and the input/output of the activities, traceability between the artefacts is also modelled. The traceability information in the model is then converted into a table and used for automatic pattern instantiation. However, the modeling of the development process is not discussed and it is not clear the automation degree of the modeling process.

Hawkins et al. [13, 14, 33] employ the method of pattern instantiation for generating GSN-based ACs. They have developed a weaving model [41] to capture the mapping between the elements to be instantiated in the argument pattern model and the elements of the artefacts' metamodels. This approach avoids the need for organizing the concrete system development data. The AC pattern is constructed as a model for automatic instantiation. The artefacts discussed in their work are the AADL models. The premise of the work is that instantiation data has to be structured models. Therefore, the work does not address the process of converting the unstructured artefacts into models. The advantage of this approach is that instantiable models can be extracted automatically from the system artefacts according to the traceability established in the weaving models.

Retouniotis et al. [42] propose a method for generating ACs using safety analysis artefacts. They use the methodology of Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) to generate safety analysis artefacts, such as fault trees, and to derive the safety integrity level, based on which the AC structure is constructed. They use a HiP-HOPS tool to parse system models, such as MATLAB Simulink, and facilitate the automated instantiation of the AC structure. The method is specifically designed for arguing the safety integrity level of the system.

Calinescu et al. [17] propose a two-step instantiation approach for generating ACs for self-adaptive systems. The instantiation process depends on whether the claims can be verified at design time or during runtime. If the claims can be verified at design time, their corresponding evidence is instantiated at design time as well. On the other hand, if the claims can only be verified during runtime,

their evidence has to be instantiated at runtime. An example of such a case is the measurement rates of sensors, which can only be obtained at runtime. Therefore, the evidence requiring this data will be instantiated at runtime. While the instantiation of the pattern is automated, the paper primarily focuses on the design-time and runtime modelling and verification aspects of self-adaptive systems. The automation of the instantiation process is not extensively discussed.

Observations

1. The AC pattern instantiation requires the identification of two types of data relationships. The first type is the mapping between system artefacts and the AC pattern elements, which can be established either at the level of artefact metamodels or of the specific artefacts. The second type is traceability among different system artefacts used for instantiation.

2. Establishing these two types of traceability requires engineering judgement and cannot be fully automated.

3. If the traceability is built between specific system artefacts and the AC pattern, updating the system artefacts will require re-establishing and re-validating the data relationship, which is a manual process and prone to errors. Instead, if traceability is built between the metamodel of the artefacts and the AC pattern, updating system artefacts will not affect traceability as long as the artefact metamodel remains unchanged. Moreover, the updated concrete value of the system artefact can be automatically extracted based on its metamodel, thereby reducing manual workload and improving AC evolution accuracy.

4. The AC generation can be largely automated using model transformation techniques, provided that the system artefacts are presented as machine-readable models. However, most research in this area assumes that the system artefacts are in the correct format for processing, i.e., as models. In cases where the system artefacts are unstructured, the instantiation data needs to be manually updated in the AC patterns, which creates an automation gap that is currently not addressed.

5. In my method, we will establish traceability between artefacts and AC patterns at the metamodel level to enable a higher degree of automation.

2.2.3 Model query

Besides the method of AC pattern instantiation using the available instantiation data, there is another method to automatically create ACs through model querying.

Gacek et al. [21] present a method for generating AC through AADL model querying. This method does not rely on a pre-defined AC pattern. A domain-specific language and a tool called ‘Resolute’ is developed for constructing the AC. The tool is integrated into the Open Source AADL Tool Environment (OSATE), which is an AADL modelling tool. The claim is formalized and decomposed by querying system models, and then further verified by formal analysis.

The generation of the AC consists of two steps. Firstly, the engineers define the top-level claim as a first-order predicate. In the second step, the rules for how the models should satisfy the claim are specified. These rules are further used to query the system models. Both the claims and rules are parameterized by variable inputs, which are instantiated using AADL models through model queries.

For example, an operating system might enforce memory protection. A way to satisfy memory protection may be to examine all the other processes, which share the same underlying memory component. Note that in AADL a "process" represents a logical memory space while a "memory" represents a physical memory space. Thus, the claim ' The memory *mem* of process *p* is protected from memory alteration caused by other processes" is formalized using the predicate '`memory_protected(p: process)`'. Next, the rules are defined to examine all the other processes which share the same underlying memory component. The query rule for the example claim is '`forall (mem: memory). bound(p, mem)` $\Longrightarrow$ `forall (q : process). bound(q, mem)` $\Longrightarrow$ `memory_safe_process(q)`'. This rule states that if other processes use the same memories as process *p*, then those processes must not have any safety impact on process *p*.

Using such predicates, the AADL models will be queried to automatically identify all the memories used by process *p* and all the other processes that use the same memories as process *p*. Next, the sub-level claims are instantiated using the identified processes and memories. Finally, the claims are verified by searching the proof using formal analysis.

Resolute includes a standard library of predefined functions for common operations, which can be combined with user-defined functions to flexibly define rules for specifying claims.

This work establishes a dependence of the assurance case on the AADL model, allowing changes to the model to automatically trigger changes to the AC. This coupling of ACs with system design ensures consistency between the ACs and system models. However, the method is constrained to a specific development environment and requires engineers to have proficiency in the Resolute language. The AC generated does not comply with SACM metamodel or GSN notations.

Gallina and Nyberg [43] generate ACs from system artefacts, excluding design models, using model query techniques. The process begins by defining a simple AC pattern based on a set of system artefacts such as algorithms, test cases, and reports, among others. All required artefacts are then constructed following the Open Services for Lifecycle Collaboration (OSLC) [44] standards to ensure structured and traceable artefacts with backward and forward traceability to other related items.

OSLC is a set of specifications for integrating tools and services used in software development and other product lifecycle management activities. It uses Resource Description Framework (RDF) [45] for data modeling and representation. RDF is a data model that allows developers to describe resources (such as documents, web pages, or even physical objects) and their relationships to one another in a machine-readable way. RDF represents data as a set of triples, each consisting of a subject, predicate, and object. The subject is typically a resource being described, the predicate describes a property or relationship of that resource, and the object is the value of that property or the related resource. For example, a triple might state that "System x has test case a", where *system x* is the subject, *has* is the predicate, and *test case a* is the object. In this way, traceability among the elements of the different artefacts is established.

Once the top-level claim is created, the AC structure beneath it is automatically constructed by querying the system artefact models. For instance, if the top claim is "Algorithm X has been successfully tested," the sub-level claims should be "Test case *i* passed," with the supporting evidence being "Test report for test case *i*." Assuming there are three test cases for algorithm X, three sub-level claims and corresponding evidence will be automatically generated by querying the algorithm, test cases, and test reports artefacts.

This work mainly uses the traceability provided by RDF formats as the basis for AC generation. Since the generation involves multiple types of artefacts, an AC pattern is required to define the relationship between the AC components and the artefacts. However, this method does not encompass the generation of ACs from system design models that are backed by metamodels, such as AADL models. Although it is technically feasible to represent the system models using RDF, it does not provide any additional benefits since these models already have metamodel support, which facilitates the utilization of model query techniques. By the time of the paper, the method was not implemented as an automated tool yet.

Observations

1. Model query is a flexible technique that enables the generation of large-scale ACs in an efficient manner. But the method is constrained to the AADL development environment. A more generalised solution utilising model query would be desirable.
2. In our work, we'll explore this direction but provide a more generalized solution that is independent of the system modelling environments.

2.3 Assurance case process using formal verification

As our third research question focuses on the generation of AC evidence using formal verification techniques, this section delves into two key aspects related to the formal verification of AC claims. Firstly, an investigation is conducted to explore the existing research on verifying AC claims using formal verification methods. Secondly, an examination is carried out to assess the automation solutions of formal verification methods in a broader context.

2.3.1 AC evidence generation by application of formal verification

Depending on the nature of the properties that are to be argued, various verification techniques can be employed to generate evidence. Formal verification is a suitable option for properties that can be described using logic and enables the automation of evidence generation for assurance cases. One significant advantage of formal verification is its ability to demonstrate the absence of errors. This section discusses the application of the formal method to the AC process for the evidence generation.

Prokhorova et al. [16] presented a method for constructing structured safety cases from Event-B formal models using different formal verification techniques. They categorised safety requirements into eight types and devised specific safety case patterns for each type based on the formal verification methods and tools employed for that particular type of property. These categories include global properties, local properties, control flow, absence of system deadlock, system termination, hierarchical properties, temporal properties, and timing properties.

The safety requirements were primarily formalized using Event-B and then verified through theorem proving. Model checking methods were also employed to fully address all classes of safety requirements. For instance, the timing requirement was formalized using Timed Computation Tree Logic (TCLT) and verified using the UPPAAL model checker, and the pattern of the safety case for timing properties was established, with UPPAAL verification integrated in the pattern as a solution.

Since the system models were developed as Event-B formal models, there was no need for an additional step of formalizing them. The instantiation of all the proposed patterns has been developed manually. The automation of safety case pattern instantiation is mentioned as a future consideration in their work.

As discussed in Section 2.2, Denney and Pai [20] propose a solution and the tool AdvoCATE for the automatic generation of ACs. This solution also supports the application of formal verification. AdvoCATE integrates the formal verification tool AutoCert [46] into the AC pattern and provides a property specification editor, which allows users to manually formalize informal requirements by providing values for several parameters using the input language of AutoCert. Based on the user input, the AC fragment can be automatically generated. Unlike other approaches that use formal verification solely for generating evidence, this method utilises formal method to generate an AC structure fragment consisting of formalised claims, decomposed claims, and evidence by invoking the AutoCert tool.

For instance, an informal claim such as 'The implementation of the PID controller for the aileron control surface is correct' can be formalized by specifying the values of a set of parameters in a dialogue editor. The claim is then decomposed into three sub-claims using a specific AC pattern [47] designed for AutoCert: 'The element of aileron output is correctly formalised', 'The requirement is correctly formalised', and 'The element meets the requirement'. Among these three claims, the third one is formally verified by AutoCert, which is a static source-code analysis tool that verifies whether the code satisfies the safety properties by calling automated theorem provers. The method of generating the AC fragment using formal verification is primarily discussed in their previous work [36, 48].

The application of formal verification in the AC process aims to verify the source code automatically generated from the system model, rather than the system model itself. A prerequisite for applying this solution is to have a comprehensive understanding of the input language of AutoCert. However, this requirement may limit the accessibility of the solution for engineers.

Foster et al. [49] explores the application of formal proof in the AC process from two perspectives. Firstly, they introduced Isabelle/SACM, a formal language for the computer-assisted construction of ACs. It is an interactive tool for construction and checking of ACs, guaranteeing their well-formedness, consistency, and traceability. Secondly, they utilized the Isabelle theorem prover to obtain system proofs that serve as evidence within the Isabelle/SACM ACs, following a standard instantiation process. They further applied this method to a case study of an autonomous underwater vehicle [50]. The construction of ACs in this approach is performed manually, which is a limitation from the practical perspective. Also, Isabelle/SACM is not integrated into a GSN/SACM graphical tool.

Calinescu et al. [17] use the formal verification results to instantiate the dynamic AC patterns for the self-adaptive software. The system is developed as formal models and verified using model checkers. Then, the formal verification results are used to instantiate the AC evidence automatically.

Jee et al. [18] construct an assurance case for a pacemaker software. The software was developed using a model-based technique that involved formal modelling of the system. The software was modelled using timed automata, and the timing properties were verified using the UPPAAL model checker. A dedicated AC was created for the software, incorporating the model-checking results as evidence, following a typical instantiation process. The study did not focus on the generation of

evidence from the claims. No automation implementation is discussed.

Moreover, there are works [51–53] that specifically address the formalization of the entire assurance arguments, which is closely intertwined with the formal development of systems. However, these works are not evaluated in this section.

Observations

1. The most common method to applying formal verification in the AC process is by instantiating the AC evidence using the results obtained from the formal verification activities conducted on the system [16–18]. However, the formal verification process is rarely integrated into the AC process.

2. An integrated method to incorporate formal verification in the AC process involves deriving formal evidence directly from the argument claim, similar to the method proposed in [20]. This integrated approach can streamline the AC process.

3. Integrated formal verification combines different formal verification techniques and tools to verify the properties of a system. It involves integrating various formal methods, such as model checking, theorem proving, to provide a comprehensive and rigorous analysis of a system's design or implementation. It is necessary for the generation of AC evidence to ensure the coverage of different types of properties [16].

4. While most research focuses on generating formal evidence directly from formal design models [16–18], it is important to note that in engineering practice, the systems are usually developed using Model-Driving Engineering (MDE) technique and system modelling languages that may or may not have formal semantics. Therefore, to effectively leverage formal verification in the AC process, the solution should encompass systems developed using MDE.

5. The automation of formal evidence generation is rarely addressed in AC process research where the formalization of requirements and models remains a manual process. This creates a gap in the automation loop of the AC process. This issue will be investigated from a broader perspective in the next section.

2.3.2 Formalization of system design models

To automate the process of formal verification, two aspects must be considered: the formalization of requirements and the system models. These aspects are not unique to AC processes that utilize formal verification but are standard procedures in the field. Hence, in this section, we provide a concise exploration of the formalization of design models.

The objective of formalizing design models is to facilitate the formal verification of the models by reusing the existing verification and analysis tools. As such, the models are typically formalized using the formal semantics compatible with the targeted verification tools.

RoboChart [54] is a UML-like modelling language specifically tailored for robotic systems. It has denotational semantics [54] based on the CSP process algebra, as well as probabilistic semantics within the high-level modelling notation used by the probabilistic model checker PRISM [55]. The RoboChart development tool, RoboTool, supports the automatic conversion of RoboChart models

into their respective formal semantics. These formal semantics enable the verification of RoboChart models using the FDR model checker [56] for non-probabilistic properties and the PRISM model checker [57] for probabilistic properties. As a result, both deterministic and probabilistic aspects of the models can undergo comprehensive verification.

Foster et al. [58] represent a subset of RoboChart using an action language for states and transitions in Isabelle/HOL. The verification of the deadlock-free property can then be carried out automatically within Isabelle/HOL. However, the transformation process remains manual.

Zou et al. [59, 60] automatically translate Stateflow diagrams to Hybrid Communicating Sequential Processes (HCSP) [61] models which are verified in Isabelle using Hoare logic.

Systems Modeling Language (SysML) [62] is a general-purpose modelling language designed for systems engineering applications. A forthcoming version of SysML is currently under development and will include a formal semantics[1].

Many studies have focused on providing formal semantics to AADL as well. Hadad et al. [63] propose an automated transformation process where a subset of the AADL model is converted into Event-B. Subsequently, the models are verified using the RODIN platform and the Atelier B prover. UML class diagrams are employed to categorize the AADL elements, and these classes are then mapped to Event-B models. Bozzano et al. [64] defined a formal semantics for a significant subset of AADL. Then, models are formally verified using the symbolic model checker NuSMV [65] and a probabilistic model checker MRMC [66]. There are many other examples of AADL transformations: formalization in Timed Abstract State Machines (TASM) [67], in Behaviour Interaction Priority (BIP) [68], as a real-time process algebra ACSR [69], as Fiacre [70], as Real-Time Maude [71], as Lustre [72], and so on.

Observation

Design model formalization is a well-established research direction with mature solutions and significant outcomes. Researchers have focused on different modelling languages in their studies. One common technique for automatic formalization is model-based engineering (MBE). The selection of the formal semantics for the system models mainly depends on the features of the system modelling languages as well as the target verification tools for which the models are transformed. In our research framework, if appropriate, it would be advantageous to use a modelling language that already has available formal semantics. However, the verification platform that the formal semantics supports also needs to be taken into account.

2.3.3 Formalization of requirements

This section investigates the methods for requirement formalization. The majority of research on automating the generation of model-checking assertions focuses on verifying hardware registers [73–75]. This emphasis on hardware registers is driven by their relatively simple assertion structures and the large number of assertions required due to the size of the registers. However, there is still some work in this area addressing software verification.

[1] `https://github.com/Systems-Modeling/SysML-v2-Release`

Bolton et al. [76] propose a method for automatically generating formal assertions in Linear Temporal Logic (LTL) using predefined templates. These assertions are generated from specifications to verify the Human-automation Interaction (HAI) behaviour of a system through model checking.

To ensure the absence of procedural issues in the system's HAI, three types of specifications are considered: state/transition reachability, which ensures that every part of a task can be executed under certain conditions; termination, which guarantees that tasks can always be completed; and deadlock freedom. These specifications are formalized using LTL and verified using the Symbolic Analysis Laboratory's Symbolic Model Checker (SAL-SMC).

The system is modelled using an Enhanced Operator Function Model (EOFM) specifically designed to describe human behaviour. EOFM has formal semantics that enables the automatic transformation from EOFM to the SAL language. The assertions are automatically generated by translating the specifications using the system models based on a predefined set of assertion templates.

Because the work focuses on verifying specific aspects of HAI, it considers a limited number of property types. These properties have relatively simple structures, which make automation implementation relatively easy. However, the flexibility of assertion generation is restricted by the templates. The work does not address refinement assertions, which have more complex structures.

Bourbouh et al. [77] address the automation of assertion generation for model checking through the use of a restricted natural language approach. The requirements are manually written in the restricted natural language of the Formal Requirements Elicitation Tool (FRET), referred to as FRETISH. FRET [78] is an open-source framework developed by NASA, designed to facilitate the elicitation, formalization, and comprehension of requirements. FRETISH encompasses six fields: *scope*, *condition*, *component*, *shall*, *timing*, and *response**, with *component*, *shall*, and *response* being mandatory. FRET automatically transforms FRETISH requirements into future-time (fmLTL) and past-time (pmLTL) linear temporal logic to facilitate model checking. For instance, consider the requirement "The rover shall not run out of battery." In FRETISH, this requirement can be expressed as "Navigation shall always satisfy battery > 0." Subsequently, it is automatically translated into the formula *H(battery > 0)*. FRET integrates with the CoCoSim tool for the analysis of Simulink models and, via CoCoSim, connects to the Kind2, Zustre, and Simulink Design Verifier (SLDV) tools for additional analysis purposes.

The study from [77] proposes a framework for system development using integrated formal methods. This framework integrates verification results from different formal verification tools, namely FRET, COCOSIM, and Event-B, and utilizes these results to generate an assurance case through the use of AdvoCATE. As the primary focus of the work is on the integration and automation of formal verification processes, we discuss this work in this section.

ARSENAL [79] facilitates the translation of natural language requirements into LTL models by employing a parser and a collection of domain-specific ontologies. The method consists of three key steps. Firstly, a domain-specific ontology is established to cater to the specific application domain. As an illustration, in the case of an infant incubator system, entities such as "Lower_Desired_Temperature" and "Higher_Desired_Temperature" are defined within the ontology to represent the lower and higher desired temperature values respectively. Subsequently, the natural language requirement is restructured into an intermediate representation utilizing the established ontology. In the second step, the intermediate representation is parsed using the Stanford Typed Dependency Parser (STDP), resulting in a set of dependency triples. Lastly, a set of mapping rules linking the dependency triples to the

LTL semantics is devised, serving as the basis for the transformation process.

In addition to utilizing restricted natural language and NLP parsers, the application of neural machine translation techniques has also been explored for this purpose. The research group for developing the ARSENAL framework has ventured into this direction as well [80]. They have adopted a generative model for natural language, employing a machine learning pipeline to generate candidate LTL formulas and models from natural language input. In this updated framework, known as ARSENAL 2, the ontologies and intermediate representation are substituted with end-to-end neural network models for semantic parsing, capitalizing on recent advancements in neural machine translation. This approach offers the advantage of removing restrictions on the natural language input for requirements. However, a key challenge lies in the lack of large training corpora that are annotated with ground truth.

Lindoso et al. [81] provides a diagrammatic solution to generate model-checking assertions for RoboChart models. They propose a diagrammatic notation which mixes the standard elements of UML activity diagram with elements of RoboChart as events and operations. The specifications are manually defined as diagrams using the diagrammatic notations and then automatically translated into formal CSP semantics and checked by FDR.

Observations

1. The methods for formalizing requirements can be broadly classified into three groups: utilizing templates in restricted natural languages [76, 77], employing a natural language parser with an ontology [79], and utilizing neural machine translation techniques [80].

2. Utilizing predefined templates is a straightforward implementation method. However, it is limited in terms of assertion expressiveness due to the fixed structure of the patterns. To accommodate various types of assertions, a larger number of patterns is required, and the existing list may not be exhaustive.

3. The effectiveness of the parser method relies on the presence of a well-defined ontology and accurate mapping rules. This approach allows for a more flexible expression of requirements. However, the manual creation of the ontology undermines the benefits of this method.

4. Neural machine translation offers an end-to-end transformation process with minimal user intervention. While widely used in natural language translation, it encounters challenges when applied to formal languages, mainly due to the lack of appropriate corpora.

2.4 RoboChart introduction

2.4.1 RoboChart notation

The core of RoboChart [19] is a subset of the UML state machine notation that excludes non-essential features that lead to an increase in the complexity of the underlying mathematical models. RoboChart offers several features that align with the objectives of this research: (i) it is a domain-specific modelling language tailored from UML for RAS applications, providing the typical elements and hierarchical structure found in UML, (ii) it is supported by an EMF metamodel, enabling easy manipulation

with MBE techniques for AC generation, and (iii) it possesses formal semantics and model checking support, which are advantageous for automating the generation of formal evidence in the next chapters.

RoboChart includes a profile of UML state machines, enriched with facilities to define real-time properties. It captures the system architecture in a hierarchical format, where a RAS system is modelled as a RoboChart module. One module can have one platform and multiple controllers. It defines the boundaries of the robotic application modelled and specifies the interface of such application through a robotic platform.

A robotic platform abstracts a physical robot in terms of the functionalities it provides to controllers. These may be of three types: variables, operations, or events. These elements can be either declared directly in the robotic platform, or grouped in interfaces. An interface groups variable lists, operations, events, and clocks. An event may or not have a type, which is an expression that defines the values that can be communicated via the connection, if any [82].

The term "controller" is used to refer to software components that interact with the hardware platform. The behaviour of the controller is defined using state machines composed of nodes and transitions and can include local variables and constants. A node can be states or junctions. A final node is a state indicating the termination of the state-machine, and an initial node is a junction indicating where the execution of a state-machine starts. Within a state, there can be an entry action, a during action, and an exit action. A state can be composite and contain states and transitions. Transitions are directed connections between two nodes: a source and a target. They can be asynchronous and bidirectional, as indicated by the boolean attributes async and bidirec. The transition may be triggered by an event, guarded by a condition, and contain an action that is executed when the transition is taken and its source node has exited [82].

A trigger of a transition is a communication. It can be of the type SIMPLE which is the event itself, the type INPUT which is the events associated with variables, the type OUTPUT which is events associated with a value corresponding to an output, or the type SYNC which is the events associated with a value corresponding to a synchronisation. RoboChart also supports shared variables, real-time constraints, and probability [58].

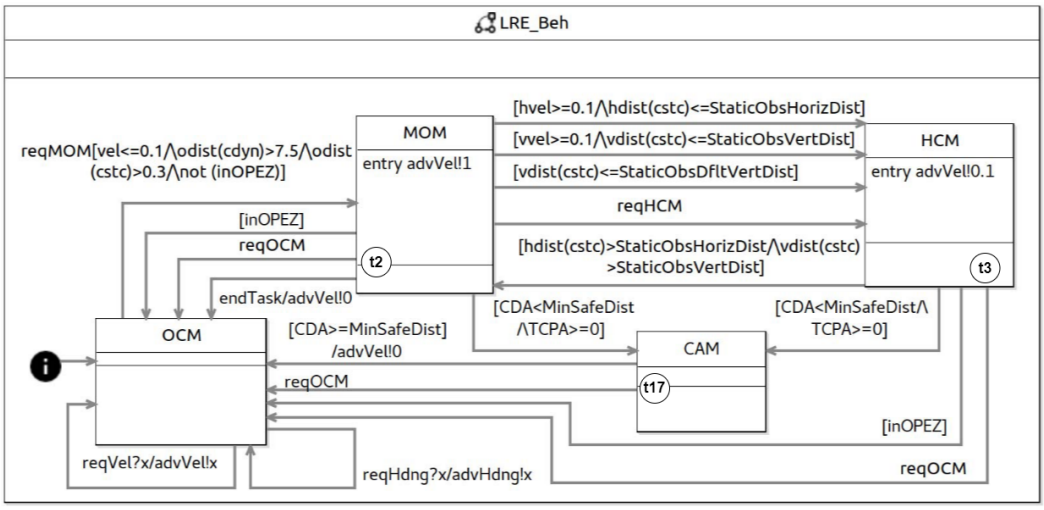


Figure 2.6: RoboChart model of LRE [50].

In this section, we use the Last Response Engine (LRE) example to briefly explain RoboChart modelling language. The behaviour of LRE controller is modelled using a state machine *LRE_Beh*, as shown in Figure 2.6. There are four operation modes Operator Control Mode (OCM), Main Operating Mode (MOM), High Caution Mode (HCM), and Colllison Aviodance Mode (CAM) modelled as states in the state machine respectively. LRE switches among these operation modes based on the system operating condition. The mode switching is modelled by transitions between states. For example, the LRE switches from MOM to HCM when the transition condition (stated in the square bracket []) is met, e.g., the horizontal velocity *hvel* is too high and the horizontal distance to an obstacle *hdist* is less than the margin. Also, the transition can be triggered by an event, for example, the operator can request LRE to switch to a certain mode whenever needed using the events *reqOCM*, *MOM*, or *HCM* which are the type of SIMPLE communication. The transition from the state *OCM* to *OCM* is triggered by an input type of communication *reqHdng?x* where *reqHdng* is an event, *x* is the variable. This is read as "to receive a value from the event *reqHdng* and stores it on the variable *x*". Further, this transition has an action (prefixed with a slash /) *advHdng!x* which is an output communication to send the value of *x* through the event *advHdng* to other components of the models, e,g, other state machines, or controllers. This example provides a general idea of the RoboChart models and main components used in this work. More detailed information on the syntax and semantics of RoboChart can be found in [82].

2.4.2 RoboChart tool support

RoboChart is supported by RoboTool[2]. RoboTool is a set of Eclipse[3] plug-ins implemented using the Xtext[4] and Sirius[5] frameworks. It enables modelling, performs type checking and analysis of well-formedness, automatically calculates CSP models and PRISM models, parses the properties written in the form of controlled English, calls the FDR and PRISM model checkers, and returns the verification reports.

The RoboTool uses Eclipse Modeling Framework (EMF) to implement RoboChart metamodel as a basis to generate a textual editor using Xtext and a graphical editor using Sirius. RoboChart metamodel is accompanied by a set of well-formedness conditions. These conditions are implemented through the validation mechanism provided by Xtext. Each condition is associated with one or more validation rules. The graphical editor constructs a RoboChart model and automatically verifies all the well-formedness conditions, and type compatibility in expressions and statements.

RoboTool calculates both the untimed and timed semantics of RoboChart models, i.e., the ASCII version of CSP called CSP_M. This semantics is used to support the automatic verification of properties within FDR. RoboTool also converts the RoboChart models into PRISM models to be used for automatic verification in the PRISM model checker.

RoboChart is equipped with the assertion languages which are in the form of controlled English to construct the properties to be verified in FDR and PRISM. The properties in these assertion languages are written in assertion files (with extension .assertions). RoboTool can process these assertion files into Communicating Sequential Processes (CSP) assertions for FDR checking or the Probabilistic

[2] www.cs.york.ac.uk/circus/RoboCalc/robotool/.

[3] www.eclipse.org.

[4] www.eclipse.org/xtext.

[5] www.eclipse.org/sirius

Computation Tree Logic (PCTL) specification for PRISM model checking. RoboTool supports calling the proper model checker through the assertions file and producing a verification report that records the result of the model checking. The advantage of the assertions files to generate the verification report is to avoid the need for users to interact directly with the model checkers [82]. RoboTool has other features that support RoboChart modelling and verification which are not used in this work. This information can be found in the RoboTool Manual [83].

2.5 Summary

The literature review conducted in this chapter has provided insights into the current state of research in the field of model-based AC generation. These findings will significantly influence the development and refinement of our research framework, ensuring its alignment with the latest advancements and addressing the existing gaps in the field.

By incorporating these insights, we aim to develop an innovative research framework that advances the field of AC engineering and contributes to the development of model-based AC supported with formal verification with real-world applicability. A framework for AC generation is proposed in the next chapter.

Chapter 3

AC Generation Framework

This chapter presents the framework for the automated generation of ACs to support the AC engineering process. In our work, a set of methods is designed for assurance case structure generation and assurance evidence generation, and the methodology is provided to guide the joint use of these methods to produce an assurance case. The methodology and the tools implemented consist our framework of AC generation. The inputs of the process are system artefacts of different formats, the predefined AC patterns, the AC metamodel to be compliant with, the system development and verification environment of the RAS systems. The output is the model-based AC modules.

The methodology comprises three phases, as depicted in Figure 3.1. Phase 1 (top left of Figure 3.1) employs a pattern-based method to generate the AC module, primarily focusing on the upper-level structure of AC. Phase 2 (top right of Figure 3.1) employs a model-query-based method to refine the lower-level structure of AC based on the formal syntax of design models. Phase 3 (bottom of Figure 3.1) involves the automatic derivation of formal evidence for AC from the claim models generated in Phases 1 and 2. This is achieved using diverse formal verification techniques facilitated by model transformation techniques.

The AC process progresses concurrently with the system development process. As the system artefacts are gradually generated during system development, the AC structure is considered complete when all the requisite system artefacts are available. Throughout the system development, updates may be made to the system artefacts, such as the specification, design models, and verification results. These updates have the potential to invalidate the AC. However, explicit traceability is established between the elements of the AC and the artefacts at each phase throughout the entire AC generation process. Consequently, the AC can maintain their validity by automatically executing the generation process using the updated system artefacts.

An abstract running example is illustrated using Figure 3.2 - 3.4 to show how the methods are joined together to create an AC. This example will be explained along with the introduction of each phase.

Phase 1

Phase 1 generates the AC structure from both unstructured and structured system development artefacts using pattern-based method. The artefacts used as inputs for the AC process are the outcomes of the system development process, such as the hazard analysis results, the system specification, the

implementation, the verification and validation, the assurance plan for the property to be argued in the ACs, etc. It should be noted that the generation and validation of these artefacts fall outside the scope of this research, which assumes that these artefacts have already been validated. The AC are then generated by instantiating a predefined AC patterns using system artefacts of different formats. A step of unification of the artefacts is involved to address the different formats used in the system artefacts. The output of Phase 1 is the Assurance Case structure that conforms to the AC pattern and the SACM metamodel. Phase 1 is discussed in detail in Sections 4.2.1 to 4.2.3 of Chapter 4.

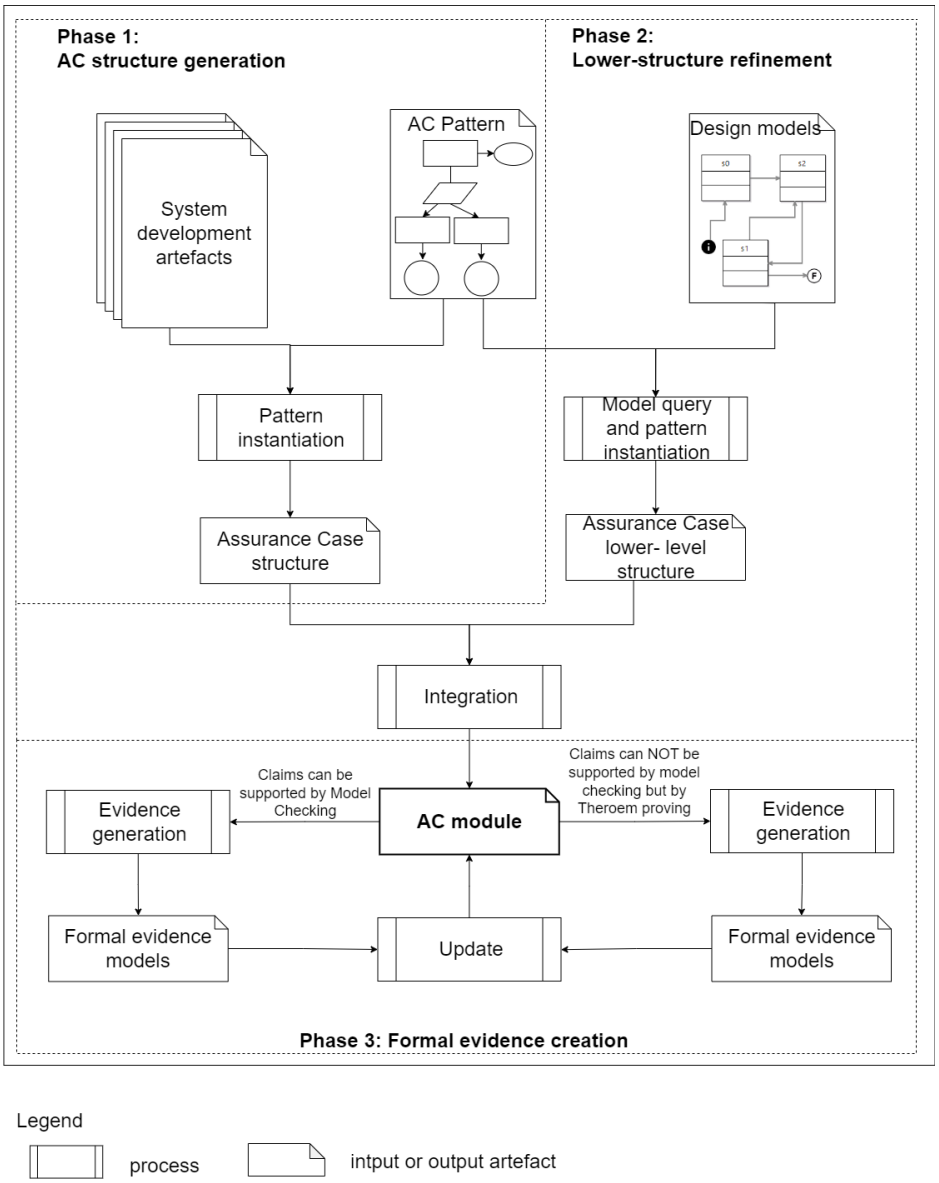


Figure 3.1: AC generation methodology.

Example of Phase 1

Figure 3.2 shows an abstract example of ACs generated in Phase 1 using pattern instantiation. The input of the process can consist of a combination of a hazard analysis result, the system specification, the system design models, and the system assurance plan, in which the corresponding verification

and validation method and result are planned for the properties to be argued in the AC.

The complete structure of AC module as shown in Figure 3.2 is generated automatically. The goals G1 - G12, the solutions So1 - So4, the context C1, and the assumption A1, are generated by pattern instantiation using the system artefacts. The strategies S1 - S7 are automatically created according to the relationships between the artefacts.

Goals G9 and G10 are supported by the system testing results, and therefore the corresponding solutions are instantiated with the test results T-1 and T-2 from the available test reports. Goals G11 and G12, on the other hand, are supported by formal verification techniques, specifically model checking and theorem proving. In this particular example, it is assumed that by the time of this AC module generation, the formal verification activities are planned, but not conducted. Therefore, the formal verification results are not yet available at the time of instantiation. As a result, solutions So3 and So4 are instantiated with placeholders indicating the type of verification results (i.e., a model checking result for So3, a theorem proving result for So4) rather than the actual results.



Figure 3.2: An example AC module generated by pattern instantiation with system artefacts.

Goal G4 for requirement R3 can be further developed into a set of sub-level claims related to the system design by reviewing the system models. In this case, the decomposition of Goal G4 into

its constituent sub-level claims can be done automatically by the model query instead of manual reviewing. The decomposition will be conducted in Phase 2 of the AC process.

Phase 2

It is assumed that the RAS system follows a model-based design development process and an AC module has been generated in Phase 1. During Phase 1, developing lower-level claims to support a system specification claim may require engineers to make design decisions based on the structure of the design models. Rather than manually defining these lower-level claims, these claims can be automatically derived from the structure of the design models using the model-query approach. This automatic derivation is accomplished in Phase 2, where the lower-level structure of the AC is refined from the formal syntax of the robotic design models. This phase specifically applies to systems developed using UML-like modelling languages that possess a hierarchical structure. In the implementation of Phase 2, the RoboChart modelling language is utilized.

The input of Phase 2 is the set of claims generated in Phase 1 that relates to the specification and needs support from the sub-claims related to the design decisions. The model query rule is then designed according to the claim to be refined and the metamodel of the system modelling language. Then, the output, i.e., the lower-level structure of AC, which includes the sub-level claims and the supporting evidence, is generated by querying the hierarchical design models using the query rules. The structure also conforms to the pre-defined AC pattern. An integration process is devised to incorporate this lower structure of the AC into the AC module generated in Phase 1. Phase 2 is presented in Section 4.3 of Chapter 4.

Example of Phase 2

In Phase 2, goal G4, which was generated in Phase 1, undergoes refinement using model query techniques. The resulting AC structure is depicted in Figure 3.3, highlighted by dotted lines. The output AC are obtained by querying the RoboChart models and instantiating the AC pattern. These models are then integrated into the overall AC module to support goal G4. The remaining structure of the AC (i.e., the portion outside the dotted frame), which was generated in Phase 1, remains unchanged and unaffected by this refinement process.

Through the model query, G4 is refined into two sub-level goals, namely G13 and G14. The corresponding solutions, So5 and So6, are also instantiated using the results of the model query. This fragment of AC adheres to the same AC pattern employed in Phase 1. Consequently, two additional goals, G15 and G16, are automatically generated to define the verification method, along with the corresponding strategies, S8 and S9, thus completing the lower-level structure for goal G4.

Phase 3

Once an AC module is generated from Phase 1 (and Phase 2), if there are claims that can be supported by formal verification, in Phase 3, the formal evidence will be automatically derived from these claims using formal verification techniques, as shown in the bottom of Figure 3.1. The formal verification techniques utilized in this phase include:

1. model checking using FDR for non-probabilistic properties
2. model checking using PRISM for probabilistic properties

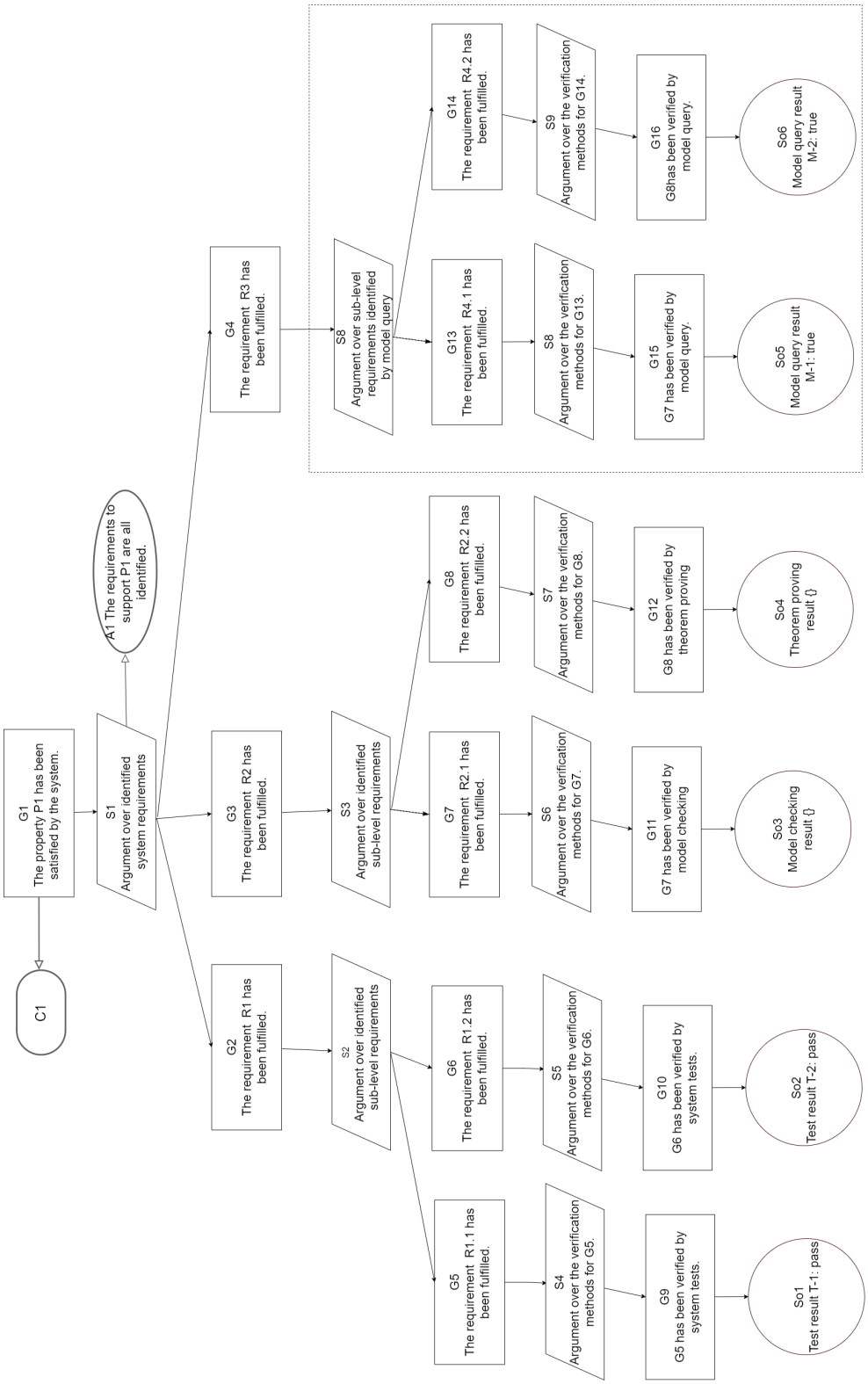


Figure 3.3: AC module of Figure 3.2 with G4 refined by model query.

3. theorem proving using Isabelle/HOL

For a comprehensive coverage of various system requirements, the model checkers FDR and PRISM are utilized in the implementation of the proposed method. The system models used in Phase 3 are still RoboChart models. One advantage of using RoboChart models is their formal semantics, which enables verification using FDR and PRISM. In addition, Isabelle/HOL is explored as an alternative verification approach to address the model checking limitations related to state space explosion and real numbers. Therefore, in Phase 3, when the claims can be supported by verification through model checking, then the evidence is generated from the result of model checking, otherwise if the claims can not be supported by verification through model checking due to the limitations of state space explosion or real numbers, the theorem proving can be considered.

The automated process encompasses four key activities: (1) formalizing the claims that require support from formal verification, (2) converting the system models into formal notations, (3) performing automatic verification using the appropriate tool, and (4) updating the AC module by instantiating it with the verification results.

When utilizing model checking, the process begins by generating formal assertions based on the claims. This involves categorizing the claims into various types, designing an assertion template for each type, matching the claim with the appropriate template, and instantiating the template using the formal semantics of RoboChart.

In the next step, the formalization of the design models is required. The implementation of Phase 3 employs RoboChart models, which can be automatically transformed into the formal semantics of the CSP language or PRISM language within the development tool "RoboTool" for RoboChart. Following this, the relevant model checker is invoked depending on the type of assertions. Subsequently, evidence models are generated based on the results of the verification process. These models are then utilized to update the AC module. The process of generating evidence models through model checking is elaborated upon in Chapter 5.

As mentioned above, it is possible certain claims can not be supported by verification through model checking due to the limitations of state space explosion or real numbers, but by theorem proving. In this case, the process involves formalizing claims as invariants, converting the system models into formal notations (Z Machine notations are used in our method), verifying these invariants in Isabelle/HOL by using automated theorem provers provided through the sledgehammer, and subsequently generating evidence models from the verification results. This is discussed in Chapter 6.

Depending on the stage of the system development process, the old evidence model to be updated can either be the one that was instantiated with placeholders indicating the type of verification results if no verification has been conducted, or it can be a verification result that needs to be re-validated due to the change of the system design.

Example of Phase 3

In Phase 3, the solutions So03 and So04 are updated (as shown in Figure 3.4) using the specific formal verification results, which are automatically generated from goals G7 and G8. The whole structure of the AC module remains unchanged from Phases 1 and 2 except for the content of So03 and So04.

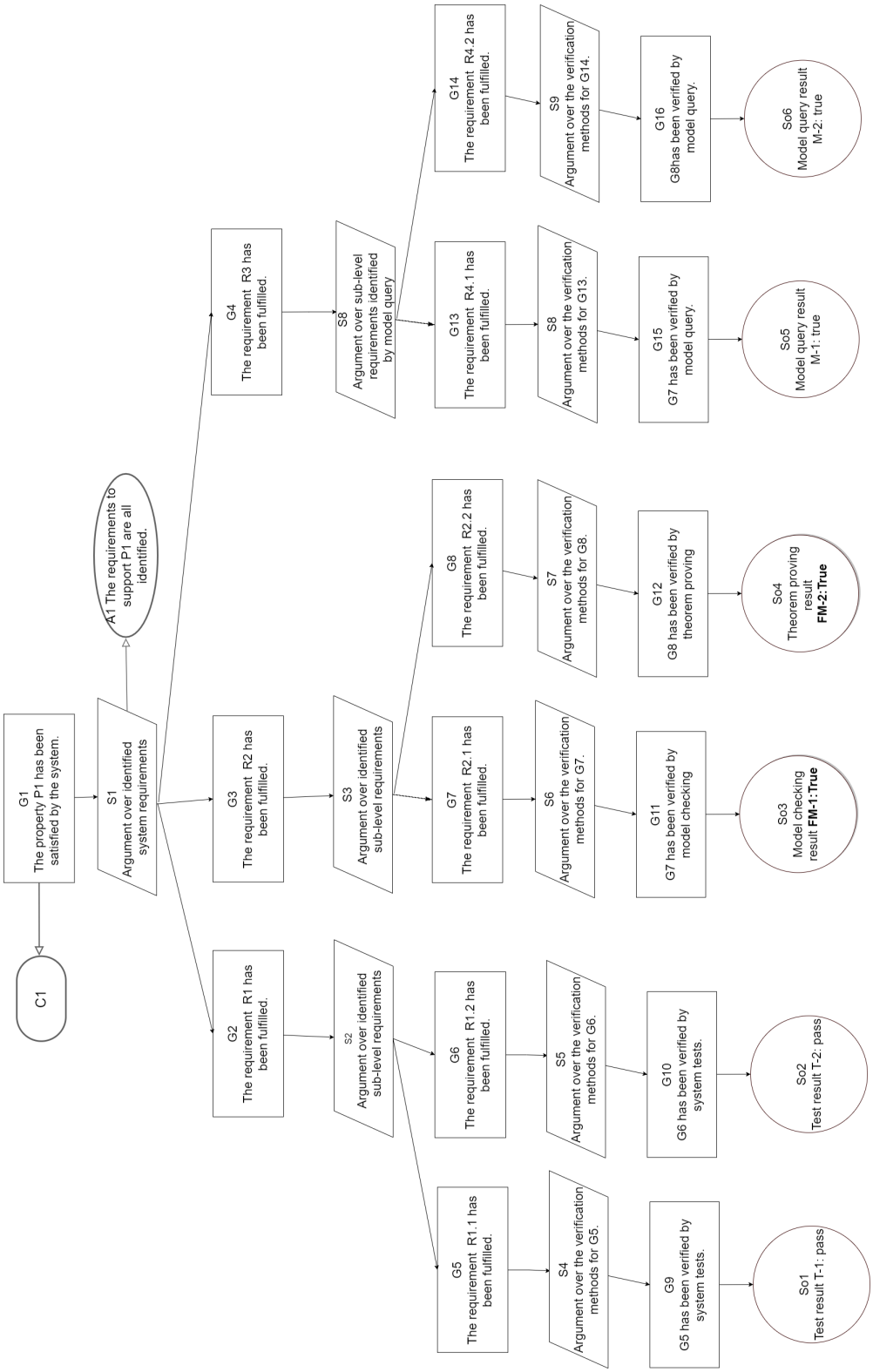


Figure 3.4: AC module of Figure 3.3 with solutions S03 and S04 instantiated with formal evidence.

Case study plan

Several case studies are utilized for evaluating the framework. All the case studies are from the validated case studies in the literature, and we try to avoid altering these cases in order to enhance confidence in their reliability and correctness. As each case study has different features that we need for evaluating our methods, we use each case study to evaluate different methods instead of the whole methodology.

Specifically, the safety monitoring component known as the "Last Response Engine" (LRE) of an autonomous underwater robot [50] is employed to apply the methods in both Phase 1 and Phase 2 in Section 4.5.2 because this example is suitable for creating claims by model query. The effectiveness of evidence generation through model checking in Phase 3 is assessed using a mail delivery robot [84] in Section 5.5.2 and a High Voltage Controller (HVC) [85] in a painting plant in Section 5.5.3; because the system models of the mail delivery robot are probabilistic and can be used to evaluate the evidence generation using PRISM model checker, and the HVC models are non-probabilistic but with timing properties that can be used to evaluate the FDR method. Furthermore, the evidence generation through theorem proving in Phase 3 is evaluated using a 2D version of the LRE in Section 6.6. This case study is used because the system models of LRE involve the parameters of real number type and can not be verified by model checking. Also, a running example of a chemical detector is provided. A summary of the case study arrangement is given in Table 3.1. All the case studies and the implementation of the research framework are accessible online[1].

Table 3.1: Case study arrangement for three phases

Phase	Activity	Chapter	Case study
1	AC structure generation by pattern instantiation	4	LRE [50]
2	Lower-level structure generation by model query	4	LRE [50]
3	Evidence generation through model checking in FDR	5	HVC [85]
3	Evidence generation through model checking in PRISM	5	Mail Delivery Robot [84]
3	Evidence generation through theorem proving	6	2D LRE [50], Chemical Detector [54]

Summary

This chapter presents the methodology for AC generation, which involves the method of AC generation, the method of evidence generation using model checking, and the method of evidence generation using theorem proving. The integration of the methods in the methodology is also explained. In the subsequent chapters, we introduce each method in separate chapters.

[1] https://github.com/uoy-fangyan/PhD_Thesis_implementation.git

Chapter 4

AC Generation

The methods presented in this chapter are aimed at answering the first two research questions concerning AC structure generation:

1. How can we generate assurance cases from the system engineering artefacts?
2. How can we elaborate on the AC structure using robotic design models?

Assurance cases are a well-established practice in high-risk industries for providing confidence in critical system properties, such as safety and security. They rely on system artefacts to support the aforementioned properties. Given the iterative nature of system development, it becomes necessary to update ACs to ensure their validity as the system evolves. Changes in design or the introduction of new hazards, for instance, would require a re-evaluation of claims or the addition of new claims. Consequently, the creation and maintenance of ACs can be a time-consuming and labour-intensive process.

The contribution of this chapter is an automated approach to AC generation that utilizes a combination of the pattern instantiation method and the model query method. A coherent assurance case is systematically and structurally derived from a variety of sources, including hazard analysis results, system models, and other available verification and validation data. A research paper [86] has been published to document the contributions arising from these methodologies, which are expected to be of value to researchers and practitioners in the field of MBE for robotics and autonomous systems.

4.1 Overview

The development of ACs progresses in parallel with the system development process. Due to the iterative nature of this process, ACs must be updated to maintain the validity of claims as the system evolves. Consequently, the continuous updates necessary for AC maintenance make it a time-consuming task prone to human error. This challenge is further compounded by the characteristics of RAS systems, which often operate within uncertain operational boundaries and necessitate frequent post-deployment updates.

This chapter presents in detail the approach to generating SACM-compliant ACs by employing a combination of pattern-based and model-query-based methods. This approach generates AC modules from system artefacts that become progressively available at different stages of system development. The process is iterative in nature and may require revisiting previous steps in response to changes in system development.

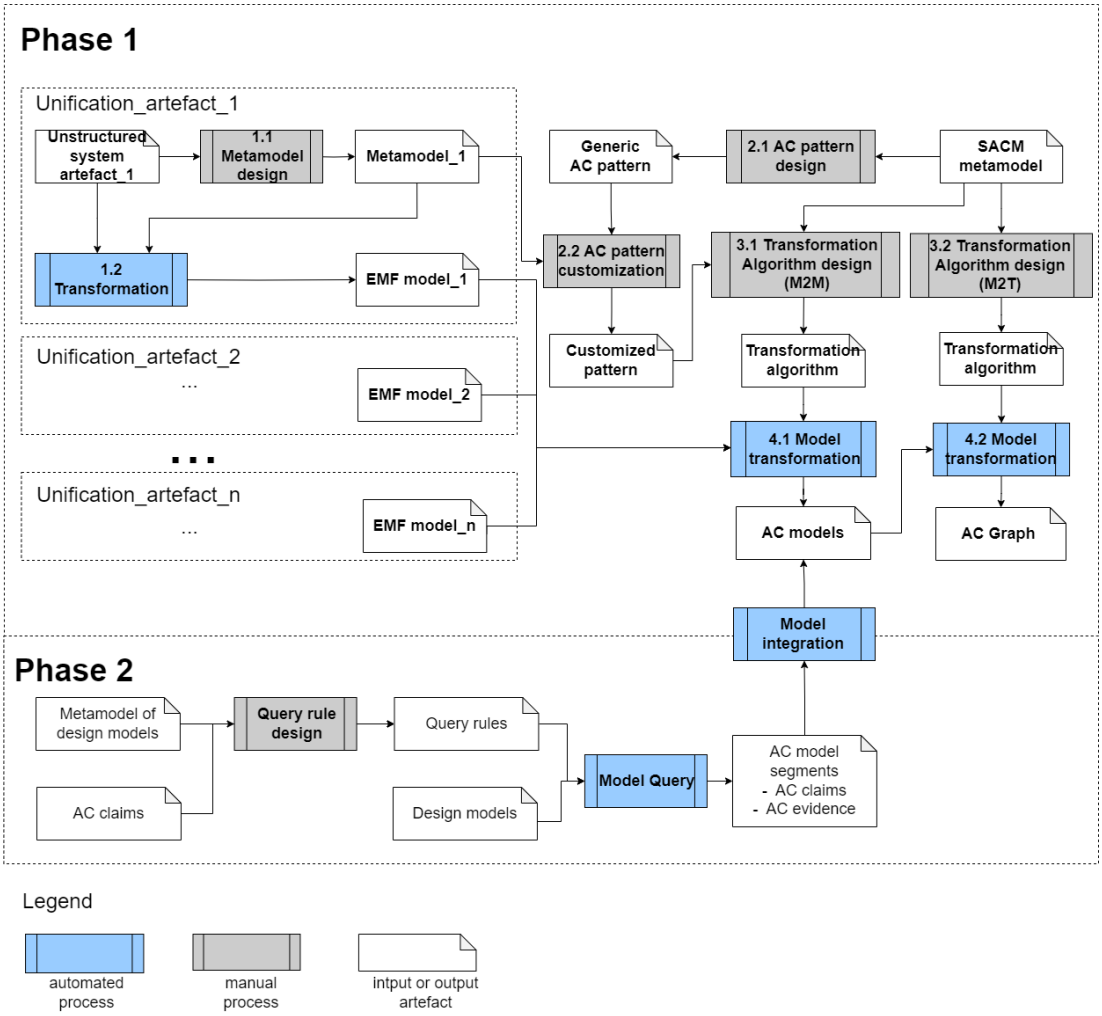


Figure 4.1: AC generation framework.

The approach encompasses the first two phases of the research framework (Figure 3.1), as illustrated in Figure 4.1. The upper half depicts Phase 1: the generation of the AC structure from various system artefacts by pattern instantiation, while the lower section illustrates Phase 2: AC generation through model query. The two phases are connected by the "integration" activity.

The method of pattern instantiation is presented in Section 4.2 and comprises four steps.

1. The initial step involves unifying system artefacts of diverse formats into EMF models. In cases where the system artefact is unstructured, which means it is unsupported by metamodels, the design of a metamodel (activity 1.1 in Figure 4.1) becomes necessary for subsequent transformation (activity 1.2). If the system artefact has a non-Ecore metamodel, it needs to undergo model-to-model transformation to be converted into EMF models. The outcome of

this step is a collection of EMF models, which serve as inputs for the AC process.

2. The second step involves the development of a generic AC pattern (activity 2.1), followed by the creation of a tailored pattern (activity 2.2) for the particular application. The pattern carries the mapping relationships i.e., the traceability between the AC pattern and the system artefacts at the metamodel level. The outcomes of this step are the two patterns.

3. In the next step, the corresponding transformation algorithm is devised based on the mapping relationship in the AC pattern (activity 3.1 and 3.2).

4. Finally, the transformation algorithm is executed (activity 4.1 and 4.2) to achieve automated pattern instantiation. The outcome is the EMF ACs that conform to SACM metamodels. Furthermore, these ACs can be visually represented using GSN notations.

The lower part of Figure 4.1 illustrates the model query method for AC structure generation. Firstly, query rules are defined based on the system metamodel and the AC claims that need to be supported, and then the system models are subjected to model query in order to automatically derive the lower-level AC structure, which encompasses AC claims and evidence. Finally, the AC fragments are integrated into the AC module generated from pattern instantiation. This is discussed in Section 4.3.

The generation of GSN graphics from the ACs will be discussed in Section 4.4.

The method is implemented within the Eclipse development environment. The transformation process makes use of the Epsilon language family [87] to implement the model query, model-to-model transformation, and model-to-text transformation.

The main technical contributions are:

1. A systematic approach for generating and presenting the upper-level structure of SACM-compliant ACs from system development artefacts using pattern instantiation.

2. A method for generating the lower-level structure of ACs from the formal syntax of robotic design models through system model query.

In the rest of this section, an introduction to Epsilon model management language which is used as the main technique for AC generation is given.

4.1.1 Epsilon model management

Epsilon, which stands for Extensible Platform of Integrated Languages for Model Management, is a platform designed for creating consistent and interoperable task-specific languages for various model management activities. These activities include model transformation, code generation, model comparison, merging, refactoring, and validation.

Epsilon languages include the following:

1. Epsilon Object Language (EOL)

2. Epsilon Validation Language (EVL)
3. Epsilon Transformation Language (ETL)
4. Epsilon Comparison Language (ECL)
5. Epsilon Merging Language (EML)
6. Epsilon Wizard Language (EWL)
7. Epsilon Generation Language (EGL)
8. Epsilon Pattern Language (EPL)

EOL is the outcome of efforts to incorporate the navigational mechanisms of OCL while introducing support for additional language features, such as multiple model access, statement sequencing, straightforward programming idioms, and model modification capabilities [88]. EOL reuses a significant portion of OCL, including model querying operations such as **select()**, *collect()*, and *iterate()*. Furthermore, it utilizes a similar syntax for defining variables: *def* <*name* >: <*type* >; and shares an identical type system. EOL serves as the core expression language within Epsilon and forms the basis for task-specific languages used in various activities, including model validation, model-to-text transformation, model-to-model transformation, and model migration. EOL can also function as a versatile standalone model management language for automating tasks that do not align with the specific patterns addressed by task-specific languages [89].

EVL's purpose is to enhance Epsilon with model validation capabilities. To be more precise, EVL enables the specification and assessment of constraints on models that utilize various metamodels and modeling technologies. While OCL's robust syntax allows users to define meaningful and succinct constraints, its purely declarative and side-effect-free characteristics impose certain limitations within a modern model management environment. These limitations in OCL inspired the creation of EVL. In EVL, validation specifications are structured within modules known as EvlModules. These modules can encompass user-defined operations and import other EOL library modules and EVL modules. In addition to operations, an EVL module also comprises a collection of invariants categorized by the context they pertain to, as well as a series of pre and post blocks.

ETL is a hybrid model transformation language that has been developed on the foundation provided by the Epsilon model management platform. Leveraging Epsilon's infrastructure, ETL seamlessly integrates with several other task-specific languages, facilitating the implementation of composite model management workflows [87]. The purpose of ETL is to enhance Epsilon with model-to-model transformation capabilities. ETL is designed to capture and execute transformation scenario specifications that involve an arbitrary number of input and output models from various modeling languages and technologies, all at a high level of abstraction. ETL transformations are structured within modules, each of which can house multiple transformation rules. Each rule is uniquely named and defines one source parameter along with multiple target parameters. Additionally, a transformation rule may inherit from several other transformation rules. Within an ETL module, there is also the option to include a series of pre and post named blocks containing EOL statements. These blocks are executed before and after the transformation rules, respectively.

EGL offers a specialized language for model-to-text transformation (M2T). With EGL, you have the capability to convert models into diverse forms of textual artifacts. These include executable

code (e.g., Java), reports (e.g., in HTML), images (e.g., using DOT), formal specifications (e.g., Z notation), or even complete applications composed of code in multiple languages (e.g., HTML, JavaScript, and CSS). EGL operates as a template-based code generator, meaning that EGL programs closely resemble the text they generate. It offers a range of features to simplify and facilitate text generation from models, including an advanced, language-independent merging engine to preserve manually-written portions of generated text. Additionally, it provides an extensible template system for generating text to various destinations, such as saving it to a file on disk, sending it to a database server, or serving it as a response via a web server. EGL also incorporates formatting algorithms to ensure the resulting text is well-formatted and easily readable, along with traceability mechanisms to establish links between generated text and source models [89].

In this work, we primarily utilized EOL, ETL, and EGL. Detailed information regarding their syntax, metamodels, as well as other languages, can be found in [89].

4.2 AC generation by pattern instantiation

4.2.1 System artefact unification

This section presents the methods employed to unify system artefacts of different formats into EMF models.

During the system development processes, a variety of artefacts are generated, including specifications, designs, architectures, verification and validation reports, etc. These artefacts are often in different formats, such as text, models, and spreadsheets. These artefacts may be unstructured and do not support MBE techniques. In this thesis, unstructured data refers to data that does not adhere to a specific metamodel.

To enable the automatic instantiation of ACs through MBE, it is necessary to transform the artefacts into machine-readable models that are in a consistent format. For instance, the artefacts can be transformed into models that conform to the Ecore metamodel. In this approach, EMF models, supported by Ecore metamodels, are utilized as the target format for unifying the artefacts. It is worth noting that alternative models can be employed as the target format for unification, replacing EMF models.

All unstructured instantiation artefacts need to be transformed into a unified and structured format, specifically EMF models, utilizing Ecore metamodels. To achieve this, a metamodel and a corresponding data structuring algorithm are defined for each type of artefact. All the metamodels should adhere to the same metamodelling language, Ecore, to facilitate the unification process. By applying the algorithm, the artefacts' structure is established in accordance with the corresponding metamodel. The detailed principles for the system artefact unification are as follows,

1. If the data are already in the form of EMF models, no unification is necessary.
2. If the data are models backed by metamodels that are not defined using Ecore (e.g., using MOF), a model-to-model transformation should be performed to obtain EMF models.
3. If the data are written in markup languages, such as XML, HTML, YAML, etc., an Ecore

metamodel should be designed for each artefact. These artefacts can then be converted into EMF models using the model query. This category also includes markup files generated from rich text documents, such as Microsoft Word documents with formatting styles.

4. If the data are in the form of spreadsheets, an Ecore metamodel should be designed for the spreadsheets. EMF models can be generated using model query within the Epsilon environment.

5. If the data are a plain text document without any formatting or markup, a manual process is required. This can involve adding formatting labels and exporting the document into a markup language (e.g., XML), or creating a new file that conforms to a predefined Ecore metamodel.

To summarize, for artefacts that have no metamodels (referred as to "unstructured" in this thesis), it is necessary to define an Ecore metamodel for each artefact type. This metamodel is then used to generate EMF models from the artefacts. For artefacts that do have metamodels but whose type differs from the target models, a model-to-model transformation is necessary.

Once all artefacts have been unified, the instantiation rule for the AC pattern should be designed based on the mapping relationship between the artefacts and the AC pattern. Alternatively, it is possible to directly use models of different metamodel types in the pattern instantiation process without prior standardization. Both approaches are viable, but they require a thorough understanding of the source and target metamodels. In this thesis, the approach adopted involves unifying the artefacts before applying the pattern instantiation rule.

4.2.1.1 System artefact metamodel design

In this section, the process of designing a metamodel for system artefacts is demonstrated using a "hazard table" as an example. The content of this hazard table is similar to a safety assurance plan. When it comes to safety properties, a safety assurance plan is usually prepared, which include the hazard analysis results, the mitigation measures, the verification and validation methods, and the expected results. This data is typically recorded in a spreadsheet. Although the spreadsheet can be seen as "structured" in certain contexts, we consider it as an unstructured format in our work due to the lacking a dedicated metamodel.

To illustrate the generation of ACs from a hazard table, we will use an excerpt from the Autonomous Underwater Vehicle (AUV) case study presented in Section 4.5.

The AUV system [50] is designed for underwater maintenance and intervention tasks, and it can operate manually or automatically under different conditions. One of its primary hazards is the potential collision with infrastructure and unidentified obstacles in the sea. To ensure the safety of system operations, a safety monitoring component called the Last Response Engine (LRE) has been developed. Drawing inspiration from existing works [20, 39], an excerpt from the hazard table of the LRE is given in Figure 4.2. This table includes a top-level hazard, two layers of causes for the hazard, two layers of safety measures to mitigate the causes, as well as the verification and validation methods and results.

The hazard table for the LRE will be utilized to instantiate the AC pattern outlined in Section 4.2.2. However, prior to the instantiation process, the spreadsheet needs to be transformed into EMF mod-

Hazard	Cause	SubCause	Safety Measure	Sub-Safety Measure	Verification	Verification Evidence	Validation	Validation Evidence
H1 AUV collides with obstacles.	C1 AUV system failure leads to AUV collision with obstacle.	C1-1 Operator can not obtain the control from the state in which the operator is not in control when requesting.	SM1 Operator shall obtain the control from any operational state in which the operator is not in control when requesting.	SSM1 Query: each state except OCM should have a outgoing transition whose trigger is reqOCM, and target is OCM.	VR1 Model query: if the required transition exists.	VRE1 The named transition returned by querying if exists.	VA1 Validation of reachability: State {OCM}	VAE1 FDR model checking results.

Figure 4.2: Excerpt of LRE hazard table spreadsheet.

els. A generalized Ecore metamodel is proposed for this artefact. This metamodel, depicted in Figure 4.3, can be customized according to the specific application.

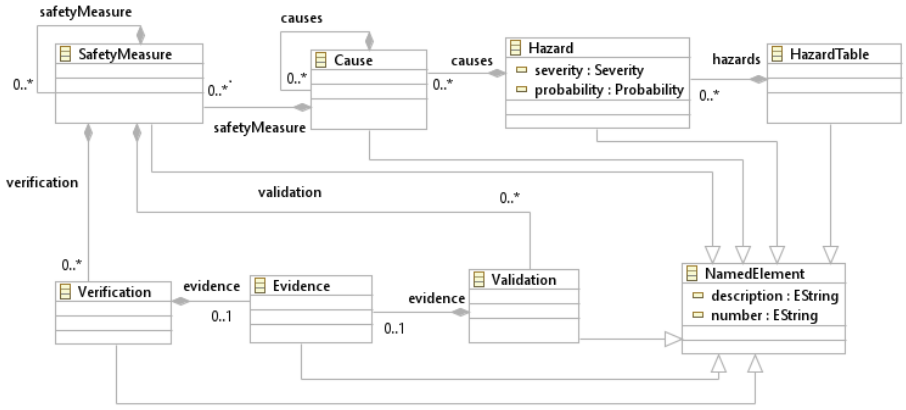


Figure 4.3: Metamodel of a hazard table.

The structure of the metamodel is compliant with the spreadsheet structure. A hazard table is a *HazardTable* in the model. It can have none to several *Hazards*. A *Hazard* has none or one or several *Causes*. A *Cause* can further have lower-level causes (i.e., sub-causes) of class *Causes*. The *Cause* of each level can have multiple safety measures *SafetyMeasure*, which also can have multiple sub-safety measures of the class *SafetyMeasure*. Each *SafetyMeasure* can have multiple *Verification* and/or *Validation* methods. Each verification method has a corresponding *Evidence* which represents the verification result. Also, each Validation method has a corresponding *Evidence* which represents the Validation result. All the classes of the metamodel have the attributes of *description* representing the content of the class, and *number* representing the identification number.

4.2.1.2 Generation of EMF model for system artefact

This section describes the method for converting the system artefact into EMF models using a spreadsheet example.

The one-to-one correspondence between the cells in the spreadsheet (Figure 4.2) and the classes in the metamodel (Figure 4.3) appears straightforward. However, when machines parse the spreadsheet, the interrelationships between the individual items are not explicitly captured. Each cell in the spreadsheet is treated as an independent entity, lack of any connection to others.

For example, human readers may readily infer that the cause **C1** is decomposed into two sub-level causes, **C1-1** and **C1-2**. However, this relationship is not explicitly encoded in the spreadsheet. Consequently, this information cannot be exploited for the purpose of AC generation.

The metamodel in Figure 4.3 provides a data structure to link all the elements in a logical manner. The rules for generating the EMF models for the hazard table are as follows.

1. Each cell in the spreadsheet is converted into an instance of its corresponding class in the metamodel. The type of the cell is decided by the column the cell is in.
2. Within a row, each cell is related to the one in the previous column.
3. If the cell in the previous column is empty, the cell is related to the cell in the previous column of the row above.

An EMF model is then generated from the spreadsheet by implementing the rules above using EOL query language. The corresponding algorithm can be seen in Appendix A Algorithm 7. The generated EMF model is shown in Figure 4.4. Each element in the spreadsheet is mapped to a component in the model. The components are logically interconnected, where the top-level component 'hazard table' has one hazard, the hazard is linked to a cause which is further decomposed into two sub-level causes, and the first sub-level cause is connected to a safety measure, and so forth.

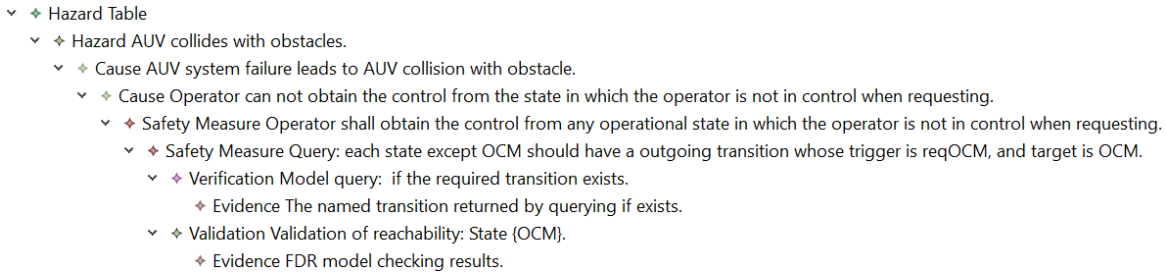


Figure 4.4: Excerpt EMF models of LRE hazard table.

In order to generate AC through pattern instantiation, it is essential to define AC patterns in advance. The subsequent section elaborates on a two-step design process for AC patterns.

4.2.2 AC pattern design

In this section, we introduce a two-step design of the AC pattern. Many of the existing patterns proposed in the literature are specific to certain development environments or applications, therefore difficult to be applied across domains. On the other hand, a generic pattern lacks details to support the application in specific domain. To enable the widespread adoption of our method of model-based AC generation across various domains, we used a two-step pattern design to balance the generality and the applicability between the generic and specific patterns.

Step 1: Generic patterns

Firstly, a generic AC pattern is designed (Figure 4.5) to set constraints on the relationships among AC elements. This facilitates the design of instantiation rules and enhances the feasibility of model-

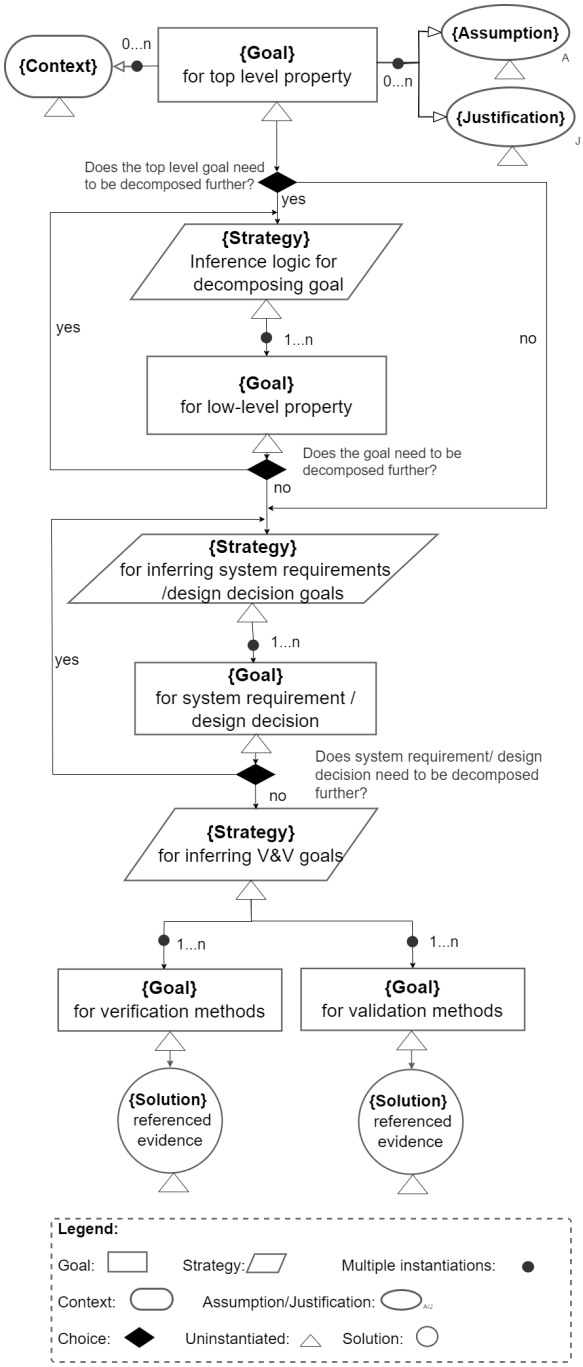


Figure 4.5: An generic AC pattern in GSN notation.

based automation for AC generation. Then, during the second step of customization, users have the flexibility to determine the system artefacts to be utilized and how they should be mapped to the AC instantiable elements in the pattern.

The pattern is presented using GSN notation. In Figure 4.5, using the GSN notations, goals are depicted as rectangles, strategies as parallelograms, solutions as circles, assumptions and justifications as ovals, and contexts as rounded rectangles. A solid arrowhead line connects goals, strategies,

and solutions, representing a top-down direction. A hollow arrowhead line connects contexts, assumptions, and justifications to goals or strategies. The GSN extensions for structural abstraction pattern [23] are utilized, including a hollow triangle to label the element that remains to be instantiated, and a solid ball with a cardinality to represent multiple instantiations. The flowchart element is also incorporated into the pattern using a solid diamond accompanied by a condition between the goal and the strategy as a choice.

The generic pattern starts from the top level goal for the top-level property, which may be supported by the assumptions, justifications, or the context. Then, if the top-level goal can be further decomposed, a set of lower-level goals is defined. The process of the decomposition iterates until there is no need to further decompose the properties. Subsequently, the goals are created for the system requirements and/or the design decisions that are used to support the decomposed property. Depending on the complexity of the system development, there might be more than one level of requirements or system design, in which case the AC structure construction follows an iterative process. The goals of the lowest level identify the verification and validation methods used to support the upper-level goals. The AC structure concludes with the solutions that reference the corresponding results of verification and validation activities.

According the description above, the generic pattern constraints the AC structure in the following ways,

1. The pattern is structured hierarchically in a top-down manner, consisting of two main layers of goals: one for the property and another for system specification and design. It begins by establishing a top-level goal for a property, which can be further decomposed into a set of goals for lower-level properties. The structure then progresses towards the goals related to system requirements and/or design decisions necessary to fulfil these properties. The element of 'multiple instantiation' is used for the decomposition process.

2. The lowest-level goals are always the claims stating that the upper-level goals have been verified/validated with certain methods. The solutions are the corresponding verification and validation results.

3. The structure is constructed recursively. Depending on the abstraction level of the top goal, the decomposition may be applied multiple times. Moreover, the complexity of the system development process can range from being simple to having multiple phases. Accordingly, the goals related to system specifications or design decisions may be decomposed into multiple sub-level goals. The pattern utilizes recursive loops to represent the iterative nature of goal decomposition at different levels. The presence of the 'choice' element in the pattern indicates when recursion may be required.

4. For the purpose of automatic instantiation, there should always be a strategy linking the goals of two adjacent levels, while the solution and the directly supported adjacent goal are linked without a strategy.

5. All the elements are instantiable and therefore labelled with the element of 'uninstantiated'.

6. The goals and strategies at any level may have optional context, assumptions or justifications. For simplicity, Figure 4.5 only shows this for the top-level goal and the first strategy.

The top-level goals may vary depending on the nature of the property being argued. However, the AC structure outlined in the pattern is generally applicable. This pattern only covers properties that are related to system design, and does not cover those pertaining to quality assurance and configuration management.

Step 2: Customized patterns

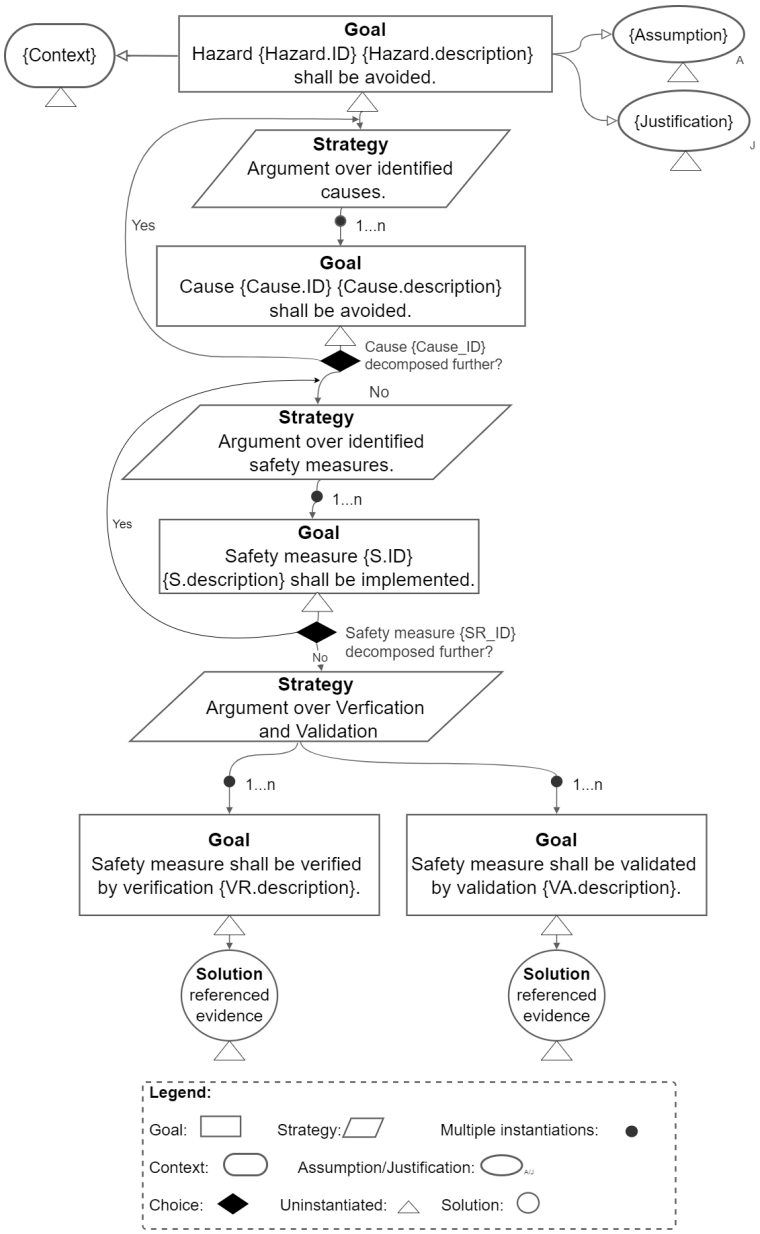


Figure 4.6: AC pattern instantiated with hazard table.

The generic AC pattern (Figure 4.5) establishes the AC structure and identifies the elements that need to be instantiated. However, the process of instantiating the pattern relies on mapping the concrete instantiation data to the instantiable elements within the pattern. Therefore, the pattern should be further customized according to the specific applications and their system artefacts involved for AC generation.

We utilize the hazard table of LRE (Figure 4.2) as an illustrative example to tailor the generic AC pattern for safety properties. In the case of instantiating the AC using hazard table models, the generic AC pattern is no long sufficient to support the process and needs to be adjusted to align with the data involved for safety property argument. For example, the top-level goal will be related to a hazard instead of a property of the system as in the generic pattern.

The modified pattern is displayed in Figure 4.6. This customized pattern for LRE incorporates specific **mapping rules** that connect the elements of the pattern to the corresponding elements within the hazard table metamodel. For example, the top-level goal is that the hazard should be mitigated. The mapping rules are summarized as follows.

1. The elements of hazard, cause, safety measures, verification, and validation of the hazard table map to AC goals, and they are integrated into the pattern's structure following the same order as in each row of the hazard table.
2. A strategy is established between every two adjacent levels of goals. There is no strategy between goals related to verification/validation methods and their corresponding solutions.
3. The content of the strategy for each level is determined by the contents of two adjacent levels of goals. For example, the strategy between a hazard and a set of causes is 'Argument over identified hazard causes'.
4. The goals for causes and safety measures can be decomposed once.

This pattern identifies the traceability between the pattern elements and the elements of the hazard table metamodel. Once this pattern is mechanized into the transformation algorithm, thanks to the established traceability, the AC can be automatically updated whenever the system artefacts change.

4.2.3 Pattern instantiation

This section discusses the automated generation of the AC through pattern instantiation. To begin with, the customized pattern needs to be mechanized into a transformation algorithm in Epsilon ETL transformation language according to the SACM metamodel. Then, the pattern should be instantiated using the EMF models of the input artefacts by executing the transformation algorithm. As a result, SACM-compliant ACs are produced.

It is possible that an AC is generated by using a set of system artefacts as instantiation data instead of using a single artefact. A generic algorithm for instantiation from a set of system artefacts is given in Algorithm 13 "AC Generation by Pattern Instantiation". The algorithm is presented in Appendix A.

To better illustrate the method, an algorithm specific to the AC pattern instantiation using the hazard table (Figure 4.4) is given in Algorithm 1 as an example. The algorithm is an implementation of the mapping rule between the hazard table and the AC elements in Figure 4.6.

The instantiation process is automated through the use of Epsilon ETL [89] model transformation techniques. The ACs conform to the SACM metamodel, and thus, they utilize the corresponding SACM terminology. For instance, an "Argument Package" corresponds to a GSN module, a "Claim"

corresponds to a GSN goal, an "Argument Reasoning" corresponds to a GSN strategy, and an "artefact Reference" corresponds to a GSN solution. A mapping of the terminologies between SACM and GSN is summarized in Section 2.1.2, and more details are in [90].

In Algorithm 1, the input is the EMF model of the hazard table *ht.model*. According to the metamodel of the hazard table in Figure 4.3, a hazard table *ht* has a set of hazards *ht.hazards*; each hazard *h* in this set may have a set of causes *h.causes*; each cause *c* in the set *h.causes* may have a set of lower-level causes *c.causes*; each lower-level cause *subc* in the set *c.causes* may have a set of safety measures *subc.safetyMeasure*; each safety measure *sm* in the set *subc.safetyMeasure* may have a set of lower-level safety measures *sm.safetyMeasure*; each lower-level safety measure *subsm* in the set *sm.safetyMeasure* may have a set of verification methode *subsm.verification*; each verificaiton method *vr* to the safety measure *subsm* in the set *subsm.verification* may have a set evidence *vr.evidence*; similarly, each *subsm* may have a set of validation methode *subsm.validation*; each validation method *va* may have a set evidence *va.evidence*.

Firstly, an AC module *ac* is created (line 1) by transform the hazard table into an *ArgumentPackage*. The corresponding ETL code for transformation is shown in Figure 4.7. The ETL code transforms each instance of class *HazardLog* into an instance of class *ArgumentPackage* of SAMC metamodel.

```
1 rule hazardlogToArgumentPackage
2 transform a : S!HazardLog
3 to b : T!ArgumentPackage
4 {
5 }
```

Figure 4.7: ETL transformation for an argument package.

Then, the top-level claim *hClaim* is created and instantiated using hazard *h*. The AC module *ac* has a set of argumentation elements *ac.argumentationElement* according to the SACM metamodel. The claims, inferences, evidence are all argumentation elements of the AC module. Therefore, the claim *hClaim* is added to the set of *ac.argumentationElement* through Algorithm 8. Next, if hazard *h* has a cause *c* defined (line 4), then a SACM AssertionInference *hinf* is created (line 5) as the linkage between *hClaim* and the claims for the causes. According to the metamodel of argument package in Figures 2.4 and 2.5, an AssertedInference which stands for the inference link between two adjacent levels of claims may have one target claim, multiple source claims, and one *Reasoning* which corresponds to a strategy of GSN notation. Taking *hinf* as an example, its target is *hinf.target*, source as *hinf.source*, and reasoning as *hinf.reasoning*. The target of *hinf* is the claim *hClaim*, and the reasoning content of *hinf* is the strategy whose content is predefined according to the nature of the two adjacent levels of claims. This is implemented by Algorithm 9. Then, all the causes of this hazard *h* are processed by creating a claim *cClaim* for the cause (line 7). The corresponding ETL code for creating a claim for a cause is shown in Figure 4.8. Using keyword *transform*, each cause is transformed into a claim. Within the transformation, the ETL code begins by generating the name of the claim (lines 14-17). Subsequently, the content of the claim description is created (lines 19-27).

Figure 4.8 provides a visual representation of the ETL code, illustrating the step-by-step process involved in transforming causes into corresponding claims.

The *cClaim* is further added as a source into *hinf* (line 8). The rest of the algorithm repeats the same pattern to create claims for the components of sub-level causes *subc*, safety measure *sm*, sub-level safety measure *subsm*, verification method *vr* and validation *va*, and generates linkages among them. Finally, the verification evidence and validation evidence are created. Algorithm 10 and

Algorithm 1: AC Generation by Pattern Instantiation - Hazard Table

```
1  ac = ht.CREATEARGUMENTPACKAGE()
2  for h in ht.hazards do
3      hClaim = CREATECLAIMANDADDTOAC(ac, h)                          ▷ Alg. 8
4      if ¬EMPTY(h.causes) then
5          hInf = CREATEASTINFANDADDTOAC(ac, hClaim)                  ▷ Alg. 9
6          for c ∈ h.causes do
7              cClaim = CREATECLAIMANDADDTOAC(ac, c)
8              hInf.source = hInf.source ∪ {cClaim}
9              if ¬EMPTY(c.causes) then
10                 cInf = CREATEASTINFANDADDTOAC(ac, cClaim)
11                 for subc ∈ c.causes do
12                     subcClaim = CREATECLAIMANDADDTOAC(ac, subc)
13                     cInf.source = cInf.source ∪ {subcClaim}
14                     if ¬EMPTY(subc.safetMeasure) then
15                         var subcInf = CREATEASTINFANDADDTOAC(ac, subcClaim)
16                         for sm ∈ subc.safetyMeasure do
17                             smClaim = CREATECLAIMANDADDTOAC(ac, sm)
18                             subcInf.source = subcInf.source ∪ {smClaim}
19                             if ¬EMPTY(sm.safetMeasure) then
20                                 smInf = CREATEASTINFANDADDTOAC(ac, smClaim)
21                                 for subsm ∈ sm.safetyMeasure do
22                                     subsmClaim = CREATECLAIMANDADDTOAC(ac, subsm)
23                                     smInf.source = smInf.source ∪ {subsmClaim}
24                                     CHECKANDCREATESTRUCTUREFORVR(ac, subsm,
                                        subsmClaim)                  ▷ Alg. 10
25                                     CHECKANDCREATESTRUCTUREFORVA(ac, subsm,
                                        subsmClaim)                  ▷ Alg. 11
26                                 end
27                             end
28                         end
29                     end
30                 end
31             end
32         end
33     end
34 end
```

Algorithm 11 implement the generation of claims for the verification and validation methods, as well as the evidence to support these claims. All the algorithms referenced in Algorithm 1 are presented in Appendix A.

The complexity of Algorithm 1 is $O(n^5)$, because of the five nested for loops from lines 2-34, 6-32, 11-30, 16-28, and 21-26, respectively. The value of n depends on the scale of the hazard table *ht*, e.g., the number of hazards in *ht*, the number of the causes of each hazard, etc. Assuming that each element has lower-level elements, the number of the lower-level elements is less than or equal to 5. Therefore, for the worst case, the complexity is 3125. Here, we also considered the for loop in the called functions in Algorithm 1.

An excerpt of the AC for LRE derived from the hazard table is shown in Figure 4.9. An *Argument-*

```
@lazy
rule causeToClaim
transform a : S!Cause
to b : T!Claim
{
var name = new T!LangString;
b.name = name;
b.name.lang = "English";
b.name.content ="Claim for " + a.number ;

var description = new T!Description;
var content = new T!MultiLangString;
var value = new T!LangString;
value.lang = "English";
value.content ="Cause " + a.number + " {" +  a.description +" } has been avoided." ;

b.description = description;
b.description.content = content;
b.description.content.value.add(value);
}
```

Figure 4.8: ETL transformation from a cause to a claim .

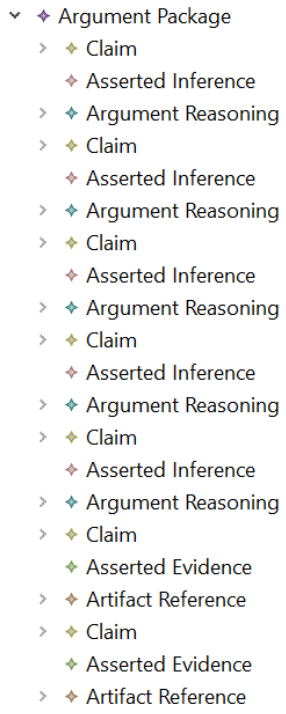


Figure 4.9: AC derived from LRE hazard.

Package is generated for the hazard table. Using the hazard, cause, safety measures, verification, and validation data from the LRE hazard table, eight claims are generated and instantiated. The logical relationships between the components are established using the relationship components of SACM, such as $AssertedInference$ and $AssertedEvidence$. The claims related to the hazard, cause, and safety measures are connected via the $AssertedInference$ component, while the claims related to the verification and validation are linked to the evidence through the $AssertedEvidence$ component.

The correctness of Algorithm 1 is verified using this LRE example. According to the hazard table in Figure 4.4, the AC module generated through Algorithm 1 should have eight claims and corresponding inference relationships, including six relationships of the asserted inference type between

claims and two relationships of the asserted inference type between claims and the evidence, and two evidence of the class *Artifact Reference*, etc. These requirements are met by checking the AC module generated as shown in Figure 4.9.

All the algorithms in this work terminate because these algorithms (except Algorithms 15, 16, 17) only used the for and if-else structure within the for loop. The for operator iterates all the elements of a given set. As the input of the algorithm can be either the hazard analysis table, the RoboChart models, or the SACM-compliant ACs, these inputs all have a **finite** set of the elements of certain class. Therefore, the for loop always terminates. For example, in line 2 of Algorithm 1, the for operator processes all the hazard *h* in the hazard table *ht*. Because the number of hazards in *ht* is finite, the for loop terminates. Algorithms 15, 16, 17 also terminate, and will be explained in Appendix A.

4.3 AC generation by model query and integration

This section discusses the method of AC generation and integration of AC segments through querying system design models as illustrated in the lower part of Figure 4.1.

Besides the pattern instantiation method described in Section 4.2, a complementary approach is to directly generate the AC structure from design models. This method facilitates the establishment of traceability between the system models and the elements of the lower structure of AC, thereby enabling automated updates of ACs. The design models refer to models written in UML-like system modeling languages (such as AADL, SysML, RoboChart) that are supported by their respective metamodels and typically exhibit hierarchical characteristics.

4.3.1 An abstract running example

In the ACs of RAS systems, certain claims within the lower structure of AC can be supported and justified by the architecture of the design models. For instance, a RAS system is developed using RoboChart models, and its hazard analysis is presented in Figure 4.10. The hazard *H1* and its corresponding hazard cause *C1* are supported by the safety measure *SM1*.

SM1: Every state meeting Condition 1 shall have an outgoing transition meeting Condition 2.

Hazard	Cause	Safety Measure	Sub-Safety Measure	Verification	Verification Evidence
H1	C1	SM1 Every state meeting Condition 1 shall have a transition meeting Condition 2.	State a shall have a transition meeting Condition 2	Manual reviewing of system models	Review result: State a has a transition t1 meeting Condition 2
			State b shall have a transition meeting Condition 2	Manual reviewing of system models	Review result: State b has a transition t2 meeting Condition 2.

Figure 4.10: Hazard table for an abstract safety property of a state machine.

Assuming that within the RoboChart state machine, there exist two states, namely *a* and *b*, both of which satisfy *Condition 1*. Additionally, each state has a transition that satisfy *Condition 2*, namely

t1 and *t2*, respectively. Consequently, *SM1* can be decomposed into two sub-level safety measures, each of which can be verified using transitions *t1* and *t2*. The information above is obtained through a model review conducted by engineers. Then, an AC module (Figure 4.11) can be generated through pattern instantiation using the hazard table.

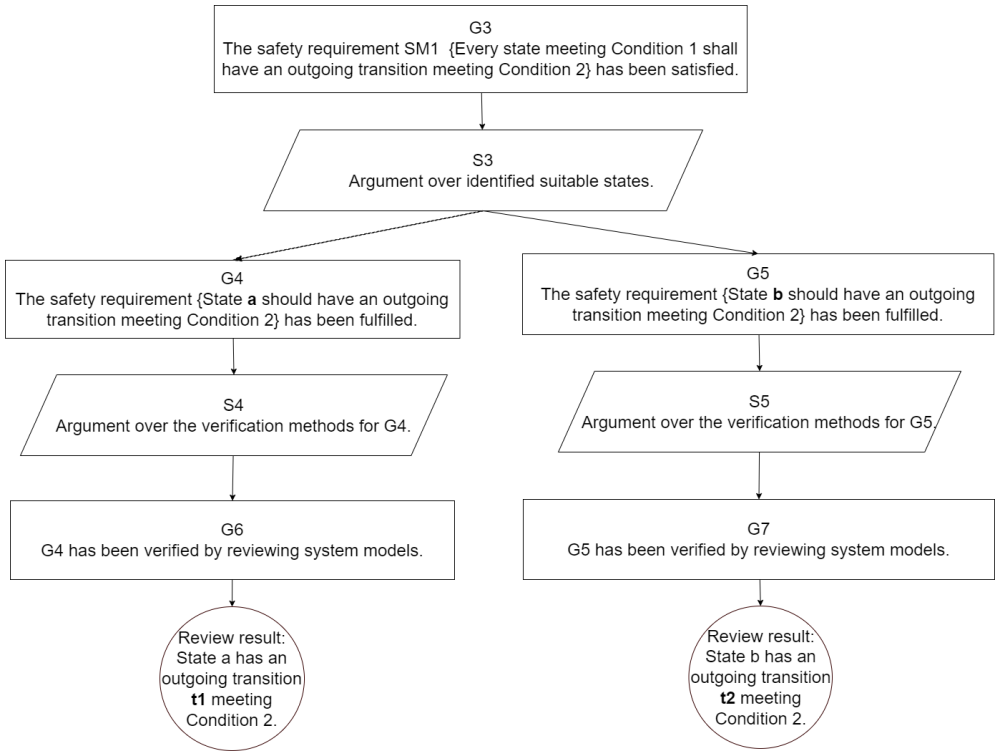


Figure 4.11: AC fragment for the abstract example.

In this example, for the instantiation of the AC pattern, the engineer is required to conduct a manual review of the design models in order to identify the states that satisfy Condition 1 and the transitions that fulfill Condition 2. Alternatively, this process can be automated through the use of system model query.

4.3.2 Model query method

This section discusses the AC generation by model query. When the model query method is employed, the content of the hazard table can be simplified, as shown in Figure 4.12, where the sub-level safety measures are left unspecified and will be further refined through the model query process.

Hazard	Cause	Safety Measure	Sub-Safety Measure	Verification	Verification Evidence
H1	C1	SM1 Every state meeting Condition 1 shall have a transition meeting Condition 2.	To be specified by model query	Model query	Model query result

Figure 4.12: Hazard table for an abstract safety property of a state machine.

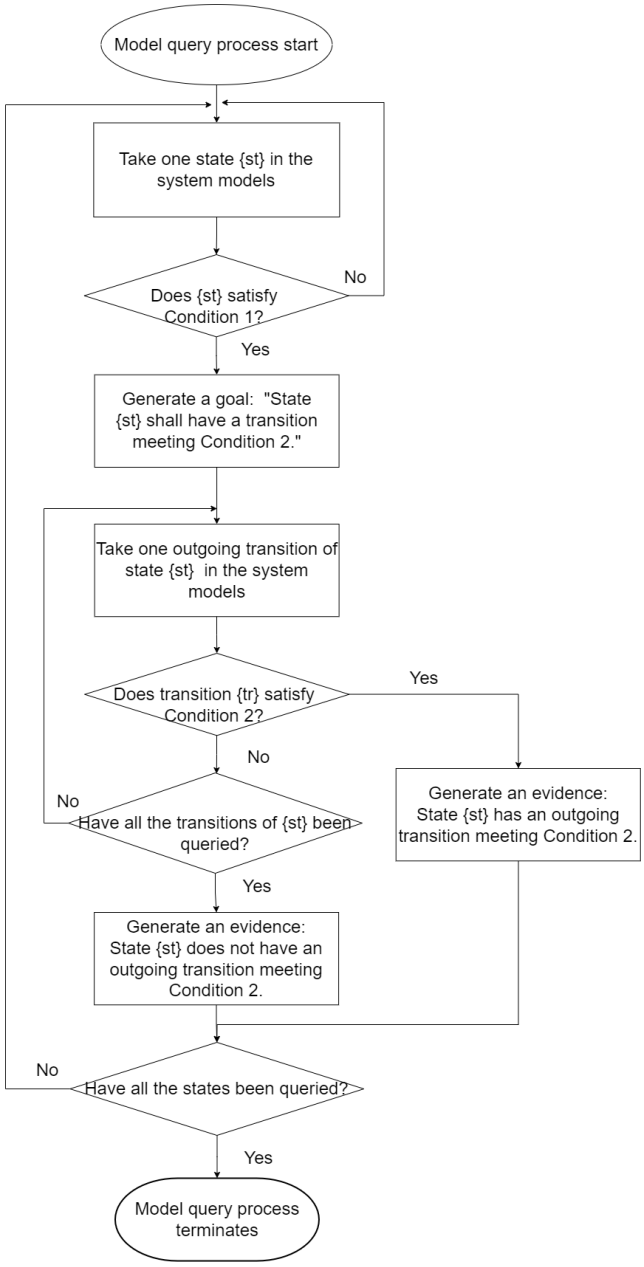


Figure 4.13: Model query process for SM1.

Then, the model query rule should be designed for *SM1*. The procedure for the query rule for this example is illustrated in Figure 4.13. Firstly, each state in the RoboChart models should be examined to determine if it satisfies Condition 1. If a state satisfies Condition 1, a goal should be created stating "State *st* should have a transition that satisfies Condition 2". *st* is the concrete state name, such as *a*. Next, all outgoing transitions from state *st* should be checked to identify any that satisfy Condition 2. If no transition meeting Condition 2 is found for a particular state, evidence should be generated indicating that the goal is not satisfied. Once all states have been queried using the aforementioned procedure, the process terminates. By implementing this query procedure, an AC structure is generated the same as the fragment of Figure 4.11 below strategy **S3** but in an automatic way. The corresponding algorithm that implements the procedures of Figure 4.13 is provided in

Algorithm 14 of Appendix A.

The query rules can be implemented in a query language suitable for the user's development environment and stored in a library for future reuse. Embedded model query rules establish traceability between design models and ACs, enabling automatic updates to the AC when modifications are made to the system models.

In this example, the structure of RoboChart models, which includes state machines, states, and transitions, is used for illustrative purposes. However, the concept of AC generation through model query is not restricted to RoboChart and can be applied to any UML-like architectural language in general.

4.3.3 Integration

The ACs generated through pattern instantiation can provide a complete AC structure, under the assumption that all the system artefacts used as inputs in the AC process will be available at a certain stage. When the claims are related to the system design models, we can employ the pattern instantiation method to generate only the upper structure of the ACs and utilize the model query to refine the lower-level structure of AC. Then, the upper and lower structures need to be integrated into a unified AC module.

A comprehensive framework for AC generation, which combines the methods of pattern instantiation and model query, is already given in Figure 4.1. The detailed procedure for integration is depicted in Figure 4.14.

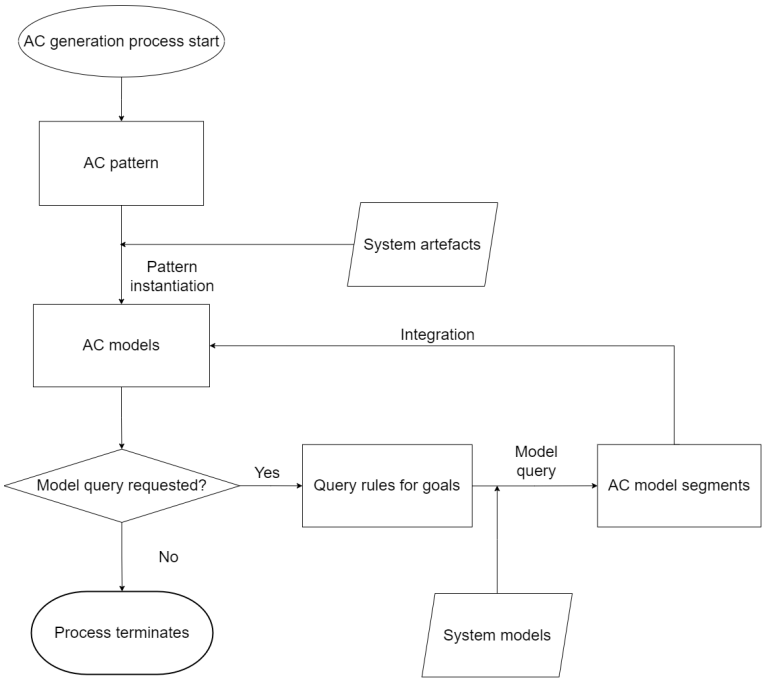


Figure 4.14: A procedure to integrate the AC structures generated through pattern instantiation and model query.

The procedure begins with AC generation through pattern instantiation, which involves designing

the AC pattern and instantiating it using system artefacts. If there are claims that can be further refined using model query, a query rule is defined for the claim. Otherwise, the procedure terminates. After designing the query rule, the lower structure of the AC is derived from the system models and integrated into the AC structure generated through pattern instantiation. Further, if there are no more claims requiring refinement through model query, the procedure terminates.

4.4 AC graphical presentation

This section discusses the strategy to generate GSN graphics from the AC modules. We use model-to-text transformation to generate DOT files from AC modules, and visualize the DOT files as graphs in PDF format.

Figure 2.1 displays the major shapes used in GSN notation, which comprise rectangles representing goals, parallelograms representing strategies, circles representing solutions, and others. To establish the relationships between goals, strategies, and solutions, solid arrowhead lines are utilized, indicating a top-down direction and referred to as "SupportedBy". The contextual elements, including context, assumptions, and justification are linked to the goals or strategies using hollow arrowhead lines and referred to as "InContextOf".

On the other hand, within the SACM metamodel, an *AssertedInference* entity captures the relationship between a target claim and a set of source claims. In this relationship, the lower-level claims serve as the sources of the link to support the upper-level claim, while the upper-level claim acts as the target. Additionally, a *AssertedInference* can have a *ArgumentReasoning* entity that provides the inference basis for decomposing the target claim into the set of source claims. This relationship is illustrated in Figure 4.15, where the context, justification, and assumption are omitted in the figure for simplicity.

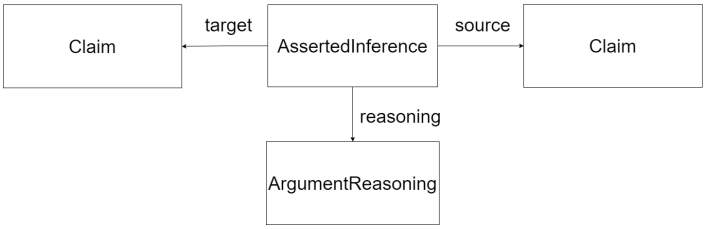


Figure 4.15: *AssertedInference* with related elements of AC model.

Similarly, an *AssertedEvidence* entity captures the relationship between a target claim and a set of evidence, with an optional *ArgumentReasoning* indicating how the evidence is related to the target claim. However, if the generic AC pattern depicted in Figure 4.5 is followed, the *AssertedEvidence* entity does not include a *ArgumentReasoning* because the solution and the directly supported adjacent goal are linked without a strategy in the pattern. This relationship is illustrated in Figure 4.16.



Figure 4.16: *AssertedEvidence* with related elements of AC model.

Furthermore, an *AssertedContext* entity establishes a connection between contextual information and either a claim or an *ArgumentReasoning* entity.

It is important to note that each *AssertedInference*, *AssertedEvidence*, and *AssertedContext* can have only one target component. Consequently, we can utilize these distinct relationship types to categorize the AC model structures into different sections. Subsequently, the graphical presentation can be generated based on these sections.

To facilitate the automation of AC graph generation, we group the SACM-based AC module into three parts, and then the AC structure in GSN notations in three parts to match the model grouping. With this grouping, we generate the graphics for each group and then integrate them into complete graphics.

An AC module is grouped into three types of "sections": (1) *ai* sections, which consist of one *AssertedInference* entity, its target claim, reasoning, and possible context, assumptions, and justifications. (2) *ae* sections, which encompass one *AssertedEvidence* entity, its target claim, a set of evidence, and the potential context, assumption, and justification. (3) *claim* sections, which encompass the undeveloped claims. The undeveloped claims refer to the ones that serve as the sources of certain *AssertedInference* relationships but are not the targets of any *AssertedInference* or *AssertedEvidence* relationships.

Correspondingly, an AC represented in GSN notation is also grouped into three sections: (1) goal-strategy sections corresponding to the *ai* section of ACs, (2) goal-solution sections corresponding to *ae* section of ACs, and (3) the undeveloped goals, which have no other components connected below them. The grouping of an AC module in GSN notation is illustrated in Figure 4.17 where there are three goal-strategy sections highlighted in yellow, two goal-solution sections highlighted in green, and two undeveloped goals marked in purple.

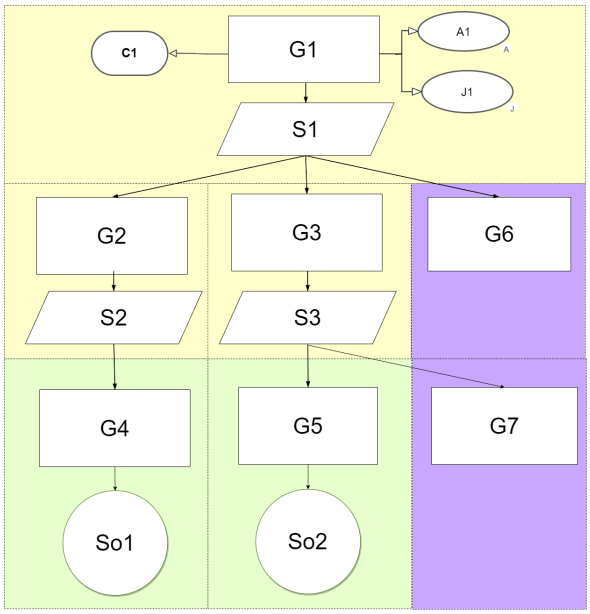


Figure 4.17: The grouping of an AC module in GSN notation.

Based on the aforementioned grouping concept, Algorithm 2 is employed to generate the AC graph. In line 1, two variables are defined: i, which represents the index of the goals to be drawn, and

ai_set_drawn, which denotes the set of the *AssertedInference* models that have been processed.

Algorithm 2: DOT generation

```
1  i = 1, ai_set_drawn = {}
2  for ai ∈ ac.argumentationElement.select(i|i.type = AssertedInference) do
3  |  ai.DRAWAI(i, ai_set_drawn)                                   ▷ Alg. 15
4  end
5  for ae ∈ ac.argumentationElement.select(i|i.type = AssertedEvidence) do
6  |  ae.DRAWAE(i, ai_set_drawn)                                   ▷ Alg. 16
7  end
8  for claim ∈ac.argumentationElement.select(i|i.type = Claim) do
9  |  claim.DRAWUNDEVELOPEDGOAL(i)                                 ▷ Alg. 17
10 end
```

The first step (line 2 to line 4) involves drawing the goal-strategy sections and connecting them together. This is achieved by invoking the DRAWAI() function for each $AssertedInference$ instance in the ACs.

The second step (line 5 to line 7) involves drawing the goal-solution sections and linking them to the goal-strategy sections. This is accomplished by calling the DRAWAE() function for each $AssertedEvidence$ instance in the ACs.

Lastly, the third step (lines 8 to 10) identifies and draws the undeveloped goals, and establishes connections between them and the corresponding strategies in the goal-strategy sections. This is done by invoking the DRAWUNDEVELOPEDGOAL() function for each *claim* in the ACs.

Algorithm 15 presents the implementation of the DRAWAI() function. It is responsible for transforming an *ai* section from the SACM-compliant AC module into a goal-strategy section in GSN notation, while also establishing a link to an upper-level *ai* section.

Similarly, Algorithm 16 outlines the DRAWAE() function. It follows a similar process to Algorithm 15, converting an *ae* section of the AC module into a goal-solution section in GSN notation, and linking it to an upper-level *ai* section.

Algorithm 17 presents the implementation of the DRAWUNDEVELOPEDGOAL() function. In certain stages of system development, it is possible that the AC module is incomplete, with some claims remaining undeveloped. DRAWUNDEVELOPEDGOAL() function is utilized to draw these claims.

Algorithm 15, Algorithm 16, and Algorithm 17 are presented and detailed explained in Appendix A.

The complexity of Algorithm 2 is $O(n)$ because of the three parallel for loops from lines 2-4, 5-7, and 8-10. n is the largest among the number of all the asserted inferences, the number of the asserted evidence, and the number of the claims in the *ac* module. If we take into account the for loops of the called functions, The complexity will be $O(n^2)$ because there are two nested for loops.

To ensure the correctness of Algorithm 2, the AC graphics generated through Algorithm 2 should have the same numbers of claims, inference relationships, and evidence as the AC module, and have the right linkages between these elements that map to the relationships recorded in the asserted inference and asserted evidence, i.e., the correct sources and targets. These requirements are verified by using the LRE example.

The transformation from AC modules to a graphical presentation can be performed at any phase of the AC process, regardless of the completeness of the ACs. The AC process is incremental, and the resulting AC graphics serve as a convenient means of presentation for the AC stakeholders throughout the development process.

4.5 Case study and evaluation

4.5.1 Implementation

The method is implemented within the Eclipse development environment. The transformation process makes use of the Epsilon language family [87], with the Epsilon Object Language (EOL) specifically employed for converting system artefacts into EMF models and querying the RoboChart design models. The automated generation of AC models from the system artefacts is achieved by executing the transformation using Epsilon Transformation Language (ETL). The generation of GSN graphics is implemented in the Epsilon Generation Language (EGL). The EMF models of AC modules are automatically transformed into dot files that can be visualized as graphs in PDF format. The Eclipse GEF plugin supports this process by offering the DOT editor and the DOT graph view functionalities. The DOT editor enables the opening and editing of Graphviz dot files associated with the AC module, while the DOT graph view is utilized to visually represent the dot file and supports exporting it in PDF format.

4.5.2 Case study

A case study of an autonomous underwater vehicle is conducted to demonstrate the application of the AC generation method proposed in this chapter.

System description

The autonomous underwater vehicle (AUV) system, as described in [50], is designed for underwater maintenance and intervention tasks. It can operate in manual or automatic mode, adapting to different conditions. Its main hazard is the potential collision with infrastructure and unknown obstacles in the sea, which can be caused by either the operator or the AUV system itself. The system incorporates machine learning techniques for local path planning. To ensure operational safety, a safety monitoring component called the “Last Response Engine” (LRE) is developed. Unlike other components that rely on machine learning, the LRE is designed without such techniques to enable thorough formal verification of its safety assurance. The RoboChart state machine model of the LRE is shown in Figure. 4.18.

The primary function of the LRE is to manage the switching between different operating modes of the system, based on safety conditions. The system has four operation modes:

1. Operator Control Mode (OCM), where the operator manually controls the system.
2. Main Operating Mode (MOM), which is the standard automatic mode used during safe conditions.

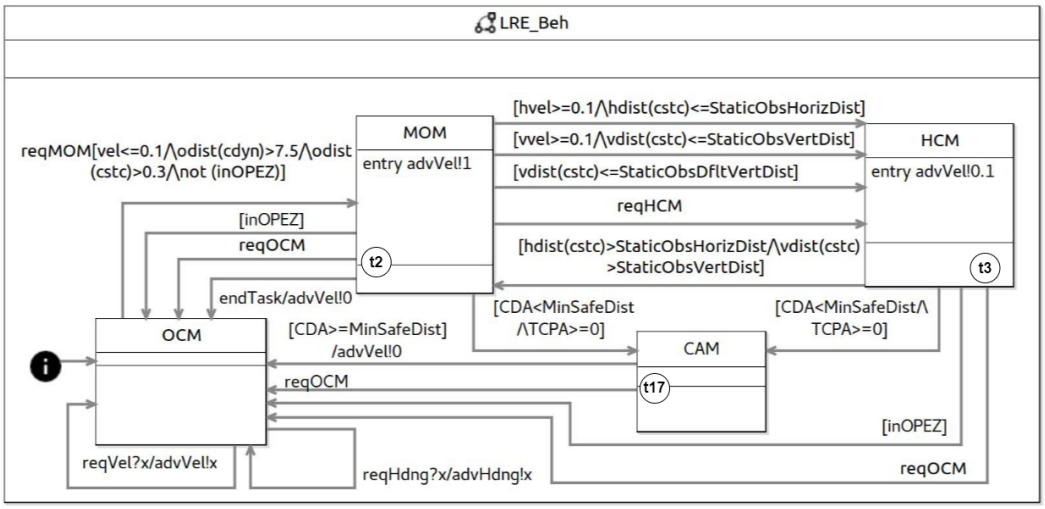


Figure 4.18: RoboChart model of LRE [50].

3. High Caution Mode (HCM), an automatic mode activated when the collision risk is present, and the risk can be reduced by decreasing the system's speed.

4. Collision Avoidance Mode (CAM), an emergency automatic mode activated when the collision risk is high and immediate manoeuvres are required to reduce the risk.

The LRE dynamically switches between these operation modes based on the system's operating condition. The four operation modes are modelled as four states in the state machine. The mode switching is modelled through transitions between states in Figure 4.18. For instance, the LRE transits from MOM to HCM when the horizontal velocity exceeds a certain threshold and the horizontal distance to an obstacle is below a predefined margin. Additionally, the operator can request a mode switch by sending events such as 'reqOCM', 'reqMOM', or 'reqHCM' to the LRE when necessary.

System artefacts processing

In this case study, the system artefact "hazard analysis table" is used for AC generation. For the LRE, one of the main hazards is as follows:

H1: AUV collides with an obstacle.

An excerpt of the detailed hazard analysis results for this specific hazard has been converted into EMF models, as depicted in Figure 4.19. It should be noted that the hazard analysis is not fully completed for certain safety measures and causes of the hazard. Consequently, the resulting AC structure derived from this artefact will contain undeveloped claims.

AC structure generation by pattern instantiation

An AC model is generated through pattern instantiation by utilizing hazard models as input, employing the predefined pattern shown in Figure 4.6. The resulting AC model is presented in GSN notations as shown in Figure 4.20. Since the system hazard analysis is incomplete at this stage, the generated AC model contains certain claims that are marked as "undeveloped". The graphics are generated in a DOT format and displayed using Eclipse GEF. The shape of the graphics has been

- Hazard Table
  - Hazard AUV collides with obstacles.
    - Cause AUV system failure leads to AUV collision with obstacle.
      - Cause Operator can not obtain the control from the state in which the operator is not in control when requesting.
        - Safety Measure Operator shall obtain the control from any operational state in which the operator is not in control when requesting.
          - Safety Measure Query: each state except OCM should have a outgoing transition whose trigger is reqOCM, and target is OCM.
            - Verification Model query: if the required transition exists.
              - Evidence The named transition returned by querying if exists.
      - Cause AUV system component failure druing automatic mode.
        - Safety Measure Sensor shall provide timely and accurate date to avoid collision with obstacle.
        - Safety Measure Path planner shall provide correct path to avoid collision with obstacle.
        - Safety Measure LRE shall provide correct control to avoid collision with obstacle.
        - Safety Measure Autopilot shall process the command from LRE correctly to avoid collision with obstacle.
        - Safety Measure Actuators shall response correctly and timely to avoid collision with obstacle.
      - Cause Operator's improper control.

Figure 4.19: EMF model of AUV Hazard analysis results for Hazard 1.

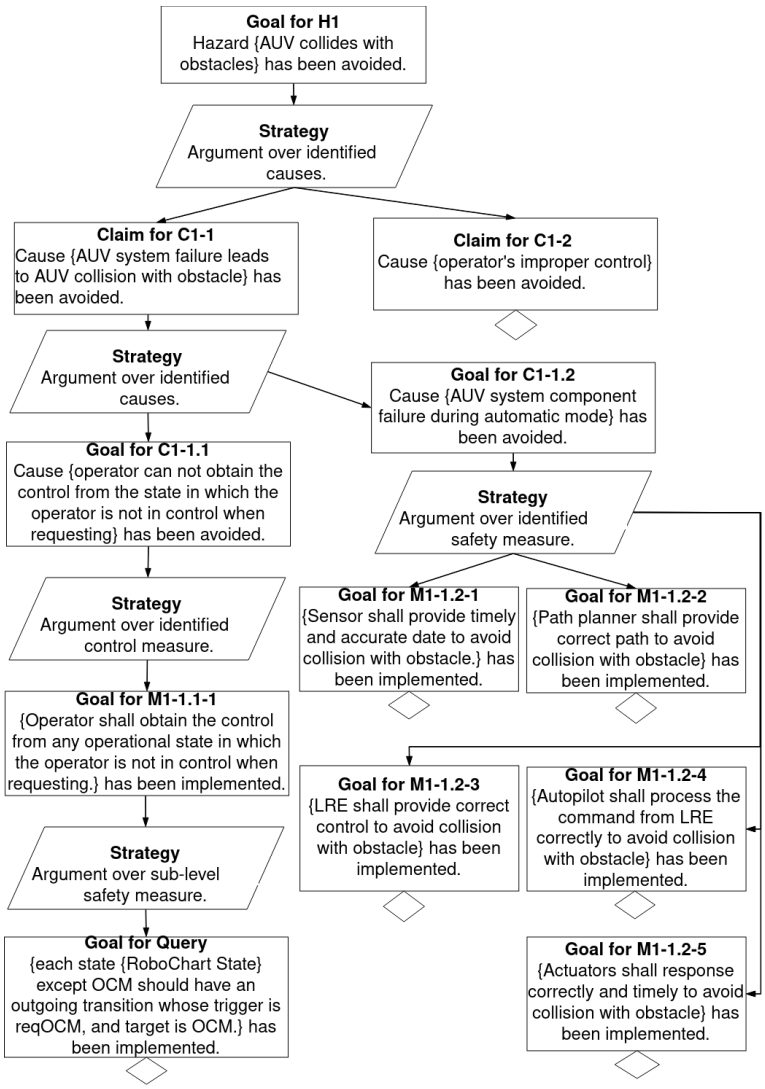


Figure 4.20: LRE AC structure generated by pattern instantiation.

adjusted here for the purpose of saving space. The context, assumptions, and justifications have been omitted for the sake of simplification.

Lower-structure refinement by model query

In Figure 4.20, the safety measure M1-1.1-1 *'Operator shall obtain the control from any state in which the operator is not in control when requesting.'* is mapped to a goal. This goal is further refined as a sub-level goal, which states that the requirement of *'each state (RoboChart State) except OCM should have an outgoing transition with the trigger reqOCM and the target state OCM'* has been implemented.

Algorithm 3: Lower-structure refinement and integration by model query- state and transition

```
1  for inf ∈ ac.argumentationElement.select(i | i.type =
     assertedInference).all.select{i | i.source.contain("Query")}
2  do
3      for c ∈ inf.source.contain("Query") do
4          inf.source.remove(c)
5          c.clear()
6          for st ∈ system.states do
7              if paraA then
8                  claim = CREATECLAIMANDADDTOAC(ac, st)
9                  inf.source = inf.source ∪ {claim}
10                 assertedInference = CREATEASTINFANDADDTOAC(ac, claim)
11                 vrclaim = CREATECLAIMANDADDTOAC(ac, claim)
12                 assertedInference.source = assertedInference.source ∪ {vrclaim}
13                 assertedEvidence = CREATEASTEVIANDADDTOAC(ac, vrclaim)
14                 for tr ∈ system.transitions do
15                     if paraB then
16                         evidence = CREATEEVIANDADDTOAC(ac, tr)
17                     end
18                 end
19                 if evidence.notDefined then
20                     evidence = CREATEEVIANDADDTOAC(ac, "not satisfied")
21                 end
22                 assertedEvidence.source = assertedEvidence.source ∪ {evidence}
23             end
24         end
25     end
26 end
```

This sub-level goal is labelled "Query" manually, indicating that it needs to be further refined into a set of goals through model querying. Therefore, in the stage of pattern instantiation, it is marked as "undeveloped". Subsequently, a query rule related to the state and transition is used for refinement of this goal. Algorithm 3 presents the structure of the query rules used for .

Claims should be created for each state except for state OCM. The query rule for identifying the states for claim generation is defined as: The state *st* must satisfy the condition *state.name* $\neq$ *"OCM"*. This condition is a user input as a parameter, *"paraA"* (line 7). *system* represents the input RoboChart state machine, it has a set of states *system.states* and a set of transitions *system.transitions*. Each state in the set *system.states* is checked against this condition. If the state satisfies the condition,

by using function CREATECLAIMANDADDTOAC(), a claim is generated for that state with the statement "State *st* should have an outgoing transition whose trigger is reqOCM, and the target is OCM" (line 8). In addition, the AC structure, including the strategies, claims for verification, and the linkages between these components, is created as shown in lines 10 to 13.

For evidence generation, the query rule to identify the transitions is: The transition t must satisfy the condition $t.source = st$ **and** $t.target =$ "*OCM*" **and** $t.trigger =$ "*reqOC*". This condition is a user input as a parameter, *"paraB"* (line 15). Each transition tr in the set *system.transitions* is checked against this condition for the state st. If the transition tr satisfies the condition, evidence is generated for *st* as "State *st* have an outgoing transition tr whose trigger is reqOCM, and the target is OCM" (line 16). Otherwise, if there is no transition that satisfies the condition, the evidence is generated as "The claim for *st* is not satisfied"(line 20).

The complexity of Algorithm 3 is $O(n^4)$ because there are four nested for loops from lines 1-26, 3-25, 6-24, and 14-18. n is the biggest value among the number of the AC inference *inf* whose source claim contains the keyword "Query", the number of claims in each queried inference *inf*, the number of the transitions in the state machine, and the number of the states.

To ensure the correctness of Algorithm 3, when applied to LRE state machine example, the AC fragment generated should have three claims derived from the states *MOM*, *HCM*, and *CAM*, three sub-level claims for the verification method, and three evidence derived from the transitions *t3*, *t2*, and *t17*; also, the relationships including asserted inference and asserted evidence should reflect the correct linkage between the sources. These requirements are verified using the generated AC module.

For the integration of the AC structure, the "Sub-Goal for M1-1.1-1" should be identified, removed from the linkage to its upper strategy, and then deleted (lines 3 - 5). Once the refined goal is generated through the query of the design models, it is linked to the strategy below "Goal for M1-1.1-1" (line 9). Thus, the lower structure is integrated into the AC model derived in the pattern instantiation stage.

The model query rule described in Algorithm 3 is implemented using the EOL language. Most of the EOL code can be reused except for the query conditions and the content of the claims and evidence, which shall be provied by the users. In this case study, the conditions used to query the states (line 7) and the transition (line 15), and the expected content of the claims (line 8) and evidence (line 20) are the input provided by users according to the claims (claim C1-2 in this case). Different claims may need different query rules to support the refinement. These rules should be designed and stored in a library for reuse.

The system models subjected to the query are the EMF models of the LRE state machine. The updated fragment of the AC, starting from the goal for M1-1.1-1, is displayed in Figure 4.21. In Figure 4.20, the "Sub-Goal for M1-1.1-1" is substituted by three new goals: "Goal for state MOM," "Goal for state HCM," and "Goal for state CAM" in Figure 4.21. The remaining portion of the lower structure, which includes the strategies, goals for verification methods, and supporting solutions, is automatically generated from the RoboChart models as well.

4.5.3 Evaluation

To evaluate the effectiveness and generality of the model-based AC generation method, we use the method to engineer the AC generation of the LRE in Section 4.5.2. The aim of the evaluation in this

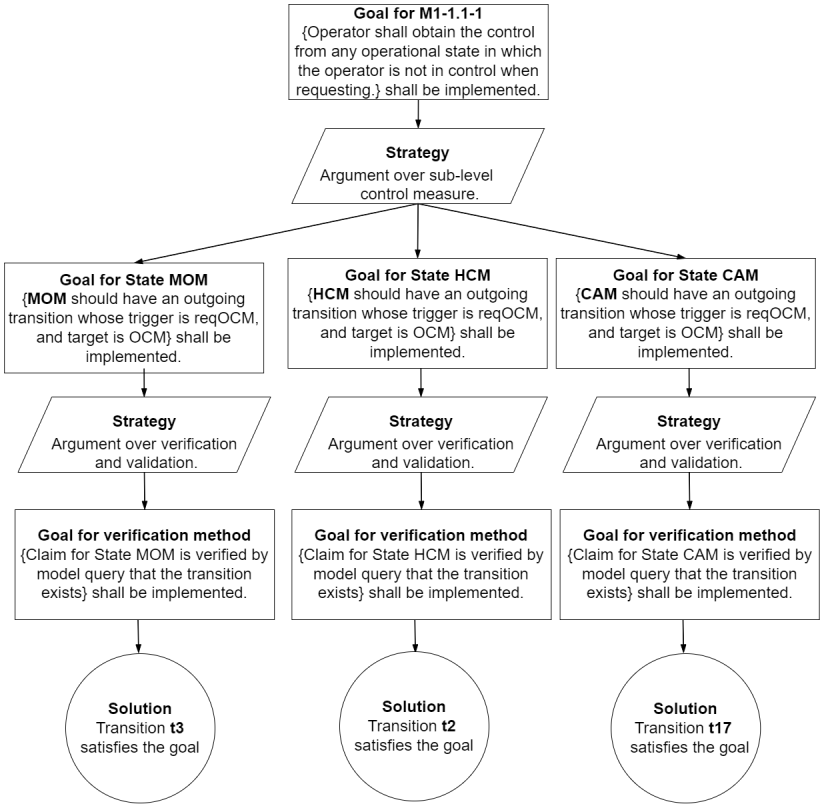


Figure 4.21: Lower-structure of LRE AC by model query.

section was to answer the following research questions.

RQ1 Correctness: Are the assurance cases generated using the proposed method complete, correct, and valid?

RQ2 Generality: Does the method support the development of ACs for different application domains? Can it also be applied to applications with various system development techniques?

RQ3 Efficiency: Does the implementation produce ACs with acceptable overheads in terms of computational resources and time?

Two assumptions have been made for the method proposed in this chapter. The first assumption is that system development adopts Model-driven Engineering, enabling the availability of system models as input for the AC process. The second assumption is that the system artefact used as input for the AC process is correct and valid. Consequently, the validation of system artefacts falls beyond the scope of the method and is not evaluated.

To answer the first question, we manually generated ACs for the LRE case study based on its system artefacts and the predefined AC pattern. The AC was presented in GSN notations and reviewed by a safety expert within the research group. Then, we employed the method to automatically create ACs. The AC for the LRE case study adhere to the predefined patterns and accurately map to the elements of the hazard analysis table. The graphical representation of the automatically generated AC is identical to that of the manually generated ones. Therefore, the correctness of the generated

AC is ensured. Also, the correctness of the algorithms are evaluated and discussed in the previous section to address RQ1.

For the second question, the primary determinant of the method's applicability lies in the formats of the system artefacts, rather than the specific application domains or the content of the artefacts themselves. The method of pattern instantiation is applicable to the applications that produce system artefacts of both structured and unstructured formats. This is achieved by incorporating a step for artefact unification. The Ecore metamodel is used to unify the artefact formats into EMF models for illustration purposes. However, the EMF framework can be replaced by other model-based frameworks without affecting the core method. As for the generality of the model query method, the method is underpinned by Model-driven Engineering (MDE), therefore is applicable to the autonomous systems developed using MDE, regardless of the application domains or specific modelling languages employed. Although the RoboChart models are used in this case study, other UML-like languages can also be utilized.

To address the third question, the efficiency of AC generation is compared between our method and a manual process. The method we propose involves several tasks, such as designing AC patterns, generating EMF models of system artefacts, creating model query rules for claims, and implementing corresponding model transformation rules. Although these tasks require initial effort, the resulting outcomes can be reused, making it cost-efficient, considering the iterative nature of system development.

The LRE case study is conducted on a 1.6GHz Core i5 computer with 8GB of RAM. The CPU time required to generate the ACs and GSN graphics from the EMF models of the LRE hazard table and the RoboChart design models is below 1 second respectively, demonstrating the efficiency of the proposed method. Any updates made to the hazard analysis table or the RoboChart models can be reflected in the AC module of the LRE by simply rerunning the generation process.

Threats to validity

Construct validity threats may be due to the assumptions made during the creation of system artefacts, including the hazard analysis table and the RoboChart models for LRE. To mitigate this threat, the system specification of the AUV system is referenced to ensure the accuracy and relevance of the hazard analysis. Furthermore, the LRE models used in this study are from a previously validated case study [50], which enhances the confidence in their reliability and suitability for analysis.

Potential threats to external validity may arise from the limited use of a single type of system artefact for the generation of ACs. To address this concern, a generic algorithm (Algorithm 13) is presented, which demonstrates the applicability of the method when multiple artefacts are involved in the AC generation.

Internal validity threats can arise from the implementation of the method. To mitigate these threats, we conducted thorough testing of the implementation by comparing the results of the code execution with the expected outcomes. Additionally, we have made all implementation data and results openly accessible in the published paper, allowing for easy replication of the study.

4.6 Summary

This chapter tackles the first two research questions proposed in Section 1.2 by presenting a combined method to generate model-based assurance cases from diverse system artefacts, including system design models, using MBE techniques. The method encompasses the integration of two techniques: pattern instantiation and model query. The techniques are versatile enough to be applied to a wide range of systems.

Our method of AC generation builds upon established methods found in the existing literature. Nevertheless, it incorporates specific features that differentiate it from existing approaches. These features include:

1. providing a method for generating ACs from different system artefacts including the system design models. Previous methods such as those presented in [14, 39, 43] focus on AC generation from artefact models, while other works address AC generation from design models [21, 33]. However, the proposed method is the first method that encompasses AC generation from both artefact models and design models.

2. having a two-phase design of AC pattern. Many studies have proposed AC patterns from different perspectives, often tailored to specific development environments or applications [16, 17, 33–35]. To enable the broad application of model-based AC generation across diverse domains, the inclusion of a generic AC pattern becomes essential. Striking a balance between generality and applicability, our method introduces a two-phase pattern design. The first phase involves a generic pattern that establishes structural constraints on the AC, aligning with the SACM metamodels. Subsequently, users can develop customized patterns specific to their respective applications.

3. being compliant with SACM metamodel and offering the option to present ACs using GSN graphics. Many existing model-based AC methods adhere to GSN standards to generate ACs [14, 20, 39], given GSN's popularity as an AC notation. These methods often involve defining GSN-compliant metamodels as the foundation for AC model generation [14, 20]. In contrast, our method leverages the SACM metamodel to ensure a unified structure of ACs, facilitating AC generation across different organizations. Additionally, the inclusion of GSN graphics enhances the readability of the ACs.

4. being able to produce the AC claims and evidence from the UML-like design models. This is influenced by existing solutions that generate AC from AADL system design models [21, 33]. One advantage of our method is its independence from the system development environment [21]. Also, unlike the work described in [33], where the AADL models are solely used to instantiate the predefined patterns, our method supports the flexible and accurate generation of AC claims and evidence through the use of model query languages.

The traceability between the system artefacts and the elements of the AC pattern is established in the customized pattern and built into the transformation rules. As such, the ACs can be automatically updated whenever changes occur in the system artefacts.

The method presented in this chapter is not limited to specific implementation techniques, such as the Ecore metamodels, Epsilon languages, or RoboChart modelling languages. For instance, the Epsilon

ETL transformation language and EOL query language are employed, but they can be substituted with other transformation languages like ATL. While the ACs are generated using Ecore metamodels within the EMF framework, the method can also be applied using alternative modelling frameworks like the Meta-Object Facility (MOF). Furthermore, although the model query method is demonstrated using RoboChart models, it is applicable to system models represented in UML-like languages that possess a hierarchical structure.

Assurance cases undergo evolution throughout the system development lifecycle. Automating the AC process using MBE techniques reduces workload and minimizes the risk of errors. The standardized metamodel SACM serves as a foundation for model-based AC generation. This method facilitates automated AC generation as well as automated updates in response to system modifications.

Chapter 5

AC Evidence Generation by Model Checking

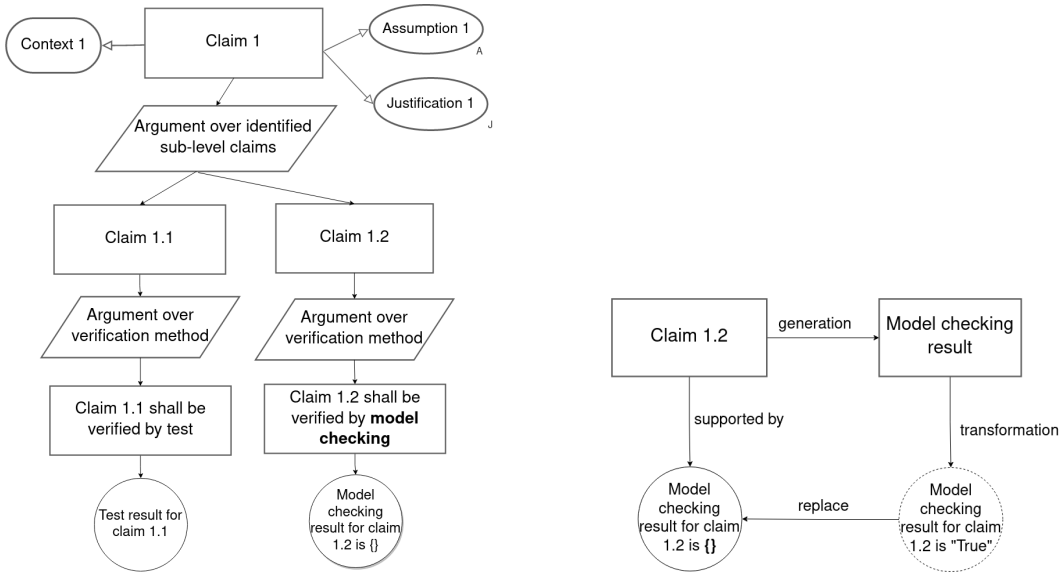
The methods in this chapter address the third research question and contribute to a process of automated generation of AC evidence models by performing model checking on RoboChart models. These methods build upon the approach described in Chapter 4 and expand it to include the generation of instantiation data for evidence models. A research paper has been prepared to state the method outlined in this chapter.

5.1 Overview and background

Chapter 4 has presented a method for generating ACs through pattern instantiation. However, it assumes that all the required system artefacts for instantiation are readily available. The method does not address the process of creating these system artefacts, including obtaining verification results as supporting evidence. This aspect has not been discussed in Chapter 4.

This chapter presents an extension of the method described in Chapter 4 to generate evidence directly from the claims for RoboChart models. In this context, an AC module is generated using pattern instantiation, as illustrated in Figure 5.1a. However, when it comes to supporting Claim 1.2 through model checking, no verification result is available at the time of AC generation. As a temporary solution, the evidence model is instantiated with the placeholder "model checking result" for Claim 1.2 instead of the actual result. Then, the method described in this chapter provides an automated process of applying model checking to RoboChart models to provide support for AC claims. Figure 5.1b illustrates the process, where the method enables the direct generation of a model-checking result from Claim 1.2. This result is then automatically transformed into an evidence model, which replaces the previous evidence model depicted in Figure 5.1a.

Previous studies have employed model checking to generate AC evidence; however, their processes are not fully automated. This is primarily because the creation of assertions from claims and the formalization of system models are typically performed manually, creating gaps in the automation of the overall process. As discussed in Section 2.3.2, several studies have focused on providing



(a) An AC module generated by instantiation. (b) Evidence generation

Figure 5.1: An illustration of the chapter's purpose with an abstract example.

formal semantics to various modelling languages. In line with these efforts, our method will adopt a modelling language that already possesses available formal semantics.

In this chapter, we continue to use RoboChart as the system modelling language. This is because RoboChart has a denotational semantics [54] based on the CSP process algebra [91], as well as a probabilistic semantics implemented in PRISM [55]. RoboChart can be automatically transformed into these formal semantics within the RoboTool environment. This allows for the use of model checkers such as FDR [56] with CSP models for non-probabilistic properties, and PRISM [57] with PRISM models for probabilistic properties. However, it is worth noting that the process of creating assertions for RoboChart claims still requires formal expertise in CSP/PRISM.

An example AC claim to be supported by model checking results is provided below:

Claim: The requirement R1 $\{$The state *St* shall be reachable in the system *Sys*$\}$ has been satisfied.

To support this claim, the requirement R1 is to be verified by model checking. The formal assertion for R1 is created as follows in machine-readable CSP (CSP_M) and is intended to be checked using the FDR model checker. Here, *P[T=Q* means "Process P is trace refined by process Q", i.e., every trace of Q is a trace of P.

```
assert not STOP [T= let
  id__ = 0
  within Sys::VS_O__(id__) |\{|Stm :: enteredV.Stm :: SID_Stm_St|}
```

To facilitate the automation of assertion generation, we further make use of the RoboChart assertion language [82]. With the support of RoboChart assertion language, the above assertion can be rephrased as follows:

assertion *A1_reachable* : *Stm::St* is reachable in *Sys*.

The assertion above is automatically parsed into CSP_M by RoboTool for FDR model checking. This RoboChart assertion language acts as an intermediate layer between RoboChart and CSP. It provides a more concise and readable representation compared to the CSP_M semantics. This concise presentation of assertions enables the feasibility of automating the generation of assertions for RoboChart claims, which is a crucial step in the automated process of AC evidence generation by model checking.

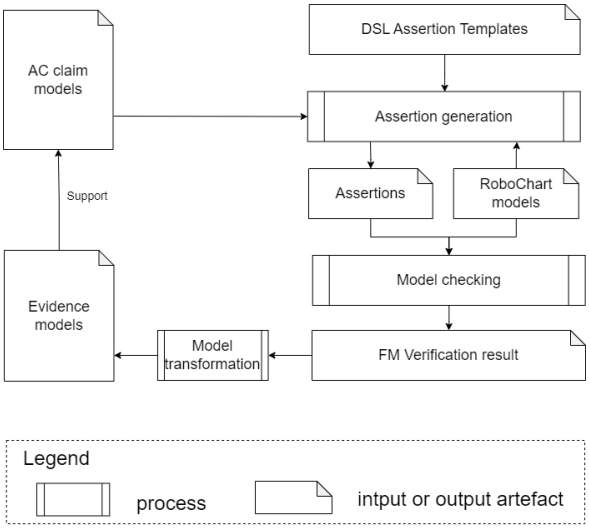


Figure 5.2: The process of evidence generation for RoboChart by model checking.

As mentioned in Section 2.3.2, various methods exist for formalizing requirements. These methods include using templates written in restricted natural languages, employing natural language parsers with ontologies, and utilizing neural machine translation techniques.

In our method, we will adopt the method of "templates in restricted natural language" for creating formal assertions. This choice is influenced by the available support we have, which includes a third-party requirement management tool capable of exporting requirements in a machine-readable format. Additionally, the use of RoboChart assertion languages will contribute to more concise and readable assertions which can facilitate the design of templates.

Therefore, the process of generating AC evidence for RoboChart through model checking is depicted in Figure 5.2. The method takes the claims that are to be supported by model checking as input and generates RoboChart DSL assertions from them using pre-defined assertion templates. These assertions are then subjected to model checking against the formalized system models. The resulting verification outcome is subsequently transformed into evidence models that are compliant with the SACM metamodel. These evidence models are integrated into the AC module to provide support for the claims. The method is implemented using the RoboTool environment, along with Epsilon and a requirement management tool. Both the FDR and PRISM model checkers are explored as part of the process.

The RoboChart assertions DSL offers the capability to define both untimed and timed assertions that are suitable for verification using FDR, as well as probabilistic assertions for verification with

PRISM [82]. These assertion languages serve as a foundation for automating the generation of assertions. To provide readers with a better understanding of the method described in this chapter, a concise overview of the DSL assertion languages will be presented in the rest of this section.

5.1.1 CSP and its RoboChart assertion language

RoboChart semantics is formalised using CSP and the dialect, tock-CSP, tailored for modelling time. CSP is a specification language used for describing the behaviour of concurrent systems, often referred to as process algebras. The main constructs of CSP are processes and channels. Processes can specify patterns of interactions, such as deterministic and nondeterministic choice, deadlock, and termination. A process can be defined via the parallel composition of processes, whose interaction occurs through channels. There are three denotational models of CSP: the traces model, the stable failures model, and the failures/divergences model. The traces model defines the meaning of a process expression as the set of sequences of events that the process can be observed to perform. The stable failures model extends the traces model with refusal sets. The failures/divergence model further extends the failures model to handle divergence. The details about the semantics and the operators of CSP can be found in [91].

By utilizing the CSP semantics, RoboChart models can be verified using the FDR refinement model checker. FDR offers a high level of automation for verifying the semantics and has the advantage of supporting the tock-CSP dialect. To facilitate the automatic verification of properties in FDR, RoboTool automatically generates an ASCII representation of CSP known as CSP_M [19]. CSP_M is a lazy functional language that includes built-in support for defining CSP processes.

The RoboChart assertion DSL for CSP is based on CSP_M and simplifies the assertion presentation. The rest of this section provides a brief overview of the syntax of RoboChart assertion DSL. For more comprehensive information, please refer to Section 5.1 of the RoboChart manual [82].

The syntax of an assertion is as follows:

```
1 Assertion Syntax
2 Assertion ::= (‘timed’ | ‘untimed’)? ‘assertion’ N ‘:’ SPEC
3               (‘in’ ‘the’ MODEL)?
4               (‘with’ (‘constant’ | ‘constants’) CONSTANTS)?
```

The RoboChart is equipped with timed and untimed CSP semantics and will be converted into both timed CSP_M models and untimed CSP_M models every time the RoboChart model is created or updated. By default, when we check an assertion for RoboChart model in FDR, we do not mark the assertion with ‘timed’ or ‘untimed’ label, which means the verification is conducted on both the timed and untimed CSP_M semantics. However, we could specify CSP_M semantics to be used in the verification as timed or untimed through the label ‘time’ and ‘untimed’ when needed.

The assertion contains a property specification ‘SPEC’, allows the specification of a model ‘MODEL’ (e.g., traces model), and allows the specification of values for constants used in the specification. ‘N’ denotes the name category, here it represents the name of the assertion.

The syntax for the specification ‘SPEC’ is shown as follows:

```
Specification Syntax
SPEC ::= N 'is' ('not')? (‘deadlock-free’ | ‘divergence-free’)
         | N (‘does’ ‘not’ ‘terminate’ | ‘terminates’)
         | N ‘is’ (‘not’)? ‘reachable’ ‘in’ N
         | N ((‘does’ ‘not’ ‘refine’) | (‘refines’) ) N
```

A specification ‘SPEC’ is either unary or binary. Unary assertions describe the general properties including termination, state reachability, deadlock freedom, divergence freedom, determinism, clock initialisation, and timelock freedom (i.e., no events are offered and time cannot pass) of specific RoboChart elements. Binary assertions compare two RoboChart elements via the refinement and equality relations. In our method, we address several types of assertions, namely termination, state reachability, deadlock freedom, divergence freedom, and refinement.

For the refinement assertions, the specification and implementation of RoboChart models are represented by two CSP_M modules. These modules are defined using the following CSP block:

```
CSP Module Syntax.
CSP module ::= (‘timed’ | ‘untimed’)? ‘csp’ N
                ‘csp-begin’
                 CSPM
                ‘csp-end’
```

In the list above, ‘CSPM’ is the formalization of the specifications or the implementations of the system using CSP processes in CSP_M.

The RoboChart assertion language simplifies the process of writing simple assertions, such as deadlock freedom, without requiring users' in-depth knowledge of the naming conventions of CSP semantics of RoboChart. However, for more complex refinement properties, it is necessary to utilize the CSP blocks and have a deeper understanding of the structure and naming conventions of the CSP semantics, in addition to the CSP_M syntax.

Our method automates the assertion generation process by providing a set of templates for the assertions combining the RoboChart assertion language and the CSP_M blocks.

5.1.2 PRISM and its RoboChart assertion language

RoboChart is also supported by the formal semantics of PRISM. RoboTool can automatically transform the RoboChart with probabilistic properties into PRISM models. These PRISM models can then be verified using the PRISM model checker.

The PRISM model checker processes the assertions written in PCTL language. RoboChart assertion DSL for PRISM provides a more concise syntax for writing assertions. In the rest of this section, a brief introduction to the syntax of the RoboChart assertion language for PRISM is provided. For a more comprehensive understanding, please refer to Section 5.2 of the RoboChart manual [82].

The syntax of a probabilistic assertion is shown below:

```
Probabilistic Assertion Syntax
ProbAssertion ::= 'prob' 'property' N ':' pExpr
                  ('with' 'constants' (ConstConfig+ '|' N))?
                  ('with' 'definitions' ((pFunction '|' pOperation)+ '|'
                     ↪ N))?
                  ('with' 'modules' (pModule+ '|' N))?
                  ('with' 'cmdoptions' STRING)?
```

The assertion starts with the keyword '*prob*'. A name 'N' is associated with the assertion to denote the name category, and functions as the property ID. The body of the assertion is an expression that denotes the property to be verified. This expression in the probabilistic assertion must be a boolean-type expression. An assertion optionally has constant configurations (given in ConstConfigs), function and operation definitions (given in pFunction or pOperation), PRISM module (given in pModule), and customised RPISM command line options.

The expressions pExpr are categorised into boolean, and numeric expressions. The boolean expression syntax of expression is omitted here. Please refer to the manual for the details.

A state formula could be one of the following:

- a probability formula (PFormula) starting with Prob and a bound (Bound) or a query (Query), followed by a boolean expression and an optional simulation method (UseMethod);
- a reward formula (RFormula) starting with Reward and an optional rewards name, and a bound (Bound) or a query (Query), followed by a reward path formula; and an optional simulation method (UseMethod);
- a non-probability property by a Forall operator followed by a boolean expression;
- a non-probability property by an Exists operator followed by a boolean expression.

The syntax of a state formula is shown as follows:

```
State Formulas Syntax
StateFormula ::= PFormula | RFormula | AFormula | EFormula
PFormula     ::= 'Prob' (Bound|Query) 'of' '['pExpr']' (UseMethod)?
RFormula     ::= 'Reward' ('{'N'}')? (Bound|Query) 'of'
   ↪ '['RPathFormula']' (UseMethod)?
AFormula     ::= 'Forall' '['pExpr']'
EFormula     ::= 'Exists' '['pExpr']'
```

For reward-based assertions in RoboChart, a reward value is defined using a guard condition 'pExpr' of boolean type, an expression, and an optional event represented by 'pEvent'.

```
Reward Syntax
Reward ::= ('[' pEvent ']')? pExpr ':' pExpr ';'
```

5.1.3 Evidence generation method

The automation process for generating evidence through model checking comprises four steps, as illustrated in Figure 5.3. This figure offers a more comprehensive overview compared to Figure 5.2.

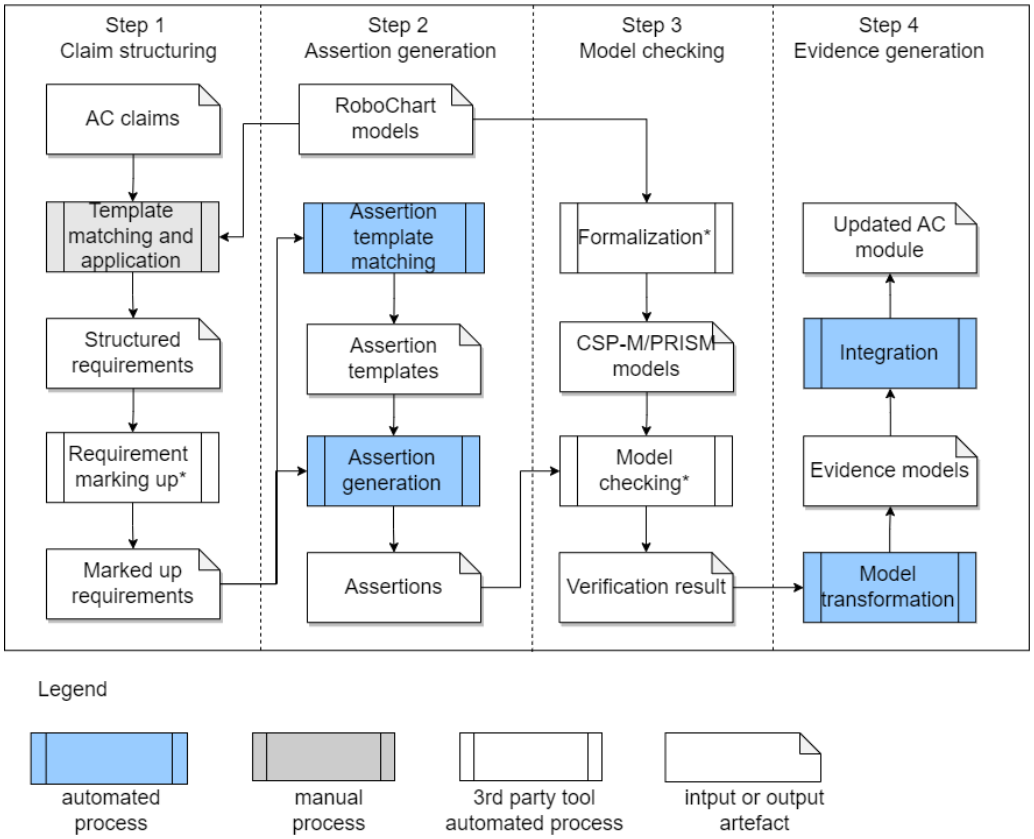


Figure 5.3: The method of evidence generation by model checking.

To automatically derive an assertion from the requirement referred to in an AC claim, the requirement must be in a machine-readable format. However, the requirement in the AC claim model is typically expressed in natural language, such as "the state *St* shall be reachable in the system *Sys*", which cannot be directly parsed by machines. As a result, **the first step** in the process involves structuring the requirement within the claims by rewriting them using predefined templates within a requirement management tool. The tool supports the export of requirements in markup formats, such as XML. Therefore, the requirements are exported from the tool as XML files, which are then used for the generation of assertions.

In **the second step** of the process, an appropriate predefined template for assertions is selected based on the class of the requirements. The marked-up requirement is used as input for the selected template. This leads to the generation of the corresponding assertions.

The third step is to execute model checking on the system models to verify the assertions. These models are formal CSP_M and/or PRISM semantics, which are automatically generated from the RoboChart models using the RoboTool.

In **the fourth step**, the verification results are transformed into SACM models and integrated into the

AC module. This process follows the same method as described in Chapter 4. The model-checking results are utilized to instantiate the AC pattern and integrate the model fragment into the AC module.

Throughout the process, there are three activities that are completed automatically by the respective tools and therefore are not a contribution of this thesis. These activities include requirement marking up in Step 1, RoboChart model formalization, and model checking in Step 3.

In the rest of this chapter, Section 5.2 discusses the activities in Step 1, Section 5.3 discusses the activities in Step 2. Step 3 is automated by tools and no further details are given in this chapter. Step 4 is described in Section 5.4.

5.2 Claim structuring

This section describes the processes involved in Step 1 including the claim classification, claim template design, claim structuring, and marked-up claim generation.

5.2.1 Claim classification

The claims, which are supported by formal verification through model checking, are classified into distinct categories to aid in the design of appropriate templates. These categories are established based on the nature of the requirements contained in the claims. Since the central focus of model checking lies in verifying system requirements within the claims, the terms “claim” and “requirement” are used interchangeably throughout this chapter. Table 5.1 presents an overview of the classification, dividing the claims into five groups.

Table 5.1: Classification of AC claims

No.	Classification of requirements referenced in claims
1	Untimed functional requirement
2	Timed functional requirement
3	General requirement
3.1	- Reachability
3.2	- Divergence freedom
3.3	- Deadlock freedom
3.4	- Termination
4	Probabilistic requirement
4.1	- Probability requirement
4.2	- Reward-based requirement
5	Temporal requirement

The non-probabilistic RoboChart models have both timed and untimed semantics. Therefore, there are two types of requirements that can be made: untimed functional and timed functional requirements. Both types of requirements can be verified through refinement checking using the FDR model checker.

The third group is referred to as the general requirements including four sub-types: deadlock-free, termination, reachability, and divergence-free. These four sub-types are verified in FDR and can be

marked as timed, untimed, or both timed and untimed.

The fourth group is probabilistic requirements for the probabilistic RoboChart models. These claims are verified in the PRISM model checker. The assertions for these requirements are written in RoboChart probabilistic assertion language. The probabilistic type is further divided into two sub-types: probability and reward. Lastly, the fifth group is the temporal requirements, which are also verified using PRISM.

According to the classification, templates are designed for each type/sub-types of requirements. In the process, a requirement management tool called Kapture[1] is utilized. Kapture provides a set of predefined templates that can be used for defining and documenting various types of requirements. These predefined templates serve as the foundation for generating customized templates. The details of the customized templates for requirements are presented in Section 5.2.3.

5.2.2 Kapture templates for requirements

In this section, the Kapture templates are customized to be used as the foundation for requirement template design. Kapture is a requirement management tool developed by D-RisQ[2], and aims to support engineers in creating well-defined and unambiguous requirements. It achieves this by providing a data dictionary and constrained text formats that assist in writing clear and concise requirements.

Kapture provides seven types of requirement templates. In the development of claim templates, three specific types are chosen and simplified: WHEN, CASE, and EVERY. The three templates that have been selected are sufficient to structure the requirements, and the other four templates share similarities with the chosen ones in certain aspects. Therefore, there is no need to use additional templates.

The WHEN template consists of five segments: “delay”, “when”, “until”, “required”, and “otherwise” among which “delay”, “until” and “otherwise” are optional (the segments within parentheses). To suit the purpose of claim template design, we use three segments: “when”, “until”, and “required”. These segments provide placeholders, namely “*GuardCondition*”, “*UntilCondition*”, and “*RequiredCondition*”, which should be replaced with the specific content of the claims.

Kapture WHEN Template

(After a delay of *DelayDuration*,)
When (AndOnlyWhen) *GuardCondition* is true,
(until *UntilCondition* becomes true,)
RequiredCondition shall be true,
(but otherwise *AlternativeCondition*.)

[1] https://www.drisq.com/product-kapture
[2] http://www.drisq.com/

Simplified WHEN Template

> When *GuardCondition* is true,
> until *UntilCondition* becomes true,
> *RequiredCondition* shall be true.

The CASE template is simplified to incorporate three segments except "delay": "until", "case" ("when" and "then"), and "otherwise". These segments provide four placeholders, namely "*UntilCondition*", "*GuardCondition*", "*RequiredCondition*", and "*DefaultCondition*", which should be replaced with the content of the claims. The original Kapture template is as follows:

Kapture CASE Template

> (After a delay of *DelayDuration*,)
> At each time step, the first of the following cases that are true shall apply:
> (until *UntilCondition* becomes true,)
> When *GuardCondition* is true,
> then *RequiredCondition* shall also be true,
> (otherwise *DefaultCondition* shall be true.)

The simplified structure of the CASE template is as follows:

Simplified CASE Template

> At each time step, the first of the following cases that are true shall apply:
> until *UntilCondition* becomes true,
> when *GuardCondition* is true,
> then *RequiredCondition* shall also be true,
> otherwise *DefaultCondition* shall be true.

The Kapture EVERY template is as follows:

Kapture EVERY Template

> (After a delay of *DelayDuration*,)
> the *Condition* shall *AlwaysOrNever* be true,
> until *UntilCondition* becomes true.
> $*$ AlwaysOrNever ::= *always* | *never*

A simplified version of EVERY template removes "dealy" and "until" segments and has two placeholders: "*Condition*" and "*AlwaysOrNever*", shown as follows:

Simplified EVERY Template

> The *Condition* shall *AlwaysOrNever* be true.
> $*$ AlwaysOrNever ::= *always* | *never*

To construct a requirement in Kapture, three hierarchical components can be utilized: the data dictionary section, the definition section, and the requirement section. The data dictionary section serves as

a repository for defining and storing the basic elements of claims, such as events, time limits, probability, reward values, and other relevant information about RoboChart properties. The definition section is used to define definitions which act as an intermediary layer between the data dictionary and the requirements. They enable the definition of functions using the elements specified in the data dictionary, allowing for more flexible requirement specifications. The requirement section is used to construct requirements by utilizing the data dictionary and the definitions.

Once the requirements are constructed in Kapture, they can be exported into various machine-readable formats, including CSV, HTML, HTML spreadsheet, and XML.

5.2.3 Requirement template design

In this section, nine templates are developed for the requirement types specified in Table 5.1, using the three customized Kapture templates as the basis. The placeholders in the templates will be replaced by RoboChart components. To enhance comprehension, an example is provided for each type, illustrating how the template can be used.

(1) Untimed functional requirement

A common pattern for the untimed functional requirements is that a "guard event" implies a "required event". This pattern can be read as "when the guard event occurs, the required event shall also occur". We design a template CT-1 for this scenario using the Kapture WHEN template. CT-1 is shown below:

CT-1 Untimed functional template

When (*guard_event*)+ is true,
(*required_event*)+ shall also be true.

∗ guard_event ::= RoboChart_event_in_CSP_semantics
∗ required_event ::= RoboChart_event_in_CSP_semantics
∗ RoboChart_event_in_CSP_semantics ::= (‘buffered_’)?scope‘::’event(‘.in’|‘.out’)
((‘.’value)?|(‘?’variable))
∗ scope ::= module(‘::’controller(‘::’state machine)?)?

CT-1 is constructed by utilizing two segments of the WHEN template, namely "When" and "Required". The "When" segment is employed to specify the guard condition, while the "Required" section is used to define the required behaviour. In this template, the terms in black represent the fixed components of the Kapture template, while the terms in blue serve as placeholders to be replaced by the RoboChart design models. The placeholders will be substituted with the corresponding RoboChart events.

The purpose of the template is to provide a consistent structure for the requirement and facilitate the generation of formal assertions that will be verified using the FDR model checker. Therefore, during this step, the placeholders in the template will be instantiated using the CSP semantics of the corresponding RoboChart events *RoboChart_event_in_CSP_semantics*. In the CSP semantics of RoboChart, the naming convention for events follows a hierarchical structure, with "::" serving as a delimiter. Additionally, events are annotated with the parameter *in* or *out* to indicate whether they

are input or output events. If the event is of type input, output, or synchronization, it may also include a *value* parameter. Replacing the placeholders in the template with the CSP semantics of RoboChart models is considered the initial step in formalizing the requirements into CSP assertions. This process primarily relies on the user's understanding of RoboChart modelling and the simple rules for naming conventions in CSP. As a result, the dependency on formal expertise is significantly reduced.

The definition of *RoboChart_event_in_CSP_semantics* mainly consists of the default strings with single quotes, e.g., *'buffered'*, *'in'*, *'out'*, and the question mark *'?'*, and the elements to be instantiated, e.g., *scope*, *event*, *value*, and *variable*. The question mark ? without the quote and the vertical bar | are the components of Backus Normal Form (BNF) to represent the optionality and the choice, respectively.

The "scope" of in the *guard_event* and *required_event* represents the context in which the event is defined. It can be a RoboChart state machine, a RoboChart controller, or a RoboChart module. This *scope* placeholder shall be replaced with a RoboChart component in their CSP semantics, incorporating all the higher-level scopes and separated by the delimiter "::".

Both placeholders, *guard_event* and the *required_event*, have an optional prefix "buffered". This prefix is used to account for the asynchronous communication in RoboChart. In RoboChart, communication with a robotic platform is always asynchronous, while communication between controllers can be either synchronous or asynchronous. In the CSP semantics, when asynchronous communication is present, a dedicated controller is created to serve as a buffer. By including the prefix "buffered" when using the template, a corresponding CSP process is generated to represent this buffer controller in the assertions.

An example of an untimed functional requirement, R1, which utilizes the CT-1 template, is presented below:

R1: The set point shall be 0 when the power is off.

Then, the requirement R1 can be expressed using the CT-1 template and RoboChart models as shown below:

Example: R1 in CT-1

> When *sys::power.in.OFF* is true,
> *sys* :: *setPoint.out.0* shall also be true.

In this example, the RoboChart event *power* of the RoboChart module *sys* represents an input communication with the value *OFF*. Similarly, the event *setPoint* of the RoboChart module *sys* represents an output communication with the value of *0*. Once the template is instantiated, requirement C1 is presented using a combination of controlled natural language (Kapture templates) and the formal semantics of RoboChart components in CSP. The further formalization will be realized through the generation of assertion templates, which is discussed in Section 5.3.1.

(2) Timed functional requirements

For the timed functional requirements, it is not feasible to present a general template that can accommodate different types of requirements with time properties due to the complexity introduced

by time constraints. In this chapter, one scenario considered is when a trigger event occurs, a target event should occur by default; alternatively, if there is a guard event, once both the trigger event and the guard event occur, the target event must happen within a specified time limit. We designed the template CT-2 for this scenario using the Kapture CASE template as follows:

CT-2 Timed functional template

> At each time step, the first of the following cases that are true shall apply,
> (until not (*trigger_event*)∗ occurs:)?
> when *guard_event* is true
> then (*guard_event* holds for at most *time_limit*) becomes *target_event* shall also be true.
> (Otherwise *target_event* shall be true)?
>
> ∗ trigger_event ::= RoboChart_event_in_CSP_semantics
> ∗ guard_event ::= RoboChart_event_in_CSP_semantics
> ∗ target_event ::= RoboChart_event_in_CSP_semantics
> ∗ RoboChart_event_in_CSP_semantics ::= (‘buffered_’)?scope‘::’event(‘.in’|‘.out’)
> ((‘.’value)?|(‘?’variable))
> ∗ scope ::= module(‘::’controller(‘::’state machine)?)? ∗ time_limit ::= unit(nat)

Four segments of the CASE template are used in CT-2, including ‘Until’ for *trigger_event*, ‘When’ for *guard_event*, ‘Then’ for *target_event* with a specified time constraint, and ‘Otherwise’ for the default status, i.e., the *target_event* in this scenario. The “Then” section for the guard event employs a user-defined function *holds for at most(guard_event, time_limit)*, with the function name as “holds for at most”, and two parameters as *guard_event* and *time_limit*. This function is defined using the Kapture definition component, which allow users to define the function definitions. Interpreting this template, it states that if the *trigger_event* occurs, *target_event* must be true. Alternatively, when *guard_event* is true, if there is a delay, it must end within the specified *time_limit* and become *target_event*.

The placeholders of *trigger_event*, *guard_event*, and *target_event* in the template are to be replaced with the RoboChart events in their CSP semantics. The time limit *time_limit* is represented as a function acting on the value of the time limit using the definition component in Kapture. For example, “10ms” will be represented as *ms(10)*.

It is important to acknowledge that Kapture provides a predefined set of requirement templates that offer limited flexibility in customizing their internal structure. In this chapter, We select the Kapture templates that best fit the requirements based on their structure. However, in certain templates, there might be extra information that is not necessary for a specific requirement. For instance, in the case of timed functional requirements, the default content in the CASE template “*At each time step, the first of the following cases that are true shall apply*” is not relevant to the timed requirement itself, but can not be removed due to software restrictions. Nevertheless, this does not affect the assertion generation process. The value of the Kapture template lies in its ability to structure the requirements in a suitable manner, enabling machine-readability and further the automatic extraction of the RoboChart components within the requirement for assertion generation.

An example of the timed functional requirement is as follows:

R2: When the power is off, if the Output signal is working normally, then the Output signal shall become 0 within 1s, otherwise the Output signal shall be 0 immediately.

In this example, the trigger event is "power off", the target event is "signal becomes 0", and the guard event is "signal working normally". Then, the requirement can be reformulated using the CT-2 template as follows:

Example: R2 in CT-2

> At each time step, the first of the following cases that are true shall apply,
> until not *sys::power.in.OFF* occurs:
> when *sys::output.out?x:*$\{x|\ x>0\}$ is true,
> then *sys::output.out?x:*$\{x|\ x>0\}$ holds for at most *s(1)* becomes *sys::output.out.0* shall also be true.
> Otherwise *sys::output.out.0* shall be true.

The placeholders in CT-2 are replaced with RoboChart events in CSP semantics. The *guard_event* is *sys::output.out?x:*$\{x|\ x>0\}$, which is a RoboChart event *sys::output.out.x* where x can take any value in the set of $\{x|\ x>0\}$.

General requirements

The general requirements encompass properties such as deadlock freedom, divergence freedom, reachability, and termination, which are verified using the FDR model checker. To develop a structured template for these general properties, the Kapture EVERY template is employed.

(3) Reachability requirements

First, a predicate is defined using the "Definitions" component in Kapture. For instance, in the case of untimed reachability, a requirement may state that "the state *st* is reachable in a state machine, a controller, or a module". Thus, a predicate should be defined as "reachable_untimed" with two parameters of *state* and *scope*. The definition is as follows:

> reachable_untimed(state, scope) $*$ scope ::= *state_machine* | *controller* | *module*

The predicate is interpreted as "the *state* is reachable in the *scope*". The function name reflects both the property type, "reachability" and the time property of the RoboChart models. Then, the general requirement template for reachability is defined using the predicate and Kapture EVERY template, as follows:

CT-3 Reachability template

> The property function(*state*, *scope*) shall *bool_statement* be true.
>
> $*$ function ::= *reachable_untimed* | *reachable_timed* | *reachable*
> $*$ scope ::= module('::'controller('::'state machine)?)?
> $*$ bool_statement ::= always | never

The function can be defined as timed by adding the suffix *timed*, as untimed by adding the suffix *untimed*, or as both timed and untimed if no suffix is present.

An example of the reachability requirement is given as follows:

R3: The *Wait24VPower* state shall be reachable in the voltage controller.

In this example, the placeholder *state* to be reached is *Wait24VPower*, the *scope* of the checking is the voltage controller which is modeled as a state machine *st_machine*. The *function* name is "reachable" because the time property is not specified and we use the default setting to check both timed and untimed models. The *bool_statement* is *always*. R3 is rewritten using the CT-3 template as follows:

Example: R3 in CT-3

> The property *reachable*(*Wait24VPower*, *st_machine*) shall *always* be true.

The predicates for deadlock freedom, divergence freedom, and termination are also defined in Kapture using similar principles. The templates for these properties are given below.

(4) Deadlock freedom requirements

In the case of deadlock freedom, a requirement may state that "the module or the controller or the state machine shall (not) be deadlock-free". The corresponding template is as follows:

CT-4 Deadlock freedom template

> The property function(*scope*) shall *bool_statement* be true.
>
> * function ::= *deadlock_free_untimed* | *deadlock_free_timed* | *deadlock_free*
> * scope ::= module(':'controller(':'state machine)?)?
> * bool_statement ::= always | never

An example of the deadlock freedom requirement is given as follows:

R4: The voltage controller shall be deadlock free.

In this example, the placeholder *scope* is the voltage controller which is modeled as a state machine *st_machine*. The *function* name is "deadlock_free". The *bool_statement* is *always*. R4 is rewritten using the CT-4 template as follows:

Example: R4 in CT-4

> The property *deadlock_free*(*st_machine*) shall *always* be true.

(5) Divergence freedom requirements

A requirement of divergence freedom may state that "the module or the controller or the state machine shall (not) be divergence-free". The corresponding template is as follows:

CT-5 Divergence freedom template

> The property function(*scope*) shall *bool_statement* be true.
>
> * function ::= *divergence_free_untimed* | *divergence_free_timed* | *divergence_free*
> * scope ::= module(‘::’controller(‘::’state machine)?)?
> * bool_statement ::= always | never

An example of the divergence freedom requirement is given as follows:

R5: The voltage controller shall be divergence free.

Similare to the deadlock freedom example above, in this example, the placeholder *scope* is the voltage controller which is modeled as a state machine *st_machine*. The *function* name is “divergence_free”. The *bool_statement* is *always*. R5 is rewritten using the CT-5 template as follows:

Example: R5 in CT-5

> The property *divergence_free*(*st_machine*) shall *always* be true.

(6) Termination requirements

A requirement of termination may state that “the module or the controller or the state machine shall (not) terminate”. The corresponding template is as follows:

CT-6 Termination template

> The property function(*scope*) shall *bool_statement* be true.
>
> * function ::= *terminate_untimed* | *terminate_timed* | *terminate*
> * scope ::= module(‘::’controller(‘::’state machine)?)?
> * bool_statement ::= always | never

An example of the termination requirement is given as follows:

R6: The voltage controller shall terminate.

Similare to the deadlock freedom example above, in this example too, the placeholder *scope* is the voltage controller which is modeled as a state machine *st_machine*. The *function* name is “terminate”. The *bool_statement* is *always*. R6 is rewritten using the CT-6 template as follows:

Example: R6 in CT-6

> The property *terminate*(*st_machine*) shall *always* be true.

(7) Probability requirements

A requirement for a probabilistic property usually takes the form of “The probability of the path shall meet the probability target when the values of the constants are set”.

We designed the template CT-7 for the probability requirements as follows:

CT-7 Probability template

> When (*constants*? | *multi_constant*?) is true,
> Until *path_formula* becomes true,
> *prob_target* shall also be true.
>
> * path_formula ::= 'pathformula_'pExpr
> * QualifiedNameToElement in pExpr ::= RoboChart element
> * prob_target ::= 'prob_target_'('>' | '>=' | '<' | '<=') Expr
> * constants ::='constant_'RoboChart_constant 'set to' Expr

In CT-7, the "When" section is utilized for constant declarations. The 'Until' segment is designated for the target events whose probabilistic property is being examined. The 'Required' section is employed to specify the quantitative probability target, i.e., the probability bound for the target events.

The "path_formula" is a boolean expression pExpr of the PRSIM assertion language prefixed with the term "pathformula_". The syntax of the boolean type pExpr can express the logic of if and only if, implication, disjunction, conjunction, logical negation, equal, inequal, larger than, larger than or equal, less than, less than or equal, etc. pExpr can also be a QualifiedNameToElement, which is presented using RoboChart element.

In CT-7, each placeholder is prefixed to indicate the type of the term and/or requirement. For example, 'prob_target' is prefixed with "prob_target_" to indicate that this is a template for "probability requirement". This information is utilized during the assertion generation stage to select the appropriate assertion template.

An example requirement of a probability type is as follows:

R7: The probability of the robot running into the wall shall be less than 0.1 when the constant speed is set to 20.

Example: R7 in CT-7

> When *constant_sys::ctrl::stm::speed set to 20* is true,
> Until *pathformula_Finally sys::ctrl::stm::distance==0* becomes true,
> *prob_target_* < 0.1 shall also be true.

In R7, the event "the robot running into the wall" is represented as *sys::ctrl::stm::distance=0* using RoboChart's PRISM formal semantics. This event is associated with the PRISM operator "Finally", which signifies the existence of a future time point at which the property stated after the operator holds.

(8) Reward-based requirement

A reward-based requirement usually takes the form of "The path shall satisfy the reward target, when the values of the constants and the reward events are set."

We designed CT-8 template using the Kapture WHEN template as follows:

CT-8 Reward-based template

When (*constants*? | *multi_constant*?) and *reward_event* and *reward_value* is true,
Until *path_formula* becomes true,
reward_target shall also be true.

* path_formula :: = 'pathFormula_'RPathFormula
* RPathFormula ::= ('Reachable' | 'LTL' | 'Cumul' | 'Total') pExpr
* QualifiedNameToElement in pExpr ::= RoboChart element
* reward_target::= 'reward_target_'('>' | '>=' | '<' | '<=' | '=') Expr
* constants:: = 'constant_'RoboChart_constant 'set to' Expr
* reward_event:: = 'reward_event_'RoboChart_event
* reward_value:: = 'reward_value_'Expr

In the template, the "When" section is used for "constant declarations", and "constant configurations" which are required for PRISM verification. It also includes information related to reward specifications, namely *reward_event*, which represents the event that triggers the reward and *reward_value*, which denotes the value assigned to the reward for each occurrence of the event. To facilitate the assertion generation process, the term *reward_event* is prefixed with the label *reward_event* to indicate its role as a reward event. Similarly, the term *reward_value* is prefixed with "reward_value" to signify its purpose as a reward value.

The "Until" section is utilized to specify the target events (*path_formula*) for which the reward property is checked. The "path_formula" is a reward path formula "RPathFormula" of the PRSIM assertion language prefixed with the term "pathformula_". The reward path formula could specify reachability cumulative, total and instantaneous rewards. Reachability rewards refer to the reward accumulated along a path until a certain point is reached; cumulative rewards are the rewards accumulated along a path until t time units have elapsed; total rewards are the accumulation of state and transition rewards in the same way as for "reachability reward" and "cumulative reward" properties, but the rewards are accumulated indefinitely, i.e. the total reward accumulated along the whole (infinite) path. Instantaneous rewards refer to the reward of a model at a particular instant in time. Expression pExpr in RPathFormula must be boolean. When it is a QualifiedNameToElement, pExpr is presented using RoboChart element.

The "Required" segment is used to specify the quantitative target, which represents the value bound of the target events. To indicate the claim type as "reward" during assertion generation, the *reward_target* is prefixed with "reward_target".

An example of a reward-based requirement is as follows:

R8: When the robot battery has a capacity of 20 and each move consumes one capacity, the robot can make at least 8 moves before running out of power and reaching the recharge station.

The requirement can be rewritten using the CT-8 template as follows:

Example: R8 in CT-8

> When *constant_sys::ctrl::stm::batterycapcity set to 20* and *reward_event_sys::ctrl::stm::move.in* and *reward_value_true:1* is true,
> Until *pathFormula_Reachable sys::ctrl::stm::barrtery==0* $\bigwedge$ *sys::ctrl::stm is in sys::ctrl::stm::battery charge* becomes true,
> *reward_target_* > 8 shall also be true.

The scenario that the robot runs out of battery and starts to recharge is modelled in RoboChart by representing the variable "battery" as zero, and the state machine *stm* being in the battery charging state *batterycharge*. This is a reachability reward, so "Reachable" is used, indicating that the reward is accumulated until the path is reached.

(9) Temporal requirement

A requirement of a temporal type is to be verified in PRISM. Unlike the probability and the reward-based requirement, this type of requirement is qualitative in nature and does not involve a quantitative target. A temporal requirement usually has the pattern "the system satisfies the temporal property when the value of the constants are set. We designed the template CT-9 for the temporal type of requirements using the Kapture WHEN template as follows:

CT-9 Temporal template

> When (*constants*? | *multi_constant*?) is true,
> Until *(temporal_operator)*? *path_formula* becomes true,
> NIL shall also be true.
>
> ∗ temporal_operator ::= 'term_'('not'? ('forall' | 'exist'))
> ∗ path_formula ::= 'pathFormula_'pExpr
> ∗ QualifiedNameToElement in pExpr ::= RoboChart element
> ∗ constants ::= 'constant_'RoboChart_constant 'set to' Expr

In the temporal requirement, no quantitative target is defined. Therefore, the "Required" section is set to "NIL" indicating the absence of a quantitative bound. The "When" section is still used for constants assignment, while the "Until" section is employed to specify the target event, which is accompanied by the temporal operator "forall" or "exist".

An example R9 is given as follows:

R9: There is a path that the robot does not get stuck.

The requirement is rewritten using the CT-9 template as follows:

Example: R9 in CT-9

> When *constant_sys::ctrl::stm::batterycapcity set to 20* is true,
> Until *term_not forall* and *pathFormula_sys::ctrl::stm is in sys::ctrl::stm::stuck* becomes true,
> *NIL* shall also be true.

The situation of the robot being stuck is modelled in RoboChart as the state machine *stm* enters the state *stuck*.

5.2.4 Claim structuring

In this section, we utilize the requirement templates introduced in Section 5.2.3 to facilitate the conversion of natural language requirements into a structured format, specifically XML, using Kapture. This conversion process is referred to as Step 1 "claim structuring" in Figure 5.3. Writing a requirement in Kapture is a manual task where users need to possess an understanding of the RoboChart models, along with a basic knowledge of the naming conventions for RoboChart events and variables in both their CSP and PRISM semantics. However, it is not necessary for users to have expertise in formal languages such as CSP and PRISM.

The first activity of Step 1 involves identifying the claims that can be supported by FDR and PRISM model checking. Once identified, the claim is then classified based on the categories outlined in Table 5.1. Following this, the subsequent task entails expressing the requirement within the claim by utilizing Kapture and the pre-defined templates.

Using R1 from Section 5.2.3 as an illustrative example, the process begins with the definition of the *guard_event* and *required_event* within Kapture's "Data Dictionary" component, as depicted in Figure 5.4.

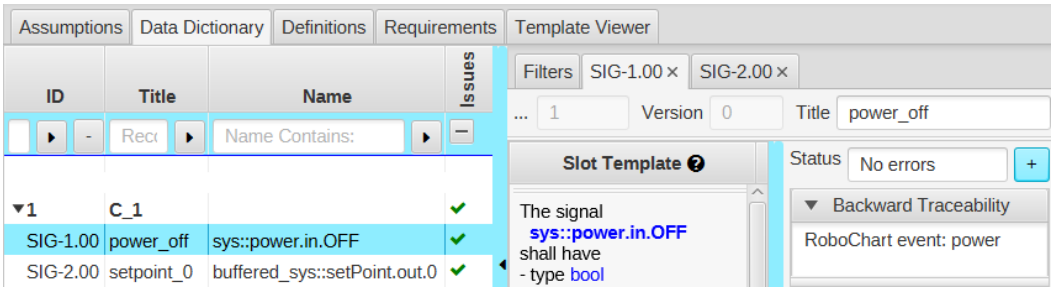


Figure 5.4: A Kapture screenshot of the definition of *guard_event* and *required_event* in R1.

Then, the WHEN template is used for creating R1. A screenshot of requirement R1 in Kapture is given in Figure 5.5. The previously defined *guard_event* and the *required_event* from "Data Dictionary" are used to formulate the requirement within the 'Requirements' section of Kapture. Notably, the "Until" and "Otherwise" segments of the template are not used in this case. On the right-hand side, traceability information is recorded to establish connections between this requirement and the AC model. This includes backward traceability to the originating claim model and forward traceability to the supporting claim that specifies the verification method. These traceability links play a crucial role in integrating the evidence model generated from model checking into the SACM AC module.

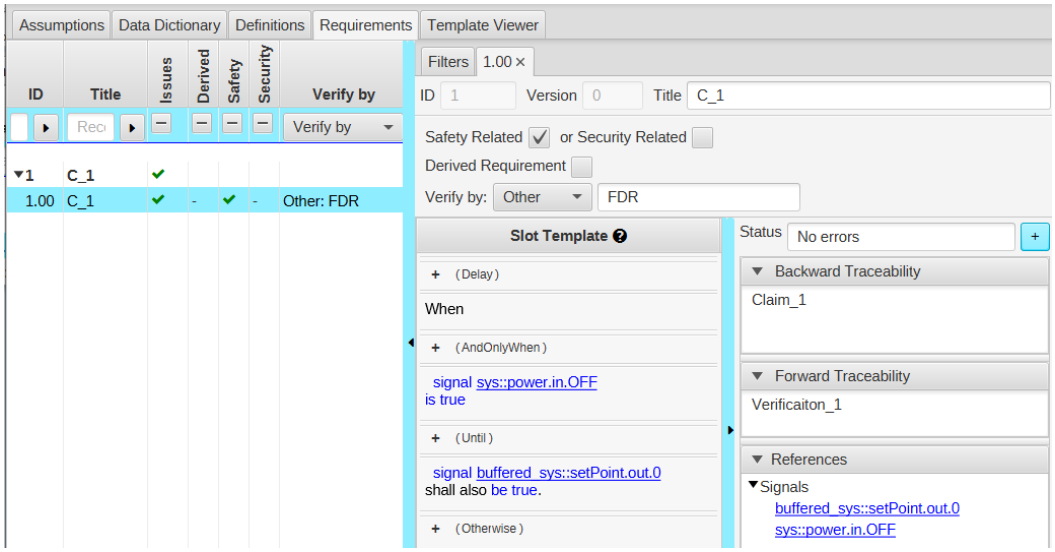


Figure 5.5: R1 in Kapture.

Kapture provides support for exporting requirements in various formats such as XML, HTML, CSV, and more. We utilize the XML format for exporting requirements. As a result, through Step 1, a requirement expressed in natural language is converted into an XML file that contains semi-formal content. This XML file will be subsequently utilized for assertion generation purposes.

5.3 Assertion generation

This section presents the process of assertion generation, which constitutes Step 2 in Figure 5.3. The desired outputs are the assertions composed in RoboChart assertion languages. These assertions are automatically generated by instantiating the assertion templates using the structured requirements. Then, they are converted into formal assertions in CSP or PRISM within RoboTool and checked in FDR or PRISM. Finally, the assertions are verified using FDR or PRISM model checkers.

To generate the assertions, the first activity involves designing the assertion templates for each type of claim. These templates are subsequently instantiated with the machine-readable requirements generated during Step 1. The design of assertion templates is detailed in Section 5.3.1. Section 5.3.2 presents the process of template instantiation. Finally, Section 5.3.3 discusses the integration of evidence models into the AC module.

5.3.1 Assertion template design

In this section, eleven assertion templates have been designed to correspond to the claims listed in Table 5.1. A clear mapping between the templates of the requirements and the corresponding assertions is provided in Table 5.2.

As described in Section 5.1.1 and Section 5.1.2, RoboChart is equipped with two domain-specific assertion languages [55] that have been specifically designed to simplify the process of creating

assertions for FDR and PRISM.

Therefore, the RoboChart assertion languages are used to design the assertions templates. The assertion generated using these templates will be automatically translated into formal CSP_M/PRISM assertions in RoboTool for the purpose of model checking.

Table 5.2: Mapping between templates of the assertions and the requirements

No.	Template of assertions	Template of requirements	Model checker
1	AT-1 Untimed refinement assertion for global invariants	CT-1 Untimed functional requirement	FDR
2	AT-2 Untimed refinement assertion for local invariants	CT-1 Untimed functional requirement	FDR
3	AT-3 Untimed refinement assertion for local invariants with buffer	CT-1 Untimed functional requirement	FDR
4	AT-4 Timed refinement assertion	CT-2 Timed functional requirement	FDR
5	AT-5 Reachability assertion	CT-3 Reachability requirement	FDR
6	AT-6 Deadlock freedom assertion	CT-4 Deadlock freedom requirement	FDR
7	AT-7 Divergence freedom assertion	CT-5 Divergence freedom requirement	FDR
8	AT-8 Termination assertion	CT-6 Termination requirement	FDR
9	AT-9 Probability assertion	CT-7 Probability requirement	PRISM
10	AT-10 Reward assertion	CT-8 Reward requirement	PRISM
11	AT-11 Temporal assertion	CT-9 Temporal requirement	PRISM

Untimed refinement assertions

For the untimed functional requirements, which use the CT-1 template, three assertion templates have been defined: AT-1, AT-2, and AT-3. A refinement assertion is verified using the FDR model checker. The assertion module comprises three sections: (1) the specification that the system should adhere to, (2) the implementation of the system (i.e., RoboChart models), and (3) the assertion on the specification and implementation written in RoboChart assertion language for CSP. The specification and implementation sections are CSP_M blocks written between the keywords **csp-begin** and **csp-end**. The content of the requirement corresponds to the specification section of the assertion file.

AT-1 is the assertion template for global invariants. The specification section takes a typical form that when a 'guard_event' occurs, the 'required_event' should occur. A specific scenario could be that a variable should always maintain a fixed target value or match the value of another variable. AT-1 is shown as follows:

AT-1

untimed csp *Spec* csp-begin
Spec = CHAOS(Events) [| {| *guard_event* |} | >
(RUN({|*guard_event*|} /\ *required_event* -> Spec)
csp-end

untimed csp *Spec*_impl **associated to** *module_name* csp-begin
*spec*_impl =
module_name::O__(0) |\{|*guard_event*, *required_event_without_value*|}) }
csp-end

untimed assertion A_*Claim_ID* :
*Spec*_impl **refines** *Spec* **in the traces model**

In AT-1 template, the keywords of the RoboChart assertion DSL are highlighted in purple. The terms in black and italic format serve as placeholders that will be instantiated using the XML requirements. The terms in black, but not in italic format, are default parts of the template that should not be altered. The identification of the assertion, denoted as *A_Claim_ID*, is established by using the claim identification in AC models, ensuring traceability between the assertion and the AC models.

In the specification section, the AT-1 template incorporates CSP_M operators **Exception** ([| | >) and **Interrupt** (/\) [92] in combination with processes **CHAOS** and **RUN** to construct a pattern that represents the requirement.

The exception P [| A | > Q, behaves as the initial process P, and if P performs an event from A (the set of exception events), it then behaves as process Q. The interruption P /\ Q behaves as process P, but if any action of process Q is performed, P is discarded and the process behaves as Q. Function CHAOS(A) offers the non-deterministic choice over all events in A, but may also deadlock at any point. Function RUN(A) offers the choice over all events in A, perpetually.

The specification *Spec* is interpreted as follows: the system behaves as any non-deterministic event from the set of 'Events', but if the system performs the *guard_event*, then *required_event* must be available, while further *guard_event* may still take place.

The implementation section employs the **Project** operator (|\) to retain the relevant events and filter out any unrelated events associated with this assertion. Given that the assertion aims to verify the refinement between the specification and the implementation, the exact value of the *required_event* in the implementation model should not be specified. Instead, the implementation section uses *required_event_without_value* to represent the *required_event* without including the input/output value.

The third part of the assertion module is the assertion section, which is written using the RoboChart assertion language for CSP. The keyword used for refinement verification is "**refines**". Since this template is specifically designed for safety-related claims, the trace model is used to perform the refinement check.

In AT-1, the placeholders *guard_event* and *required_event* will be replaced with the corresponding RoboChart models *guard_event* and *required_event*, which are extracted from XML requirement written in template CT-1.

In Step 1 of the method, the first phase of formalization is the claim structuring, which involves template instantiation using the formal semantics of RoboChart models. In this section, the design of the assertion template serves as the second phase of formalization by incorporating CSP operators and modelling pattern into the template. The formalization of the requirements will be completed when the generated assertions in RoboChart assertion languages are transformed into pure CSP or PRISM models in RoboTool.

AT-1 is designed for global invariants, capturing system behavior without any scope constraints. However, when the requirement is limited to a specific scope, such as a certain state, it becomes a local invariant. The template for local invariants, AT-2, is derived from AT-1 by introducing an additional layer in the specification section. AT-2 is defined as follows:

AT-2

untimed csp *Spec* csp-begin
Spec = CHAOS(Events) [| {| *guard_event_1* |} | >
(LocalBehaviour −> Spec)

LocalBehaviour = *guard_event_2* − > *required_event* −> LocalBehaviour
csp-end

untimed csp *Spec*_impl **associated to** *module_name* csp-begin
*spec*_impl =
module_name::O__(0) |\{|*guard_event_1*, *guard_event_2*, *required_event_without_value*|}) }
csp-end

untimed assertion A_*Claim_ID* :
*Spec*_impl **refines** *Spec* **in the traces model**

In AT-2, the process *Spec* follows a similar pattern to AT-1, employing the operators **Exception** ([| | >) and the process **CHAOS**. *Spec* is interpreted as follows: "The system behaves as any non-deterministic event in the set of 'Events'. However, if *guard_event_1* evaluates to true, then the process *LocalBehaviour* must be available, while allowing for the possibility of further occurrences of *guard_event_1*". The *guard_event_1* serves to define the scope of the invariants. The newly introduced process, *LocalBehaviour*, is defined as "if the guard event *guard_event_2* occurs, the required event *required_event* also occurs". The implementation section is similar to that of AT-1, utilizing the **Project** operator (|\) to retain the relevant events in the model.

Assertion template AT-3 is derived from AT-2 and is designed for local invariants with consideration of a buffer. In RoboChart, when the events of a robotic platform are connected to the events of controllers, these connections are asynchronous in nature, as the software does not block the platform. To model asynchronous communication, a CSP process is used to define a buffer. Although the buffer is hidden in RoboChart models, its semantics are represented by a CSP process. The inclusion of asynchronous communication adds complexity to the assertions. AT-3 is presented below:

AT-3

untimed csp *Spec* csp-begin
channel buffered_*require_event*: core_real
channel read: core_real

BufferedOutput = (Buffer [| {|read|} |] Output) \ {|read|}

– Output simply outputs what it reads.
Output = read?x__ − > *require_event* − > Output

– A buffer, whose current 'value' can be overwritten.
Buffer = buffered_*require_event*?x__ − > (read!x__ − > Buffer [] Buffer)

Spec = (Behaviour [| {| buffered_*require_event* | } |] BufferedOutput) \ {| buffered_*require_event* | }

Behaviour = CHAOS(Events) [| {|*guard_event_1*|} | > (InnerLoop /\ Behaviour)

InnerLoop = *guard_event_2* − > buffered_*require_event* − > InnerLoop csp-end

untimed csp *Spec*_impl **associated to** *module_name* csp-begin
*spec*_impl =
module_name::O__(0) |\{|*guard_event_1*, *guard_event_2*, *required_event_without_value*|}) }
csp-end

untimed assertion A_*Claim_ID* : *Spec*_impl **refines** *Spec* **in the traces model**

Asynchronous connections are implemented through the use of a buffer, which is represented by a process that runs in parallel with the controller processes. Therefore, in AT-3, if the "required_event" is an asynchronous communication, an intermediate communication event called "buffered_required _event" is introduced. The event is labelled with "buffered" as a prefix. In the template, the value communicated by "require_event" is assumed to be of type *real*. However, it should be adjusted based on the actual event type in the system.

Then, the buffer is represented by the process *BufferedOutput*, which concurrently executes the processes *Buffer* and *Output*. The process *Buffer* models a buffer with a capacity of one. It accepts inputs and overwrites the current value in the buffer. On the other hand, the process *Output* models the output of the buffer. Additionally, an event called *read* is introduced to simulate the buffer's reading operation. This event has the same type as *buffered_required_event*.

Then, the specification *Spec* is represented as a process that executes the controller behaviour *Behaviour* and the buffer behaviour *BufferedOutput* concurrently. These two parallel processes are synchronized on the event *buffered_required_event*, which is hidden using the **Hide** (\) operator. P \ Q Behaves like process P, but if P performs any event from the event set A, the event is hidden and turned into an internal event.

The requirement is actually modelled by the process *Behaviour*, which follows the same pattern as the *Spec* process in AT-2.

Timed refinement assertion

RoboChart's timed behaviour is underpinned by the timed formal semantics of *tock*-CSP [93]. *tock*-CSP is a variant of CSP that introduces discrete time modelling through the use of the event *tock* to represent the passage of time. CSP_M incorporates a timed section syntax that automatically translates processes into *tock*-CSP, enabling verification in tools such as FDR.

AT-4 is the assertion template designed for timed refinement. It addresses the scenario where, if a trigger event occurs, a required event must take place within a specified time limit when the corresponding guard event occurs. This template aligns with timed functional requirements that utilize the requirement template CT-2. It is formulated using the RoboChart assertion DSL for FDR and CSP_M blocks. The template is presented below:

AT-4

timed csp *Spec* csp-begin
Timed(OneStep) {
Spec = **timed_priority**(Process_1)
Process_1 =
target_event− > Process_1 [ ]
guard_event− >Process_2 [ ]
trigger_event − >Process_1
Process_2 =
target_event− >Process_1 [ ]
guard_event− >Process_2 [ ]
trigger_event− >**TRUN**
{| *trigger_event*, *target_event_without_value*|}
/\(**WAIT**(*time_limit*);
USTOP [|{|*target_tevent*|}| > **SKIP**; Process_1}
csp-end

timed csp *Spec*_impl **associated to** *module_name* csp-begin
Timed(OneStep) {
*spec*_impl =
timed_priority(
module_name::O__(0,*time_limit*,1)
|\ {|*trigger_event*, *target_event_without_value*, tock|}) }
csp-end

timed assertion A__*Claim_ID* :
*Spec*_impl **refines** *Spec* **in the traces model**

The process for the specification is defined within a **Timed** section[3], which designates the modules as timed and automatically introduces *tock* events to account for the passage of time. The **OneStep** construct determines the duration of each synchronisation with the environment [85]. By default, it is set to 0, indicating that no time elapses during the synchronization. The **timed_priority** function enables the prioritization of *tock* events over internal behaviour, ensuring that the internal behavior does not consume any time.

Four *tock*-CSP processes are employed in the specification section of AT-4. The **SKIP** process terminates a process. **WAIT**(d) terminates exactly after a delay of d units of time. Similar to **RUN**(X) in CSP, the **TRUN**(X) process continuously offers events from the set X to the environment while

[3]https://cocotec.io/fdr/manual/cspm/definitions.html#csp-timed-section

allowing time to pass. On the other hand, **USTOP** denotes timelock, i.e., no events are offered (deadlock) and time cannot pass.

The specification is formulated using two processes, namely *Process_1* and *Process_2*. The interpretation of *Process_1* is as follows: there is an external choice where either *target_event* can occur, or *trigger_event* can occur, or once the *guard_event* is triggered, the process transitions to behave as *Process_2*.

In *Process_2*, there are other choices: *guard_event* can occur, or *target_event* can occur and then the process behaves as *Process_1*, or if the *trigger_event* continuously occurs and so does the *target_event* but with any possible value, then the *target_event* with a specific value can be observed within given time limit, denoted as *time_limit* time units, then the process behaves as SKIP that terminates immediately and resumes the behaviour of *Process_1*.

Similar to the implementation section in AT-1, the implementation section of AT-4 also employs the **Project** operator ($|\backslash$) to restrict the events to the relevant ones. However, an additional event, *tock*, is included in this process to model the pass of the time.

General assertions

Four assertion templates, namely AT-5, AT-6, AT-7, and AT-8, are designed to address reachability, deadlock freedom, divergence freedom, and termination. These assertions are formulated using the RoboChart assertion DSL for FDR. They can be applied to the timed, untimed or both timed and untimed semantics of RoboChart models.

Each assertion template is labelled as either "untimed" for using untimed semantics or "timed" for using timed semantics. In cases where neither label is specified, the assertion is checked against both timed and untimed semantics. The specific definitions of AT-5, AT-6, AT-7, and AT-8 are presented below:

AT-5

(**untimed** | **timed**)? **assertion** A_*Claim_ID*:
state bool_term **reachable in** *scope*.

∗ *bool_term* ::= **is** | **is not**

AT-6

(**untimed** | **timed**)? **assertion** A_*Claim_ID*:
scope bool_term **deadlock-free**.

∗ *bool_term* ::= **is** | **is not**

AT-7

(**untimed** | **timed**)? **assertion** A_*Claim_ID*:
scope bool_term **divergence-free**.

∗ *bool_term* ::= **is** | **is not**

AT-8

(**untimed** | **timed**)? **assertion** A_*Claim_ID*:
scope bool_term

∗ *bool_term* ::= **terminates** | **does not terminate**

In the above templates, the placeholders *state* and *scope* will be replaced with the elements of the same placeholders from the corresponding requirements which are structured using CT-3, CT-4, CT-5, and CT-6.

Three templates, namely AT-9, AT-10, and AT-11, are designed for probabilistic and temporal assertions. These templates utilize the RoboChart assertion language for PRISM. The resulting assertions are automatically converted into PRISM assertions and verified in PRISM within RoboTool. The instantiation data for the placeholders will be extracted from XML requirements generated using CT-7, CT-8, and CT-9.

Probability assertion

AT-9 is the template for probability assertions as shown as follows:

AT-9

prob property P_*Claim_ID*:
Prob *prob_target* **of** [*path_formula*]

(**with constant**
(*constant_i*)∗ ,
(*multi_constant_j*)∗)∗
(**const** *x*: core::nat)∗

The template consists of two sections. The assertion section begins with the keywords **prob property**. The assertion includes the property name and content, which verifies whether the probability of *path_formula* satisfies the quantitative bound specified in *prob_target*. The identification of the assertion, *P_Claim_ID*, is defined based on the claim identification in AC models, ensuring traceability between the assertions and AC models.

The second section 'with constant' is used for the constant configuration. *constant_i* is employed to assign a single value to the constant, while *multi_constant_j* is used to assign a set of values to the constant. The final line is dedicated to declaring the constants.

The placeholders within AT-9 will be substituted with elements of the corresponding type found in requirements written using CT-7.

Reward assertion

AT-10 is the template for reward-based assertions, aligning with the corresponding requirement template CT-8. AT-10 is presented below:

AT-10

rewards reward_*Claim_ID*
=[*reward_event*] (true | false) : *reward_value*
endrewards

prob property R_*Claim_ID*:
Reward reward_*Claim_ID reward_target* **of** [*path_formula*]

(**with constant**
(*constant_i*)∗ ,
(*multi_constant_j*)∗)∗
(**const** *x*: core::nat)∗

AT-10 comprises three sections. The first section, titled "rewards" specifies the event that generates a cost or reward, along with its corresponding reward value. The second section introduces the assertion segment, denoted by the keywords **prob property**. Within this section, the assertion identification *R_Claim_ID* is established, utilizing the claim identification from AC models. The assertion assesses whether the resultant reward value complies with the defined bound (*reward_target*) while satisfying the designated *path_formula*. The final section pertains to the constant configuration.

Depending on the nature of the 'path_formula', the calculated reward can take various forms. It can represent the accumulated value along a path until reaching a specific point or a state that satisfies a given property. Alternatively, it can denote the accumulated value along a path within a specified time limit, the total accumulated value along the entire path, or the value at a particular instant in time [94].

Temporal assertion

AT-11 serves as the template for temporal assertions, featuring two sections similar to AT-9. These assertions assess whether the property specified in the *path_formula* is satisfied by either all paths or some paths. The placeholders *temporal_operator* can have the values of *forall* or *exists*. Both placeholders will be substituted with the corresponding elements found in the requirements documented in CT-9.

AT-11

prob property T_*Claim_ID*:
temporal_operator [*path_formula*]

(**with constant**
(*constant_i*)∗ ,
(*multi_constant_j*)∗)∗

The assertion templates and the requirement templates utilise the identical set of placeholders, which

simplifies the process of generating assertions by instantiating the assertion placeholders with the content from the requirement placeholders. The assertion generation is discussed in Section 5.3.2.

5.3.2 Assertion template instantiation

An assertion is generated by instantiating the assertion template with the elements extracted from the requirements structured in XML format. To establish the correspondence between the elements of the requirements and the placeholders of the assertion templates, the same names are employed in both sets of templates. For example, the *guard_event* in CT-1 is utilised to instantiate the *guard_event* in AT-1. This process is automated through the execution of the Epsilon EGL [87] model to text transformation, which establishes the linkage between the requirement and the assertion template.

The instantiation process takes as input the XML requirements, which contain information about the requirement type and the data for instantiation. To select the appropriate assertion template, the type of the requirement template, such as WHEN, CASE, or EVERY, is considered along with the labels associated with certain terms in the requirements. For example, if the Kapture template is "WHEN" and its *RequiredCondition* section is prefixed with *prob_target*, the corresponding assertion template would be AT-9. The elements of the requirement are then extracted from the XML files, following the structure of the requirement template, and used to instantiate the assertion template.

To automate the instantiation of assertion templates, the mapping relationship between the requirement and assertion templates is incorporated into EGL transformation codes. The EGL codes specify the structure of the assertion template and define how it relates to the elements of the source requirement templates.

The transformation rules outlined in Algorithm 4 demonstrate the instantiation process applicable to each type of assertion template.

Algorithm 4: Abstract procedure of assertion template instantiation

```
1 for r in kapture_xml.reqs do
2     rTemp = r.readKapTemp()
3     emlements = r.fetch(rTemp)
4     asstTemp = readAsstTemp(rTemp, elements.getPrefix())
5     emlements = elements.removePrefix()
6     asstTemp.instantiate(elements)
7 end
```

In Algorithm 4, the source model *kapture_xml* is generated in Kapture and may have multiple requirements that require support from model checking and are stored in the set *kapture_xml.reqs*. For each requirement *r* in the set *kapture_xml.reqs*, the process begins by identifying the Kapture template *rTemp* used for the corresponding claim by calling the function *readKapTemp*(line 2). Subsequently, the claim's *elements* are retrieved based on their locations within the template structure (lines 3). The next step involves determining the appropriate assertion template based on the type of requirement template and associated prefixes (line 4). The extracted elements from the requirement undergo processing to remove any prefixes (line 5), after which they are used to instantiate the assertion template (line 6).

The complexity of Algorithm 4 is $O(n)$ because of the for loop from lines 1-7. n is the number of the requirments in the XML file. The correctness of Algorithm 4 is considered correct if the assertions generated with the different input requirements are compliant with the right type of assertion template. We verified the correctness of this algorithm by using different case studies in Section 5.5 to compare the automatically generated assertions with the manually generated ones.

An example process is demonstrated using R1. The requirement R1 written in CT-1 is repeated below:

> When *sys::power.in.OFF* is true,
> *sys::setPoint.out.0* shall also be true.
>
> ∗ guard_event ::= sys::power.in.OFF
> ∗ required_event ::= sys::setPoint.out.0

CT-1 contains two placeholders, namely *guard_event* and *required_event*. The values assigned to these placeholders are *sys::power.in.OFF* and *sys::setPoint.out.0*, respectively. These values are used to instantiate the placeholders *guard_event*, *required_event*, and *required_event_without_value* in AT-1. Since CT-1 does not employ any prefixes, there is no need to remove any prefixes. The generated assertion is as follows:

> **untimed csp** *Spec* csp-begin
> *Spec* = CHAOS(Events) [| {| *sys::power.in.OFF* |} | >
> (RUN({|*sys::power.in.OFF*|} /\ *sys::setPoint.out.0* -> Spec)
> csp-end
>
> **untimed csp** *Spec*_impl **associated to** *module_name* csp-begin
> *spec*_impl =
> *module_name*::O__(0) |\{|*sys::power.in.OFF*, *sys::setPoint.out*|}) }
> csp-end
>
> **untimed assertion** A_*C_1* :
> *Spec*_impl **refines** *Spec* **in the traces model**

In RoboTool, the generated assertion A_C_1 is translated into formal assertions in CSP_M. This translation primarily involves converting the RoboChart assertion languages into the pure CSP_M language. This step represents the final phase of requirement formalization. Subsequently, the assertion module is subjected to verification using FDR to assess the satisfaction of the R1 by the RoboChart models. The translation and verification processes are automated by RoboTool and the FDR model checker. The verification result obtained in RoboTool is further utilized for the generation of AC evidence models, which will be elaborated on in Section 5.4.

5.3.3 Assertion generation integration

The processes of assertion generation for each type of assertion, as discussed in Section 5.3.2, are combined into a unified standalone implementation, as depicted in Algorithm 5. Algorithm 5 encompasses the identification of the assertion template and the subsequent generation of the assertion using the appropriate template. For instance, if a source claim employs the “WHEN” template and

its segment "RequestCondition" is prefixed with "prob_target", then the resulting assertion belongs to the probability type. The generation process follows procedures in Algorithm 4.

Algorithm 5: The integrated process of assertion generation

```
1  for r ∈ kapture_xml.reqs do
2      claim_ID = r.backwardsTr
3      if r.template = "WHEN" then
4          if r.requiredCondition.prefixed("prob_target") then
5              GENERATEPROBASST(r, claim_ID)                          ▷ Alg. 18
6          end
7          if r.requiredCondition.prefixed("reward_target") then
8              GENERATEREWARDASST(r, claim_ID)                        ▷ Alg. 19
9          end
10         if r.requiredCondition.prefixed("NIL") then
11             GENERATETEMPASST(r, claim_ID)                          ▷ Alg. 20
12         end
13         numOfCons = 0
14         hasConst = false
15         for elem in r.guardCondition.elems do
16             if elem.prefix ∈ {"constant", "multi_constants"} then
17                 hasConst = true
18                 numOfCons++
19             end
20         end
21         i = 1
22         if hasConst then
23             for elem ∈ r.guardCondition.elems do
24                 if elem.prefixed("constant") and i ≤ numOfCons then
25                     constant_i = elem.removePrefix()
26                     i++
27                 end
28             end
29             generateConstant(numOfCons, constant_i)
30         end
31         if r.requiredCondition.prefix ∉ {"prob_target", "reward_target", "NIL"} then
32             GENERAUNTIMEDASST(r, claim_ID)                         ▷ Alg. 22
33         end
34     end
35     if r.template = "CASE" then
36         GENERATETIMEDASST(r, claim_ID)                             ▷ Alg. 23
37     end
38     if r.template = "EVERY" then
39         GENERATEGENERALASST(r, claim_ID)                           ▷ Alg. 24
40     end
41 end
```

In **Algorithm** 5, each requirement *r* in the set *kapture_xml.reqs* is processed to generate the formalized assertions. The requirement *r* corresponds to an entry record in the XML file. It has an attribute *backwardsTr* which represents the identification of the associated claim.. This claim identification is used to generate the assertion identification. Rerquirement *r* has an attribute of requriement type *r.template*. The requirement may also have the attributes *requiredCondtion* representing the RequiredCondition section, and *guardCondition* representing GuardCondition section in the Kapture

template. Each section contains a set of elements that consist of the whole condition. For example, the requirement *r* has a set *r.guardCondition.elems* which are the guard conditions and logical operators among the conditions. The process begins with the "WHEN" template. If the requirement template *r.template* is "WHEN", then the generated assertions can be of various types, such as probability, reward, temporal, or untimed refinement. The generation of these assertions is described in Algorithm 18, Algorithm 19, Algorithm 20, and Algorithm 22, respectively. If the requirement template is "CASE", the generated assertion belongs to the timed refinement type, which is discussed in Algorithm 23. In the case of the "EVERY" requirement template, the generated assertion is of a general type, as discussed in Algorithm 24.

The complexity of Algorithm 5 is $O(n^2)$ because there are two parallel nested for loops. The first nested for loops are from lines 1-39 and lines 13-18. The second nested for loops are from lines 1-39 and 21-26. n is the larger value between the number of requirements and the number of constant declarations within each requirement.

Algorithm 5 is considered correct if the assertions generated with the different input requirements are compliant with the right type of assertion template. We verified the correctness of the algorithm by using different case studies in Section 5.5 to compare the automatically generated assertions with the manually generated ones.

Please refer to Appendix B for the detailed presentation of Algorithm 18, 19, 20, 22, 23, and 24.

The assertion templates for probability, reward, and temporal types each include a 'constant configuration' section. This section is generated in lines 15 - 30 of the code, separate from Algorithm 18, Algorithm 19, and Algorithm 20. This presentation is to avoid repetition in the algorithm. Each generated assertion is saved in a separate file following the naming format "assertion.assertions".

5.4 Evidence model generation and integration

After obtaining the model checking result, an evidence model is generated using model transformation techniques in accordance with the SACM metamodel. This evidence model incorporates the traceability information linking the result to the supported claim. With this traceability information, the old evidence model associated with the claim is identified. Subsequently, this outdated evidence model is removed from the association with the claim and deleted from AC module, and the new evidence model is connected to the claim it supports.

The evidence model generation process involves utilizing the verification results obtained from FDR/PRISM. These results are provided by RoboTool in the form of an HTML report, which includes the checked assertions and their pass/fail outcomes. Each assertion file corresponds to a single assertion being verified, resulting in a report that presents the results for a specific assertion. Examples of these HTML reports can be seen in Figure 5.6 and Figure 5.7.

During the verification process using FDR, the generated report includes separate tables for untimed analysis and timed analysis, if applicable. When using PRISM, the report consists of a single table. Depending on the assertions being evaluated, this table may contain multiple results. The title of the table in the HTML report indicates the claim that is supported by the verification, such as "Claim_3" as shown in Figure 5.6. In order to generate and integrate an evidence model into the AC models, the

Results of untimed analysis of assertions in Claim_3.assertions using FDR

Assertion	States	Transitions	Result
untimed mod_sys_setpoint refines Claim_3 [traces model]	210022	4674325	true

Results of timed analysis of assertions in Claim_3.assertions using FDR

There are no timed assertions.

Figure 5.6: An example of a FDR model-checking report in RoboTool.

Results of probabilistic analysis of assertions in Claim_4.assertions using PRISM

Assertion: P_Claim_4

Assertion	Const	Const	Const	states:	transitions:	result:	checkTime:
P_Claim_4	x=1	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	43.522 seconds
P_Claim_4	x=2	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	42.446 seconds
P_Claim_4	x=3	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	40.292 seconds

Figure 5.7: An example of a PRISM model-checking report in RoboTool.

HTML report, XML claim, and AC models generated through pattern instantiation are utilized. It is necessary to establish traceability among these input artifacts. The algorithm outlining this process is presented in Algorithm 6.

Initially, it is crucial to identify the claim that is supported by the model checking process. The HTML report denoted as *report* contains one or two tables in the set *report.tables*. The identification of the claim, which is to be processed, is denoted as *claim_id*. Each table in the report has a title, for example, the title of the first table is *report.tables. f irst.title*. This title contains the information of the identification as well as the model checking tool. For example, a table title could be "Results of probabilistic analysis of assertions in SR1_1_1.assertions using PRISM" in which the tool is "PRISM" and the requirement referenced in the claim is "SR1_1_1". Therefore, the identification can be found in the title of the first table (line 1). However, it should be noted that this claim does not directly correspond to the claim that the evidence supports within the AC models. Referring to the AC pattern illustrated in Figure 4.5, if a claim is intended to be supported by a specific verification method, it must possess a lower-level claim that specifies the employed verification method. Consequently, the evidence will be directed towards supporting this lower-level claim and indirectly the higher-level claim.

The traceability between the evidence and the supported claims, including both higher-level and lower-level claims, is documented within the requirement attributes *backwardsTr* for backward traceabilit" and *forwardTr* for forward traceability. The allocation of the claim *claim_xml* in the XML file is accomplished using the *claim_id* (line 2). Subsequently, the claim that directly receives support from the evidence is identified within the "forwardsTr" section (line 3).

To generate evidence models, the model checking tool used is identified as FDR or PRISM. In the case of PRISM verification, there is only one table in the report. This table has a set of rows in the set *report.table.first.rows*. The final result is considered successful if each row in the table yields a "true" result (line 6 - 10). For FDR, the final result is deemed successful when both tables' results

Algorithm 6: Model checking evidence generation and integration

```
1  claim_id = report.tables.first.title.getClaimID()
2  claim_xml = kapture_xml.reqs.select(c | c.backwardsTr = claim_id)
3  vclaim_id = claim_xml.forwardTr
4  tool = report.tables.first.title.getToolType()
5  result = true
6  if tool=PRISM then
7      for r ∈ report.tables.first.rows do
8          result = result ∧ r.getResult()
9      end
10 end
11 if tool=FDR then
12     for result_table ∈ report.tables do
13         result = result ∧ result_table.getResult()
14     end
15 end
16 evidence = result.genEvi(tool)
17 for assertedEv ∈ ac.argumentationElement.select(c | c.type = AssertedEvicence) do
18     if assertedEv.target=vclaim_id then
19         for c ∈ ac.argumentationElement.select(c | c.type = Claim) do
20             if c ∈ assertedEv.source then
21                 c.delete()
22             end
23         end
24         assertedEv.source.clear()
25         assertedEv.source = {evidence}
26     end
27 end
```

are evaluated as "true" (line 11 - 15).

Upon obtaining the verification result, the evidence model *evidence* is generated (line 16). To integrate this evidence into the AC module generated through pattern instantiation, it is necessary to remove the old evidence and its associated claim linkage (line 18-24). Subsequently, the generated evidence is linked to the supporting claim, thus completing the integration process. The complexity of this algorithm is $O(n^2)$. Algorithm 6 is considered correct if the evidence model generated has the content expected. We verified the correctness of the algorithm by using different case studies in Section 5.5 to compare the automatically generated models with the expected models.

During the system development process, multiple iterations of model checking may be necessary as the RoboChart models are updated. Consequently, the process of evidence generation and integration follows a similar iterative manner to accommodate the evolving system development.

5.5 Case study and evaluation

In this section, two case studies are presented to demonstrate the application of evidence model generation through FDR and PRISM model checking, as discussed in Section 5.5.3 and Section 5.5.2 respectively. The evaluation presented in Section 5.5.4 is conducted based on the outcomes of these

two case studies.

5.5.1 Implementation

According to Figure 5.3, the method outlined in Section 5.1.3 consists of four distinct steps. Step 1 involves manual processes, whereas Step 3 is fully implemented by RoboTool and does not contribute to our work. A comprehensive summary of the tasks associated with each step and the corresponding supporting tools is provided in Table 5.3. The implementation is developed within the Eclipse EMF framework and relies on three tools: RoboTool[4], Epsilon[5], and Kapture[6].

Table 5.3: Status of tool support for the steps in AC process

No.	Step	Automated/ manual	Tool support
1	AC Claim structuring	Manual	Kapture
2	DSL Assertion generation from XML claims	Automated	Epsilon/ EGL
3	Model checking	Automated	RoboTool/ FDR&PRISM
4	Evidence model generation and integration	Automated	Epsilon/ EOL

RoboTool, an Eclipse plugin, functions as a modelling environment for RoboChart and integrates FDR and PRISM model checkers to enable formal verification. It provides support for generating formal semantics from RoboChart models, converting assertions in RoboChart assertion languages into formal assertions, and retrieving model checking results by invoking the model checkers. Epsilon is used to facilitate the transformation process, converting XML claims into RoboChart DSL assertions and translating model checking results into SACM evidence models. Additionally, Kapture is employed to organize and export claims into XML files.

5.5.2 Case study: mail delivery robot

A case study of a mail-delivering robot [84] is conducted to demonstrate the application of the method for generating evidence models using PRISM model checking.

System description

A mail-delivering robot is responsible for delivering mail to eight offices, labelled as 1 to 8, as depicted in Figure 5.8. The battery charging station is represented by Office 0. The robot adheres to the rule that an operator in one office can request mail delivery to another office, excluding the charging station. The robot is designed to handle a single delivery task at a time. Upon receiving a request from an office, the robot first travels to the source office to retrieve the mail and then proceeds to the designated destination office for delivery. Once the task is completed, the robot becomes available for the next request. The robot's next move is determined randomly, with an equal probability assigned to all available options. It can choose to remain at its current location or move to either of the adjacent offices. For example, if the robot is currently located in Office 3, it can either stay at Office 3 or move to either Office 2 or Office 5, each option having a probability of 1/3.

[4]https://www.cs.york.ac.uk/robostar/robotool/update/
[5]http://download.eclipse.org/epsilon/updates/2.2/
[6]https://www.drisq.com/product-kapture

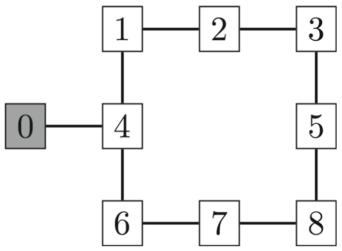


Figure 5.8: Map of the workplace [84]

A top-level hazard of the robot is identified as follows::

H1: Robot fails to deliver the mail.

This hazard arises when the robot exhausts its power supply while situated at a location away from the charging station. A detailed analysis of this hazard has been performed, and the results of the hazard analysis have been transformed into EMF models, illustrated in Figure 5.9.

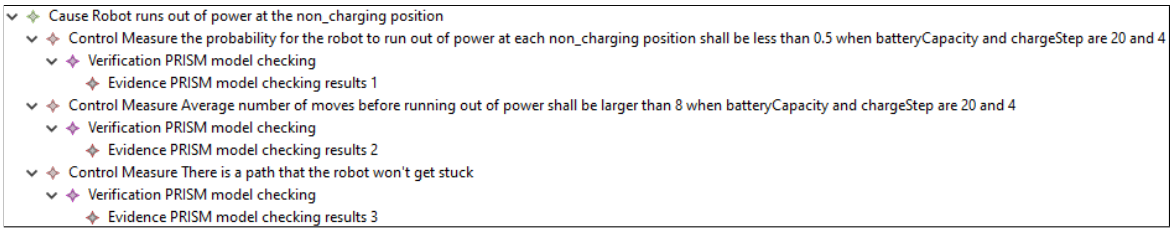


Figure 5.9: Hazard analysis results for hazard cause of running out of power.

Preliminary AC generation

Utilizing the results of the hazard analysis, the initial construction of the AC argumentation structure is accomplished through the application of the pattern instantiation method. The resulting AC graph is generated by transforming the models into a Graphviz DOT file, and an illustration of the AC graph is provided in Figure 5.10. For the sake of simplicity, the context elements of the AC are excluded from the diagram.

In this initial AC argumentation structure, there are three claims that can be supported by PRISM model checking, namely “Goal for VR1_1_1”, “Goal for VR1_1_2”, and “Goal for VR1_1_3”. The actual requirements to be verified in PRISM are referenced in the higher level claims of “Goal for SR1_1_1”, “Goal for SR1_1_2”, and “Goal for SR1_1_3”. It’s assumed that the verification activities have not been conducted at the time of the AC generation; therefore, the model checking results that are to be referenced to as the solutions are unavailable at this stage. The subsequent step involves generating the evidence model from the set of goals, i.e., “Goal for SR1_1_1”, “Goal for SR1_1_2”, and “Goal for SR1_1_3”. The first activity is to structure the requirements in the three claims using the tool “Kapture” and the requirement templates outlined in Section 5.2.3.

Claim structuring

Within the three claims, the three requirements referenced are as follows:

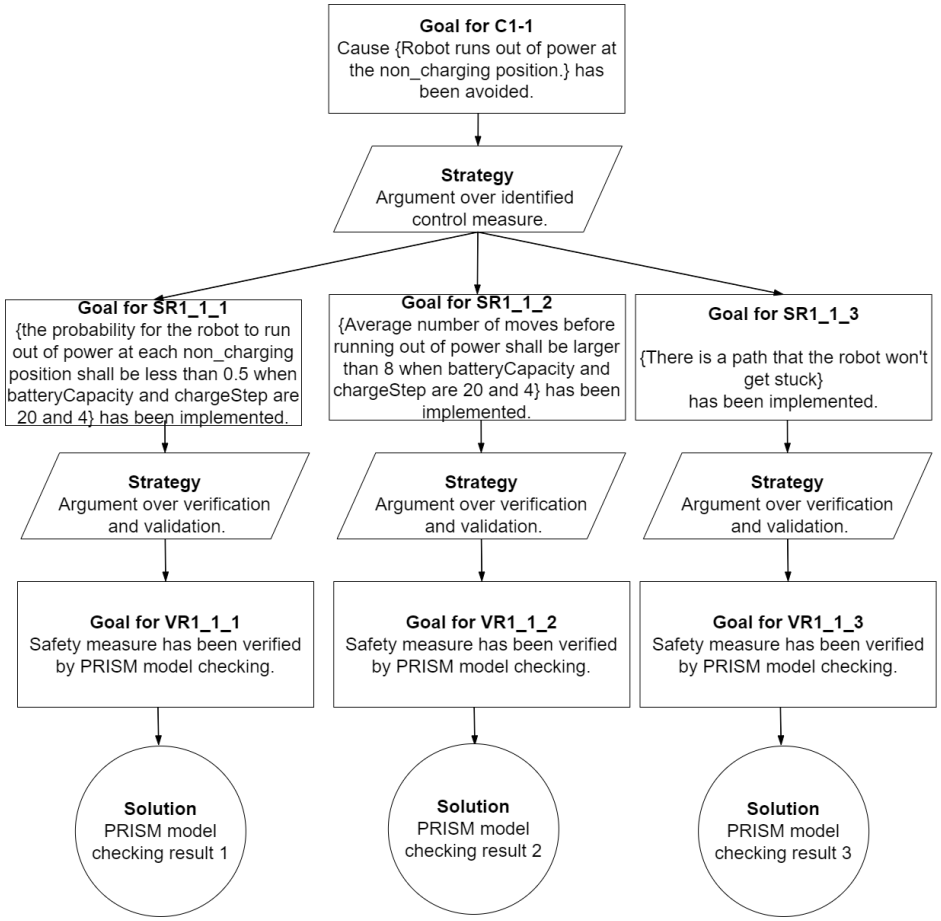


Figure 5.10: AC in GSN notation for the mail-delivering robot.

1. SR1_1_1: The probability for the robot to run out of power at each non_charging position shall be less than 0.5 when battery capacity and charge step are 20 and 4.
2. SR1_1_2: Average number of moves before running out of power shall be larger than 8 when battery capacity and charge step are 20 and 4.
3. SR1_1_3: There is a path that the robot won't get stuck.

SR1_1_1 is a probabilistic requirement. SR1_1_2 can be categorized as a reward-type requirement. SR1_1_3 is a requirement of temporal type. Thus, the corresponding requirement templates for these three are CT-7, CT-8, and CT-9, respectively.

To structure the requirements using the Kapture templates, the input required is the RoboChart models. The RoboChart models for the mail delivery robot are depicted in Figures 5.11, 5.12, 5.13, 5.14, and 5.15.

The RoboChart module is shown in Figure 5.11. The module deliverMOD is composed of a robotic platform deliverRP and a controller deliverCTRL. A platform includes variables, events, and operations that represent sensors and actuators. It may also define constants. RoboChart interfaces are used to group elements for reuse and to describe dependencies. In this example, the ctrlVarInf interface defines the variables p and c to represent the position and battery level of the robot, respectively.

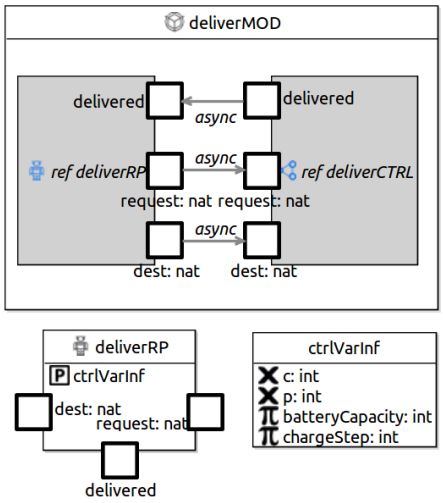


Figure 5.11: Module and Robotic platform in RoboChart model for delivery robot [55].

The constants batteryCapacity and chargeStep are also defined to represent the battery capacity and the amount of battery charged per update, respectively. The interface ctrlVarInf is provided by the robotic platform deliverRP. Therefore, the variables and constants defined within ctrlVarInf are used in the controller and all its state machines, which require this interface.

The platform and the controller in the RoboChart module communicate with each other through connected events. The controller deliverCTRL, receives a mail delivery request from the platform. The request includes the office number where the mail is located and the destination office number. This information is transmitted to the controller through the typed events request and dest. Once the delivery is completed, the controller notifies the platform via the delivered event.

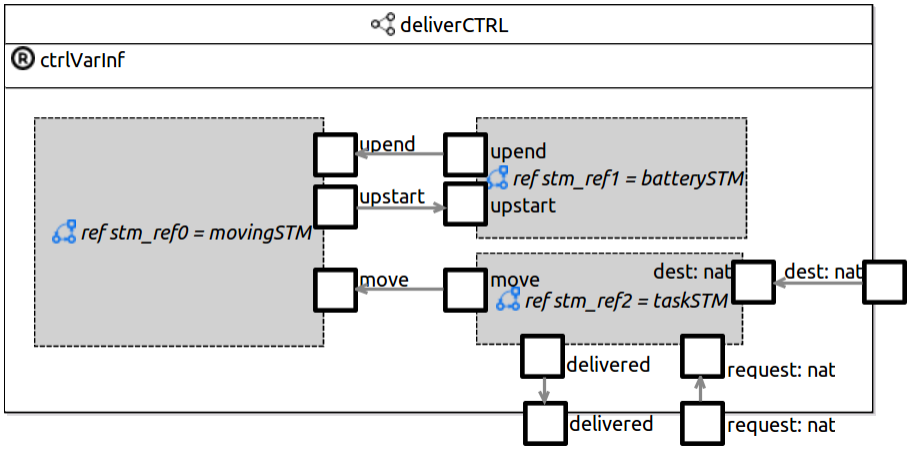


Figure 5.12: Controller in RoboChart model for delivery robot [55].

The deliverCTRL controller is depicted in Figure 5.12. It consists of three state machines: movingSTM, batterySTM, and taskSTM, which handle movement control, battery management, and task management, respectively. The definitions of these machines can be found in Figures 5.13, 5.14, and 5.15. As shown in Figure 5.12, the movingSTM and taskSTM machines synchronize with each

other using a move event. This event is used to trigger movement when the current position differs from the desired goal, which can be either the mail request office or the mail destination office. Additionally, the movingSTM and batterySTM machines synchronize using the upstart and upend events. These events are used to trigger the start and end of a battery level update after a movement has taken place.

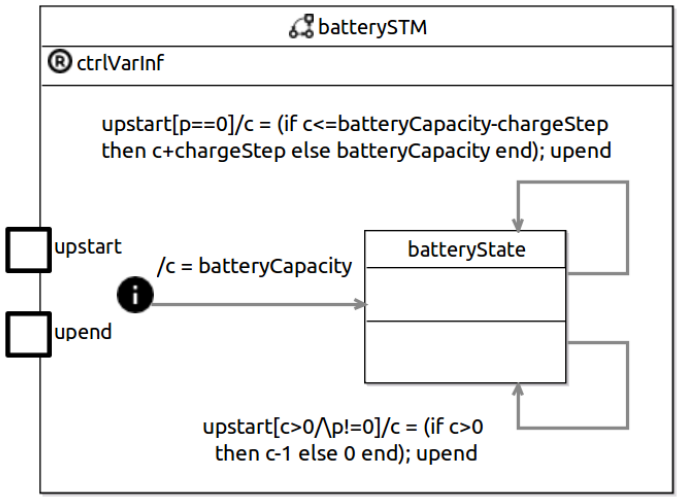


Figure 5.13: Battery state machine [55].

The batterySTM machine, displayed in Figure 5.13, is responsible for managing the robot's battery. It consists of a single state, batteryState. The upstart and upend events in the batterySTM machine are synchronized with events in the movingSTM machine. This synchronization enables the battery to be updated while the robot is in motion.

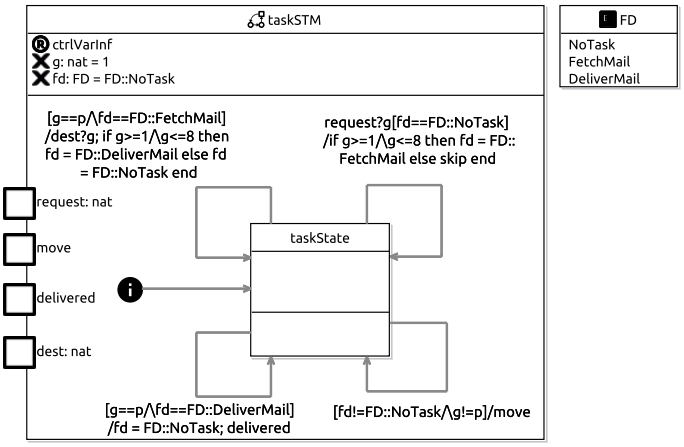


Figure 5.14: Task state machine [55].

The taskSTM machine, depicted in Figure 5.14, consists of a single state, taskState, and has four self-transitions. However, since the three safety requirements are unrelated to the behaviour of the taskSTM machine, its functionality will not be discussed in this context.

The movingSTM machine, shown in Figure 5.15, is responsible for controlling the movement of

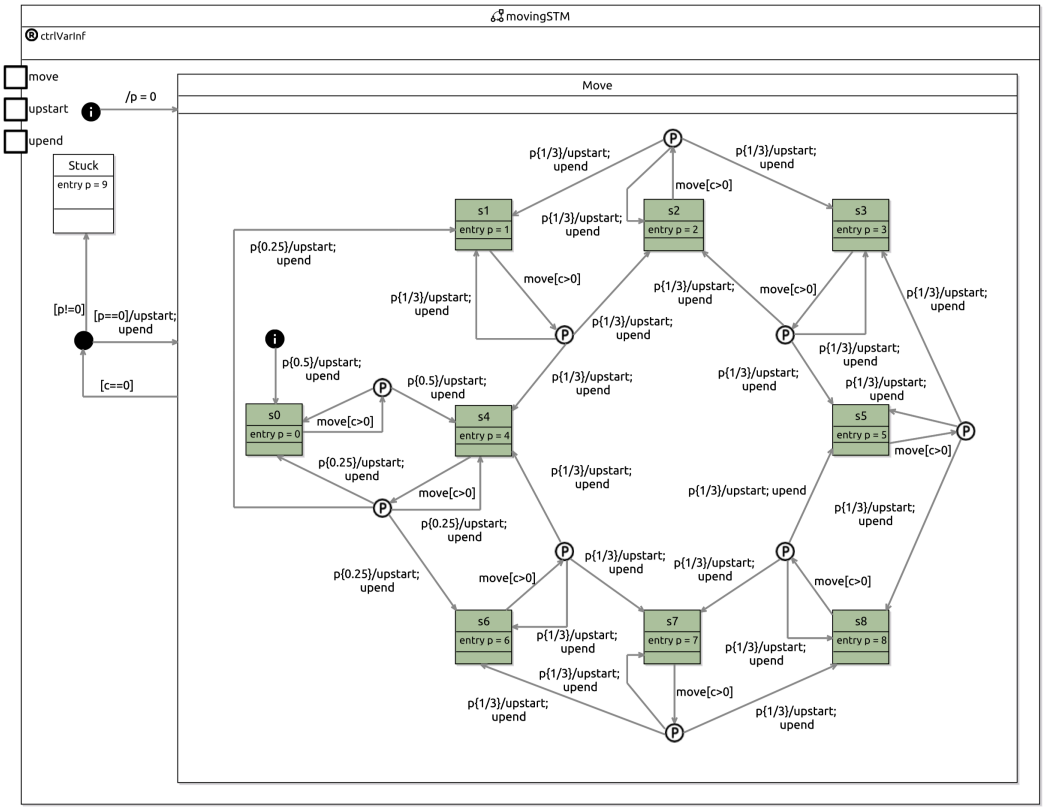


Figure 5.15: Movement state machine [55].

the robot. It consists of two states: Stuck and Move. The Move state is a composite state that contains nine sub-states (s0 to s8). Each sub-state corresponds to an office and has an entry action, such as p=2, which sets the value of p to 2 when entering state s2. This action records the current office location of the robot in the variable p. The transitions from these sub-states are connected to probabilistic junctions and are labeled with a trigger move and a guard condition (c>0). As a result, these transitions are only taken when the movement is triggered by the taskSTM machine and the battery level is not empty.

According to the RoboChart models, the requirements in the AC claims are structured using Kapture and the claim templates CT-7/8/9, as depicted in Figures 5.16, 5.17, and 5.18. For the sake of readability, the requirements are presented here in HTML format. However, the XML format is employed for subsequent assertion generation.

For SR1_1_1, the scenario in which the robot runs out of power is modelled by the condition *c==0* representing a battery capacity of zero. Additionally, the robot position not being at the charging position, represented by Office 0, is denoted by the condition $p==x$, where x can take any value from 1 to 8 excluding Office 0. Therefore, the placeholder *path_formula* is a conjunction of "$p==x$" and "*c==0*" using the $path_formula_operator$ as "Finally". The constants *batteryCapacity* and *chargeStep* are assigned values of *20* and *4* respectively.

SR1_1_2 requires that the robot should enter the battery charging station when its battery capacity is depleted. The robot's entry into the charging status is represented by the batterySTM state machine being in the batteryState state. Therefore, the *path_formula* is a conjunction of the battery capacity

When
signal constant_deliverMOD::rp_ref0::batteryCapacity set to 20
and signal constant_deliverMOD::rp_ref0::chargeStep set to 4
and signal multi_constant_x from set_{1 to 8}
is true, until
signal pathFormula_Finally deliverMOD::rp_ref0::p==x /\ deliverMOD::rp_ref0::c==0
becomes true,
signal prob_target_<0.5
shall also be true,

Derived Requirement:	no
Safety Related:	no
Security Related:	no
Verify by:	Other

Backwards Traceability: SR1_1_1

Forwards Traceability: VR1_1_1_1

Figure 5.16: SR1_1_1 exported from Kapture.

When
signal constant_deliverMOD::rp_ref0::batteryCapacity set to 20
and signal constant_deliverMOD::rp_ref0::chargeStep set to 4
and signal reward_event_deliverMOD::ctrl_ref0::stm_ref0::move.in
and signal reward_value_true:1
is true, until
signal pathFormula_Reachable (deliverMOD::rp_ref0::c==0 /\
deliverMOD::ctrl_ref0::stm_ref1 is in deliverMOD::ctrl_ref0::stm_ref1:: batteryState)
becomes true,
signal reward_target_>8
shall also be true,

Derived Requirement:	no
Safety Related:	yes
Security Related:	no
Verify by:	Other

Backwards Traceability: SR1_1_2

Forwards Traceability: VR1_1_1_2

Figure 5.17: SR1_1_2 exported from Kapture.

condition "*c==0*" and the robot's entry into the charging state "*stm_ref1 is in batteryState*". For a reward-type requirement, the reward event and its corresponding value must be defined. In this case, the reward event is the robot's movement which is defined using synchronisation over the move event from the movingSTM machine referenced by stm_ref0. The reward value is assigned as 1 whenever a movement occurs.

SR1_1_3 requires that the robot must not get stuck. This requirement is represented by the mov-

When
signal constant_deliverMOD::rp_ref0::batteryCapacity set to 20
and signal constant_deliverMOD::rp_ref0::chargeStep set to 4
is true, until

constant term_not Forall
and signal pathFormula_Finally deliverMOD::ctrl_ref0::stm_ref0 is in deliverMOD::ctrl_ref0::stm_ref0::Stuck
becomes true,
signal NIL
shall also be true,

Derived Requirement:	no
Safety Related:	yes
Security Related:	no
Verify by:	Other

Backwards Traceability: SR1_1_3

Forwards Traceability: VR1_1_1_3

Figure 5.18: SR1_1_3 exported from Kapture.

ingSTM state machine not being in the Stuck state. Therefore, the placeholder *path_formula* in CT-9 is "*stm_ref0 is in Stuck*" with a temporal operator of "*not Forall*".

Assertion generation and evidence generation

With the requirements restructured in XML format, the corresponding assertion templates AT-9, AT-10 and AT-11 are used to generate RoboChart DSL assertions as shown in Fig. 5.19, 5.20 and 5.21. Within RoboTool, the RoboChart models are automatically converted into PRISM models. Then, by calling the PRISM model checker, the assertions will first be transformed into PRISM assertions and then verified. The verification results are then presented by RoboTool in the HTML format. An example of the verification result for SR1_1_1 is shown in Figure 5.22.

```
1 prob property P_SR1_1_1:
2     Prob<0.5 of [Finally deliverMOD::rp_ref0::p==x /\
  deliverMOD::rp_ref0::c==0]
3 with constant
4 deliverMOD::rp_ref0::batteryCapacity set to 20,
5 deliverMOD::rp_ref0::chargeStep set to 4,
6 x from set  {1 to 8}
7 const x : core::nat
```

Figure 5.19: Probability assertion for SR1_1_1.

AC module update with new evidence models

The evidence models are created using the pattern instantiation method based on the verification results. Using the traceability information provided in the XML requirements and the HTML verification reports, the corresponding AC claim to be supported is identified, and the evidence model

```
rewards reward_SR1_1_2 =
 [deliverMOD::ctrl_ref0::stm_ref0::move.in] true:1;
endrewards

prob property R_SR1_1_2:
Reward {reward_SR1_1_2} >8 of
[Reachable (deliverMOD::rp_ref0::c==0 /\
 deliverMOD::ctrl_ref0::stm_ref1 is in
 deliverMOD::ctrl_ref0::stm_ref1:: batteryState)]

with constant
deliverMOD::rp_ref0::batteryCapacity set to 20,
deliverMOD::rp_ref0::chargeStep set to 4
```

Figure 5.20: Reward assertion for SR1_1_2.

```
prob property T_SR1_1_3:
not Forall [Finally deliverMOD::ctrl_ref0::stm_ref0 is in
 deliverMOD::ctrl_ref0::stm_ref0::Stuck]

with constant
deliverMOD::rp_ref0::batteryCapacity set to 20,
deliverMOD::rp_ref0::chargeStep set to 4
```

Figure 5.21: Temporal assertion for SR1_1_3.

Results of probabilistic analysis of assertions in SR1_1_1.assertions using PRISM

Assertion: P_SR1_1_1

Assertion	Const	Const	Const	states:	transitions:	result:	checkTime:
P_SR1_1_1	x=1	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	43.522 seconds
P_SR1_1_1	x=2	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	42.446 seconds
P_SR1_1_1	x=3	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	40.292 seconds
P_SR1_1_1	x=4	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	41.461 seconds
P_SR1_1_1	x=5	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	40.97 seconds
P_SR1_1_1	x=6	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	43.843 seconds
P_SR1_1_1	x=7	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	42.66 seconds
P_SR1_1_1	x=8	deliverMOD::rp_ref0::chargeStep=4	batteryCapacity=20	643972	1405869	true	39.676 seconds

Figure 5.22: Model checking result for SR1_1_1.

is integrated into the AC module. In the case of SR1_1_1, the evidence model generated from the verification result shown in Figure 5.22 is linked to the claim for VR1_1_1_1. The complete AC depicted in Figure 5.10 are updated by substituting the “Solution” elements with the three new evidence models, as shown in Figure 5.23.

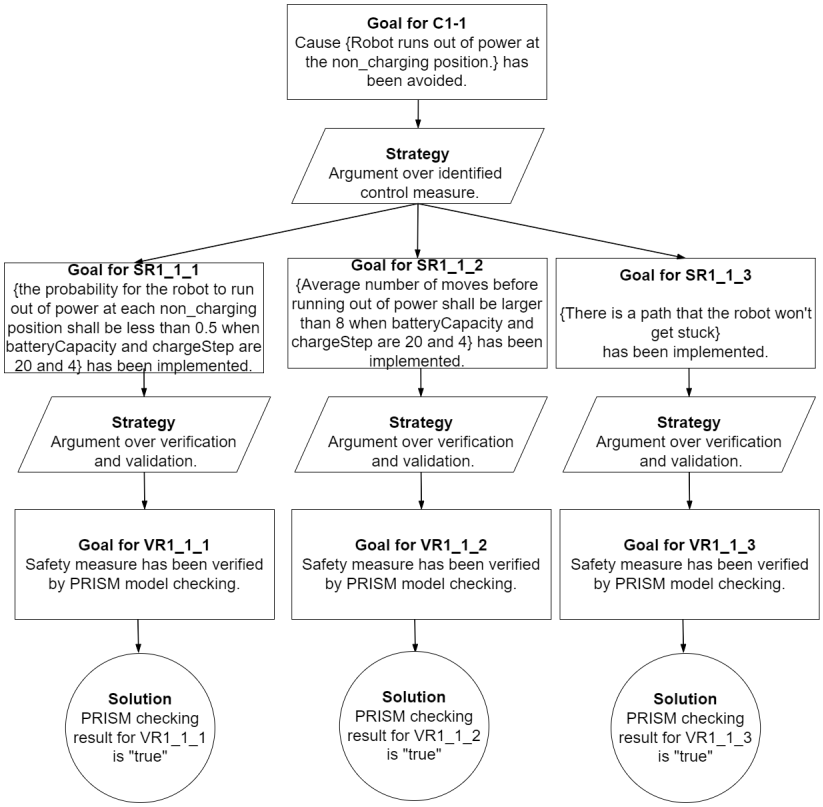


Figure 5.23: AC module for mail delivery robot updated with formal evidence.

5.5.3 Case study: High Voltage Controller

A case study of the High Voltage Controller (HVC) [85] is conducted to demonstrate the application of the proposed method for generating evidence models using FDR model checking.

System description

The system examined in this case study is an industrial painting robot [85]. The robot utilizes high voltage to charge the painting particles, which are then attracted to the object being painted. While this method offers efficiency and quality assurance, it also presents safety concerns. The high voltage has the potential to generate discharge sparks, which can serve as an ignition source. Given the presence of painting particles in the surrounding atmosphere, a fire incident could lead to significant damage. To ensure safe operation, a dedicated control unit called the "High Voltage Controller" (HVC) has been developed.

A simplified block diagram of the painting robot is illustrated in Figure 5.24. The *24V power signal* is the external power supply for the HVC. *HV.setPoint* represents the desired voltage level. Using these inputs, the HVC calculates the Pulse Width Modulation (PWM) value, denoted as PWM_Output, which subsequently serves as the input for the high voltage regulator. *PWM_Output* is a percentage ranging from 0 to 100%, corresponding to an analogue voltage signal from 0 to 10, as amplified by the transformer. Within the Cockcroft-Walton (CW) generator, the voltage is rectified and further amplified through multiple voltage doubling circuits. *HV.Actual* and *IM* are real-time measurements of the robot's voltage and current, respectively. These two values are fed back to the HVC for the

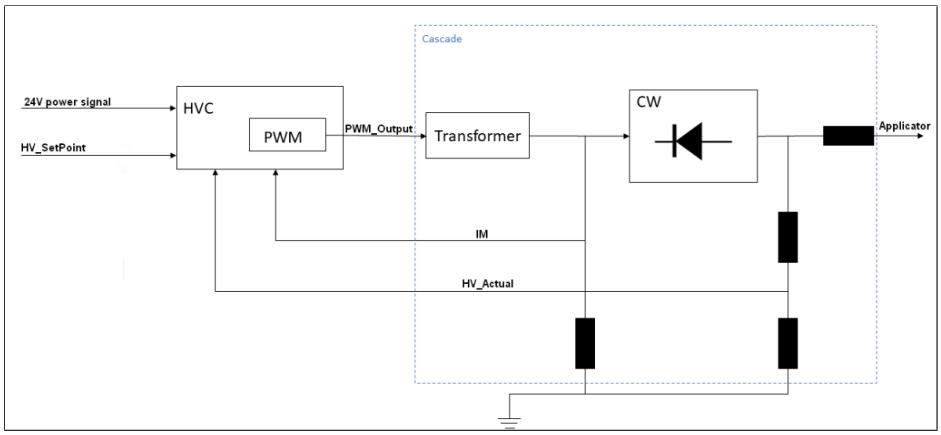


Figure 5.24: Block diagram of one part of the paint robot, containing the HVC [85].

calculation of the *PWM_Output*.

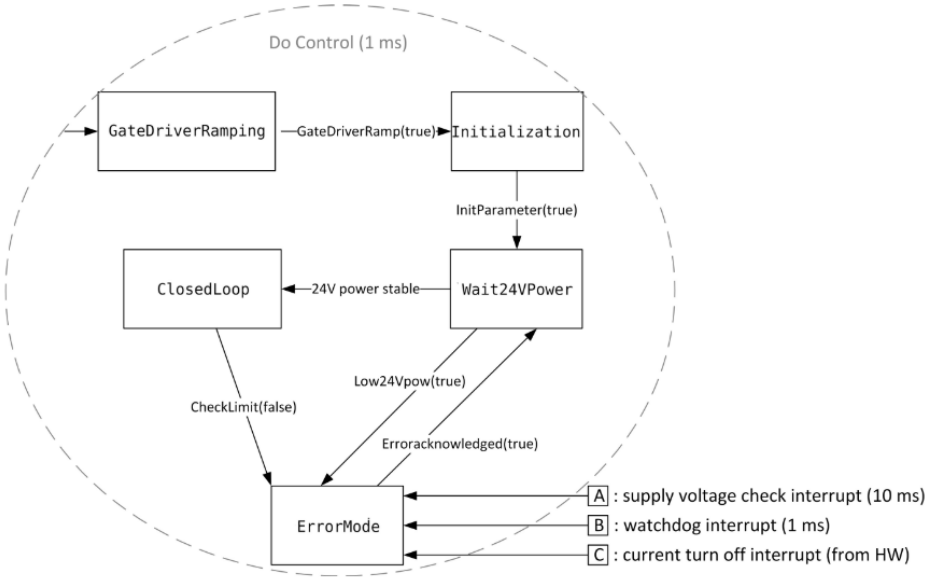


Figure 5.25: Finite state diagram of the HVC controller [85].

The behaviour of the HVC controller is abstracted as five distinct operation modes, as depicted in Figure 5.25. Upon activation, the HVC initially enters the *GateDriverRamping* state. During this phase, the PWM duty cycle is gradually increased to augment both current and voltage levels. Following this, the HVC transitions into the *Initialization* state, where parameters are configured, including voltage limits. Subsequently, the HVC proceeds to the *Wait24VPower* state, which ensures the stability of the external 24V power supply. Upon reaching this stage, the robot enters the normal operation mode known as *ClosedLoop*, wherein the HVC regulates the voltage output based on the setpoint. In case that the voltage exceeds the predefined upper or lower bounds, the HVC transitions to the *ErrorMode*. To revert from the *ErrorMode*, manual acknowledgement from the operator is required, after which the operation reverts to the *Wait24VPower* state. It is worth noting that time-related properties are not considered within the system. The corresponding RoboChart state machine can be observed in Figure 5.26 and 5.27.

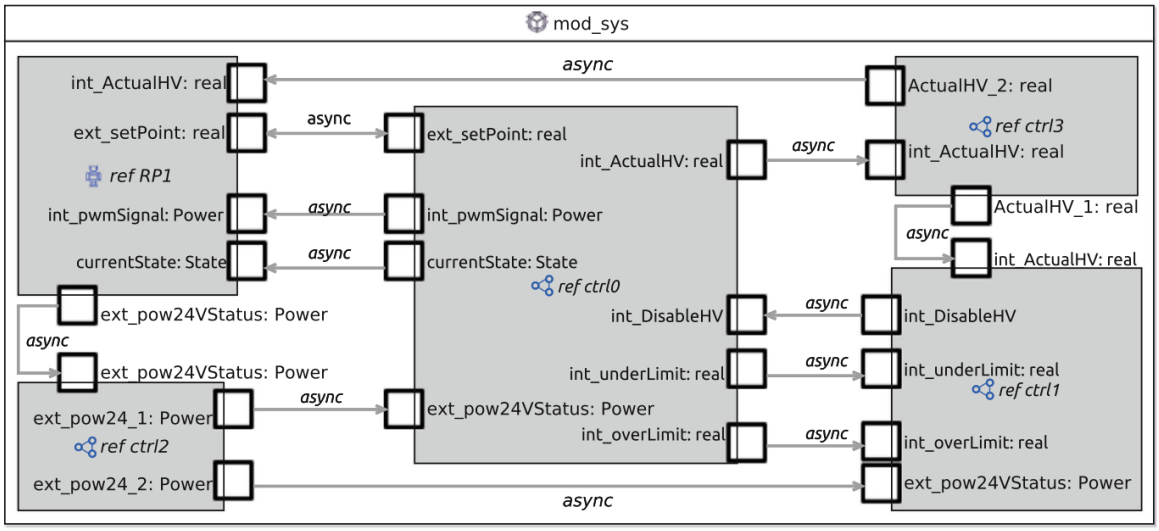


Figure 5.26: HVC RoboChart module [85].

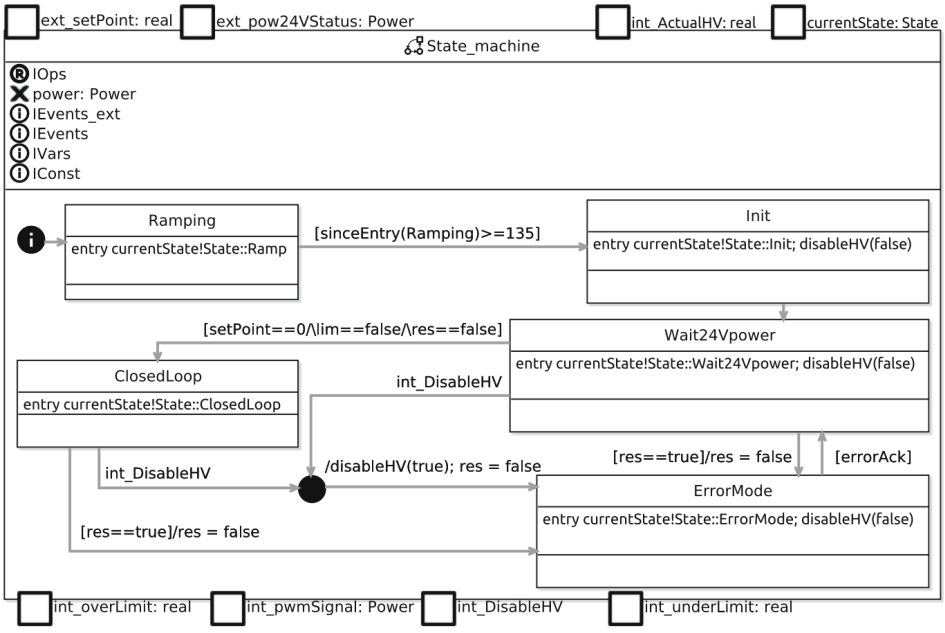


Figure 5.27: HVC RoboChart state machine [85].

The HVC is represented by the RoboChart module mod_sys, as shown in Figure 5.26. RP1 is the robotic platform representing the physical system. It communicates with the system software through events. The RoboChart controller ctrl0 captures the actual behaviour of the HVC software. It recasts the operational characteristics illustrated in Figure 5.25. ctrl0 contains a state machine State_machine, as shown in Figure 5.27. ctrl1 functions as a watchdog. ctrl2 and ctrl3 are used to relay the events ext_pow24VStatus and int_ActualHV between the platform and other controllers, respectively.

The behavior of the HVC software is modelled in the RoboChart state machine, presented in Figure 5.27. In each state, an entry action is specified to indicate the current state of the machine. This

is accomplished by utilizing the currentState event, which is specifically designed and employed for properties that are relevant only within particular states.

The primary hazard associated with the robot is the occurrence of fire in the painting cell resulting from sparks discharge caused by the high voltage output. To address this hazard, a hazard analysis is conducted specifically focused on the HVC unit. The EMF model of the hazard analysis results is displayed in Figure 5.28. The analysis identifies three causes of the high voltage output, which are as follows:

- HVC does not respond to *HV.setPoint* changes and has a constant voltage output,
- *HV.setPoint* is not disabled and HVC continues to set out high voltage when 24V power fails,
- *PWM_Output* is not disabled and HVC continues to set out high voltage when 24V power is off.

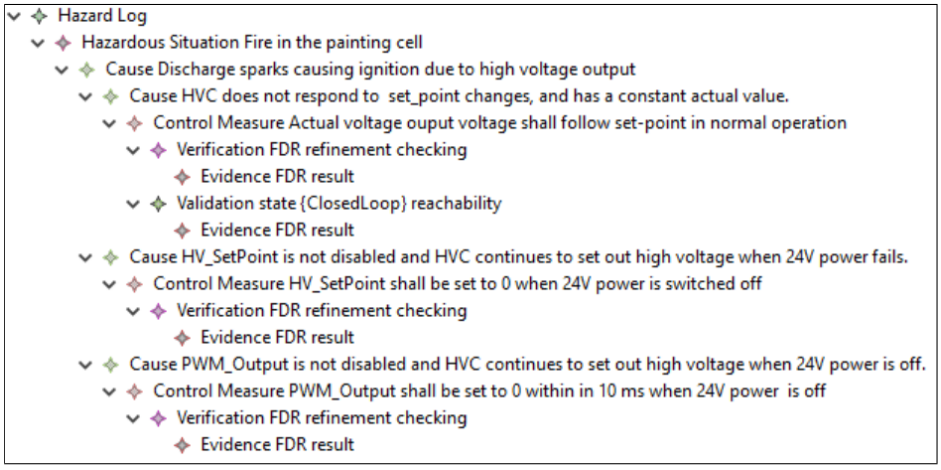


Figure 5.28: HVC hazard analysis EMF models.

Preliminary AC generation

A preliminary AC structure (Figure 5.29) is generated by instantiating it with the results of the hazard analysis.

There are three claims, namely *Goal for SR1_1_1*, *Goal for SR1_2_1*, and *Goal for SR1_3_1*, that require support from FDR model checking results. Therefore, four verification and validation activities will be conducted in FDR, as outlined below:

1. SR1_1_1 verification: Actual voltage shall follow set-point in normal operation.
2. SR1_1_2 verification: HV_SetPoint shall be set to 0 when 24V power is switched off.
3. SR1_1_3 verification: PWM_Output shall be set to 0 when 24V power is off.
4. SR1_1_1 validation: The state *ClosedLoop* is reachable in the state machine.

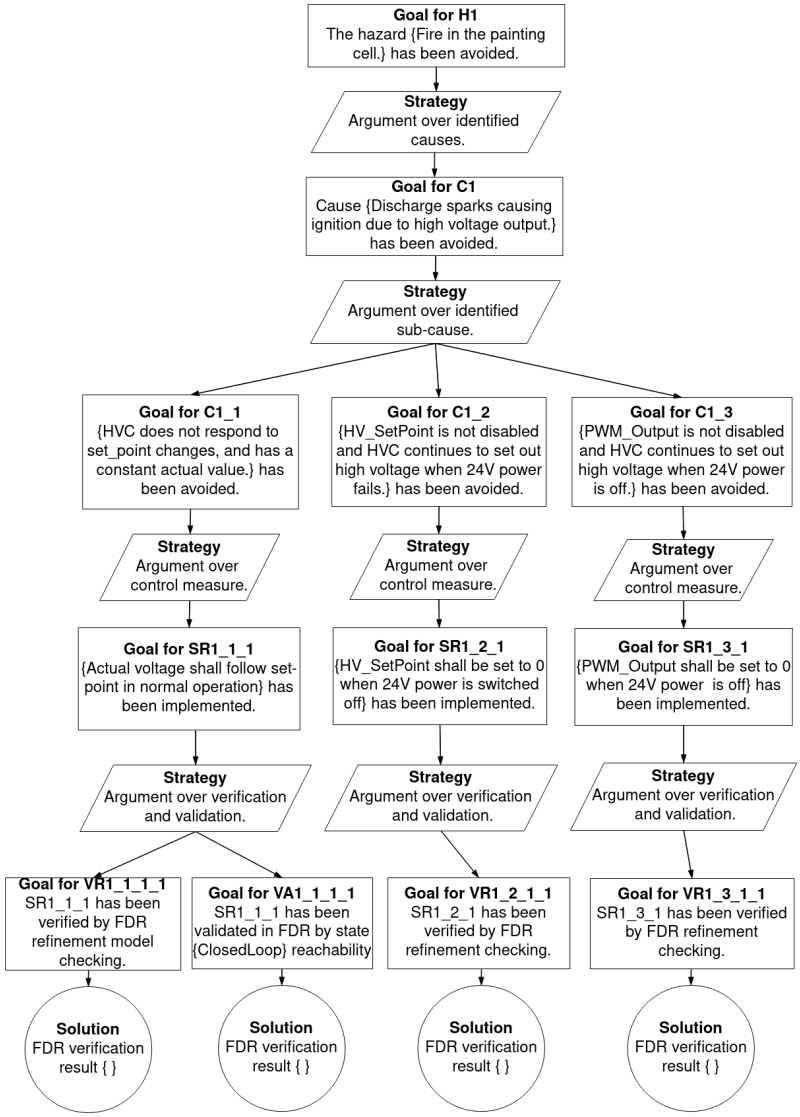


Figure 5.29: Preliminary AC of HVC.

Claim structuring

The requirements referenced in the claims are all untimed properties. To structure the requirements in Kapture, SR1_1_1, SR1_2_1 and SR1_3_1 verification use the template CT-1, SR1_1_1 validation requirement uses template CT-3. The requirements are structured and presented in Figures 5.30, 5.31, 5.37, and 5.32, respectively. In order to enhance readability, the requirements are displayed here in HTML format. However, for the purpose of generating assertions, the XML format is utilized.

SR1_1_1 requires the actual voltage to follow the set-point during normal operation, i.e., in the ClosedLoop state. To address this requirement, the placeholder *guard_event* in CT-1 incorporates two RoboChart events: currentState with a value of State_ClosedLoop, indicating that the HVC is in the normal operation mode, and event ext_setPoint to receive an input of any possible value for the setPoint. As for the placeholder *required_event* in CT-1, it is instantiated with the event int_ActualHV, which should output the same value as the setPoint. Due to the asynchronous nature of

the int_ActualHV output, this event is prefixed with *buffered* to facilitate the assertion generation in the next step.

When
signal mod_sys::currentState.out.State_ClosedLoop
and signal mod_sys::ext_setPoint.in?x__
is true
signal buffered_mod_sys::int_ActualHV.out!x__
shall also be true.

Derived Requirement:	no
Safety Related:	yes
Security Related:	no
Verify by:	Other

Backwards Traceability: SR1_1_1

Forwards Traceability: VR1_1_1_1

Figure 5.30: HVC Claim for SR1_1_1.

SR1_2_1 requires that HV_SetPoint should be set to 0 when the external 24V power is switched off. This requirement is considered a global property and should hold true for any state. To address this, the placeholder *guard_event* in CT-1 is instantiated with one RoboChart event ext_pow24Vstatus with an input value of Power_Off. The placeholder *required_event* in CT-1 is instantiated with the event ext_setPoint, which should output the value *0*.

When
signal mod_sys::ext_pow24VStatus.in.Power_Off
is true
signal mod_sys::ext_setPoint.out.0
shall also be true.

Derived Requirement:	no
Safety Related:	yes
Security Related:	no
Verify by:	Other

Backwards Traceability: SR1_2_1

Forwards Traceability: VR1_2_1_1

Figure 5.31: HVC Claim for SR1_2_1.

SR1_3_1 is similar to SR1_2_1 and is omitted here.

The validation of SR1_1_1 involves checking the reachability of the ClosedLoop state in the state machine. To accomplish this, the placeholder *function* in CT-3 is instantiated with the predefined function component *reachable_untimed*. The placeholder *scope* is used to confine the reachability analysis to the state machine.

Assertion generation and evidence generation

With the restructured claim statements exported from Kapture in XML files, the corresponding as-

The property
reachable_untimed(signal State_machine :: ClosedLoop, signal State_machine)
shall always be true.

Derived Requirement:	no
Safety Related:	no
Security Related:	no
Verify by:	Other

Backwards Traceability: SR1_1_1

Forwards Traceability: VA1_1_1_1

Figure 5.32: HVC Claim for SR1_1_1 validation.

sertion templates used to generate RoboChart DSL assertions are AT-1, AT-3, and AT-5. These assertions are depicted in Figure 5.33, Figure 5.34, Figure 5.38, and Figure 5.35, respectively.

```
1  untimed csp SR1_1_1 csp-begin
2  channel buffered_int_ActualHV : core_real
3  channel read : core_real
4  BufferedOutput = (Buffer [| {|read|} |] Output) \{|
   read|}
5  Output = read?x__ -> mod_sys::int_ActualHV.out!x__
   -> Output
6  Buffer = buffered_int_ActualHV?x__ -> (read!x__ ->
   Buffer [] Buffer)
7
8  SR1_1_1 = (Behaviour [| {| buffered_int_ActualHV |}
   |] BufferedOutput) \ {|buffered_int_ActualHV|}
9
10 Behaviour = CHAOS(Events)
11         [| {|
   mod_sys::currentState.out.State_ClosedLoop|} |>
12         (ClosedLoop_ctrl /\  Behaviour)
13 ClosedLoop_ctrl = mod_sys::ext_setPoint.in?x__ ->
   buffered_int_ActualHV!x__ -> ClosedLoop_ctrl
14 csp-end
15
16 untimed csp SR1_1_1_implement associated to mod_sys
   csp-begin
17 SR1_1_1_implement = mod_sys::O__(0) |\ {|
   mod_sys::currentState.out.State_ClosedLoop,mod_sys:
   :int_ActualHV.out,mod_sys::ext_setPoint.in?x__|}
18 csp-end
19
20 untimed assertion A_SR1_1_1 : SR1_1_1_implement
   refines SR1 1 1  in the traces model
```

Figure 5.33: HVC assertion for SR1_1_1 verification.

SR1_1_1 involves communication between the platform RP1 and the state machine in the controller ctrl0 via events ext_setPoint and int_ActualHV. Hence, a buffer needs to be created in the CSPM module of the assertion. Additionally, the property is constrained within the ClosedLoop state, making it a local property. Thus, the assertion for SR1_1_1 is generated using AT-3, as shown in Figure 5.33.

The assertion for SR1_2_1, as shown in Figure 5.34 uses AT-1. The CSPM module for the specification states that initially, any event is possible (CHAOS). Then, if the event *mod_sys::ext_pow24VStatus.in.Power_Off* occurs, it can continuously happen until *mod_sys::ext_setPoint.out.0* occurs, and then

```
untimed csp SR1_2_1 csp-begin
SR1_3_1 = CHAOS(Events) [| {|
mod_sys::ext_pow24VStatus.in.Power_Off|} |> (RUN
({|mod_sys::ext_pow24VStatus.in.Power_Off|}) /\
mod_sys::ext_setPoint.out.0 -> SR1_2_1 )
csp-end

untimed csp SR1_2_1_implement associated to
mod_sys csp-begin
SR1_3_1_implement = mod_sys::O__(0) |\ {|
mod_sys::ext_pow24VStatus.in.Power_Off,mod_sys::
ext_setPoint.out|}
csp-end

untimed assertion A_SR1_2_1 : SR1_2_1_implement
refines SR1_2_1  in the traces model
```

Figure 5.34: HVC assertion for SR1_2_1 verification.

it recurses to behave as *CHAOS*. The CSPM module for the implementation is the tailored RoboChart models in CSP semantics that only keep the behaviour of the two events that are relevant this specification: *mod_sys::ext_setPoint.out.0* and *mod_sys::ext_setPoint.out*. The structure of the assertion for SR1_3_1 is similar to that of the assertion for SR1_2_1 and is omitted here.

```
timed assertion A_SR1_1_1_VA : State_machine::ClosedLoop
is reachable in State_machine
```

Figure 5.35: HVC assertion for SR1_1_1 validation.

The assertion for the validation of SR1_1_1 is shown in Figure 5.35.

Within RoboTool, the RoboChart models are automatically converted into CSP models for verification purposes. By calling the FDR model checker, the assertions are then first transformed into CSPM assertions and then verified. The RoboTool outputs the verification results in the HTML format same as for the PRISM model checking. The RoboTool outputs the verification results in the HTML format with the same pattern used for PRISM model checking. The results are omitted here.

AC module update with new evidence models

The evidence models are created from the verification results using the pattern instantiation method. Then, according to the traceability information present in the XML requirements and the HTML verification reports, the corresponding AC claim to be supported is identified, and the evidence model is integrated into the AC module. In the case of SR1_1_1, the evidence model generated from the verification result shown in Figure 5.22 is linked to the claim for 'VR1_1_1_1'. To update the AC depicted in Figure 5.10, the "Solution" elements are replaced with with three new evidence models, as shown in Figure 5.36.

Timed version of HVC

A variation of HVC models has been developed to incorporate timed properties of the system. The RoboChart models can be accessed online[7]. The hazard analysis process for the HVC system with timed properties has a similar result as the one for the untimed version depicted in Figure 5.28 but with extra time properties, and thus, it has been omitted to avoid redundancy. The procedures for

[7] https://robostar.cs.york.ac.uk/case_studies/hvc/index.html

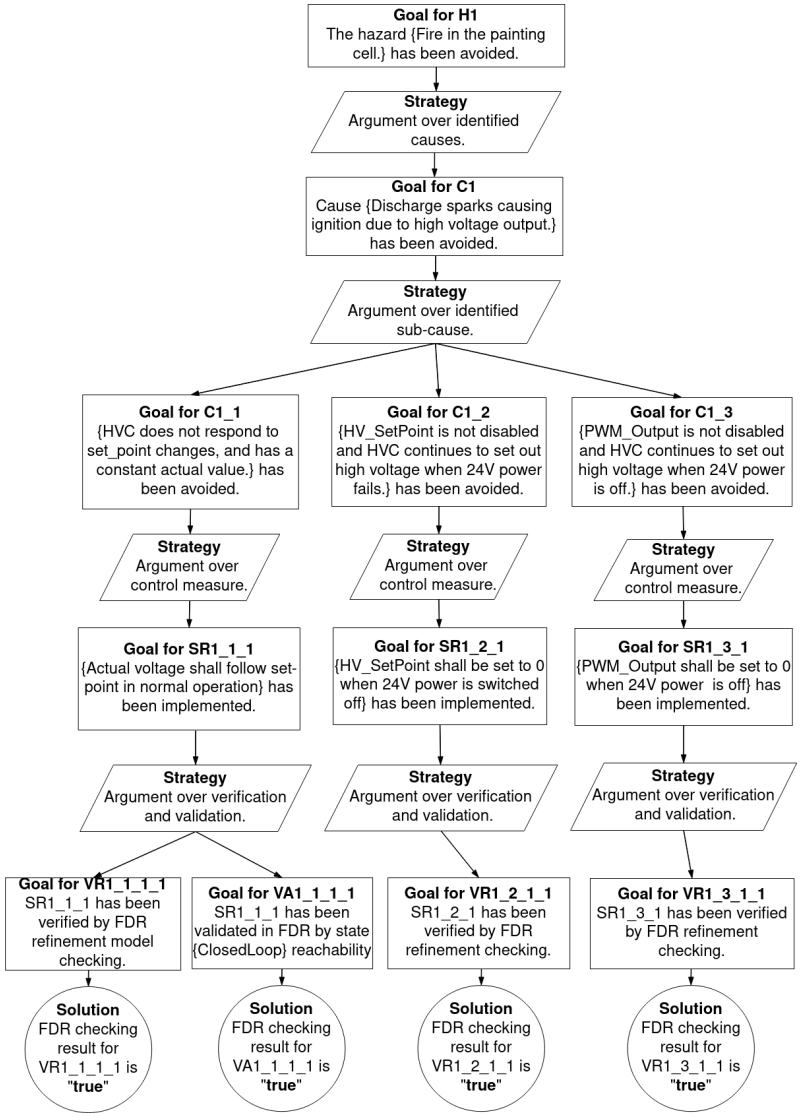
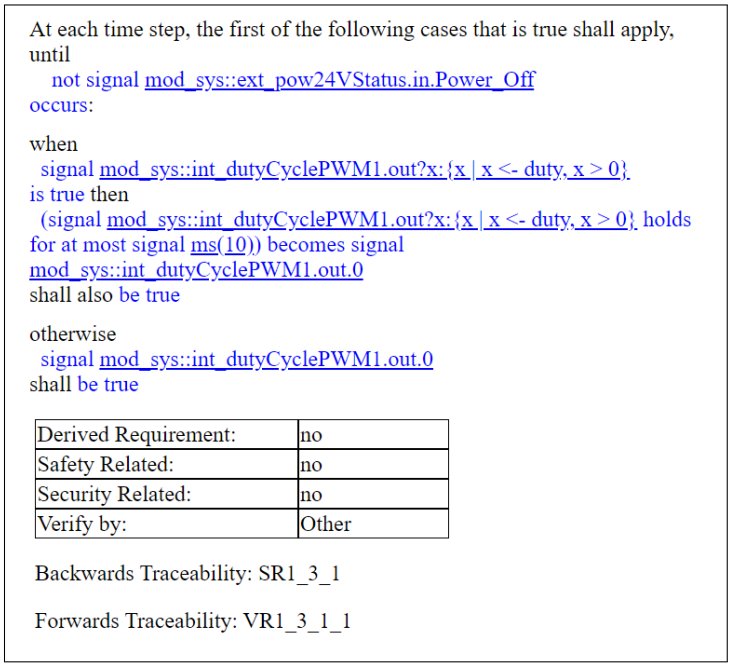


Figure 5.36: Updated AC of HVC using FDR verification results

generating the evidence from FDR verification results remain the same as for the untimed properties and are therefore omitted as well. One example of assertion generation for SR1_1_3 with the inclusion of time properties is provided below:

SR1_1_3_timed: PWM_Output shall be set to 0 within 10 ms when 24V power is off.

The requirement is structured in Kapture using template CT-2 to accommodate timed properties. SR1_1_3_timed can be restated as follows: it requires that the observation of the input ext_pow24VStatus with value Power::Off is followed by the output int_dutyCyclePWM1 with value 0 within a timeframe of 10 ms. In CT-2, there are three placeholders: *trigger_event*, *guard_event* and *target_event*. The *trigger_event* is the event of the 24V power being switched off, which is modelled using RoboChart event *mod_sys::ext_pow24VStatus.in.Power_Off*. The *target_event* is the event of PWM_Output being equal to zero, which is modelled by the event *mod_sys::int_dutyCyclePWM1.out.0*. The *guard_event* is that the output *int_dutyCyclePWM1* has been set to a value other than zero. The rewritten require-


At each time step, the first of the following cases that is true shall apply, until
not signal mod_sys::ext_pow24VStatus.in.Power_Off
occurs:

when
signal mod_sys::int_dutyCyclePWM1.out?x:{x | x <- duty, x > 0}
is true then
(signal mod_sys::int_dutyCyclePWM1.out?x:{x | x <- duty, x > 0} holds for at most signal ms(10)) becomes signal mod_sys::int_dutyCyclePWM1.out.0
shall also be true

otherwise
signal mod_sys::int_dutyCyclePWM1.out.0
shall be true

Derived Requirement:	no
Safety Related:	no
Security Related:	no
Verify by:	Other

Backwards Traceability: SR1_3_1

Forwards Traceability: VR1_3_1_1

Figure 5.37: SR1_3_1_timed in Kapture template.

ment is shown in Figure 5.37.

The assertion for SR1_3_1_timed is generated using the assertion template AT-4 and is shown in Figure 5.38.

5.5.4 Evaluation

To evaluate the effectiveness and generality of the formal evidence generation method through model checking, the method was applied to generate the AC modules for two case studies: the mail robot case study in Section 5.5.2 and the HVC case study in Section 5.5.3. The evaluation in this section aims to address the following research questions.

RQ1 Completeness: Are the ACs constructed using the method complete, i.e., does the AC structure comply with the AC template, and does the AC module conclude with evidence that refers to a formal verification result?

RQ2 Generality: Does the method support the development of ACs for different application domains? Can it be applied to applications with various system development techniques?

RQ3 Efficiency: Does the implementation produce ACs with acceptable overheads in terms of computational resources and time?

RQ4 Coverage: Does the classification of the requirements referred to in the claims sufficiently cover the different requirements of safety-critical RAS systems?

One assumption has been made for the method proposed in this chapter: the modelling language

```
timed csp SR1_3_1 csp-begin
Timed(OneStep) {
SR1_1_1= timed_priority(Process_1)

Process_1=
mod_sys::int_dutyCyclePWM1.out.0 -> Process_1
[]
mod_sys::int_dutyCyclePWM1.out?x:{x | x <- duty, x > 0}
-> Process_2
[]
mod_sys::ext_pow24VStatus.in.Power_Off -> Process_1

Process_2 =
mod_sys::int_dutyCyclePWM1.out.0 -> Process_1
[]
mod_sys::int_dutyCyclePWM1.out?x:{x | x <- duty, x > 0}
-> Process_2
[]
mod_sys::ext_pow24VStatus.in.Power_Off -> TRUN {|
mod_sys::ext_pow24VStatus.in.Power_Off,
mod_sys::int_dutyCyclePWM1.out|} /\ (WAIT(ms(10)));
USTOP [|{|mod_sys::int_dutyCyclePWM1.out.0|} |> SKIP;
Process_1
}
csp-end

timed csp SR1_3_1_implement associated to mod_sys csp-
begin
Timed(OneStep){
SR1_1_1_implement =
timed_priority(
mod_sys::O__(0, ms(10), 1)
|\ {| mod_sys::ext_pow24VStatus.in.Power_Off,
mod_sys::int_dutyCyclePWM1.out, tock|})
}
csp-end
```

Figure 5.38: Assertion for SR1_3_1_timed.

used in system development is supported with formal semantics to enable proper model checking of the models. Therefore, the formalization of the system models is outside the scope of our method and is not evaluated.

To address the completeness of the AC modules and answer **RQ1**, two aspects are considered. Firstly, traceability is established throughout the entire process, connecting the claim to be supported by formal evidence, the structured requirement referred in the claim, the assertion automatically generated from the requirement, and the verification result obtained from the model checking tool. This traceability ensures that formal evidence is generated for each claim to be supported by the model checking result and is accurately integrated into the AC modules. Secondly, during the process of evidence model instantiation from the model checking results, the AC pattern (Figure 4.6) introduced in Section 4.2.2 is strictly followed by implementing Algorithm 6. This guarantees the completeness of the AC module's structure after integrating the evidence model. The mail robot and HVC case studies provide empirical evidence that the proposed method is capable of generating complete ACs.

In the context of **RQ2**, the proposed method aims to facilitate the generation of formal evidence by verifying the satisfaction of claims through the use of model checkers on system models. Therefore, the applicability of the method primarily depends on the system modeling languages and model checking tools employed during the system development process, and is not strongly tied to the specific domain of RAS applications.

The two case studies of the mail delivery robot and the high voltage controller are applications of different domains. However, the domains themselves do not impact the process of evidence model generation. The utilization of RoboChart models in the method implementation and the case studies

is advantageous due to their formal semantics of CSP and PRISM. Consequently, other modeling languages supported by CSP and PRISM semantics can replace RoboChart models.

Regarding the model checking tools utilized, the current implementation of the method employs FDR and PRISM, and the assertion templates are specifically designed using CSP and PRISM operators. Nevertheless, the core process of the method remains applicable to other model checking tools and their associated formal semantics. By creating new assertion templates using the appropriate formal semantics, the method can be extended to incorporate other model checking tools.

Any changes made to the system models can trigger the update of the verification result, and further the update of the ACs.

To address **RQ3**, the efficiency of formal evidence generation is compared between our method and a manual process. The comparison focuses on the step of evidence model generation from model checking results through pattern instantiation, while excluding the tasks of assertion generation and model checking from the efficiency analysis. The purpose of assertion generation is to reduce the need for formal expertise in the process of claim formalization, rather than aiming for efficiency improvement. Additionally, the time efficiency of model checking using FDR and PRISM shows no difference between our method and a manual process.

The two case studies are conducted on a 1.6GHz Core i5 computer with 8GB of RAM. The CPU time required to generate the formal assertions from the structured requirements, and the evidence models from the verification results in the two case studies is below 1 second respectively, indicating the efficiency of the proposed method compared to a manual process.

To address **RQ4**, the requirements referenced in the claims are classified into five distinct groups, as illustrated in Table 5.1. These groups encompass untimed functional requirements, timed functional requirements, general requirements, probabilistic requirements, and temporal requirements. Due to the high-level nature of these groups, they can encompass all types of requirements that can be verified through model checking in RAS systems.

These five groups are further divided into nine sub-types. In terms of the sub-types, the level of coverage differs depending on the requirement templates designed for each specific type. Generally, the templates are designed to encompass all types of requirements, as users are allowed to provide flexible definitions for the placeholders, with the exception of timed functional requirements. The template of untimed function requirement (CT-1) employs a typical pattern for functional requirement, offering flexibility and expressiveness through the placeholders of *guard_event* and *required_event*. These two placeholders allow for the introduction of complex requirements. Reachability, deadlock freedom, divergence freedom, and termination requirements, on the other hand, are specific and concrete, making their templates (CT-3/4/5/6) directly applicable to various types of applications.

The templates for probabilistic requirements and temporal requirements provide a general format with the flexibility and expressiveness achieved through the placeholder of *path_formula*, enabling coverage of requirements with different levels of complexity.

The CT-2 template, designed for timed requirements, has a limitation in terms of requirement coverage due to the incorporation of the time property. This template is suitable for scenarios where a delay before the target event is permitted. However, in order to adequately address timed functional requirements of different types, it is necessary to develop additional templates.

Threats to validity

Construct validity threats may arise from the assumptions made regarding system development artifacts, specifically the safety requirement and design models. To address this potential threat, the safety requirements and the RoboChart models of the mail delivery robot and the HVC are sourced from previously validated case studies [55, 85]. This choice contributes to an increased level of confidence in their reliability and appropriateness for analysis.

Potential threats to external validity may emerge due to the restricted usage of RoboChart as the system modeling tool, as well as FDR and PRISM as the model checkers employed in the implementation and case studies. To address this concern, the fundamental concept of the proposed method can be applied to alternative system modeling languages that possess formal semantics, along with their corresponding model checking tools. By extending the scope beyond the current set of tools, the validity and generalizability of the findings can be enhanced.

Internal validity threats may be present due to the implementation of the method, particularly concerning the accuracy of generating assertions from the requirements. To address these threats, a strategy has been employed where formalized assertions from existing literature are utilized for comparison purposes. This ensures the correctness of the assertions generated using the proposed method. Additionally, to strengthen internal validity, extensive testing has been conducted on the implementation of the evidence model generation. This involves comparing the results of the code execution with the expected outcomes, thus ensuring the reliability and accuracy of the generated evidence models.

5.6 Summary

This chapter presented a method for generating evidence models from claim models through the application of model checking verification. The proposed method serves as a complement to the AC generation process discussed in Chapter 4, with a specific focus on deriving formal evidence models from claims. By employing this method, the manual generation of formal assertions is eliminated, and the necessity of formal expertise is reduced, thus bridging the existing gap. The process consists of four key steps:

1. Firstly, claim structuring is performed, resulting in an XML file that expresses the requirement using RoboChart components.
2. Secondly, assertion generation takes place by instantiating templates based on the structured requirement.
3. Thirdly, model checking is executed, producing a verification report in HTML format.
4. Finally, evidence model generation and integration are accomplished using the pattern instantiation method.

The process is automated by using MBE techniques and tool support, except for the step of claim structuring which requires user input.

The AC claims are categorized into five main groups (nine sub-types) to cover different claims that require support through model checking verification. Correspondingly, the requirement templates have been designed for each sub-type by customizing the basic templates of a third party requirement management tool. In terms of the assertion generation, the method takes the advantage of the RoboChart assertion language, which simplifies the process of defining formal assertions to a certain extent. To enable verification of both probabilistic and non-probabilistic properties, the two model checkers, FDR and PRISM, have been employed. Eleven assertion templates have been defined, using non-probabilistic RoboCert language and CSP_M blocks for FDR, while the probabilistic RoboCert language is employed for PRISM. The mapping is established between the elements of the claims templates and the corresponding assertion templates.

The generation of assertions is achieved through automatic instantiation of templates, utilizing the structured claims and following the prescribed mapping rules. Subsequently, the assertions are verified by invoking FDR/PRISM in RoboTool, and the verification results are presented in the form of an HTML report. These results are then transformed into AC evidence models, which are subsequently integrated into AC modules. The implementation of the proposed method employs RoboChart models. However, with suitable adjustments, the method can be generalized and applied to other modeling languages as well.

The common approach to applying model checking verification in the AC process is to reference the existing formal verification results within the AC evidence component [16–18]. Thus, the process of obtaining the model checking results to support the claims is usually not addressed. Furthermore, previous research work [16–18] typically focuses on generating formal evidence using formal design models. However, it is essential to note that RAS systems are commonly developed using MDE techniques and the system modelling languages, rather than formal semantics. In light of this practical consideration, a comprehensive solution is needed to effectively incorporate model checking into the AC process, encompassing systems developed through MDE techniques. Considering these two issues above, this chapter proposes an integrated method to directly derive formal evidence from the argument claim for RAS systems modelled using RoboChart modelling language. This integrated method can streamline the AC process with model checking application.

Our method draws inspiration from the work of Denney and Pai [20], which integrates claim formalization and formal verification into the AC generation process and the corresponding tool. Their work utilizes a static source-code analysis tool to verify the source code generated from system models. Similarly, our method encompasses claim formalization and verification as integral parts of the AC process. However, it differs in terms of the specific techniques and methods employed for implementation. In contrast to the aforementioned work, our method applies formal verification to the system models using model checking, rather than focusing on the source code. Furthermore, our method facilitates the requirement formalization by utilizing system models. This can eliminate the need for users to have an in-depth understanding of formal semantics. This feature greatly facilitates the practical application of formal verification in engineering practice.

Another advantage of our method is its ability to coordinate various formal verification methods. By employing both probabilistic and non-probabilistic model checkers, different types of requirements can be effectively addressed, ensuring the broad applicability of the method. This coordination of verification methods aligns with the method demonstrated by Prokhorova et al. [16], who exemplified the use of multiple formal verification methods, including model checking and theorem proving, to cover different requirement types. However, the automation of the process was not realized in the original work by Prokhorova et al. The extension of the method proposed in this chapter is further

explored and detailed in the subsequent chapter, where theorem proving techniques are incorporated into the methodology.

Chapter 6

AC Evidence Generation by Theorem Proving

The methods presented in this chapter are focused on addressing the third research question and making a contribution to the automatic generation of AC evidence through the verification of design models in the Isabelle/HOL theorem prover. The research paper "Automated Compositional Verification for Robotic State Machines using Isabelle/HOL" [95] has been published in the ICECCS 2023 conference to state the contribution of RoboChart model verification in Isabelle/HOL, with the exception of Section IV.

RoboChart is a graphical language for model-based engineering of robotic systems, in the style of UML and SysML. It contains notations for data structures, system architecture, and the behaviour of individual robotic controllers using state machines. Crucially, RoboChart has a formal semantics in the CSP process algebra, which provides a precise foundation for software engineering and formal verification using model checking. The application of the model checking as a verification method to automatically generate the AC evidence models has been explored in Chapter 5. However, due to the issue of state space explosion and constraints of real numbers, the application of model checking has its limitation that it does not scale well.

In this chapter, the theorem-proving platform Isabelle/HOL is explored. The contribution is a process to automatically generate the AC evidence by theorem proving using Isabelle/HOL. The key step is a compositional verification technique that uses Isabelle/HOL to model and verify the RoboChart state machines symbolically. The technique uses state invariants to capture AC claims over a very large or infinite state, similar to the B method, and the proving process is highly automated using Isabelle's sledgehammer tool. The verification results are manually converted into AC evidence models.

6.1 Overview and background

Model checking provides a high degree of automation in terms of verification, an unavoidable issue is the state space explosion introduced by large models [96]. On the other hand, theorem proving allows us to represent the state space symbolically, so it scales well without the issue of the state

space explosion. Though theorem proving has traditionally been a manual process, the integration of automated proof tools like SMT solvers has the potential to greatly improve its usability. For example, *sledgehammer* [97] is a tool that applies Automated Theorem provers (ATP) and Satisfiability Modulo Theories (SMT) solvers in Isabelle/HOL [98].

To support the AC generation process in a comprehensive way, the integrated formal verification is considered in this research, which combines different formal verification techniques and tools to verify the properties of a system. The integrated formal verification involves integrating various formal methods, such as model checking, theorem proving, to provide a comprehensive and rigorous analysis of a system's design or implementation. Therefore, as a complement to the method in Chapter 5, this chapter proposed a method to generate formal evidence models through theorem proving.

Similar to the context of the method in Chapter 5, assuming that an AC module is generated by pattern instantiation, some claims are to be supported by theorem proving results. Then, the method in this chapter provides an automated process of applying theorem-proving verification to RoboChart models whose results are used to support the AC claims.

The core contribution in this chapter is a compositional verification technique for generating AC evidence of RoboChart state machine models. The technique can support the seamless integration of theorem proving into the process of model-based AC generation. The work bridges the gap between the domain-specific models and formalization using a set of intermediary notations that balance usability by software engineers with suitable rigour provided by formal semantics.

The technical contributions in this chapter are (1) a pipeline for generating AC evidence for the RoboChart state machine by automated theorem proving; (2) the mapping rules from RoboChart to an abstract machine language with verification support of Isabelle/HOL; (3) formalization method for a set of claims. A running example is used throughout this chapter for better illustration.

6.1.1 Z Machine

This section introduces Z Machine, the core formal semantics underpinning RoboChart transformation.

The Z Machine language is an abstract machine model developed by RoboStar group at the University of York. It is similar to the Z [99] and B [100] methods, which is built as a conservative extension of Isabelle/HOL. A Z Machine consists of (1) a rich state space using types defined in HOL and including state invariants; (2) operations acting over the state space; and (3) the machine itself, which initialises the state and groups together the operations.

As with the B method [100], Z Machine is essentially an action system [101], where operations can be called repeatedly depending on their preconditions. Isabelle has a general theory document model, where a theory consists of a sequence of commands; therefore, the Z Machine models are stored in a theory file. The "Birthday Book" example taken from [99] is presented using Z Machine notations and given in Listing 6.1.

Listing 6.1: Birthday book example

```
theory BirthdayBook
imports "Z_Machines.Z_Machine"
begin
type_synonym name = string
type_synonym date = string

consts
  NAME :: "name set"
  DATE :: "date set"

zstore BirthdayBook =
  known :: "ℙ name"
  birthday :: "name ↦ date"
  where "dom(birthday) = known"
        "known ⊆ NAME"
        "ran(birthday) ⊆ DATE"

zoperation AddBirthday =
  params name ∈ NAME date ∈ DATE
  pre "name ∉ known"
  update "[known′ = known ∪ {name}
          , birthday′ = birthday ⊕ {name ↦ date} ]"

zoperation FindBirthday =
  params name ∈ NAME date ∈ DATE
  pre "name ∈ dom birthday"
  where "date = birthday(name)"

zoperation Remind =
  params today ∈ DATE cards ∈ "ℙ NAME"
  where "cards = {n ∈ known.birthday(n) = today}"

definition Init :: "BirthdayBook subset" where
  "Init = [known ⇝ {}, birthday ⇝ {↦}]"

zmachine BirthdayBookSys =
  over BirthdayBook
  init Init
  invariant BirthdayBook_inv
  operations AddBirthday FindBirthday Remind

lemma Init_inv: "Init establishes BirthdayBook_inv"
  by zpog_full

lemma AddBirthday_inv: "AddBirthday (n, d) preserves BirthdayBook_inv"
  by zpog_full

lemma FindBirthday_inv: "FindBirthday (n, d) preserves BirthdayBook_inv"
  by zpog_full

lemma Remind_inv: "Remind (n, d) preserves BirthdayBook_inv"
  by zpog_full

definition [z_defs]: "BirthdayBook_axioms = (NAME ≠ {} ∧ DATE ≠ {})"
lemma BBook_ddl_free: "BirthdayBook_axioms⟹deadlock_free BirthdayBookSys"
  by (deadlock_free invs: AddBirthday_inv FindBirthday_inv Remind_inv; auto)
end
```

The theory file of "Birthday Book" example starts with the type declaration. We used Isabelle keyword **type_synonym** to define two types name and date. Then, the set of all names and the set of all dates are defined using keyword **consts**. The state space is declared as BirthdayBook using keyword **zstore**. There are two state variables declared as known, the set of names with birthdays recorded, and birthday, a function, which, when applied to certain names, gives the birthdays associated with them. A set of structural invariants are defined in the **where** section of the state space to set the set known as the domain of the function birthday. This set of structure invariants identified in zstore is automatically named as BirthdayBook_inv.

There are three operations over the state space to describe the behaviour of the system. Each operation may consist of (1) a number of parameters (**params**), which model inputs and outputs; (2) a precondition (**pre**), which must be true for the operation to be executed; and (3) an **update**, which simultaneously assigns new values to state variables.

AddBirthday operation is to add a new birthday record with the precondition that the name to be added must not have been one of those known names to the system. FindBirthday operation is to find the birthday of a person known to the system. The state of the system does not change in this operation, therefore there is no **update** section. The operation takes a name as input and yields the corresponding birthday as output through section **params**. The precondition of this operation is that the input name is one of the names known to the system; if so, the output date is the value of the birthday function at argument name. Remind operation is used to find which people have birthdays on a given date. The operation has one input today, and one output cards, which is a set of name set. cards is to be equal to the set of all values n drawn from the set known such that the value of the birthday function at n is today.

An initial state Init of the system is given for the system operation to start with. The whole system is modelled as BirthdayBookSys using the keyword **zmachine**, and consists of the initialization state Init, the structural invariant BirthdayBook_inv, and all the operations.

The verification of Z Machines consists of identifying suitable state invariants to satisfy the critical requirements, and then showing (1) that the initialisation *establishes* the invariants, i.e. $\{true\}\, Init\, \{I\}$; and (2) that each operation *preserves* them, i.e. $\{I\}\, Op_i(x,y,z)\, \{I\}$. The zpog method is provided to generate a set of proof obligations for such a specification triple, via calculation of the operation's weakest precondition. The resulting proof obligations can then often be discharged automatically using the Isabelle tools *auto* and *sledgehammer*.

In the "Birthday Book" example, the structure invariants BirthdayBook_inv identified in zstore are proved using zpog method to show the initialisation *establishes* the invariants and then each operation *preserves* them. Finally, a property of deadlock freedom is proved using the method deadlock_free designed for Z Machine.

An abstract theory template of Z Machine models including the verification process is abstracted in Appendix D Listing D.1. This template will be used as the basis for implementing the RoboChart transformation.

6.1.2 Methodology

This section presents the methodology for generating AC evidence for RoboChart state machines through theorem proving. Figure 6.1 illustrates the seven activities involved in the method.

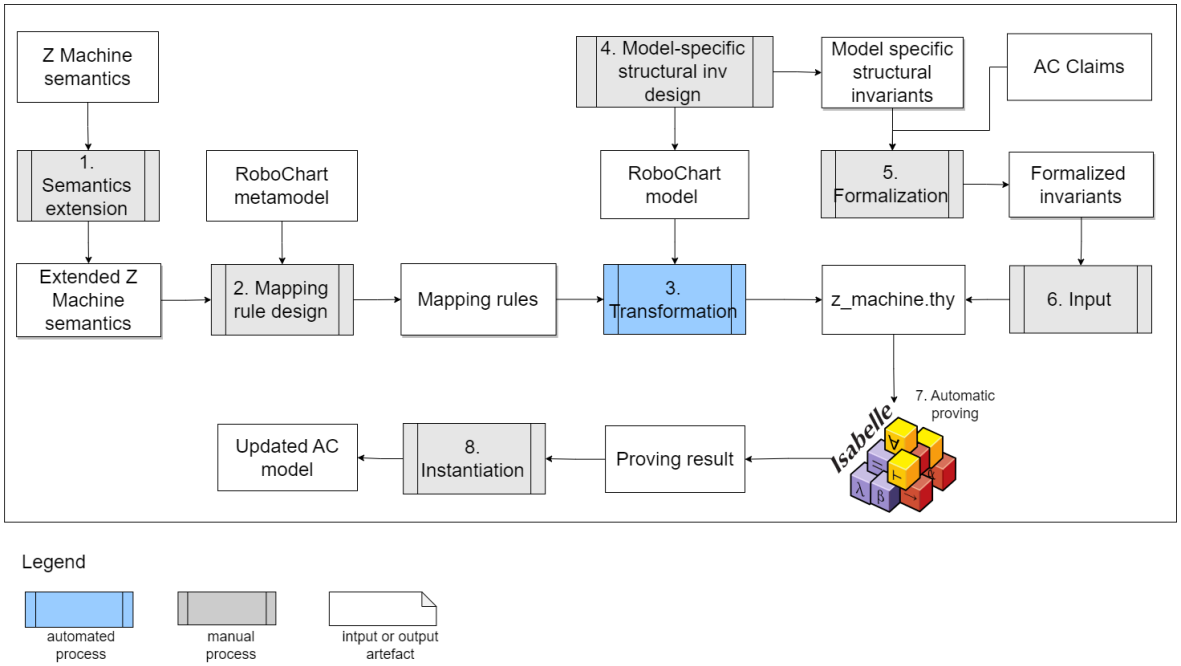


Figure 6.1: Pipeline for generating AC evidence for RoboChart state machine in Isabelle/HOL.

The first activity involves providing RoboChart with a formal semantics by extending the Z Machine's semantic domain. As described in Section 6.1.1, the Z Machine serves as the target notation for transforming RoboChart. However, additional features are required in the Z Machine's semantic domain to accurately represent the behavior of RoboChart, the CSP semantics of RoboChart models being as the reference. The details are discussed in Section 6.2.

The second activity involves defining the mapping rules from RoboChart elements to Z Machine notations, according to both the Z Machine semantic domain and the RoboChart metamodel. The details of these mapping rules can be found in Section 6.3.

The third activity is to convert the RoboChart state machine into a Z Machine by automatic model-to-text transformation. The implementation of the transformation is provided in Section 6.5.

The remaining activities of the methods address the evidence model generation through theorem proving, and are discussed in Sections 6.4. The fourth activity is the generation of structural invariants that are specific to the state machine under analysis and is discussed in Section 6.4.1. These invariants show the inner relationship of the state components of the Z-Machine, and are necessary for requirement verification in the next step. These model-specific properties should be formalized and provided by the user. The fifth activity formalizes the AC claims of interest as invariants as stated in Section 6.4.2. The sixth activity is to integrate the formalized invariants into the theory file. The seventh activity is to prove that the Z Machine notations satisfy the invariants by using the *auto* proof methods and the *sledgehammer* interface to call ATPs. This is discussed in Section 6.4.3. With the verification result obtained from Isabelle/HOL, the eighth activity is to update the AC evidence

models with the result of theorem proving. This step follows the same process of pattern instantiation in Chapter 4, therefore is not discussed in this chapter.

6.1.3 A running example

A running example of an autonomous Chemical detector robot [54, 102] is used to describe the method of this chapter. The robot performs a random walk with obstacle avoidance and analyses the air to detect dangerous gases. Once a chemical with its intensity above a threshold is detected, the robot drops a flag, reports the location of the gas, and stops the walk.

The top-level structure of a RoboChart model is a module containing one robotic platform (i.e., the abstracted physical robot to indicate the facilities available), and several controllers to specify software behaviours. The module ChemicalDetector is shown in Fig. 6.2 which contains one platform Vehicle and two controller references MainController and MicroController. The platform provides several services to the controllers including (1) reading sensor data through events: gas, obstacle, and odometer; (2) movement actuation through operations: move, randomWalk, and shortRandomWalk which are grouped in an interface Operations; and (3) dropping a flag when receiving event flag.

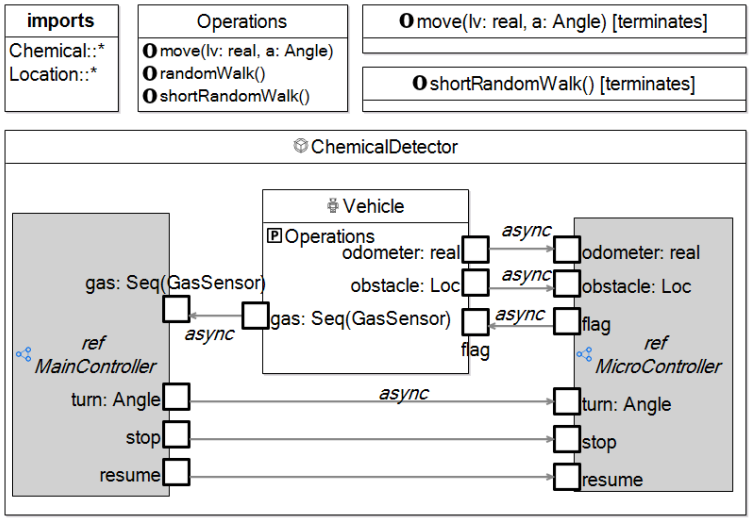


Figure 6.2: RoboChart module of Chemical detector [54].

The types are defined in the two packages and imported as shown in Fig. 6.2: Chemical and Location (the diagrams for the two packages are omitted). In the two packages, primitive types Chem and Intensity are declared for chemicals to be detected and the intensity of the chemicals, enumerations Status for the detection status, Angle for the turning directions of the robot, and Loc for the chemical locations. The packages declare a record GasSensor containing two fields (c of type Chem and i of type Intensity) and define six functions and an operation changeDireciton using a state machine. The behaviour of MicroController is specified using state machine Movement, and MainController using state machine GasAnalysis as shown in Fig. 6.3.

The machine GasAnalysis declares a constant thr as intensity threshold, and four variables including sts as detection status, gs as the list of sensor readings, ins as the gas intensity, and anl as turning direction. The machine has one initial junction, four normal states of Reading, Analysis, NoGas,

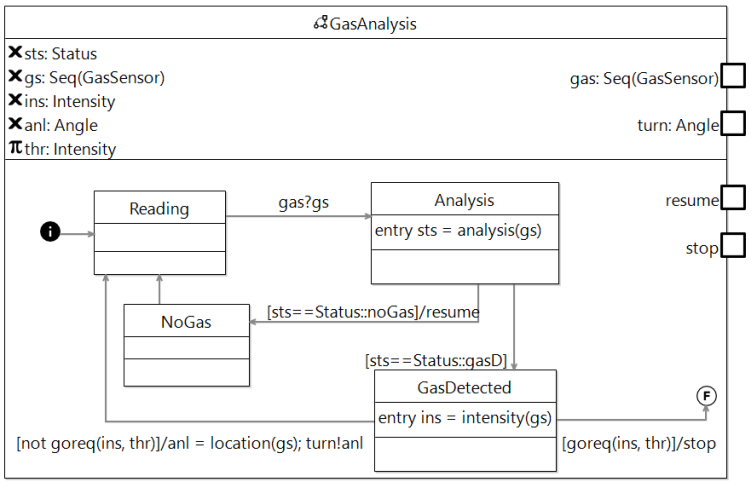


Figure 6.3: RoboChart state machine model of Gas Analysis [54].

GasDetected, and a final state. Each state may have an entry action such as sts = analysis(gs), a during action, and an exit action. A Transition is a directed connection from a source node to a target node. It can have a trigger event (e.g., event gas of input type), a guard condition (e.g., "not goreq(ins,thr)"), and an action that is executed during the transition (e.g., the assignment action "anl = location(gs)", and the action of event turn).

GasAnalysis models the gas detection behaviour of the robot. After the machine is initialized, it takes the sensor readings from event gas, then in state Analysis it analyses the gas status according to the readings. If there is no gas detected, a command of resume is sent to Movement to let the robot continue its random walk, and GasAnalysis reads the new reading. If there is gas detected, the machine enters GasDetected and calculates the gas intensity. If the intensity reaches the threshold, it is identified as dangerous gas detected, a stop signal is sent to the machine Movement, GasAnalysis terminates; otherwise, it is identified as no danger and a turn signal is sent to Movement to change the direction of the robot, and GasAnalysis enters Reading to take the new readings.

6.2 Semantic domain

To verify RoboChart models in Isabelle/HOL, I propose a modelling pattern for a RoboChart subset using Z-Machines. Z-Machine operations are used to represent RoboChart transition behaviours. A set of observational variables are defined to encode the semantic structure of RoboChart elements. These variables are accompanied by a set of healthiness conditions, following the approach of Unifying Theories of Programming [103]. The healthiness conditions characterise state machines with a coherent semantics, and all operations shall satisfy them.

6.2.1 Observational variables

RoboChart states and transitions have separate CSP semantics [54] represented by processes. The transition process starts with an enabled trigger event (if one exists). If the transition condition is

met, the process exits the current state, executes any transition actions, and then enters the target state. From here, the state process starts and executes state entry actions and then during actions.

Two types are defined as *St* and *Evt*, to represent the sets of states and events defined in RoboChart models. A parametric datatype *tag* is defined as follows for an observational variable *tr* to tag semantic transition information:

```
datatype ('s, 'e) tag = State 's | Event 'e
```

To define *tr* using the type *tag*, *'s* is instantiated with *St*, and *'e* with *Evt*. The two constructors *State* and *Event* are used to define state-related and event-related requirements. *tag* comes with the built-in functions of *is_State* and *is_Event*, and both are of Boolean type.

Three observational variables are further defined to enrich the structure of the Z Machine, and to enable a more precise mapping to the RoboChart semantics:

- *st :: St* type, as the current state the machine is in;
- *triggers* of type "*Evt set*", a set of possible trigger events that can be enabled in the following transitions; and
- *tr :: (St, Evt) tag list* as a list representing the sequential trace of the behaviours of each transition.

These variables form part of the Z-Machine store. *tr* is used to mimic a combined process of states and transitions based on the CSP semantics, but is a simplified version of the CSP process of RoboChart without the process of entering and exiting states. tr is used to record sequentially the trigger event, the exit action of event type in source state, the actions of event type executed, the entry action and during action of event type in target state, and the target state entered. tr is mainly used to define the event-related and state-related safety requirements.

6.2.2 Healthiness conditions

With the observational variables defined, we give a set of predicates as healthiness conditions that ensure well-formedness of the semantic variables. For example, when a transition is completed, the entered state in the transition process shall be the current state. These conditions will be the basis for property requirements proving.

There are three generic healthiness conditions defined below:

HC1 $\triangleq (\#tr > 0)$. The trace shall not be an empty list. In RoboChart, state machines must contain at least one state, therefore after initialization, *tr* is not empty. *#xs* is a function on *xs* to return the length of the list *xs*.

HC2 $\triangleq (last(tr) = State\ st)$. The last element of *tr* shall be the current state *st* the state machine is in after the transition.

HC3 $\triangleq (\forall i < \#tr.tr!i = State\ final \longrightarrow i = \#tr - 1)$. If the state machine terminates, the last element of *tr* shall be one of the termination states. *tr ! i* means to get the ith element of *tr*.

The healthiness conditions are collected in a predicate called `wf_rcstore`, which is imposed on every state machine store.

6.3 Model transformation rules

This section discusses the second step of mapping RoboChart state machines to the semantic domain of Z-Machines. We group the elements of RoboChart meta-models into six categories according to the elements of RoboChart: Type Declarations, Context, Nodes, Transitions, Statements and Expressions, and State machines. The Statements and Expressions mapping is straightforward and omitted. Therefore, five groups of mapping rules are designed and presented in this section. The rules proposed in this section are summarized in Table 6.1. These rules will be implemented into EGL code for model-to-text transformation. The whole process is illustrated with the running example of the GasAnalysis state machine of the Chemical Detector. In the rest of this section, we present in detail the rules we designed.

Table 6.1: RoboChart element, rules, and mapping

RoboChart element	Rules for RoboChart elements	Mapping in Z Machine
A. Types	R1 Primitive datatype	Isabelle/HOL primitive type
	R2 Type declarations	HOL type declaration
B. Context	R4 Constants	Polymorphic constants
	R5 Variables	zstore variable
	R6 Events	Enumeration type: Evt
C. Nodes	R7 Nodes	Enumeration type: St
D. Transition behavior	R8 Transition behavior	operations
	- value of input/synchronization event	params section
	- transition condition	pre section
	- value assignment	update section
E. State machine	R9 state machine	zmachine
F. Statement and expression	Omitted	

6.3.1 Types

Rule 1: **Primitive datatypes** of RoboChart are mapped to Isabelle/HOL primitive types including nat to `nat`, int to `integer`, boolean to `bool`, real to `real`, and char to `string`.

Rule 2: **Type declarations** in RoboChart are mapped to HOL type declarations. These include abstract types without a definition (i.e. given sets), record types, enumerations, Cartesian products, and sets. HOL has similar elements for these types. An example mapping for GasAnalysis is shown in Fig. 6.4. We use the text format of RoboChart (on the left of the image) here to compare it with the models (one the right of the image) in Isabelle/HOL.

```
datatype GasSensor {
    c : Chem
    i : Intensity }
type Chem
type Intensity
enumeration Status {noGas gasD}
```

(a) RoboChart types

```
type_synonym Chem= "nat"
type_synonym Intensity= "nat"
record GasSensor =
  c :: Chem
  i :: Intensity
enumtype Status = noGas | gasD
definition "Status= {noGas, gasD}"
```

(b) Isabelle types

Figure 6.4: Gas Analysis example for mapping **Rule** 2.

Rule 3: **Types in the Z mathematical toolkit** including functions, relations and sequences, are mapped to Isabelle/HOL functions, relations, and lists, using the available Z mathematical toolkit[1]. RoboChart functions are partial, so have a precondition and postcondition pattern. However, for the purpose of theorem proving, an abstract function declaring the type of the domain and the range is often sufficient.

6.3.2 RoboChart context

RoboChart context includes Variables, Constants, Operations, and Events. These elements can either be declared directly in the robotic platform, or grouped in interfaces [54]. Operations are omitted in our transformation as they represent the hardware platform behaviours.

Rule 4: **Constants** are mapped to polymorphic constants in HOL using the **consts** keyword. Such constants are uninterpreted symbols of a given type, which can be later assigned a value using the **def_consts** command.

Rule 5: **Variables** are mapped to store variables. The **zstore** command is used to collect all the variables.

Rule 6: **Events** are collected and defined as an enumeration type `Evt` with all the events as the constructors in Isabelle/HOL. Events may be used as a transition trigger, a transition action, or a state action.

The transformation examples using Rule 4-6 is as below:

```
1  consts thr::"Intensity"                                          ▷ Rule 4
2
3  enumtype Evt = gas | turn | resume | stop                        ▷ Rule 6
4
5  zstore GasAnalysis =
6    sts::"Status"                                                  ▷ Rule 5
7    gs:: "GasSensor list"                                          ▷ Rule 5
8    ins::"Intensity"                                               ▷ Rule 5
9    anl::"Angle"                                                   ▷ Rule 5
10 where invs...
```

According to Figure 6.3, one constant thr is defined for the state machine, and is mapped to the constant in Z Machine using keyword **consts**. There are also four variables sts, gs, ins, and anl in the

[1] https://isabelle-utp.york.ac.uk/theories/z-toolkit

state machine. Therefore, Rule 5 is used to present these variables in the state space `GasAnalysis`. Based on Rule 6, an enumeration type Evt is defined with four events `gas`, `turn`, `resume`, and `stop` as constructors. The **where** section contains the healthiness conditions and structural invariants, which are explained in detail in Sections 6.2.2 and 6.4.1.

6.3.3 Nodes

A RoboChart node can either be a state or a junction, with Final States and Initial Junctions as special types of States and Junctions [54]. There is no need to differentiate the initial junction from other states for transformation.

Rule 7: **Nodes** are collected and defined as an enumeration type `St` with all the nodes as the constructors.

In GasAnalysis there are six nodes, which are used to construct the enumeration type `St` as below:

```
enumtype St = initial | NoGas | GasDetected | Analysis  | Reading  | final
```

6.3.4 RoboChart transition behaviour

Rule 8: **Transition behaviour** of RoboChart, including entering and exiting states, and transitions between the states, is mapped to **zoperation**s. One transition matches one **zoperation**. This means that traceability to the original state machine model is retained. Moreover, it makes the technique compositional, since it enables the independent verification of whether each individual transition preserves the invariants, without affecting any of the other transitions. Thus, the incremental nature of the Isabelle document model is kept, whereby individual commands are processed independently [104]. The mapping between a RoboChart transition and the Z Machine operation is depicted in the Z Machine template for an operation, as shown in Listing 6.2.

Listing 6.2: Z Machine template for operation

```
zoperation TransN =
  params <parameter_input> ∈ "<PARAMETER>"
  pre "st=<src_st> (∧ <tran_cond>)*"
  update "[<state_var>′ = <new_val_state_var>
          ,st′ = <tgt_st>
          ,triggers′={<possible_trg_events>}
          ,tr′= tr @ [<trg_event>, <src_exit_action>, <tran_action>,
              ↪ <tgt_action>, <tgt_st>]
          ,<parameter>′= <parameter_input> ]"
```

This template preserves the healthiness conditions defined in Section 6.2.2 by construction. For example, in the **update** section, the length of trace `tr` may get longer by being appended with events or states. Thus, HC1 (`#tr>0`) is preserved.

Now, each of the components of this template is explained in turn with examples mapping of GasAnalysis shown in Fig. 6.5.

```
state Reading { }
state Analysis {
    entry st = analysis (gs)}
state NoGas { }
transition t0 {
    from Reading
    to Analysis
    trigger gas ? gs
    }
transition t3 {
    from Analysis
    to NoGas
    condition st == Status::noGas
    action send resume
    }
```

(a) RoboChart transitions

```
zoperation ReadingToAnalysis =
  params gs_input∈"SeqGasSensor"
  pre "st= Reading "
  update "[ sts′ =analysis(gs_input)
           ,gs′ = gs_input
           ,st′= Analysis
           ,tr′=tr @ [Event gas, State Analysis]
           ,triggers′={}
          ]"

zoperation AnalysisToNoGas =
  pre "st= Analysis ∧ sts=(noGas)"
  update "[ st′= NoGas
           ,tr′=tr @ [Event resume, State NoGas]
           ,triggers′={}
          ]"
```

(b) Z Machine operations

Figure 6.5: Examples for mapping Rule 8, 8.1, 8.2, 8.3.

Rule 8.1: For RoboChart events of input and synchronization types that communicate a value to a variable, `params` section lets the variable `parameter` first read a value from a set `PARAMETER` that represents the possible input values. The variable to be updated will then be assigned the value of `parameter` in the `update` section. To differentiate the parameter in the `params` section and the parameter in the `update` section, a suffix *_input* to the parameter is used in the `params` section. .

For example, in transition t0 of Figure 6.5, `gs` is to be updated through event `gas` of input type. After transformation, the input value is first read by `gs_input` from the set `SeqGasSensor`. Then `gs` is assigned with `gs_input` for an update.

Rule 8.2: For each operation, there are two possible sources of precondition including transition conditions (e.g., `sts = Status::noGas`), and the source state of the transition (e.g., `st = Reading`).

Rule 8.3: The `update` section updates the store variables if involved in this transition. Depending on the transition to be executed, the updated variables vary. However, the observational variables are always updated in each operation:

- `st` is updated to the target state of the transition.
- `triggers` is updated to a set of events that may be enabled while the machine is at the target state.
- `tr` is updated by appending to the list sequentially the trigger event, the exit action of event type in the source state, the transition action of event type, the entry action and the during action of the event type in the target state as appropriate, and the target state.
- For the variable to be updated through an input event, it is assigned with the value of `parameter` from the `params` section. Besides, if there is an assignment in the same operation using this variable as a function parameter, the assignment shall use `parameter` instead of the variable itself.

  For example, variable `sts` is updated through the assignment `sts' = analysis(gs_input)` where `gs_input` is the input parameter for `gs`.

The rules are further explained using examples in Fig. 6.5. In RoboChart, transition t0 is from source state Reading to target state Analysis, which is transformed to the operation `ReadingToAnalysis`. This transition is first triggered by gas; a value is then assigned to variable gs; then the source state is exited and state Analysis is entered; and an entry action assigns a new value to sts.

Accordingly, in Z Machine operation `ReadingToAnalysis`, **pre** uses the source state as a precondition, i.e., `st = Reading`. Another precondition is a constraint on the event. As state Reading is the target state of three transitions, there are three corresponding conditions for the event before the operation including (i) there is no event if Reading is transited from initial, or (ii) the event shall be turn if transited from GasDetected, or (iii) the event shall be resume if transited from NoGas which is transited from Analysis. The event gas is of input type, so its value is communicated through the **params** section. In the **update** section, `st` is updated to `Analysis`, `tr` gains two new elements of trigger event `Event gas` and target state `State Analysis`. Since there are no trigger events in the following transitions with Analysis as the source state, `trigger` is an empty set.

In the second example of Figure 6.5, transition t3 is mapped to the operation `AnalysisToNoGas`. A difference from t0 is that t3 has a transition condition sts==Status::noGas, which shall also be added into **pre** section in operation.

6.3.5 RoboChart state machine

Rule 9: **State machine** is mapped to a **zmachine** which is defined using the initialisation, invariants and all of the transition operations. The initialisation assigns the initial value to all the store variables to provide a starting point for the operation. For example, `tr` is initialized as `[State initial]`. The values of the state variables are provided by users manually. Keyword **invariant** points to the set of state machine invariants (including healthiness conditions and the structural invariants) defined in zstore using the naming format of "Machine_name_inv". For `GasAnalyais`, the invariants is represented by `GasAnalysis_inv`.

The **zmachine** for GasAnalysis is as below:

```
definition Init::"GasAnalysis subst" where
"Init = "[gs⇝[]
        ,ins⇝0
        ,sts⇝noGas
        ,anl⇝Front
        ,st ⇝initial
        ,tr ⇝[State initial]
        ,triggers⇝{}]"

zmachine GasAnalysisMachine =
    over GasAnalysis
    init Init
    invariant GasAnalysis_inv
    operations InitialToReading AnalysisToNoGas AnalysisToGasDetected
        ↪ GasDetectedToFinal GasDetectedToReading ReadingToAnalysis
        ↪ NoGasToReading
```

6.4 Verification method

Section 6.3 has provided the complete transformation rules from RoboChart models to Z Machine based on which the Z Machine notations can be generated automatically. Then, the process of verification that the Z Machine notations representing the RoboChart state machine satisfy the AC claims is discussed in this section.

There are two parts of the verification for the state machine modelled in Z Machine, including the verification of the structural invariants and of the system requirements referenced in the AC claims. Both types of verification use the same verification structure as described in Section 6.1.1: (1) a lemma is generated to prove that the initialization of the machine establishes the invariant, and (2) a lemma is generated for each operation to that the operation preserves the invariant. This section provides the principles to define invariants and requirement specifications, and a method to verify them.

After a RoboChart Z-Machine is generated, the developer needs to provide a set of invariants and requirement specifications, and then apply Isabelle to them. `zpog` method is used to generate the prove obligations. These obligations may be discharged immediately or a set of residual proof obligations is provided. For the latter, *sledgehammer* is called as needed to complete the proof automatically, or we can use the command *nitpick* to find counterexample traces, which may facilitate the error identification in the models.

6.4.1 Structural invariants

Structural invariants impose constraints on the state space, and they are typically supplied by the user. However, some simple heuristics to infer invariants for state variables may also be implemented according to some common patterns employed in RoboChart.

A variable can be updated with an assignment of a function value. If the variable is not updated with this assignment in each operation, an invariant is inferred to record the relationship between the variable and the function parameters and to impose this relation to all operations so that those operations where the variable is not updated know their relation's existence.

There are two ways to assign values to a variable using the value of a function.

Case 1: The variable and the function parameters are updated during the same transition and not updated anywhere else, an invariant shall state that the variable equals the function value for all operations.

Case 2: The operation where the variable is updated is different from the operation where the function parameter is updated, an invariant shall state that the variable equals the function value under certain states excluding the state where only the parameters are updated.

The invariants are defined in the **`where`** section of the **`zstore`**, as shown below:

```
zstore GasAnalysis =
 ...
 where
  hc:"wf_rcstore tr st (Some final)"
  inv_1:"sts = analysis(gs)"
  inv_2:"st=GasDetected⟶ ins=intensity(gs)"
```

The state space shall satisfy all the healthiness conditions by the inclusion of the `wf_rcstore` predicate, and then the state the structural invariants.

Invariant `inv_1` is an example of Case 1. In operation `ReadingToAnalysis`, the input event `gas` updates the gas sensor reading list `gs`, and the detection status `sts` is updated by an assignment of `analysis(gs)` to `sts`. The variable `sts` and the parameter `gs` are updated in the same operation, and nowhere else in the state machine. Consequently, `sts = analysis(gs)` is a structural invariant.

Invariant `inv_2` is an example of Case 2. In operation `AnalysisToGasDetected`, `ins` is updated with `intensity(gs)`, while the parameter `gs` is updated in the operation `ReadingToAnalysis`. Thus, the invariant `ins = intensity(gs)` holds when in the state `GasDetected`.

Once the invariants are specified, it should be proved that each transition operation preserves them using `zpog`, a process which is fully automated.

6.4.2 AC claim formalization

The requirements referenced in the AC claims for the RoboChart state machine are manually formalized in HOL, and then defined within the theory file using a command keyword **`zexpr`**, which creates a named expression for the invariant. The method can verify the claims related to events, states, and state variables based on the available store variables. For example, the trace variable `tr` can be used to impose constraints about sequencing of events. There are five types of claims that can be verified against the state machine in Isabelle including:

- The state reachability driven by the variables.
- The sequential relationship between states and events.
- The sequential relationship of states, i.e., the correctness of the transition source and target.
- The relationship between state variables.
- Deadlock freedom.

The example requirements of GasAnalysis for each type above and their formalisation as invariant expressions, are given below. “Deadlock freedom” does not require a formalization process and its verification is discussed in Section 6.4.3.

R1: If no gas is detected, the machine does not terminate.

$sts = noGas \longrightarrow \forall i < \#tr.tr \,!\, i \neq State\ final$

R2: The machine shall terminate only on arrival of a stop event.

$\forall\, i < \#tr.tr \,!\, i = State\ final \longrightarrow tr\,!(i-1) = Event\ stop$

R3: The readings from the sensors shall be analyzed before the gas detection status is identified.

$\forall i < \#(states\ tr) - 1.\ (states\ tr) \,!\, i = State\ Reading \longrightarrow (states\ tr) \,!\, (i+1) = State\ Analysis$

R4: If intensity exceeds the threshold, gas detected shall be indicated.

$intensity(gs) \geq thr \longrightarrow sts = gasD$

Here, $\#xs$ denotes the length of list xs, and $xs!i$ returns the ith element of xs.

R1 is an example of the type of the state reachability driven by the variables. It states that whenever the status is noGas and the threshold is positive, then the state $final$ does not appear in the trace, i.e., not reachable.

R2 verifies the sequential relationship between states and events. It states that when there are more than two elements in tr, then if the final state is present, then the element before the *State final* must be the event $stop$.

R3 verifies the relationship of two states of *Reading* and *Analysis* within one operation, i.e., the correctness of the transition direction from the source state to the target state. It states that after entering the state *Reading*, the following state must be *Analysis*.

R4 verifies the relationship between the state variables. It states that when the intensity of the detected gas reach the threshold, the variable *sts* should be assigned the value of *gasD*.

6.4.3 Invariant proving

Both the structural invariants and requirements referenced in the claims are regarded as invariants for the Z Machine models. To prove the Z Machine model of the RoboChart state machine satisfies an invariant, two steps are needed: (1) the initialisation establishes the requirement; and (2) each operation preserves the requirement. An excerpt of **R1** verification is given in list below:

```
zexpr R1 is
"sts=noGas ⟶ (∀i<(length tr). tr ! i ≠ State final)"

lemma "Init establishes R1"
  by zpog_full

lemma "InitialToReading() preserves R1"
  apply zpog_full
  by (metis St.distinct(30) less_SucE nth_append nth_append_length tag.inject(1))

lemma "GasDetectedToFinal() preserves R1 under GasAnalysis_inv"
  by zpog_full
```

The initialisation `Init` establishing **R1** is proved automatically by applying `zpog` method. The operation `InitialToReading` preserving **R1** is proved by applying `zpog` method and calling *sledgehammer*. After applying `zpog`, one sub-goal remained as shown in Figure 6.6. Then we applying *sledgehammer*, which returned several options of solutions from different external tools. In this example, we chose the solution returned by the theorem prover Vampire [105] to complete the proof, as shown in Figure 6.7.

```
lemma "InitialToReading() preserves R1"
  apply zpog_full
```

Proof state | Auto update | Update | Search:

```
proof (prove)
goal (1 subgoal):
 1. ⋀tr i.
       ∀i<length tr. tr ! i ≠ State final ⟹
       0 < thr ⟹ i < Suc (length tr) ⟹ (tr @ [State Reading]) ! i ≠ State final
```

Figure 6.6: Screenshot of applying `zpog_full` for proving `InitialToReading()` preserving R1.

```
lemma "InitialToReading() preserves R1"
  apply zpog_full
  sledgehammer
```

Proof state | Auto update | Update | Search:

```
Sledgehammering...
cvc4 found a proof...
verit found a proof...
vampire found a proof...
cvc4: Try this: by (simp add: nth_append) (8 ms)
verit: Try this: by (simp add: nth_append) (5 ms)
cvc4 found a proof...
vampire: Try this: by (metis St.distinct(30) less_Suc_eq nth_append nth_append_length tag.inject(1)) (28 ms)
cvc4: Try this: by (simp add: nth_append) (4 ms)
QED
```

Figure 6.7: Screenshot of applying *sledgehammer* for proving `InitialToReading()` preserving R1.

The operation `GasdetectedToFinal` is proved by `zpog` method but with the support of the invariants `GasAnalysis_inv`. This `GasAnalysis_inv` includes `inv_1`, `inv_2`, and the healthiness

conditions hc listed the state space. The lemma for GasdetectedToFinal is to say that

```
{C1 ∧ GasAnalysis_inv} GasdetectedToFinal() {C1}
```

that is, when the operation is started in a state where both the structural invariants and **R1** are satisfied, then following execution the requirement still holds.

Though users need to interact with Isabelle/HOL to give the commands and instructions, the proof itself is completely automated. Concerning the automation of the invariant proving, the verification template, i.e., the **zexpr** section is automatically generated with the requirement content left blank and the lemma expressions for the initialization and each operation. The users need to fill in the formalized invariants manually.

Another type of requirement that can be verified is the deadlock freedom of the Z Machine using an automated method [95]. For RoboChart, this means that each state has at least one enabled transition when the structural invariants are satisfied. In the example of GasAnalysis, a claim of deadlock freedom is that the state machine only deadlocks when it enters final state, i.e., it is deadlock-free when not entering final state. The solution to verify this property is to introduce an operation to bypass final state and use the deadlock_free proving method to verify the deadlock freedom. The bypass operation and the deadlock freedom requirement of GasAnalysis are as below:

```
zoperation Bypass =
  pre "st= final"

lemma GasAnalysis_deadlock_free:
"deadlock_free GasAnalysisMachine"
    apply deadlock_free
    sledgehammer ...
```

The proof is done by applying deadlock_free method to provide the sub-goals, and completing the proving by calling *sledgehammer*.

6.5 Implementation

The generation of AC evidence through theorem proving was implemented in the environments of Eclipse and Isabelle/HOL as shown in Figure 6.8.

There are seven activities including:

1. Export of EMF model from RoboChart models. This is done automatically within RoboTool.
2. Generation of EGL template based on the mapping rules. This is a manual process that builds the base for automated transformation.
3. Design of transformation algorithm based on the mapping rules. This is the design and coding process of the implementation.

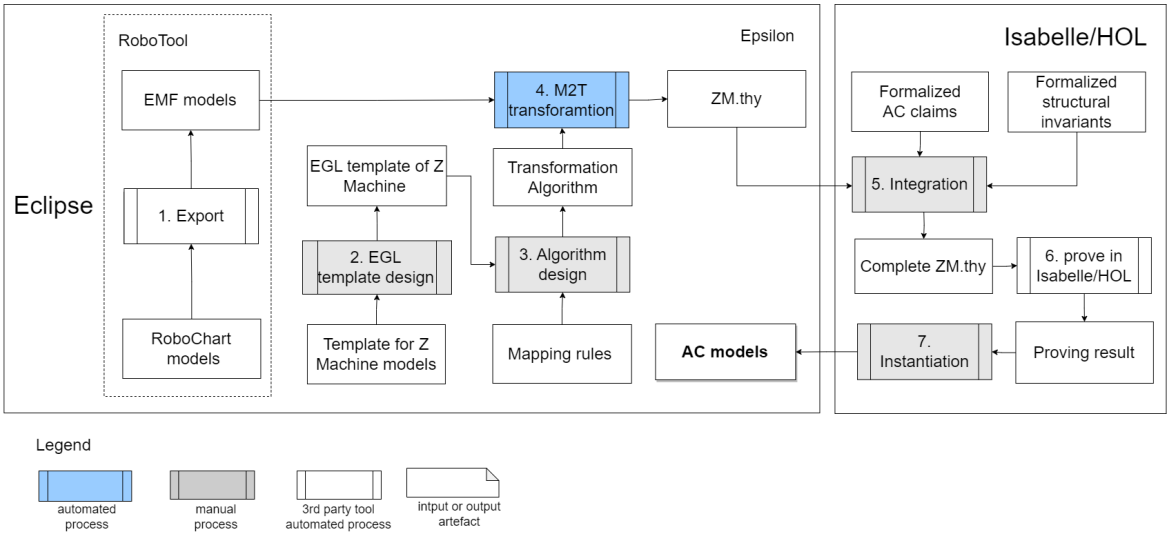


Figure 6.8: Implementation environment of AC evidence generation by theorem proving.

4. Transformation of RoboChart models into Z machine notations in the form of a theory file. This is executed automatically within Epsilon using the transformation rule in Step 3 mentioned above.

5. Manual input of the model-specific invariants and the formalized AC claims to the theory file.

6. Verification of the AC claims of the RoboChart models in Isabelle/HOL. This is supported by automated proving capability of Isabelle/HOL.

7. Generation of evidence models from the proving results. This is a manual process to update the AC evidence models with the proving results.

The implementation of Step 2 is discussed in detail in Section 6.5.1. Step 3 is described in Section 6.5.2. Other steps mainly involve the usage of the tools or the manual process, therefore are omitted here.

6.5.1 EGL template design

A template for the Z Machine notations that represents a RoboChart state machine is defined in Listing 6.3, and the corresponding state machine verification template is defined in Listing 6.4. These two templates are created by incorporating the mapping rules stated in Section 6.3 into the general template in Listing D.1. The instantiable elements in the Z Machine templates are to be replaced by the elements of the RoboChart state machine.

The templates in Listing 6.3 and Listing 6.4 are implemented using EGL to realized the model-to-text transformation. The placeholders of the templates are replaced by the EMF models of the RoboChart models that are automatically generated and exported from the RoboTool environment.

Listing 6.3: Z Machine template for EGL transformation

```
subsection < Types and constants>
datatype ('s, 'e) tag = State 's | Event 'e
abbreviation "is_Event x ≡ ¬ is_State x"
type_synonym ('s, 'e) rctrace = "('s, 'e)  tag list"

definition wf_rcstore :: "('s, 'e) rctrace ⇒ 's ⇒ 's option ⇒ bool"
where [z_defs]: "wf_rcstore tr st final =
   length tr > 0
 ⋀ last tr = State st
 ⋀ final ≠ None⟶ (∀i< #tr. tr ! i = State (the final) ⟶ i= #tr -1)"

enumtype St = <St>                              ▷ <St> = initial (| <state>)* (| final)?
definition "St = {<St_set> }"                 ▷ <St_set> = initial (,<state>)* (, final)?
(enumtype Evt = <Evt>                                    ▷ <Evt> = <event> (| <event>)*)?
(definition "Evt = {<Evt_set>}")?                      ▷ <Evt_set> = <event> (, <event>)*)?
(enumtype <EType> = <literals> )?
(definition "<EType> = {<literal> (, <literal>)+}")?
(type_synonym <Primtype> = <type>)*
(record <Record> = (<field>:: "<type>")+)*
(consts <consts_type> )*                            ▷ <consts_type> = <Consts> ::  "<type>"
consts <fun_name> :: "<fun_type>"                        ▷ <fun_type> = <type> (⇒ <type>)+
fun <fun_name> :: "<fun_type>"     where "<definitionOfFun>"

subsection < State space >
zstore <StateMachine> = st:: "St"
                        tr:: "(St, Evt) tag list"
                        triggers:: "Evt set"
                        (<stateVar> :: "<type>")+
                        where hc:"wf_rcstore tr st (Some final)"
                              ("inv: "Sturctural_Invariant")+

subsection < Operations >
(zoperation <op_name> =
  (params <params>                          ▷ <params> = <parameter>_input ∈ "<PARAMETER>")?
  pre "st=<src_st> (∧ <tran_cond>)*"
  (update "[st′ = <tgt_st>
          (,<parameter>′= <parameter_input>)?
           ,triggers′={<trg_events>}
           ,tr′= tr <tr>          ▷ <tr> = tr (@ [<trg>])?  (@ [<src_exit_action>])?  (@
               [<tran_action>])?  (@ [<tgt_action>])?  @ [<tgt_st>]
          (,<stateVarAssign>)?
                                  ▷ <stateVarAssign> = "<stateVar>′ = <stateVar_new_value>"
           ]")? )+

subsection < Initialization and machine >
definition Init::"<StateMachine> subst" where
"Init = "[st ⇝ initial
          ,tr ⇝ [State initial]
          ,triggers ⇝ {}
         (,<stateVar>⇝ <Val> )*]"

zmachine <StateMachine>Machine =
    over <StateMachine>
    init Init
    invariant <StateMachine>_inv
    operations <operations>                                  ▷ <operations> = (<op_name>)*
```

To facilitate the implementation of EGL transformation, some placeholders of the basic Z Machine template (Listing D.1) are combined to simplify the presentation of the templates. For example, the definition of enumeration type $< St >$, which is composed of the all the nodes in the state machine, is represented as a single placeholder $< St >$. This is commented at the end of line 12 to show the actual composition of $< St >$.

Listing 6.4: Z Machine verification template for EGL transformation

```
1  subsection < Structural Invariants >
2
3  lemma Init: "Init establishes <StateMachine>_inv"
4     by (zpog_full; auto)
5
6  (lemma <Opname>_int:"<op_name_()> preserves <StateMachine>_inv"
7     by (zpog_full; auto))+                              ▷ <op_name_()> = <op_name>(p*)
8
9  subsection < Requirements >
10
11 (zexpr Req_1 is "<formalized_requirement>"
12
13 lemma Init:"Init establishes Req_1 under <StateMachine>_inv"
14    by (zpog_full; auto)
15
16 (lemma <op_name>_int:"<op_name> preserves Req_1 under <StateMachine>_inv"
17    by (zpog_full; auto))+                              ▷ <op_name_()> = <op_name>(p*)
18    )+
19
20 subsection < Deadlock freedom >
21
22 definition [z_defs]: "<StateMachine>_axioms = <>"
23
24 lemma deadlock_free: "<StateMachine>_axioms  ⇒ deadlock_free
       ↪ <StateMachine>Machine"
25   apply deadlock_free
26   sledgehammer
```

6.5.2 Transformation from RoboChart to Z Machine

To be instantiated automatically, the templates in Listing 6.3 and Listing 6.4 are represented as an EGL template and the instantiable elements are obtained by querying the RoboChart EMF models. All the algorithms used for the implementation are presented and described in Appendix C. The steps to query RoboChart models and transform them into Z Machine notations are listed below:

1. Generation of enumeration types for states and events by searching all the nodes, events of the state machine, the detailed algorithm can be seen in Algorithm 25.
2. Generation of primitive types by searching all primitive types and to generate type declarations by searching RoboChart type declarations. The detailed algorithm can be seen in Algorithm 26.
3. Generation of the constants for all the constants defined in the state machine and for the parameters communicated through the events. The detailed algorithm can be seen in Algorithm 27 and Algorithm 28.

4. Generation of function definition by querying the functions used in the state machine. The abstract functions without concrete definitions are created with the keyword **consts**, and the explicit functions which have detailed definitions are created with keyword **fun**. The detailed algorithm can be seen in Algorithm 29. The definition of explicit functions is not supported by automated transformation because the definition involves concrete design decisions which require the engineering judgement.

5. Generation of **zstore** section by searching for the variables defined for the state machine. The detailed algorithm can be seen in Algorithm 30.

6. Generation of operations by querying each transition behaviour. The detailed algorithm can be seen in Algorithm_Operation. The detailed algorithm can be seen in Algorithm 31.

7. Generation of the initialization and the machine by collecting all the state variables and operations generated in the previous steps. The detailed algorithm can be seen in Algorithm 39.

8. Generation of the proving process of the invariants including the structural invariants and system requirements. The detailed algorithm can be seen in Algorithm 40.

6.6 Case studies and evaluation

In addition to the previously discussed chemical detector example in Sections 6.3 and 6.4, this section applies the proposed method using the LRE case study (as introduced in [50]) which is also utilized in Section 4.5.2. The LRE unit used in this section is a modified version, where the operation dimensions have been reduced from three to two, and additional transitions have been introduced between states for the purpose of verification.

The activities of system hazard analysis, creating the EMF model from the hazard analysis results, and constructing the initial AC structure follow the same processes as those used in the LRE case study discussed in Section 4.5.2. Therefore, these processes are omitted in this section, and only the system model transformation and verification processes are presented for the case study.

6.6.1 System description

Despite the adjustment made to the LRE unit, the function description remains unchanged from Section 4.5.2 and is therefore omitted. Figure 6.9 illustrates the RoboChart state machine model of the modified LRE. To provide a better understanding of the RoboChart models and the subsequent verification, additional details are provided here for the LRE state machine. The state machine incorporates two variables: `pos` for the robot's position and `vel` for its velocity. Furthermore, four constants are defined: `Obsts`, which represents a list of static obstacles; `HCMVel`, the advisory velocity for HCM mode; `MOMVel`, the advisory velocity for MOM mode; `StaticObsDist`, the acceptable distance to the obstacles; and `MinSafeDist`, which represents the assumed non-negative minimum safe distance to the obstacles.

There are five functions utilized in the state machine. The `inOPEZ` function determines whether the robot has entered a restricted area referred to as the "object proximity exclusion zones" (OPEZ),

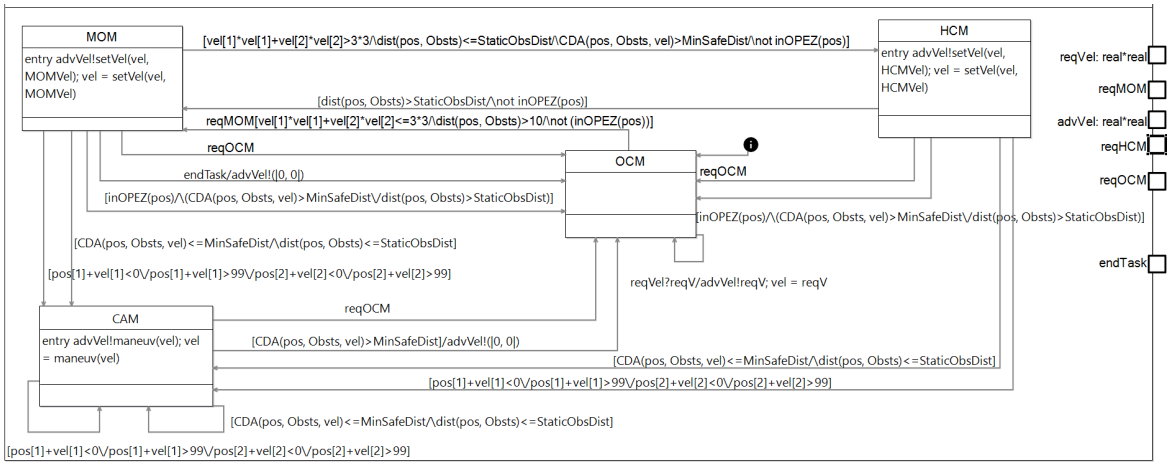


Figure 6.9: RoboChart state machine of LRE adapted from [50].

returning a Boolean value. The `dist` function calculates the distance between the robot and the nearest obstacle based on their respective positions. The `CDA` function (closest distance of approach) computes the minimum distance at which the robot will approach an obstacle in the near future, taking into account both positions and velocity. The `maneuv` function can be employed to alter the robot's direction by 90 degrees to avoid collision with an obstacle. Lastly, the `setVel` function assigns a new value to the velocity. The machine establishes communication with other components through six events, consisting of five input events reqMOM, reqHCM, reqOCM, endTask, reqVel, and one output event advVel.

6.6.2 Z Machine model

The complete Z machine notations generated from the LRE state machine can be found in Appendix E. In this section, the essential aspects of the Z Machine models are highlighted. The state space named *LRE_Beh* encompasses two state variables, three observational variables, and the healthiness conditions.

```
zstore LRE_Beh =
  pos::  "real×real"
  vel::  "real×real"
  st::   "St"
  tr::   "(St, Evt) tag list"
  triggers:: "Evt set"
where
  hc:"wf_rcstore tr st None"
```

According to Figure 6.9, the machine requires the current position as input, which is provided by the robot platform at each state. The position is used as a parameter in multiple functions. Although the transformation does not account for the behaviour of the RoboChart platform or the communication with it, a dedicated operation named `Move` is created to depict the position update. This operation straightforwardly updates the position based on the current velocity. The `pos` variable is the type of pairs, we use the functions *fst* and *snd* to get the first and the second value of the pair.

```
zoperation Move =
  update "[pos′ = (fst(pos)+fst(vel), snd(pos)+snd(vel))]"
```

A possible initialization of the machine is defined as below:

```
definition Init::"LRE subset" where
"Init = "[pos⇝(0,0)
        , vel⇝(0,0)
        , st⇝ initial
        , tr⇝ [State initial]
        , triggers⇝{reqOCM}]"
```

The whole state machine is modelled as `LRE_BehMachine` using the keyword **zmachine**, and consists of the initialization state `Init`, the structural invariant `LRE_Beh_inv`, and all the operations. The state machine is defined as follows:

```
zmachine LRE_BehMachine =
  over LRE_Beh
  init Init
  invariant LRE_Beh_inv
  operations InitialToOCM  OCMToOCM OCMToMOM MOMToOCM MOMToOCM_1  MOMToOCM_2
      ↪ HCMToOCM HCMToOCM_1 MOMToHCM  HCMToMOM  HCMToCAM HCMToCAM_1  MOMToCAM
      ↪ MOMToCAM_1  CAMToCAM CAMToCAM_1  CAMToOCM CAMToOCM_1 Move
```

6.6.3 Requirement formalization and verification

In the AC module for the LRE, there are four claims that require support from theorem proving results, the corresponding requirements referenced in the claims need to be formalized and verified in Isabelle/HOL. The complete verification process is presented in Appendix E. The four requirements and their formalization are presented as follows:

R1: When LRE is in CAM mode, it can only change to OCM or re-enter CAM.

$$\forall i < \#(states\ tr) - 1.\ (states\ tr)\,!\,i = State\ CAM \longrightarrow (states\ tr)\,!\,(i+1) \in \{State\ OCM,\ State\ CAM\}$$

We use the trace *tr* to formalize R1. *tr* records the states entered during operations. According to R1, if there is a state *CAM* in the trace *tr*, then the next state in the trace can only be *CAM* or *OCM*. Here, #*xs* denotes the length of list *xs* and *xs*!*i* returns the *i*th element of *xs*. *states tr* is defined as *(filter is_State tr)* using functions *filter* and *is_State*. The list *states tr* is the list consisting of all the states in *tr*. If state *CAM* is the ith element of the list *states tr*, then the next state in *tr* will be *states tr ! (i+1)*

R2: On entering MOM, the LRE shall advise a new velocity with a value between 0.4 m/s and 0.6 m/s for MOM mode.

$$st = MOM \longrightarrow (tr\,!\,(\#tr-2) = Event\ advVel \wedge (xvel^2+yvel^2 > 0.16) \wedge (xvel^2+yvel^2 < 0.36))$$

We represent the state machine entering the state *MOM* by using the current state *st = MOM*. As stated in R2, the advised velocity should fall within the range of 0.4m/s and 0.6m/s. To formalize

this requirement, we consider two aspects: (1) the occurrence of the event *advVel* that communicates the advisory velocity with the hardware platform, and (2) the advised velocity should be within the range of 0.4m/s and 0.6m/s.

Based on the definition of *tr* (Listing 6.2), during an operation, the last element appended to *tr* is the state *st* entered during the operation, and the second last element represents the occurrence of events in the entry action of the entered state. Therefore, we formalize the occurrence of the event *advVel* as *tr ! (#tr-2) = Event advVel*. Due to the limitation of Z Machine not supporting the square root function, we employ the workaround of using the square value of the velocity $xvel^2+yvel^2$.

R3: When LRE is not in OCM mode, the trigger *reqOCM* can always be enabled.

$st \neq OCM \longrightarrow reqOCM \in triggers$

R3 requires that *reqOCM* can be enabled. This is formalized by stating that *reqOCM* is a member of the set *triggers*, which represents the set of events that can be enabled by the environment.

For each requirement of **R1**-**R3**, a total of 18 lemmas are proven. These include one lemma for initialization, which establishes the requirement, and 17 lemmas for operations generated from the transitions to preserve the requirement.

The fourth requirement **R4** is the property of deadlock freedom of LRE state machine. **R4** is formalized as below and verified by applying $deadlock_free$ method and calling *sledgehammer*.

```
lemma R4_LRE_Beh_deadlock_free:
"deadlock_free LREMachine"
  apply deadlock_free
  by (metis St.exhaust_disc)
```

6.6.4 Evaluation

To evaluate the effectiveness and generality of the method of formal evidence generation through theorem proving, the method has been applied to LRE case study. Based on the case studies of the chemical detector and the LRE, this evaluation section seeks to address the following research questions:

RQ1 Completeness: Are the ACs constructed using the method complete?

RQ2 Generality: Does the method support the development of ACs for different application domains? Can it be applied to applications using various system development techniques?

RQ3 Efficiency: Does the implementation produce ACs with acceptable overheads in terms of computational resources and time?

RQ4 Coverage: Does the method support the generation of evidence models for different requirements referenced in the claims?

One assumption has been made for the method proposed in this chapter: the system is developed using the Model-Driven Engineering (MDE) technique and a UML-like modelling language.

To address the completeness of the AC modules and answer RQ1, the proposed pipeline of the evidence generation process shown in Figure 6.1 ensures that traceability is established throughout the entire process, connecting the AC claims that require support of theorem proving, the formalized invariants, the verification results, and finally the evidence models. Although the identification and formalization of the claims and the update of the evidence model with the verification results are manual processes, with the correctly established traceability, the pipeline, when followed diligently, can ensure the completeness of the generated AC module.

In the context of RQ2, the applicability of the method primarily depends on whether the system models can be verified in the theorem prover and is not strongly tied to the specific domain of RAS applications. The two case studies, namely the chemical detector and the LRE, represent applications from different domains. However, the domains themselves do not significantly impact the process of evidence model generation. Although RoboChart is utilized in the implementation of the method and the case studies, most parts of the method are applicable to EMF-based languages with formal semantics that can be mechanised in Isabelle/HOL. For instance, a new version of SysML is expected to be released soon with a formal semantics[2]. The pipeline can be readily applied to this new version with appropriate model transformations. Furthermore, Z Machines is a fairly generic notation that can give formal semantics to other languages (e.g., SysML).

To answer RQ3, the two case studies of LRE and the chemical detector are conducted on a 1.6GHz Core i5 computer with 8GB of RAM. The CPU time required to generate the Z Machine notations from the RoboChart models is below 1 second. The time taken to verify a claim in Isabelle/HOL varies depending on the scale and complexity of the system models. When proving that the system models satisfy a claim, the number of lemmas to be proved corresponds to the number of transitions in the state machine. In the LRE case study with 18 transitions, the time required to prove a lemma ranges from a few seconds to one or two minutes. Such overhead is deemed acceptable for AC generation during the design phase.

To answer RQ4, the analysis focuses on the coverage of different types of requirements. Currently, the transformation from RoboChart models to Z Machine notations is implemented at the state machine level, allowing for the verification of requirements related to the properties of the state machine using Isabelle/HOL. The communication between state machine and other higher level components is not considered. Moreover, the current implementation does not consider probabilistic and timed properties. However, by employing the method proposed in this chapter, the implementation can be extended to encompass a wider range of RoboChart models and incorporate probabilistic and timed properties by providing new semantics for RoboChart.

Threats to validity

Construct validity threats may arise from the assumptions made during the creation of system artifacts, including safety requirements and RoboChart models. To mitigate this threat, for the LRE case study, the system specification of the AUV system is referenced to ensure the accuracy and relevance of the hazard analysis. Additionally, the LRE models used in this study are adapted from a previously validated case study [50], which enhances confidence in their reliability and suitability for analysis. Regarding the chemical detector, the safety requirements and RoboChart models are sourced from a previously validated case study[3] specifically designed for verification in the FDR model checker.

[2] https://github.com/Systems-Modeling/SysML-v2-Release

[3] https://robostar.cs.york.ac.uk/case_studies/autonomous-chemical-detector/autonomous-chemical-detector.html

This choice contributes to an increased level of confidence in their reliability and appropriateness for analysis.

Potential threats to external validity may arise from the limited use of RoboChart as the system modeling language in the implementation and case studies. This restriction may limit the generalizability of the findings. To address this concern, it is worth noting that the majority of the proposed method can be applied to any EMF-based language that has a formal semantics and can be mechanized in Isabelle/HOL. By extending the application of the method to other modeling languages, the validity and generalizability of the research can be enhanced. Also, Z Machine notation and its verification approach do generalise and can be used for other modelling languages.

Internal validity threats may be present due to the implementation of the method, specifically regarding the accuracy of the model transformation and the generation of assertions from the requirements. To address these concerns, several measures have been taken. Firstly, besides analyzing the verification results of the assertions, the assertions are thoroughly reviewed by a formal expert in our research group to ensure their correctness. Additionally, the automated model transformation from RoboChart to Z Machine notations is validated by comparing the output with manually generated versions of Z Machine notations for the case studies. As a future work, the semantics of the RoboChart state machine will be formally proven for correctness, using RoboChart CSP semantics [19] as the baseline. This step will further enhance the internal validity of the method.

6.7 Summary

This chapter introduces a method for generating AC evidence from claim models by utilizing theorem proving. The proposed method acts as a complement to the AC generation process discussed in Chapter 4, specifically focusing on deriving formal evidence models from claims. Additionally, it complements the method of AC evidence generation through model checking presented in Chapter 5, with an emphasis on theorem proving verification. While model checking is a valuable and highly automated technique, it encounters challenges related to the state explosion problem [96] and cannot be applied to robotic models that involve unbounded data types and real numbers. In contrast, theorem proving represents the state space symbolically, eliminating the need for abstractions and overcoming these limitations. Moreover, the automated theorem provers provided by Isabelle/HOL, through the sledgehammer interface, contribute to the automation of the verification process.

The method defines a pipeline (Figure 6.1) to automate claim verification and evidence generation using the Isabelle/HOL theorem prover. The RoboChart system models are equipped with a Z Machine-based formal semantics that is mechanised within Isabelle/HOL. Mapping rules are established to define the transformation of RoboChart elements into Z Machines through model-to-text transformation. Claims that receive support of the theorem proving results are manually formalised using Z Machine notations. Following verification in Isabelle/HOL, the evidence models of the AC module are updated manually with the corresponding verification results. This method can be applied to other system modelling languages that possess a formal semantics that can be mechanised in Isabelle/HOL, allowing for generalisation beyond RoboChart.

The literature has explored the application of theorem proving in the AC generation process, as demonstrated in previous works [16, 20, 49]. These studies utilize the verification results of theorem proving to instantiate the evidence components of the AC module. However, these works do

not address the complete process of deriving a formal evidence model from the argument claim model. Additionally, most research primarily focuses on generating formal evidence using formal models [16, 49]. In practice, systems are often developed using Model-Driven Engineering (MDE) techniques and system modelling languages that may or may not have formal semantics. Hence, to effectively leverage formal verification in the AC process, the evidence generation process should encompass systems developed using MDE.

Denney and Pai [20] introduced a process and tool called AdvoCATE for AC generation, which integrates claim formalization and formal verification. Because their approach involves using a static source-code analysis tool to verify if the generated source code from system models satisfies the claims, therefore, the process of model formalization is not involved. Compared with these available research work, our method has provided a complete process and solution for the application of theorem proving in the AC generation with the capability to support the system developed with MDE techniques.

In our method, the transformation of system models into Z Machine notations offers several advantages. Previous work by Zou et al. [59, 60] translate Stateflow diagrams into HCSP [61] models, which are verified in Isabelle using Hoare logic. In contrast, our method employs Z Machines as a higher-level representation, facilitating a direct mapping between RoboChart elements and Z Machine constructs. Foster et al. [58] present an automated verification technique for a subset of RoboChart by mechanizing an action language for states and transitions in Isabelle/HOL. Building upon their work, we incorporate Z Machines to realize the mechanization of RoboChart in Isabelle/HOL. The mapping rules from RoboChart to Z Machines draw inspiration from the dynamic semantics of a state diagram described in their research. While their work focuses on verifying only the deadlock-free property, our method enables the verification of various types of properties, including deadlock freedom. Moreover, in their work, the semantics of the entire state machine are generated as a monolithic entity. In contrast, our semantic translation allows for compositional verification. This means that each transition in the RoboChart state machine can be verified independently, without affecting other transitions within the state machine.

Additionally, Foster et al. [106] mechanize an Interaction Tree (ITree) based CSP semantics and a Z mathematical toolkit in Isabelle/HOL. Ye et al. [107] further utilize this ITree-based semantics to provide RoboChart with an operational semantics in Isabelle, while we employ Z Machines as a higher-level semantics. Z Machines is also built on top of the ITree CSP semantics. However, their work does not focus on developing a verification method for this semantics.

Chapter 7

Conclusion

7.1 Summary of contributions

In this thesis, we introduced an integrated framework for deriving and evolving model-based ACs from structured and unstructured system artefacts, as well as robotic design models. The automation of this process is achieved through a range of MBE and formal verification techniques. This framework enables the generation of complete AC structures for RAS systems where development processes yield artefacts in different formats. The various formal verification techniques are exploited to ensure a high level of claim support coverage.

We demonstrated that the ACs can be generated from the artefacts in various formats, including spreadsheets, using predefined AC patterns, and that the lower-level structure of ACs can be automatically derived from the system design models. We also showed that the evidence models can be derived automatically using model checking and theorem proving techniques to provide support for the claims. We have met the research objective set in Section 1.2.

In the remainder of this section, we summarise our contributions, explain how each contribution supports the research hypothesis, and addresses the research questions.

In Chapter 4, we tackle the first research questions proposed in Section 1.2: How can we generate assurance cases from the system engineering artefacts? We addressed this question by presenting an AC-pattern-based method to generate a comprehensive AC structure from the system artefacts in different formats.

Our contribution complements the existing solutions in this area in several ways. We provided an automated process to create model-based ACs that are compliant with the SACM metamodel. By leveraging the features provided by the SACM standard, the resulting ACs can be more expressive. Moreover, the utilization of the SACM metamodel ensures a consistent structure for ACs, facilitating the generation of ACs across various organizations and domains. Our method includes a unification step to unify system artefacts of diverse formats, which are used as input in the AC generation process, into EMF models. Additionally, the method introduces a two-phase design of AC patterns. Initially, a general pattern establishes the fundamental constraints for the AC structure, aiding in the implementation of model transformations. This broadens the applicability of our method across

different domains. Subsequently, users develop their customized patterns tailored to their specific applications, considering the properties to be argued and the involved system artefacts. This two-phase design of AC patterns strikes a balance between generality and applicability. The method is applied to a safety monitoring unit of an underwater vehicle to show the effectiveness of the method.

In Chapter 4, we have also addressed the second research question: "How can we elaborate on the AC structure using robotic design models?" To tackle this question, we have developed a model-query-based method that allows the automated generation of the lower-level structure of ACs based on the formal syntax of robotic models.

When the RAS system follows a model-based design development process, the design models may be involved in the construction of ACs to support the claims of system specifications. A common verification technique employed to support these claims is model review through which we check the presence of certain features required by the specification within the implementation. Then, the AC structure is instantiated with the review purposes (for the sub-level claims) and the review results (for the supporting evidence). We use model query technique to enable the automation of this manual process, reduce the need for manual intervention and enhance the efficiency and the accuracy of the AC generation.

While employing the AC-pattern-based method for AC structure generation, the claims that are supported by the model review will be left as undeveloped at this stage. These claims are further elaborated into a lower-level structure using the model-query-based method. This structure encompasses the AC claims, the corresponding supporting evidence, and the strategies employed as linkages. The undeveloped claim serves as the input for the definition of query rules. Both the claims and evidence are generated through querying the system models. The query rules establish natural traceability between the claims and the design models, thereby facilitating the automated update of the AC whenever modifications are made to the design model.

Our method is independent from the system development environment, making it applicable to systems developed with any UML-like modeling languages. This feature enhances the versatility and adaptability of our method across different development contexts.

The third question is addressed in Chapters 5 and 6, which presented an integrated process that applies model checking and theorem proving techniques in the AC generation to derive AC evidence from the formal semantics of the design models.

The AC-pattern-based method assumes the availability of all system artefacts for instantiation. The generation of these artefacts is not incorporated into the AC process itself. However, in Chapters 5 and 6, we extend the AC-pattern-based method and introduce automated processes for generating AC evidence models from claims using model checking and theorem proving techniques applied to RoboChart models, respectively.

In Chapters 5, after an AC structure is generated using AC-pattern-based method and there are claims that are supported by model checking, we have proposed a method to automatically derive from the claims the evidence models that references the model checking results. The core step of this method is the automated generation of assertions using predefined templates. To achieve this, we categorize the claims into nine types and define a requirement template for each type using controlled natural language. The placeholders of the template are then instantiated using the formal semantics of RoboChart models. Further, we have designed eleven assertion templates to match the requirement

templates. To ensure comprehensive coverage of the requirements, we employ two model checkers: FDR and PRISM. The assertion templates are constructed by utilizing CSP_M and PRISM operators, as well as the RoboChart assertion language. The assertion language, as an intermediate layer between RoboChart and its formal semantics, provides a more concise and readable representation compared to the CSP_M or PRISM semantics, simplifying the process of defining formal assertions to a certain extent.

To instantiate the assertion template, we established the mapping between the claims templates and the assertion templates, by ensuring that they have identical placeholders. Therefore, when instantiating a placeholder in the assertion template, we locate the corresponding placeholder in the related requirement template and use its content to instantiate the assertion placeholder.

Once the formal assertions are generated, the subsequent steps of model checking and evidence model generation are automated processes facilitated by model checking tools and model transformation techniques. The formalization of RoboChart models is automated by RoboTool according to the formal semantics of RoboChart.

Similarly, when an AC structure is generated using the AC-pattern-based method and there are claims that are supported by theorem proving, the verification results referenced in the evidence models can be derived from the claims automatically. This method is discussed in detail in Chapter 6.

Chapter 6 shares the same objective as Chapter 5, which is to provide automated support for formal evidence generation from the claims. However, it employs a different formal verification technique, namely theorem proving, and uses different techniques to achieve the desired outcomes.

In Chapter 6, we defined a pipeline to automate claim verification and evidence generation using the Isabelle/HOL theorem prover, including formal semantics definition, model transformation, claim formalization, formal verification, and the evidence model generation. The RoboChart system models cannot be verified in Isabelle/HOL due to the lack of formal semantics supported in the theorem prover. So we have defined a formal semantics for RoboChart state machines based on the Z Machine notation and established a set of mapping rules between RoboChart and its corresponding semantics. With these mapping rules in place, the transformation of RoboChart models into Z Machine notations is fully automated. Since Z Machine is mechanised within Isabelle/HOL, the transformed RoboChart models can then be verified in Isabelle/HOL.

Then, we defined the verification process for RoboChart models within Isabelle/HOL. This process encompasses the identification of structural invariants, formalization of claims, and their subsequent verification. Currently, the formalization of claims using Z Machine notations is carried out manually. However, we acknowledge the need for automating this process and consider it as a topic for future research. Following the verification in Isabelle/HOL, the evidence models within the AC module are manually updated with the corresponding verification results.

With the methods presented for application of model checking and theorem proving in the AC process, we provided a comprehensive solution for the generation of formal evidence to support AC claims. This solution exhibits a high level of generality across different domains and applicability to various development techniques. Moreover, the automation of formal verification processes significantly reduces the reliance on formal expertise, thereby promoting the practical adoption of formal verification. We also acknowledge the potential inclusion of other verification techniques to support the AC process, which we intend to explore in future work.

The entire framework, encompassing all the methods presented in this section, has been predominantly automated through the utilization of MBE techniques and existing tools. As a result, our framework offers engineers an efficient solution for the derivation and evolution of ACs. As the framework is supported by MBE techniques, there is potential for its capability to be extended in various ways, which will be discussed in the following section.

7.2 Future Work

There are multiple ways in which the research presented in this thesis may be refined and extended. In this section, we present a number of research directions for future work.

7.2.1 Automation of claim formalization using NLP

Efforts are needed to streamline the AC process automation and bridge the existing gap in the requirement formalization process. While this thesis has already implemented a certain degree of formalization automation for model checking evidence generation, fully automated formalization from the requirements for both model checking and theorem proving is not realized. When employing the model checking technique to generate formal evidence for AC claims, the requirements referenced in these claims are formalized as assertions using a specific set of requirement templates and corresponding assertion templates. This process requires manual input to the requirement management tool using RoboChart models. Also, when providing formal evidence for the AC claims through theorem proving, the requirements are manually formalized as invariants using Z Machine notations. Natural Language Processing (NLP) techniques could provide a solution to address this issue. By exploring NLP for requirement formalization, the goal is that the users can automatically obtain the formalized assertions from the requirements written in natural languages. The work in [80] has provided end-to-end neural network models for semantic parsing and employed a machine learning pipeline to generate candidate LTL formulas and models from natural language input. One of the challenges to be addressed is the lack of large training corpora that are annotated with ground truth.

7.2.2 Extension of formal semantics for RoboChart

A more expressive and realistic model is desired to represent RoboChart within Isabelle/HOL to enable a thorough verification of RoboChart behavior. Currently, RoboChart is presented using extended Z Machine semantics, which supports the verification of the behavior of RoboChart state machines. The semantics needs to be extended in several aspects to enhance the representation of RoboChart behavior in Isabelle/HOL. Firstly, the semantics should incorporate the hierarchy of RoboChart models in Isabelle/HOL. Higher-level components such as RoboChart controllers, and the communication among controllers and state machines require a more comprehensive semantics to support them. Secondly, the semantics should incorporate probabilistic and timed features to enable thorough requirement verification. Lastly, the semantics for RoboChart models are defined with reference to their CSP semantics provided in RoboTool. To ensure the robustness and correctness of the semantics, a rigorous proof of correctness is necessary to validate the semantics against the RoboChart CSP semantics in future research.

7.2.3 Incremental system development and verification supported by bidirectional communication between environments

One interesting research direction is to provide a pipeline and environment for incremental development and verification of the system. A key is to establish a bidirectional communication between the system modelling environment (e.g., Eclipse) and the theorem proving environment, i.e., Isabelle/HOL, and to define a communication process to implement the incremental development and verification. Currently, the evidence generation from theorem proving is unidirectional. The system models are converted into formal semantics and taken as input of Isabelle/HOL to generate verification results. The results can not be feedback to the system models automatically as the basis to make the modification. By incorporating a trace-back mechanism from Isabelle/HOL to the system development environment, the identification and feedback of any proving errors to the specific claims in the ACs and their corresponding RoboChart components within the Eclipse environment are enabled. The automated and bidirectional communication also facilitates incremental development and assurance by allowing models to be developed and verified one component at a time. This incremental development and verification can provide significant time efficiency [108]. Furthermore, bidirectional communication has the potential to reduce users' direct interaction with Isabelle/HOL by providing the capability to trigger the proving within the Eclipse environment.

7.2.4 Integration of other verification techniques into AC process

AC claims can be supported by various verification and validation techniques, such as review, simulation, testing, and analysis, etc. In this thesis, we have integrated formal verification techniques into the automated AC generation process. In future work, more verification techniques can be explored for AC generation, especially the simulation and testing techniques.

Simulation is a widely used technique for analyzing robotic systems and has the potential to be automated. Therefore, integrating simulation into the automation process of AC generation is feasible. Specifically, for RAS systems developed with RoboChart RoboSim [109] modelling language can be used. RoboSim allows for the independent modeling of robotic system simulations using state machines to define concurrent and distributed designs that utilize specified services of a hardware platform. The work to derive RoboSim models from RoboChart models has been currently undergoing. Supported with this automated transformation from RoboChart system models, it is feasible to propose an automated process that incorporates RoboSim simulations into the AC generation process to provide simulation evidence for the AC claims. Another advantage to use RoboSim is that RoboSim models can be formally verified using FDR and PRISM. Further, it will also bring benefit to give a Z Machine semantics to RoboSim to support the verification using theorem proving technique.

Testing is another commonly used verification technique. The integration of testing into the automated AC process can be supported by the Isabelle/HOL code generator. Once the RoboChart models are verified in Isabelle/HOL, the implementation code can be generated from the models using the Isabelle code generator. Isabelle supports the conversion of HOL specifications into executable programs in languages such as SML, OCaml, Haskell, and Scala. A case study would be necessary to investigate the possibility of embedding Haskell code generated from Isabelle into a ROS-powered physical robot for testing purposes. The test results obtained can then be used to support the corresponding AC claims.

Appendix A

Algorithms Referenced from Chapter 4

This appendix presents the algorithms referenced from Chapter 4.

Algorithm 7

Algorithm 7 is used to convert the spreadsheet of hazard analysis table into EMF models according to the corresponding metamodels. The template of the input spreadsheet used in Algorithm 7 is given in Figure A.1. It is the same table as in Figure 4.2 with the content left blank. For the algorithm design, only the header of the spreadsheet is the main information needed. In this template of hazard table, there are nine columns for each row including *Hazard*, *Cause*, *SubCause*, *Safety Measure*, *SubSafety Measure*, *Verification*, *Validation*, *Verification Evidence*, and *Validation Evidence*. It has two layers of causes and safety measures. It is allowed for certain hazard, there is only one layer of causes or one layer of safety measures. The implicit relationship between the cells exists that each element belongs to its left element of the same row except the first element of the row; and that if the the left cell is empty, then the element belongs to the its left element of the upper row.

The metamodel used for the unification is the one shown in Figure 4.3 of Section 4.2.1.1.

Hazard	Cause	SubCause	Safety Measure	SubSafety Measure	Verification	Verification Evidence	Validation	Validation Evidence

Figure A.1: A template of hazard table spreadsheet.

Algorithm 7 scans the cells from left to right in each row of the spreadsheet, creates a model according to the type of cell, assigns the value of the cell to the model, and establishes the inner relationship among the cells.

A hazard table *ht* is first created for the spreadsheet (line 1). Then, a set of global variables are defined as *h* for hazard models, *c* for cause models, *subc* for sub-level cause models, *sm* for safety measure models, *subsm* for sub-level safety measure models, *vr* for verification models, *vre* for verification evidence models, *va* for validation models, and *vae* for validation evidence models.

For each row in the table, the cells are queried one by one from left to right. The query starts from

Algorithm 7: Data structuring - Hazard table

```
1  ht = hazardTable.CREATEHAZARDTABLE()
2  for r in hazardTable.rows do
3      if r.hazard.isDefined then
4          h ← r.CREATEHAZARD()
5          ht.hazards ← ht.hazards ∪ {h}
6      end
7      if r.cause.isDefined then
8          c ← r.CREATECAUSE()
9          h.causes ← h.causes ∪ {c}
10     end
11     if r.subCause.isDefined then
12         subc ← r.CREATESUBCAUSE()
13         c.causes ← c.causes ∪ {subc}
14     end
15     if r.safetyMeasure.isDefined then
16         sm ← r.CREATESM()
17         if c.causes.notEmpty then
18             subc.safetyMeasure ← subc.safetyMeasure ∪ {sm}
19         else
20             c.safetyMeasure ← c.safetyMeasure ∪ {sm}
21         end
22     end
23     if r.subSafetyMeasure.isDefined then
24         subsm ← r.CREATESUBSM()
25         sm.safetyMeasure ← sm.safetyMeasure ∪ {subsm}
26     end
27     if r.verification.isDefined then
28         vr ← r.CREATEVERIFICATION()
29         if sm.safetyMeasure.notEmpty then
30             subsm.verification ← subsm.verification ∪ {vr}
31         else
32             sm.verification ← sm.verification ∪ {vr}
33         end
34     end
35     if r.verificationEvidence.isDefined then
36         vre ← r.CREATEEVIDENCEANDLINKTOVR(vr)
37     end
38     if r.validation.isDefined then
39         va ← r.CREATEVALIDATION()
40         if sm.safetyMeasure.notEmpty then
41             subsm.validation ← subsm.validation ∪ {va}
42         else
43             sm.validation ← sm.validation ∪ {va}
44         end
45     end
46     if r.validationEvidence.isDefined then
47         vae ← r.CREATEEVIDENCEANDLINKTOVA(va)
48     end
49 end
```

the *Hazard* column. If a value is defined in *Hazard* column, a hazard *h* of class *Hazard* should be created and assigned with the value of the cell by calling function CREATEHAZARD (line 4). The next line builds the relationship between the table *ht* and the hazard *h* by adding *h* into the hazard set of the table *ht.hazards* (line 5). Then, the query moves to the next cell and repeats the same actions of creating the models and the relationships until the last column is processed.

It is possible for certain hazards there is only one layer of causes in which case, the safety measure *sm* is linked to the cause *c* (line 20) instead of the subcause *subc* (line 18). With this algorithm, the hazard table can be automatically structured as EMF models whenever the table is updated.

The complexity of this algorithm is $O(n)$. n is the number of entries in the hazard table. The algorithm is considered correct if (1) all the hazards are under the element *Hazard Table*, (2) for each row, the element of one column is under the element of the previous column, (3) if the previous column is empty, the element shall be under the element of the previous column of the above row. The correctness of the algorithm is verified by comparing the spreadsheet with the generated EMF models using LRE case study in Section 4.5.2.

Algorithm 8

Algorithm 8 has two inputs, *ac*, the AC module, and *element* which is the element in the hazard table that maps to a claim, e.g., a hazard, or a hazard cause. The function creates a claim class and instantiates it using *element*, then add it into the set of *ac.argumentationElement*.

Algorithm 8: Function: CREATECLAIMANDADDTOAC

```
1 Function createClaimAndAddToAC(ac, element):
      Result: claim
2     claim = CreateClaim(element);
3     ac.argumentationElement.add(claim);
4     return (claim)
```

Algorithm 9

Algorithm 9 has two inputs, *ac*, the AC module, and *claim*. This function is used to create an inference class whose target claim is set as intput *claim*. This *inference* is also added into the set of *ac.argumentationElement*.

Algorithm 9: Function: CREATEASTINFANDADDTOAC

```
1 Function createAstInfAndAddToAC(ac, claim):
      Result: inference
2     inference = CreateAssertedInference(element);
3     inference.reasoning = CreateReasoning(inference, element) ;
4     inference.target = element;
5     ac.argumentationElement.add(inference);
6     return inference
```

Algorithm 10 and Algorithm 11

Algorithm 10 and Algorithm 11 have a similar structure, address the creation of claims for verifi-

cation and validation respectively, and the evidence related. Each function has three inputs, *ac*, the AC module, *subsm*, the sub-level safety measure, and the claim *subsmclaim* which is generated from *subsm*. The function first checks if *subsm* is provided with proper verification or validation methods, if so, the inference relationship is created as *subsmInf* whose target is the claim *subsmclaim*. Then, the claims are created for each verification or validation method using function CREATECLAIMANDADDTOAC, and added as the source of inference relationship. Further, the evidence elements of the AC module are created by calling function CHECKANDCREATEVVEVI.

Algorithm 10: Function: CHECKANDCREATESTRUCTUREFORVR

```
1 Function CheckAndCreateStructureForVr(ac, subsm, subsmClaim):
      Result: subsmInf, vrClaim, evidence
2     if ¬EMPTY(subsm.verification) then
3         subsmInf = CREATEASTINFANDADDTOAC(ac, subsmClaim) ;
4         for vr ∈ subsm.verification do
5             vrClaim = CREATECLAIMANDADDTOAC(ac, vr);
6             subsmInf.source = subsmInf.source ∪ {vrClaim} ;
7             CHECKANDCREATEVVEVI(ac, vr.evidence, vrClaim)        ▷ Alg. 12;
8         end
9     end
```

Algorithm 11: Function: CHECKANDCREATESTRUCTUREFORVA

```
1 Function CheckAndCreateStructureForVa(ac, subsm, subsmClaim):
      Result: subsmInf, vaClaim, evidence
2     if ¬EMPTY(subsm.validation) then
3         subsmInf = CREATEASTINFANDADDTOAC(ac, subsmClaim) ;
4         for va ∈ subsm.validation do
5             vaClaim = CREATECLAIMANDADDTOAC(ac, va);
6             subsmInf.source = subsmInf.source ∪ {vrClaim} ;
7             CHECKANDCREATEVVEVI(ac, va.evidence, vaClaim)        ▷ Alg. 12;
8         end
9     end
```

Algorithm 12

Algorithm 12 has three inputs, *ac*, the AC module, *v.evidence*, the evidence of the verification or validation method, and *vClaim*, the claim created for *v*. This function is used to create an inference object *vInf* of the class *AssertedEvidence* whose target claim is input *VClaim*. Then, this *vInf* is added into the set of *ac.argumentationElement*. After creating *evidence* for each evidence reference of *v*, each *evidence* are added as the source of the inference relationship *vInf*.

Algorithm 13

The general algorithm for AC pattern instantiation from a set of system artefacts is given in **Algorithm 13** . To make it a general case, we assume each type of instantiation data is from an individual artefact. The number of artefact types needed varies depend on the specific system and the property to be argued. We assume there are three artefacts whose contents map to the AC claims, one artefact for the verification result and one artefact for the validation result.

Algorithm 12: Function: CHECKANDCREATEVVEVI

```
1  Function CHECKANDCREATEVVEVI(ac, v.evidence, vClaim):
       Result: evidence
2      if ¬EMPTY(v.evidence) then
3          vInf = CREATEASTEVIANDADDTOAC(ac,vClaim);
4          ac.argumentationElement.add(vInf);
5          for evi ∈ v.evidence do
6              evidence = CREATEEVIANDADDTOAC(ac, evi);
7              vInf.source = vrInf.source ∪ {evidence};
8          end
9      end
```

Algorithm 13: AC Generation from a set of artefacts by Pattern Instantiation

```
1  ac = ht.CREATEARGUMENTPACKAGE()
2  for elem1 in artefact1.elems_1 do
3      claim1 = CREATECLAIMANDADDTOAC(ac, elem1)                    ▷ Alg. 8
4      if ¬EMPTY(artefact2.elems_2_Of_elem1) and artefact2.elems_2.mapToClaim then
5          Inf = CREATEASTINFANDADDTOAC(ac, claim1)                 ▷ Alg. 9
6          for elem2 in artefact2.elems_2_Of_elem1 do
7              claim2 = CREATECLAIMANDADDTOAC(ac, elem2)
8              inf.source = inf.source ∪ {claim2}
9              if ¬EMPTY(artefact3.elems_3_Of_elem2) and artefact3.elems_3.mapToClaim then
10                 Inf = CREATEASTINFANDADDTOAC(ac, claim2)
11                 for elem3 in artefact3.elems_3_Of_elem2 do
12                     claim3 = CREATECLAIMANDADDTOAC(ac, elem3)
13                     inf.source = inf.source ∪ {claim3}
14                     CHECKANDCREATESTRUCTUREFORVR(ac, artefact4.elems_4_Of_elem3,
                         claim3)                                   ▷ Alg. 10
15                     CHECKANDCREATESTRUCTUREFORVA(ac, artefact5.elems_5_Of_elem3,
                         claim3)                                   ▷ Alg. 11
16                 end
17             end
18         end
19     end
20 end
```

The algorithm first generates an *ArgumentPackage* to represent the AC module (line 1). Then, it follows the mapping rule to create the corresponding AC models using each instantiation element in the artefacts. Assuming the pattern needs three elements from three individual artefacts to instantiate the AC claims, e.g., the 1st element $elem1$ comes from the 1st artefact *artefact1* (line 1), and so on. However, it is possible that there is only one artefact needed in the pattern for the instantiation, then $artefact1$ to $artefact3$ will be replaced by the same artefact.

The process starts from the top-level claim instantiation. If an element *elem1* for the top-level claim is defined in the artefact, a top-level claim *claim1* is created and instantiated using *elem1* in *artefact1* (line 3). Then, the next element related to *elem1* is considered. For example, if *elem1* is a hazard in a hazard list, and *elem2* is a set of safety requirements in the safety requirement document, only those safety requirements addressing *elem1* hazard are regarded as 'related to' *elem1* and marked as

elems_2_Of_elem1 (line 4).

For *elem1*, if its related $elem2$ is defined in $artefact2$ (line 4) and maps to a claim, it means the AC structure is further developed below *claim1* to create the inference relationship. Further, a *for* loop is used to generate claims for each *elem2* related to *elem1* and set them as the source claims of *inf*.

Now, a process (line 5 to line 8) is complete for $claim1$ which builds its adjacent lower-level of claims and their inference relation. Then, the algorithm repeats this process for *claim2* and *claim3* of the next levels. In the next step, Function CHECKANDCREATESTRUCTUREFORVR and Function CHECKANDCREATESTRUCTUREFORVA are used for the generation of claims for the verification and validation methods, as well as the evidence to support these claims.

The complexity of Algorithm 13 is $O(n^m)$. where n is the number of elements from each artefact involved, and m is the number of the types of elements involved in the AC instantiation. This is a general algorithm, its correctness is verified by the correctness of its specific version Algorithm 1.

Algorithm 14

Algorithm 14 implements the model query method of Section 4.3 using EOL model query language. According to Section 4.3, if a claim can be supported through system model query, no further manual activity of model review is needed. The claim can be decomposed and verified automatically by querying system models.

Algorithm 14: AC Generation by Model Query

```
1  for inf ∈ ac.assertedInference.all.select(i | i.source.Contain("Model query") do
2      for c ∈ inf.source.Contain("Model query") do
3          inf.source.remove(c)
4          c.clear()
5          for compOfClaimQuery ∈ system.model do
6              if paraA then
7                  claim = CREATECLAIMANDADDTOAC(ac, compOfClaimQuery)
8                  inf.source = inf.source ∪ {claim}
9                  assertedInference = CREATEASTINFANDADDTOAC(ac, claim)
10                 vrclaim = CREATECLAIMANDADDTOAC(ac, claim)   ▷ claim is verified by model
                     query
11                 assertedInference.source = assertedInference.source ∪ {vrclaim}
12                 assertedEvidence = CREATEASTEVIANDADDTOAC(ac, vrclaim)
13                 for compOfEviQuery ∈ system.mode do
14                     if paraB then
15                         evidence = CREATEEVIANDADDTOAC(ac, compOfEviQuery)
16                     end
17                 end
18                 if evidence.notDefined then
19                     evidence = CREATEEVIANDADDTOAC(ac, "not satisftied")
20                 end
21                 assertEvidence.source = assertEvidence.source ∪ {evidence}
22             end
23         end
24     end
25 end
```

An AC model *ac* is generated by pattern instantiation using Algorithm 13. *ac* is the input of Algorithm 14. In *ac* models, the claims that can be supported by "model query" is labelled with "query". Also, these claims are regraded as "undeveloped" which means no further structure is attached below them during pattern instantiation.

Algorithm 14 takes the model *ac* as the source model to search for the AC claims that can be supported by model query (line 1). The identified claims is used to define the query rules for the generation of claims and their evidence.

Each of the identified claim is first removed from the set of source claims of their upper-level inference relationship (line 3), and deleted from *ac* model (line 4).

Next step is to create the new AC segments by querying the system design models (line 5 to 12). The claim query rule identifies the type of the components of the system model that compose the new claims, e.g., state, transition, etc. Then, each component in the system model *system.model* is checked against the claim query rule , which is presented as a parameter *paraA* provided by the users. If the component satisfies the rule, the algorithm creates a claim *claim* for this component, adds the claim as a source claim of the upper-level inference relation inf, creates its lower-level claim *vrclaim* for the verification method and its inference relation *assertedInference* to link *claim* with *vrclaim*; then creates an inference relation *assertedEvidence* which take *vrclaim* as the target claim.

Then, the evidence are generated for the claims by querying system models using the evidence query rules (line 13 to 21). Again, the type of the components of the system models is identified by evidence query rule , which is presented as a parameter *paraB* provided by the users. Each component is queried for generating the evidence. If there is a component satisfies the evidence query rule, an *evidence* model of class *ArtifaceReference* is created for the component (line 15); otherwise an *evidence* is generated as "the *claim* is not satisfied."(line 19). Finally, this *evidence* model is added as the source of *assertedEvidence*.

The complexity of this algorithm is $O(n^4)$ where n is taken as the largest value among the number of inferences in the AC module whose source claims contain keyword "query", the number of claims of each inference that contain keyword "query", the number of components in the system models that are queried for claim generation, the number of components that are to be queried for evidence generation. This is a general algorithm, its correctness is verified by the correctness of its specific version Algorithm 3.

Algorithm 15

Algorithm 15 describes Function DRAWAI(). The algorithm first draws the whole $assertedinference$ section including a goal for the target claim of the *ai*, a strategy for *reasoning* of *ai*, and the edge from the goal to the strategy (line 2 to line 4). Each shape is named using the GSN component with an index, e.g., $goal_i$ for the ith goal, str_i for the ith strategy. Then, this section shall be connected to an upper-level section if the target claim of *ai* is one of source claims of that upper-level $assertedinference$ (line 8). A while loop (line 7)is used to identify the related upper-level from the set of *ai_set_drawn*. When the upper-level is identified, **found** is labelled to stop the search. The complexity of this algorithm is $O(n)$. The function is called by Algorithm 2, therefore its correctness is verified through correctness verification of Algorithm 2. The algorithm uses a *while* control structure. *ai_set_drawn* is a finite set, therefore the algorithm terminates when the parameter t increases to the size of *ai_set_drawn*.

Algorithm 15: Function: DRAWAI

```
1  Function DrawAI(i, ai_set_drawn):
       Result: str_i, goal_i
2      goal_i = ai.target.DRAWGOAL()
3      str_i = ai.reasoning.DRAWSTRATEGY()
4      goal_i.DRAWLINKTO(str_i)
5      found = false
6      t = 0
7      while t < #ai_set_drawn and found = false do
8          if ai.target ∈ ai_set_drawn.at(t).sources then
9              str_(t+1).DRAWLINKTO(goal_i)
10             found = true
11         end
12         t++
13     end
14     ai_set_drawn = ai_set_drawn ∪ {ai}
15     i++
16     return (i, ai_set_drawn)
```

Algorithm 16: Function: DRAWAE

```
1  Function DrawAE(i, ai_set_drawn):
       Result: goal_i, solution_j
2      goal_i = ae.target.DRAWGOAL()
3      ai_set_drawn = ai_set_drawn ∪ {ai}
4      found = false, t = 0
5      while t < #ai_set_drawn and found = false do
6          if ae.target ∈ ai_set_drawn.at(t).sources then
7              str_(t+1).DRAWLINKTO(goal_i)
8              found = true
9          end
10         t++
11     end
12     j = 1
13     while j ≤ #ae.sources do
14         solution_j = ae.sources.at(j−1).DRAWSOLUTION()
15         goal_i.DRAWLINKTO(solution_j)
16         j++
17     end
18     i++
19     return (i)
```

Algorithm 16

Algorithm 16 describes Function DRAWAE(). It has a similar process as Algorithm 15. It turns one *assertedEvidence* section of AC models into a goal-solution section in GSN, and links it to an upper-level $assertedinference$ section.

Algorithm 17: Function: DRAWUNDEVELOPEDGOAL

```
1  Function DrawUndevelopedGoal(i):
       Result: goal_i
2      developed = false
3      for ai ∈ ac.assertedInference.all do
4          if claim = ai.target then
5              developed = true
6          end
7      end
8      for ae ∈ ac.assertedEvidence.all do
9          if claim = ae.target then
10             developed = true
11         end
12     end
13     if developed= false then
14         goal_i = claim.DRAWGOAL()
15         k = 1
16         while k ≤ #ac.assertedInference.all do
17             if claim∈ ac.assertedInference.at(k-1).sources then
18                 str_k.DRAWLINKTO(goal_i)
19             end
20             k++
21         end
22         i++
23     end
24     return (i)
```

ae is a *AssertedEvidence* model in the AC module. The algorithm first draws a goal for the target claim of *ae*, then links it to the upper-level section. Again, a while loop (line 5)is used to identify the related upper-level from the set of *ai_set_drawn*. No strategy is drawn for *ae* because according to the general AC pattern designed in Figure 4.5 of Section4.2.2, the solutions are linked directly to the upper-level goal without a strategy in between. Then, a set of solutions are drawn for the sources of *ae*. The complexity of this algorithm is $O(n)$. The function is called by Algorithm 2, therefore its correctness is verified through correctness verification of Algorithm 2. This algorithm terminates for the same reason as Algorithm 15.

Algorithm 17

Algorithm 17 describes Function DRAWUNDEVELOPEDGOAL(). The undeveloped claims can not be identified by searching the target claim of an *AssertedInference* or *AssertedEvidence*. The function is called for each claim in ACs to identify if it is an undeveloped claim. If it is neither a target of any *AssertedInference* nor of any *AssertedEvidence* (line 3 to line 12), a goal is drawn for the claim. Then, a while loop is used to search the *AssertedInference* that takes this claim as a source. The link is further drawn the strategy generated for that identified *AssertedInference* to the undeveloped goal. The complexity of this algorithm is $O(n)$. The function is called by Algorithm 2, therefore its correctness is verified through correctness verification of Algorithm 2. This algorithm terminates for the same reason as Algorithm 15.

Appendix B

Algorithms Referenced from Chapter 5

In this appendix, six algorithms are presented which are referenced from Chapter 5. They are functions called by Algorithm 5; therefore their correctness is verified throught the correctness verification of Algorithm 5.

The "WHEN" template has three instantiable sections *GuardedCondtion*, *UntilCondtion*, and *RequiredCondition*. Depending on the types of claims, these three map to different sets of elements. In **Algorithm** 18, the section *RequiredCondition* is prefixed with "prob_target", which means this claim is the type of probability. Therefore, the assertion shall be generated using template AT- 9 which has three instantiable elements of *prob_target*, *path_formula*, *constant_i*. Then, *RequiredCondition* in the requirement *r* is de-prefixed by function *removePrefix()*, and assigned to *prob_target* (line 2), and *UntilCondtion* to *path_formula* (line 3). *constant_i* is assigned the value using the de-prefixed *GuardCondtion* lines 15 - 30 of Algorithm 5. The assertion is generated partially by instantiating the AT-9 template using *prob_target*, *path_formula* in line 4 of Algorithm 18 and partially by instantiation using *constant_i* in line 29 of Algorithm 5.

Algorithm 18: Function: GENERATEPROBASST

```
1 Function GENERATEPROBASST(r, claim_ID):
2     prob_target = r.RequiredCondition.removePrefix()
3     path_formula = r.UntilCondition.removePrefix()
4     INSTPROBASST(claim_ID, prob_target, path_formula)
```

Algorithm 19 is similar to Algorithm 18 but has two more elements of *reward_event* and *reward_value* in the assertion template AT-10. These two are mapped back to the *GuardCondition* section of the claim.

Algorithm 20 follows the same pattern as the above. The assertion template is AT-11. The two instantiable elements *temporal_operator* and *path_formula* are mapped to the section *UntilCondition* of the WHEN template for the requirement *r*.

Algorithm 22 generates the assertions of untimed refinement, using the assertion template AT-1 for global invariant type, AT-2 for local invariant type, and AT-3 for local invariant with buffer involved. The algorithm first decides whether the requirement is a global or local type. If there are two elements defined in *GuardCondition* section of the requirement (line 2), the claim is a type of "local invariant".

Algorithm 19: Function: GENERATEREWARDASST

```
1  Function GENERATEREWARDASST(r, claim_ID):
2      reward_target = r.requiredCondition.removePrefix()
3      path_formula = r.untilCondition.removePrefix()
4      reward_event =
         r.guardCondition.elems.select(m|m.prefixed("reward_event")).removePrefix()
5      reward_value =
         r.guardCondition.all.elems.select(m|m.prefixed("reward_value")).removePrefix()
6      INSTREWARDASST(claim_ID, reward_target, path_formula, reward_event, reward_value)
```

Algorithm 21: Function: GENERATETEMPASST

```
1  Function GENERATETEMPASST(r, claim_ID):
2      temporal_operator = r.untilCondition.elems.select(m|m.prefixed("term_")).removePrefix()
3      path_formula =
         r.untilCondition.elems.select(m|m.prefixed("path_formula")).removePrefix()
4      INSTTEMPASST(claim_ID, temporal_operator, path_formula)
```

Then, the first element of *GuardCondition* is the scope constraint mapped to *guard_event_1* in the assertion template, the second is the local guard condition mapped to *guard_event_2*, *required_event* in the assertion is assigned the value with *RequiredCondition* (lines 3- 6). When rewriting the untimed refinement claims in Kapture, the prefix is only used to label the events that involve buffers. This prefix shall be kept in the generated assertions. Therefore, the content of the three claim sections can be used directly without removing any prefixes.

Algorithm 22: Function: GENERATEUNTIMEDASST

```
1  Function GENERATEUNTIMEDASST(r, claim_ID):
2      if r.guardCondition.elems.at(1).isdefined() then
3          guard_event_1 = r.guardCondtion.elems.at(0)
4          guard_event_2 = r.guardCondtion.elems.at(1)
5          required_event = r.requiredCondtion
6          required_event_without_value = required_event.removeValue()
7          if required_event.prefixed("buffered") then
8              INSTLOCBUFUNTIMEDASST(claim_ID, guard_event_1, guard_event_2, required_event,
                 required_event_without_value)
9          else
10             INSTLOCUNTIMEDASST(claim_ID, guard_event_1, guard_event_2, required_event,
                 required_event_without_value)
11         end
12     else
13         guard_event = r.guardCondtion
14         required_event = r.requiredCondtion
15         required_event_without_value = required_event.removeValue
16         INSTGLBUNTIMEDASST(claim_ID, guard_event, required_event, required_event_without_value)
17     end
```

Next, the algorithm decides whether the local claim involves a buffer, i.e., whether the claim involves communication with the RoboChart platform. This feature is marked by users' labelling the event using the prefix 'buffered' if an event is communicated between a controller and a platform. Then, if

the *required_event* is prefixed with 'buffered' (line 7), the requirement is the type of local invariant with buffer. The assertion shall be generated by instantiating the template AT-3 (line 8). Otherwise, the requirement is the type of local invariant without a buffer. The assertion shall be generated by instantiating the template AT-2 (line 10). If there is one element defined in *GuardCondition* (line 12), the requirement is a global type. The assertion shall be generated by instantiating AT-1 (line 16). The complexity of this algorithm is $O(1)$.

Algorithm 23 and **Algorithm** 24 also follow the same pattern of fetching the elements and instantiating the template using the elements. In **Algorithm** 24 for general assertions, the claim uses "EVERY" kapture template whose sections of *Condition* and *AlwaysOrNever* are used to identify and generate the elements of the assertion template. The complexity of these two algorithms is $O(1)$.

Algorithm 23: Function: GENERATETIMEDASST

```
1  Function GenerateTimedAsst(r, claim_ID):
2      target_event = r.defaultCondition
3      guard_event = r.guardCondition
4      trigger_event = r.untilCondition
5      target_event_without_value = target_event.removeValue
6      time_limit = r.requiredCondtion.getTimeLimit()
7      INSTTIMEDASST(claim_ID, target_event, guard_event, trigger_event,
         target_event_without_value, time_limit)
```

Algorithm 24: Function: GENERATEGENERALASST

```
1  Function GenerateTimedAsst(r, claim_ID):
2      asst_type = r.condition.getType
3      asst_time_feature = asst_type.time
4      if asst_type = reachable then
5          scope = r.condition.getScope
6          state = r.condition.getState
7          bool_term = c.alwaysOrNever.generateBool()
8          INSTREACHASST(claim_ID, scope, state, bool_term, asst_time_feature)
9      end
10     if asst_type = deadlock_free then
11         scope = r.condition.getScope
12         bool_term = r.alwaysOrNever.generateBool()
13         INSTDDLKFASST(claim_ID, scope, bool_term, asst_time_feature)
14     end
15     if asst_type = divergence then
16         scope = r.condition.getScope
17         bool_term = r.alwaysOrNever.generateBool()
18         INSTDIVFASST(claim_ID, scope, bool_term, asst_time_feature)
19     end
20     if asst_type = terminate then
21         scope = r.condition.getScope
22         bool_term = r.alwaysOrNever.generateBool()
23         INSTTMNTASST(claim_ID, scope, bool_term, asst_time_feature)
24     end
```

Appendix C

Algorithms Referenced from Chapter 6

This appendix presents the algorithms referenced from Chapter 6.

Algorithm 25

Section 6.5.2 lists the steps to query RoboChart models and transform them into Z Machine notations. Algorithm 25 is designed for Step 1: generation of enumeration types for states and events by searching all the nodes, events of the state machine.

The states and junctions are the instantiation of class "node" in RoboChart metamodel. By querying all the nodes, a collection of nodes which are separated by "|" is created as a string. This string *St* is used to replace the placeholder $< St >$ at line 12 of the EGL template in Listing 6.3 for further model-to-text transformation.

Similarly, if there is communication through events for the state machine, a collection of events which are separated by "|" is created as a string *Evt*. This string is used to replace the placeholder $< Evt >$ at line 14 of the template (Listing 6.3).

Algorithm 25: Enumtype definition for states and events

```
1 for n in machine.nodes do
2 |   St = St + n.name + "|"
3 end
4 if ¬EMPTY(machine.events) then
5 |   for e in machine.events do
6 |   |   Evt = Evt + e.name + "|"
7 |   end
8 end
```

Algorithm 26

Algorithm 26 is designed for Step 2. For the primitive types defined in RoboChart, a string *PrimT* is created to define a HOL primitive type. All the types are assigned as a *nat* as default. For the record types defined in RoboChart, a string *record* is created to define the name of a HOL record type. The field of the record is composed of the field name and field type. For the enumeration types defined in RoboChart, a string *Enum* is created to define the name of a HOL enumeration type. A string

Enum_literals is created to define the literals of an Isabelle enumeration type which are separated by |. These three type declarations are directly mapped to the HOL type declarations. Algorithm 26 converts the formats of the declarations.

As stated in Rule 2 of Section 6.3, the primitive datatypes of RoboChart include *nat*, *int*, *boolean*, *real*, *char*. These types map to HOL types of *nat*, *int*, *bool*, *real*, *string*. The function *zmType* in line 12 is used to convert these RoboChart types to HOL types. The complexity of this algorithm is $O(n^2)$.

Algorithm 26: Type Declaration

```
1  for t in package.types do
2      if t.type = "PrimitiveType" then
3          Primtype = t.name
4          type = "nat"
5      end
6  end
7  for t in package.types do
8      if t.type = "RecordType" then
9          Record = t.name
10         for f in t.fields do
11             field = f.name
12             type = f.type.zmType
13         end
14     end
15 end
16 for t in package.types do
17     if t.type = "Enumeration" then
18         EType = t.name
19         for l in t.literals do
20             literals = literals + l + "|"
21         end
22     end
23 end
```

Algorithm 27

Algorithm 27 implements Step 3 of Section 6.5.2, which queries the constants defined for or required by the state machine. Also, a set shall be generated as a constant for the parameter in an input event. As shown in line 34 of the EGL template, the value of the input parameter shall be selected from a set which is defined as a constant.

Both variables and constants in RoboChart models are stored in the *varialeList*. To queries the constants defined for or required by the state machine, the *varialeList* that stores constants shall be identified first. Then, each constant in the list is converted to a Z Machine constant. The algorithm is given in Algorithm 27.

Algorithm 28

To define the constants of the set type for input events, all the transitions and their target states are queried to identify the input events. The algorithm is given in Algorithm 28. For a transition t, its trigger can be an input event (line 2), it may have an input event as a transition action (line 7), or its

Algorithm 27: Constant definition for RoboChart constants

```
1 for varlist in machine.varialeList do
2     if varlist stores "CONST" then
3         for c in varlist.vars do
4             Consts = c.name
5             type = c.zmType
6             consts = Consts + " :: " + type
7         end
8     end
9 end
```

target state may have an event as an state action (line 24).

There are four types of communication in RoboChart including INPUT, OUTPUT, SIMPLE, and SYNC. Currently, only INPUT type is modelled in Z Machine. The class of *Communication* has the attributes of *parameter*, *type*, etc. In the RoboChart metamodel, *trigger* of a transition is an instantiation of class *Communication*. Thus, the event of INPUT type can be identified using the condition $trigger.type = INPUT$ (line 2), the parameter of a trigger event can be obtained by *parameter.name* (line 3), and its datatype by *parameter.type*(line 4). Again, *zmType* function is used to converted the RoboChart types into Z Machine types.

An event in a transition action is an instantiation of class *CommunicationStmt* (line 8)which has an attribute of *communication* of *Communication* Class. Thus, the event of INPUT type can be identified using the condition $t.action.communication.type = INPUT$ (line 8), the parameter of the event can be obtained by $t.action.communication.parameter.name$ (line 9, and its datatype by *parameter.type* (line 10). However, it is possible to have multiple transition actions in which case, each action shall be checked to identify the events (line 13).

For the events in the state actions, the event is an instantiation of class *CommunicationStmt*. So its query is similar to the events in the transition actions. It is also possible to have several actions in a state, and each action shall be checked to identify the events in the actions (line 29).

Only target state of a transition needs to be considered because each state except the initial state shall be the target state of at least one transition. The initial state is not a target of a transition, but it also has no actions. The complexity of this algorithm is $O(n^2)$.

Algorithm 29

Step 4 of Section 6.5.2 queries the functions with or without the concrete definitions. The automated transformation generates the name *fun_name* and types *fun_type* of the functions as shown in Algorithm 29. The concrete definition of the functions *definitionOfFun* in the template (line 22) is input manually by users.

Algorithm 30

Step 5 of Section 6.5.2 transforms RoboChart variables to the store variables in **zstore** section as shown in line 28 of the template. Other elements of the state space including observational variables and healthiness conditions as part of structural invariant are embedded in the template as default. The structural invariants which are state machine specific as shown in line 30 of the template shall

Algorithm 28: Constant definition for event parameters

```
1  for t in machine.transitions do
2      if t.trigger.isDefined and t.trigger.type= "INPUT" then
3          Consts = t.trigger.parameter.name
4          type = (t.trigger.parameter.type.zmType) set
5          consts_type = Consts + " :: " + type
6      end
7      if t.action.isDefined then
8          if t.action.type= "CommunicationStmt" and t.action.communication.type= "INPUT" then
9              Consts = t.action.communication.parameter.name
10             type = (t.action.communication.parameter.type.zmType) set
11             consts_type = Consts + " :: " + type
12         end
13         if t.action.type = "SeqStatement" then
14             for s in t.action.statements do
15                 if s.type= "CommunicationStmt" and s.communication.type= "INPUT" then
16                     Consts = s.communication.parameter.name
17                     type = (s.communication.parameter.type.zmType) set
18                     consts_type = Consts + " :: " + type
19                 end
20             end
21         end
22     end
23     if t.target.actions.isDefined then
24         if t.target.actions.action.type= "CommunicationStmt" and
            t.target.actions.action.communication.type= "INPUT" then
25             Consts = t.target.actions.action.communication.parameter.name
26             type = (t.target.actions.action.communication.parameter.type.zmType) set
27             consts_type = Consts + " :: " + type
28         end
29         if t.target.actions.action.type = "SeqStatement" then
30             for s in t.target.actions.action.statements do
31                 if s.type= "CommunicationStmt" and s.communication.type= "INPUT" then
32                     Consts = s.communication.parameter.name
33                     type = (s.communication.parameter.type.zmType) set
34                     consts_type = Consts + " :: " + type
35                 end
36             end
37         end
38     end
39 end
```

be manually input by the user.

The algorithm for Step 5 is given in Algorithm 30. Same as constants, variables in RoboChart models are stored in the *varialeList*. To query the variables defined for or required by the state machine, the attribute $modifier$ of *varialeList* shall be "VAR" which means this list stores variables.

Algorithm 31

Step 6 of Section 6.5.2 transforms RoboChart transitions to operations. The corresponding template

Algorithm 29: Algorithm_Function

```
1  for f in package.functions do
2      fun_name = f.name
3      if f.parameters.size = 1 then
4          fun_type = f.parameter.type.zmType + " ⇒ " + f.type.zmType
5      end
6      if f.parameters.size > 1 then
7          for p in f.parameters do
8              fun_type = fun_type + f.parameter.type.zmType + " ⇒ "
9          end
10         fun_type = fun_type + f.type.zmType
11     end
12 end
```

Algorithm 30: Algorithm_Variable

```
1  for varlist in machine.varialeList do
2      if varlist stores "VAR" then
3          for v in varlist.vars do
4              stateVar = v.name
5              type = v.zmType
6          end
7      end
8  end
```

is line 33 to line 42. The algorithm is given in Algorithm 31.

The algorithm starts with defining a global variable *opCollection* (line 1) which is used to store all of the names of operations generated. *opSet* is a global variable of *set* type that will be used in Algorithm 40 to generate the verification lemmas. It stores a set of tuples which have the elements of the name, the existence of the parameters, and the parameters of the operations. It then calls function CREATOPNAME to generate the operation name and passes *opCollection* to the funciton.

Variable *zop_tuple* is defined as a tuple. It has two elements of *hasPara* as a Boolean representing if there is parameter communicated in this operation, and *paras* as a set to collect all the parameters.

The algorithm futher calls the function CREATEPARA (line 9) to generate the instantiable element *params* of the template (line 34) by searching all the input events occurred during the operation. The variable *zop_tuple* is passed to function CREATEPARA to assign the values to the two elements of the tuple. Whenever there is an input event during the operation, *zop_tuple.hasPara* is set to *true*, and the parameter of the event is added into *zop_tuple.paras* (e.g., line 5 - 6 of Algorithm 33).

The algorithm (line 11 to line 16) uses the variable *zop_tuple* updated from function CREATEPARA to generate $<$ *parameter* $>$ and $<$ *parameter_input* $>$ of the template . It firstly check if there is a parameter for communication (line 11), if so, then for each parameter, it generates a pair of string *parameter* and *parameter_input*. This follows the naming convention defined in Rule 8.1 of Section 6.3.4 that the parameter in the `update` section uses the name of the RoboChart parameter, and the parameter in the `params` section has a suffix "*_input*" to differentiate from the parameter in the `update` section which is further updated through assigning the value of *parameter_input*.

Algorithm 31: Algorithm_Operation

```
1  opCollection = " "
2  opSet = {}
3  var zop_tuple
4  zop_tuple.hasPara = false
5  zop_tuple.paras = {}
6  for t in machine.transitions do
7      op_name = CREATEOPNAME(t, opCollection, opSet)                ▷ Alg. 32
8      zop_tuple.name = op_name
9      params = CREATEPARA(t, zop_tuple)                             ▷ Alg. 33
10     opSet = opSet ∪ {zop_tuple}
11     if zop_tuple.hasPara then
12         for p in zop_tuple.paras do
13             parameter = p
14             parameter_input = p + "_input"
15         end
16     end
17     scr_st = t.source.name
18     tran_cond = CREATETRANCOND(t)                                 ▷ Alg. 35
19     tgt_st = t.target.name
20     trg_events = CREATETRGEVTS(t)                                 ▷ Alg. 36
21     tr = [ ]
22     stateVarAssign = " "
23     CREATETR(t, tr, stateVarAssign, zop_tuple)                    ▷ Alg. 37
24 end
```

The algorithm generates $< scr_st >$ for `pre` section and $< tgt_st >$ `update` section using the source state and the target state of the transition respectively (line 17, line 19).

The algorithm calls another three functions CREATETRANCOND, CREATETRGEVTS, and CREATETR for generating the instantiable elements including $< tran_cond >$, $< trg_events >$, and $< tr >$ respectively. Two global variables tr and *stateVarAssign* are passed to function CREATETR to define their values. The complexity of this algorithm is $O(n^2)$.

Algorithm 32

The Algorithm 32 describes the CREATEOPNAME function called in Algorithm 31, i.e., how the name of the operation *op_name* in the template (line 33) is generated for one transition. Generally, *op_name* is simply represented by using the source state and the target state of the transition as shown in line 2. However, it is common that there are multiple transitions between a source and a target state in which case an index (the variable *i* in line 3) to the operation name is used to distinguish the transitions. For example, three transitions from state A to state B can be named as *AToB*, *AToB*_1, *AToB*_2.

In the function, every time a valid name is generated for an operation, it is stored in a global string *opCollection*. The *While* loop in line 4 used this string to determine whether and which index number shall be used. If the name generated in line 2 has been stored in the string *opCollection*, it means that a transition has used this name. Then, the name is updated by adding an index to the end of the name, and the while loop continues the check. If the updated name is also stored in the string, the index is increased by one, and used to update the name. The while loop stops until the name is

not used for any other operations.

Algorithm 32: Function: CREATEOPNAME

```
1  Function createOpName(t, opCollection, opSet):
       Result: op_name
2      op_name = t.source.name + "To" + t.target.name
3      i = 1
4      while op_name is substring of opCollection do
5          op_name = t.source.name + "To" + t.target.name + "_" + i.asString()
6          i++
7      end
8      opCollection = opCollection + " " + op_name
9      return op_name
```

Algorithm 33

Algorithm 33: Function: CREATEPARA

```
1  Function createPara(t, zop_tuple):
       Result: (params)
2      params = ""
3      if t.trigger.isDefined and t.trigger.type= "INPUT" then
4          params = params + t.trigger.parameter.name + "_input ∈
             " + t.trigger.parameter.name.firstToUpperCase
5          zop_tuple.hasPara = true
6          zop_tuple.paras.add(t.trigger.parameter.name)
7      end
8      if t.action.isDefined then
9          params = CREATEPARAFORACTION(t, zop_tuple)                ▷ Alg. 34
10     end
11     if t.target.actions.isDefined and (t.target.actions.isTypeOf(Entry) or
         t.target.actions.isTypeOf(During)) then
12         params = CREATEPARAFORACTION(t.target.actions, zop_tuple)
13     end
14     if t.source.actions.isDefined and t.source.actions.isTypeOf(Exit) then
15         params = CREATEPARAFORACTION(t.source.actions, zop_tuple)
16     end
17     return params
```

Algorithm 33 defines *params* in line 34 of the template 6.3. Similar to Algorithm 28, the process identifies the input events. However, it distinguishes the events from the source and target states. In Algorithm 28, only events from target state of the transitions are considered because it is sufficient to cover all the events in the state machine. For *params*, the algorithm searches for the input events of entry and during actions in the target state, and of the exit actions in the source state of the transition, i.e., all the input events occurred within the operation. The complexity of this algorithm is $O(1)$.

Algorithm 34

Algorithm 34: Function: CREATEPARAFORACTION

```
1  Function CreateParaForAction(t, zop_tuple):
       Result: (params)
2      if t.action.type= "CommunicationStmt" and t.action.communication.type= "INPUT"
        then
3          params = params + t.action.communication.parameter.name + "_input ∈
            " + t.action.communication.parameter.name.firstToUpperCase
4          zop_tuple.hasPara = true
5          zop_tuple.paras.add(t.action.communication.parameter.name)
6      end
7      if t.action.type = "SeqStatement" then
8          for s in t.action.statements do
9              if s.type= "CommunicationStmt" and s.communication.type= "INPUT" then
10                 params = params + s.communication.parameter.name + "_input ∈
                    " + s.communication.parameter.name.firstToUpperCase
11                 zop_tuple.hasPara = true
12                 zop_tuple.paras.add(t.action.communication.parameter.name)
13             end
14         end
15     end
16     return params
```

Algorithm 34 is called in Algorithm 33 to generate parameters involved in the actions. The actions can be a transition action or a state action. The complexity of this algorithm is $O(n)$.

Algorithm 35

Algorithm 35 defines the function CREATETRANCOND which converted the transition conditions to preconditions of an operation. The funciton returns the string *tran_cond* that is used to instantiate *tran_cond* in line 35 of the template. The function calls another funciton BINARYEXPSTRING to convert the RoboChart expressions into Z Machine expressions.

Algorithm 35: Function: CREATETRANCOND

```
1  Function createTranCond(t):
       Result: (tran_cond)
2      if t.condition.isDefined then
3          tran_cond = BINARYEXPSTRING(t.condition)
4      end
5      return tran_cond
```

Algorithm 36

Algorithm 36 defines the function CREATETRGEVTS which identifies the transitions whose source state is the target state of this transition (line 3), and for each relevant transition, collects all its trigger events. The function return a string *trg_events* that is used to instantiate *trg_events* in line 38 of the template.

Algorithm 36: Function: TRGEVTS

```
1  Function CREATETRGEVTS(t):
       Result: trg_events
2      i = 0
3      for tr in machine.transitions.select(i | i.source = t.target) do
4          if tr.trigger.isDefined and i = 0 then
5              trg_events = tr.trigger.event.name
6              i++
7          else
8              if tr.trigger.isDefined and i>0 then
9                  trg_events = trg_events + ", " + tr.trigger.event.name
10                 i++
11             end
12         end
13     end
14     return trg_events
```

Algorithm 37

Algorithm 37 defines the function CREATETR which assigns values to the global variables tr and *stateVarAssign*. The function has four parameters t as the transition, tr as the operation behaviour concerning events and target state, *stateVarAssign* to show the state variable update, and *zop_tuple* which is used in the assignment of state variables through functions.

Algorithm 37: Function: TR

```
1  Function createTr(t, tr, stateVarAssign, zop_tuple):
       Result: (tr, stateVarAssign)
2      if t.trigger.event.isDefined then
3          tr = tr + "@[Event " + t.trigger.event.name + "]"
4      end
5      if t.source.actions.isDefined and t.target.actions.isTypeOf(Exit) then
6          tr = CREATETRFORACTION(t.source.actions, tr, stateVarAssign, zop_tuple)   ▷
             Alg. 38
7      end
8      if t.action.isDefined then
9          tr = CREATETRFORACTION(t, tr, stateVarAssign, zop_tuple)
10     end
11     if t.target.actions.isDefined and (t.target.actions.isTypeOf(Entry) or
         t.target.actions.isTypeOf(During) ) then
12         tr = CREATETRFORACTION(t.target.actions, tr, stateVarAssign, zop_tuple)
13     end
14     tr = tr + "@[State " + t.target.name + "]"
```

There are five types of elements in tr including the trigger event of the transition, the events in the *Exit* action of the source state, the events in the transition actions, the events in the *Entry* or *During* actions of the target state, and the target state the transition enters. Besides the events, the actions

in the source state, the transition and the target state can have assignments and operations. The assignments are to update the state variables and are represented by the string variable *stateVarAssign* which is used to instantiate the element $< stateVarAssign >$ in the template (line 40). The operations are called during the actions, and they are not transformed into Z Machines.

The algorithm first queries the trigger events to add the first element to tr in this transition (line 2), then moves to the actions of the source state. If there is *Exit* action in the source state (line 5), an action of the transition (line 5), or an entry or during action in the target state (line 5), the function CREATETRFORACTION in Algorithm 38 is called.

The order to query the trigger, the source state, the transition, and the target state shall be kept in order to show a trace of the operation's behaviour. In each operation, tr is ended with the target state that the transition enters (line 14). The complexity of this algorithm is $O(1)$.

Algorithm 38

Algorithm 38: Function: CREATETRFORACTION

```
1  Function createTrForAction(t, tr, stateVarAssign, zop_tuple):
       Result: (tr, stateVarAssign)
2      if t.action.isTypeOf(SeqStatement) then
3          for s in t.action.statements do
4              if s.isTypeOf(Assignment) then
5                  stateVarAssign = s.left + "'= " + s.right.oneSideOfBinaryString
6                  for p in zop_tuple.paras do
7                      if p isSubstringOf (stateVarAssign ) then
8                          stateVarAssign = stateVarAssign.replace(p, (p + "_input"))
9                      end
10                 end
11             end
12             if s.isTypeOf(CommunicationStmt) then
13                 tr = tr + "@[Event " + s.communication.event.name + "]"
14             end
15         end
16     end
17     if t.action.isTypeOf(CommunicationStmt) then
18         tr = tr + "@[Event " + t.action.communication.event.name + "]"
19     end
20     if t.action.isTypeOf(Assignment) then
21         stateVarAssign =
             t.action.left + "'= " + t.action.right.oneSideOfBinaryString
22         for p in zop_tuple.paras do
23             if p isSubstringOf (stateVarAssign ) then
24                 stateVarAssign = astateVarAssign.replace(p, (p + "_input"))
25             end
26         end
27     end
```

In Algorithm 38, three types of actions are considered including (1) *SeqStatement* type (line 2)

meaning there are several actions, (2) *CommunicationStmt* (line 17) type indicating this action is an event, (3) *Assignment* (line 20) type which means the action is an update of a state variable by assignment. For the action of *SeqStatement* type, each statement in the actions are further checked to identify whether it is an event or an assignment. For the assignments, *stateVarAssign* is assigned with new value by converting the RoboChart assignment statement into the Z Machine format. The function ONESIDEOFBINARYSTRING is called to convert the RoboChart expressions into Z Machine expressions. The events identified are added into the list tr to record the operation's behaviour. The complexity of this algorithm is $O(n^2)$.

Algorithm 39

Algorithm 39 defines the initialization and the machine. There are three placeholders to be instantiated in the template of Listing 6.3 (line 3 to line 55) including the names of the machine $< StateMachine >$, the state variables $< stateVar >$, the name of the operations $< operations >$.

Similar to Algorithm 30, all the state variables are listed (line 6) in the initialization, but their value are manually assigned. This gives a starting point for the machine's operation. $< operations >$ are assigned the value of the string variable *opCollection* which is defined in Algorithm 31 where *opCollection* collects the names of all the operations with a space to separate them.

Algorithm 39: Initialization and Machine

```
1  for stm in StateMAchineDef.allInstances do
2  |  StateMachine = stm.name
3  end
4  for varlist in machine.varialeList do
5  |  if varlist stores "VAR" then
6  |  |  for v in varlist.vars do
7  |  |  |  stateVar = v.name
8  |  |  end
9  |  end
10 end
11 operations = opCollection                    ▷ opCollection defined in Alg. 31
```

Algorithm 40

Algorithm 40 generates the instantiable elements for Step 8 of Section 6.5.2, which is used for the verification of the structural invariants and the system requirements (line 45 to line 18).

Algorithm 40: Verification

```
1  for tuple in opSet do
2  |  op_name = tuple.name                      ▷ opSet defined in Alg. 31
3  |  if tuple.hasPara then
4  |  |  op_name_() = tuple.name + tuple.paras.TOPARATHSTRING
5  |  else
6  |  |  op_name_() = tuple.name + "( )"
7  |  end
8  end
```

The structural invariants given in the **zstore** section are automatically grouped and named as $<$

$StateMachine > _inv$. The system requirements shall be manually formalized and presented with the keyword `expr`.

To verify that the state machine satisfies an invariant (i.e., the structural invariants or the system requirement), a lemma that shows the initialization establishes the invariant is generated as default in the template, the lemmas that shows the operations preserve the invariant are generated by providing the name of each operation $< op_name >$ / $< op_name_() >$ to the template.

opSet is a set of tuples that is defined in Algorithm 31. Algorithm 40 queries each tuple of the set *opSet* (line 1), and assigns the *name* of the tuple to op_name. Then, the algorithm uses the elements *haspara* of the tuple to determine if parameters are involved in an operation (line 3). If so, the parameters shall be presented in the parentheses as shown in the template (line 18 of Listing 6.4).

The function toPARATHSTRING converts the set $tuple.paras$ to a single string that uses a space to separate all the parameters of the operation. If no parameter of input events is involved in the operation, the parentheses is left empty (line 7). For the template of deadlock freedom proving, only the name of the RoboChart state machine $< StateMachine >$ is required. This has been generated in Algorithm 39.

Appendix D

Templates for Z Machine Notation

The basic Z Machine template is given in Listing D.1. The template comprises five sections: the definition of types, constants, and functions; the state space; the operations on the state space; the zmachine; and the verification of invariants. The values within the brackets '<>' are to be replaced with specific values.

Listing D.1: A general Z Machine template

```
subsection < Types, constants, functions >
type_synonym <type_name> = <type>
record <Record> =
  (<field>:: "<type>")+
enumtype <Etype> = <literal> (| <literal>)+
consts <Consts> :: "<type>"
consts <fun_name> :: "<type> (⇒ <type>)+"

fun <fun_name> :: "<type> (⇒ <type>)+"
    where "<fun_name> para = <definitionOfFun>"

subsection < State space >
zstore <Zstore> = (<stateVar> :: "<type>")+
                  (where "(Invariant)+")?

subsection <operations>
(zoperation <Zop> =
  (params <parameter> ∈ "<PARAMETER>")?
  (pre "<precond> (∧ <precond>)*" )?
  (update "[<stateVar>′ = <stateVar_new_value>
          (,<stateVar>′ = <stateVar_new_value>)*
          ]" )? )+

subsection < Initialization, machine>
definition Init::"<Zstore> subst" where
"Init = "[<stateVar>⇝ <Val>
        (,<stateVar>⇝ <Val> )* ]"

zmachine <Zstore>Machine =
    over <Zstore>
    init Init
    invariant <Zstore>_inv
    operations (<Zop>)*

subsection < Structural Invariants >
lemma Init: "Init establishes <Zstore>_inv"
   by (zpog_full; auto)

(lemma <Zop>_int: "<Zop>() preserves <Zstore>_inv"
   by (zpog_full; auto))+

subsection < Requirements >
zexpr Req_1 is "<formalized_requirement>"

lemma Init: "Init establishes Req_1 under <Zstore>_inv"
   by (zpog_full; auto)

(lemma <Zop>_int: "<Zop>() preserves Req_1 under <Zstore>_inv"
   by (zpog_full; auto))+

subsection < Deadlock freedom >
definition [z_defs]: "<zstore>_axioms = <>"

lemma deadlock_free: "<zstore>_axioms  ⇒ deadlock_free <zstore>Machine"
  apply deadlock_free
  sledgehammer
```

Appendix E

Z Machine notations for LRE state machine

In this appendix, the automatically generated Z Machine notations as well as the verification process for LRE are presented in the following list.

```
theory LRE
imports "Z_Machines.Z_Machine"
begin

notation undefined ("???")

subsection ⟨ Introduction ⟩

datatype ('s, 'e) tag =
  State (ofState: 's) | Event (ofEvent: 'e)

abbreviation "is_Event x ≡ ¬ is_State x"

type_synonym ('s, 'e) rctrace = "('s, 'e) tag list"

definition wf_rcstore :: "('s, 'e) rctrace ⇒ 's ⇒ 's option ⇒ bool" where
[z_defs]: "wf_rcstore tr st final =  (
     length(tr) > 0
   ∧ tr ! ((length tr) -1) = State st
   ∧ ( final ≠ None ⟶ (∀i<length tr. tr ! i = State (the final) ⟶ i= (length tr)
       ↪ -1))
   ∧ (filter is_State tr) ! (length (filter is_State tr) -1) = State  st)"


subsection ⟨ type_definition ⟩

enumtype St = OCM | MOM | HCM | CAM | initial

definition [z_defs]: "St = {OCM, MOM, HCM, CAM, initial}"


enumtype Evt = advVel | reqHCM | reqOCM | reqMOM | endTask | reqVel

definition "Evt = {advVel, reqHCM, reqOCM, reqMOM, endTask, reqVel}"
```

```
34
35 type_synonym coord="real×real"
36
37 record Obstacle =
38   obspos :: coord
39   id :: nat
40
41 consts Obsts :: " coord list"
42
43 consts Positions::"(coord) set"
44
45 consts Velocities:: "(real×real) set"
46
47 consts ReqV:: "(real×real) set"
48
49 consts HCMVel:: "real"
50
51 consts MOMVel:: "real"
52
53 consts MinSafeDist :: "real"
54
55 consts Opez_min:: "coord"
56
57 consts Opez_max:: "coord"
58
59 consts SafeVel :: "real"
60
61 consts ZeroVel:: "coord"
62
63 text ⟨ function_definition ⟩
64
65 fun inOPEZ:: "coord⇒ bool"
66   where "inOPEZ (x,y) = ( x≥ fst Opez_min ∧ x< fst Opez_max ∧ y≥snd Opez_min ∧
       ↪ y< snd Opez_max)"
67
68 fun single_dist:: " coord × coord  ⇒ real"
69   where "single_dist((x,y), (m,n)) =(x-m)^2+ (y-n)^2"
70
71 fun dist:: " coord × (coord list) ⇒ real"
72   where
73 "dist((x,y),[]) = 200^2+ 200^2" |
74  " dist((x,y), g#gs) =( if  single_dist((x,y),g) ≤ dist ((x,y),gs) then
       ↪ single_dist((x,y),g) else dist ((x,y),gs))"
75
76 fun obst_index:: "coord  × (coord list) ⇒ nat"
77   where
78 "obst_index ((x,y), [])=100 " |
79  "obst_index ((x,y), g#gs)= (if single_dist ((x,y),g) = dist ((x,y),g#gs) then 0
       ↪ else  (obst_index ((x,y), gs)+1))"
80
81
82 fun abslt:: "real⇒ real"
83   where "abslt(x) = (if x≥0 then x else -x)"
84
85 fun closestObs_xpos :: "coord × (coord list) ⇒ real"
86   where "closestObs_xpos((xp,yp),[]) = 1000000"|
87         "closestObs_xpos((xp,yp),g#gs) = fst ((g#gs) ! obst_index((xp,yp),
              ↪ g#gs))"
88
```

```
fun closestObs_ypos :: "coord × (coord list) ⇒ real"
  where "closestObs_ypos((xp,yp),[]) = 1000000"|
        "closestObs_ypos((xp,yp),g#gs) = snd ((g#gs) ! obst_index((xp,yp),
            ↪ g#gs))"


fun CDA :: " coord × (coord list)× (real × real) ⇒ real"
  where "CDA((xp,yp),[], (xv,yv)) = (10+MinSafeDist+5)^2" |
"CDA ((xp,yp), g#gs, (xv,yv)) =
(if xv≠0 ∧ yv=0
 then
  (if (closestObs_xpos((xp,yp),g#gs)- xp) * xv≥0
   then dist((xp,yp),g#gs) -(closestObs_xpos((xp,yp),g#gs) - xp)^2
   else dist((xp,yp),g#gs)  )
else
  (if xv=0 ∧ yv≠0
   then
    (if (closestObs_ypos((xp,yp),g#gs) - yp) * yv≥0
     then dist((xp,yp),g#gs)- (closestObs_ypos((xp,yp),g#gs) - yp)^2
     else dist((xp,yp),g#gs)  )
   else dist((xp,yp),g#gs)  )
)"


fun maneuv :: "real× real ⇒ real× real"
  where "maneuv(x,y) = (y,-x)"


fun setVel :: "(real × real) × real   ⇒   (real × real) "
  where
  "setVel((xv, yv), setpoint) =
  (if xv=0
  then
   (if yv>0
    then (0, setpoint)
    else  ( if yv<0 then(0, (- setpoint) )
              else (setpoint, 0) ) )
  else
   (if xv>0
    then (setpoint, 0)
    else ((-setpoint), 0) ) )"

subsection ⟨ State Space ⟩

instantiation real :: default
begin
definition "default_real = (0::real)"
instance ..
end

zstore LRE_Beh =
  pos:: "coord"
  xvel :: "real"
  yvel :: "real"
  reqV:: "real × real"
  advV::"real × real"
  st::"St"
  tr :: "(St, Evt)tag list"
  triggers:: "Evt set"
```

```
  where inv:
        "wf_rcstore tr st None"

subsection ⟨Operations⟩

zoperation Move =
update "[pos′ =(fst(pos) + xvel, snd(pos) + yvel)
        ]"

zoperation InitialToOCM =
  pre "st= initial"
  update "[ st′ = OCM
          , tr′ =tr @ [State OCM]
          , triggers′ = {reqMOM, reqVel}        ]"

zoperation OCMToMOM =
  pre "st= OCM  ∧ ( xvel^2 +  yvel^2)≤  SafeVel^2 ∧ dist(pos,Obsts)>
      ↪ (MinSafeDist+10)^2 ∧ ¬inOPEZ(pos)"
  update "[ st′ = MOM
          , tr′ =tr  @ [Event reqMOM]  @ [Event advVel]@ [State MOM]
          , advV′ = setVel((xvel, yvel), MOMVel)
          , (xvel,yvel)′ = setVel((xvel, yvel), MOMVel)
          , triggers′ = {endTask, reqOCM}
          ]"

zoperation MOMToOCM =
  pre "st= MOM"
  update "[ st′ = OCM
          , tr′ =tr @ [Event reqOCM] @ [State OCM]
          , triggers′ = {reqMOM, reqVel}
          ]"

zoperation MOMToOCM_1 =
  pre "st= MOM ∧ inOPEZ(pos) ∧ (dist(pos,Obsts)>  (MinSafeDist+5)^2  ∨
      ↪ CDA(pos,Obsts, (xvel,yvel))>  MinSafeDist^2)"
  update "[ st′ = OCM
          , tr′ =tr @ [State OCM]
          , triggers′ = {reqMOM, reqVel}
          ]"

zoperation MOMToOCM_2 =
  pre "st= MOM "
  update "[ st′ = OCM
          , tr′ =tr @ [Event endTask] @ [Event advVel] @ [State OCM]
          , advV′ = ZeroVel
          , (xvel,yvel)′ = ZeroVel
          , triggers′ = {reqMOM, reqVel}
          ]"

zoperation HCMToOCM =
  pre "st= HCM "
  update "[ st′ = OCM
          , tr′ =tr @ [Event reqOCM] @ [State OCM]
          , triggers′ = {reqMOM, reqVel}
          ]"

zoperation HCMToOCM_1 =
  pre "st= HCM ∧ inOPEZ(pos)∧ (dist(pos,Obsts)>  (MinSafeDist+5)^2  ∨
      ↪ CDA(pos,Obsts, (xvel,yvel))>  MinSafeDist^2)"
```

```
  update "[ st′ = OCM
          , tr′ =tr @ [State OCM]
          , triggers′ = {reqMOM, reqVel}
          ]"

zoperation MOMToHCM =
  over LRE_Beh
  pre "st= MOM ∧  ( xvel^2 +  yvel^2)>  SafeVel^2 ∧  dist(pos,Obsts)≤
      ↪ (MinSafeDist+5)^2  ∧ CDA(pos,Obsts, (xvel,yvel))>  MinSafeDist^2"
  update "[ st′ = HCM
          , tr′ =tr @ [Event advVel] @ [State HCM]
          , advV′ = setVel((xvel, yvel), HCMVel)
          , (xvel,yvel)′ = setVel((xvel, yvel), HCMVel)
          , triggers′ = {reqOCM}
          ]"

zoperation HCMToMOM =
  over LRE_Beh
  pre "st= HCM ∧ dist(pos,Obsts)>  (MinSafeDist+5)^2   ∧ ¬inOPEZ(pos) "
  update "[ st′ = MOM
         , tr′ =tr @ [Event advVel] @ [State MOM]
         , advV′ = setVel((xvel, yvel), MOMVel)
         , (xvel,yvel)′ =  setVel((xvel, yvel), MOMVel)
         , triggers′ = { endTask, reqOCM}
         ]"

zoperation OCMToOCM =
  params reqV_input ∈ " ReqV"
  pre "st= OCM "
  update "[ st′ = OCM
          , tr′ =tr @ [Event reqVel] @ [Event advVel] @[State OCM]
          ,reqV′ = reqV_input
          , advV′ = reqV_input
          , (xvel,yvel)′ = reqV_input
          , triggers′ = {reqMOM, reqVel}
          ]"

zoperation HCMToCAM =
  pre "st= HCM ∧ CDA(pos,Obsts, (xvel,yvel))≤  MinSafeDist^2 ∧dist(pos,Obsts) ≤
      ↪ (MinSafeDist+5)^2 "
  update "[ st′ = CAM
          , tr′ =tr @ [Event advVel] @ [State CAM]
          , advV′ = maneuv(xvel, yvel)
          , (xvel,yvel)′ = maneuv(xvel, yvel)
          , triggers′ = {reqOCM}
          ]"

zoperation HCMToCAM_1 =
  pre "st= HCM ∧ (-100> (fst(pos) + xvel) ∨ (fst(pos) + xvel) >100 ∨  -100>
      ↪ (snd(pos) + yvel) ∨ (snd(pos) + yvel) >100)"
  update "[ st′ = CAM
          , tr′ =tr @ [Event advVel] @ [State CAM]
          , advV′ = maneuv(xvel, yvel)
          , (xvel,yvel)′ = maneuv(xvel, yvel)
          , triggers′ = {reqOCM}
          ]"

zoperation MOMToCAM =
  pre "st= MOM ∧ CDA(pos,Obsts, (xvel,yvel))≤  MinSafeDist^2 ∧dist(pos,Obsts)≤
```

```
      ↪ (MinSafeDist+5)^2 "
  update "[ st′ = CAM
          , tr′ =tr @ [Event advVel] @ [State CAM]
          , advV′ = maneuv(xvel, yvel)
          , (xvel,yvel)′ = maneuv(xvel, yvel)
          , triggers′ = {reqOCM}
          ]"

zoperation MOMToCAM_1 =
  pre "st= MOM ∧ (-100> (fst(pos) + xvel) ∨ (fst(pos) + xvel) >100 ∨  -100>
      ↪ (snd(pos) + yvel) ∨ (snd(pos) + yvel) >100)"
  update "[ st′ = CAM
          , tr′ =tr @ [Event advVel] @ [State CAM]
          , advV′ = maneuv(xvel, yvel)
          , (xvel,yvel)′ = maneuv(xvel, yvel)
          , triggers′ = {reqOCM}
          ]"

zoperation CAMToCAM =
  pre "st= CAM ∧ CDA(pos,Obsts, (xvel,yvel))≤  MinSafeDist^2 ∧dist(pos,Obsts)≤
      ↪ (MinSafeDist+5)^2 "
  update "[ st′ = CAM
          , tr′ =tr @ [Event advVel] @ [State CAM]
          , advV′ = maneuv(xvel, yvel)
          , (xvel,yvel)′ = maneuv(xvel, yvel)
          , triggers′ = {reqOCM}
          ]"

zoperation CAMToCAM_1 =
  pre "st= CAM ∧  (-100> (fst(pos) + xvel) ∨ (fst(pos) + xvel) >100 ∨  -100>
      ↪ (snd(pos) + yvel) ∨ (snd(pos) + yvel) >100)"
  update "[ st′ = CAM
          , tr′ =tr @ [Event advVel] @ [State CAM]
         , advV′ = maneuv(xvel, yvel)
         , (xvel,yvel)′ = maneuv(xvel, yvel)
         , triggers′ = {reqOCM}
          ]"

zoperation CAMToOCM =
  pre "st= CAM ∧ CDA(pos,Obsts, (xvel,yvel))>  MinSafeDist^2"
  update "[ st′ = OCM
          , tr′ =tr @[Event advVel] @ [State OCM]
          , advV′ = ZeroVel
          , (xvel,yvel)′ = ZeroVel
          , triggers′ = {reqMOM, reqVel}
          ]"

zoperation CAMToOCM_1 =
  pre "st= CAM "
  update "[ st′ = OCM
          , tr′ =tr @ [Event reqOCM] @ [State OCM]
          , triggers′ = {reqMOM, reqVel}
          ]"

definition Init :: "LRE_Beh subst" where
  [z_defs]:
  "Init =
  [pos⇝(0,0),
   xvel ⇝ 0,
```

```
   yvel ⇝ 0,
   advV ⇝ (0,0),
   reqV⇝ (0,0),
   st ⇝ initial,
   tr  ⇝ [State initial],
   triggers ⇝  {reqOCM}
   ]"

zmachine LRE_BehMachine =
  over LRE_Beh
  init Init
  invariant LRE_Beh_inv
  operations  InitialToOCM  OCMToOCM OCMToMOM MOMToOCM MOMToOCM_1  MOMToOCM_2
      ↪ HCMToOCM HCMToOCM_1 MOMToHCM  HCMToMOM  HCMToCAM HCMToCAM_1  MOMToCAM
      ↪ MOMToCAM_1  CAMToCAM CAMToCAM_1  CAMToOCM CAMToOCM_1

def_consts Velocities = "{(0,1),(0,-2), (2,0),(-4,0)}"
declare Velocities_def [z_defs]

def_consts MinSafeDist= "2"
declare MinSafeDist_def [z_defs]

def_consts HCMVel = "0.3"
declare HCMVel_def [z_defs]

def_consts MOMVel = "0.5"
declare MOMVel_def [z_defs]

def_consts ZeroVel = "(0,0)"
declare ZeroVel_def [z_defs]

def_consts SafeVel = "0.3"
declare SafeVel_def [z_defs]

subsection ⟨ Structural_Invariants ⟩

lemma Init_inv [hoare_lemmas]: "Init establishes LRE_Beh_inv"
  by (zpog_full; auto)

lemma InitialToOCM_inv [hoare_lemmas]: "InitialToOCM () preserves LRE_Beh_inv"
  by (zpog_full; auto)

lemma OCMToMOM_inv [hoare_lemmas]: "OCMToMOM() preserves LRE_Beh_inv"
  apply (zpog_full; auto)
  apply (metis add_2_eq_Suc' cancel_ab_semigroup_add_class.add_diff_cancel_left'
      ↪ not_add_less1 nth_Cons_0 nth_Cons_Suc nth_append numeral_2_eq_2)
  apply (metis add_2_eq_Suc' cancel_ab_semigroup_add_class.add_diff_cancel_left'
      ↪ not_add_less1 nth_Cons_0 nth_Cons_Suc nth_append numeral_2_eq_2)
  apply (metis add_2_eq_Suc' cancel_ab_semigroup_add_class.add_diff_cancel_left'
      ↪ not_add_less1 nth_Cons_0 nth_Cons_Suc nth_append numeral_2_eq_2)
  by (metis add_2_eq_Suc' cancel_ab_semigroup_add_class.add_diff_cancel_left'
      ↪ not_add_less1 nth_Cons_0 nth_Cons_Suc nth_append numeral_2_eq_2)

lemma MOMToOCM_inv [hoare_lemmas]: "MOMToOCM () preserves LRE_Beh_inv"
  apply (zpog_full add: prod_var_decomp)
  by (simp add: nth_append)

lemma MOMToOCM_1_inv [hoare_lemmas]: "MOMToOCM_1 () preserves LRE_Beh_inv"
   by (zpog_full; auto)
```

```
367
368 lemma MOMToOCM_2_inv [hoare_lemmas]: "MOMToOCM_2 () preserves LRE_Beh_inv"
369   apply (zpog_full add: prod_var_decomp)
370   by (simp add: nth_append)
371
372 lemma HCMToOCM_inv [hoare_lemmas]: "HCMToOCM () preserves LRE_Beh_inv"
373   apply (zpog_full add: prod_var_decomp)
374   by (metis One_nat_def Suc_eq_plus1 Suc_n_not_le_n
      ↪ cancel_ab_semigroup_add_class.add_diff_cancel_left' nat_less_le
      ↪ nth_Cons_0 nth_Cons_Suc nth_append)
375
376 lemma HCMToOCM_1_inv [hoare_lemmas]: "HCMToOCM_1() preserves LRE_Beh_inv"
377   by (zpog_full; auto)
378
379 lemma MOMToHCM_inv [hoare_lemmas]: "MOMToHCM() preserves LRE_Beh_inv"
380   apply (zpog_full add: prod_var_decomp)
381   by (metis append.assoc append_Cons append_Nil length_append_singleton
      ↪ nth_append_length)
382
383 lemma HCMToMOM_inv [hoare_lemmas]: "HCMToMOM() preserves LRE_Beh_inv"
384   apply (zpog_full add: prod_var_decomp)
385   by (metis One_nat_def add_diff_cancel_right' not_add_less2 nth_Cons_0
      ↪ nth_Cons_Suc nth_append plus_1_eq_Suc)
386
387 lemma OCMToOCM_inv [hoare_lemmas]: "OCMToOCM (v) preserves LRE_Beh_inv"
388   apply (zpog_full add: prod_var_decomp)
389   by (simp add: nth_append)
390
391 lemma HCMToCAM_inv [hoare_lemmas]: "HCMToCAM() preserves LRE_Beh_inv"
392   apply (zpog_full add: prod_var_decomp)
393   by (metis append.left_neutral append_Cons append_assoc length_append_singleton
      ↪ nth_append_length)
394
395 lemma HCMToCAM_1_inv [hoare_lemmas]: "HCMToCAM_1() preserves LRE_Beh_inv"
396   apply (zpog_full add: prod_var_decomp)
397   apply (simp add: nth_append)
398   by (simp add: nth_append)
399
400 lemma MOMToCAM_inv [hoare_lemmas]: "MOMToCAM() preserves LRE_Beh_inv"
401    apply (zpog_full add: prod_var_decomp)
402   by (simp add: nth_append)
403
404 lemma MOMToCAM_1_inv [hoare_lemmas]: "MOMToCAM_1() preserves LRE_Beh_inv"
405   apply (zpog_full add: prod_var_decomp)
406   apply (simp add: nth_append)
407   by (simp add: nth_append)
408
409 lemma CAMToCAM_inv [hoare_lemmas]: "CAMToCAM () preserves LRE_Beh_inv"
410   apply (zpog_full add: prod_var_decomp)
411   by (metis One_nat_def add_diff_cancel_right' not_add_less2 nth_Cons_0
      ↪ nth_Cons_Suc nth_append plus_1_eq_Suc)
412
413 lemma CAMToCAM_1_inv [hoare_lemmas]: "CAMToCAM_1 () preserves LRE_Beh_inv"
414   apply (zpog_full add: prod_var_decomp)
415   apply (simp add: nth_append)
416   by (simp add: nth_append)
417
418 lemma CAMToOCM_inv [hoare_lemmas]: "CAMToOCM () preserves LRE_Beh_inv"
419   apply (zpog_full add: prod_var_decomp)
```

```
  by (simp add: nth_append)

lemma CAMToOCM_1_inv [hoare_lemmas]: "CAMToOCM_1() preserves LRE_Beh_inv"
  apply (zpog_full add: prod_var_decomp)
  by (metis append_Cons append_Nil append_assoc length_append_singleton
      ↪ nth_append_length)

subsection ⟨ Safety Requirements ⟩

zexpr R1 is
"∀ i < length (filter is_State tr)-1 . (filter is_State tr) ! i= State CAM⟶
    ↪ (filter is_State tr) ! (i+1) ∈ {State OCM, State CAM}"
lemma "Init establishes R1"
  by zpog_full

lemma  "InitialToOCM () preserves R1  under LRE_Beh_inv"
  apply (zpog_full add: prod_var_decomp)
  by (metis (no_types, lifting) Nat.lessE Suc_less_eq Suc_pred nth_append
      ↪ nth_append_length zero_less_Suc)

lemma "OCMToMOM() preserves  R1 under LRE_Beh_inv"
 apply (zpog_full add: prod_var_decomp)
  by (metis (no_types, lifting) Nat.lessE One_nat_def St.distinct_disc(6)
      ↪ Suc_lessI diff_Suc_1 nth_append tag.inject(1))

lemma  "MOMToOCM() preserves  R1 under LRE_Beh_inv"
  apply (zpog_full add: prod_var_decomp)
  by (metis (no_types, opaque_lifting) Nat.lessE One_nat_def
      ↪ St.distinct_disc(12) Suc_lessI diff_Suc_1 nth_append tag.inject(1))

lemma  "HCMToOCM () preserves R1 under LRE_Beh_inv"
  apply (zpog_full add: prod_var_decomp)
  by (metis (no_types, opaque_lifting) Nat.lessE One_nat_def
      ↪ St.distinct_disc(15) Suc_lessI diff_Suc_1 nth_append tag.inject(1))

lemma  "MOMToHCM() preserves  R1 under LRE_Beh_inv"
  apply (zpog_full add: prod_var_decomp)
  by (metis (no_types, lifting) Nat.lessE One_nat_def St.distinct_disc(11)
      ↪ Suc_less_eq diff_Suc_1 nth_append tag.inject(1))

lemma  "MOMToOCM_1() preserves  R1 under LRE_Beh_inv"
  apply (zpog_full add: prod_var_decomp)
  apply (simp add: nth_append)
  apply (metis Suc_less_eq Suc_pred length_greater_0_conv list.size(3)
      ↪ not_less_zero)
  by (metis (no_types, opaque_lifting) St.simps(12) Suc_lessI add.right_neutral
      ↪ add_Suc_right cancel_comm_monoid_add_class.diff_zero diff_Suc_Suc
      ↪ less_diff_conv nth_append tag.inject(1))

lemma  "HCMToOCM_1() preserves R1 under LRE_Beh_inv"
  apply (zpog_full add: prod_var_decomp)
  apply (metis (no_types, lifting) Nat.lessE Suc_less_eq Suc_pred nth_append
      ↪ nth_append_length zero_less_Suc)
  by (metis (no_types, lifting) Nat.lessE Suc_less_eq Suc_pred nth_append
      ↪ nth_append_length zero_less_Suc)

lemma  "HCMToMOM() preserves R1 under LRE_Beh_inv"
  apply (zpog_full add: prod_var_decomp)
  by (smt (verit, ccfv_threshold) Nat.lessE St.simps(16) Suc_less_eq Suc_pred
```

```
      ↪ diff_Suc_1 length_greater_0_conv list.size(3) not_less_zero nth_append
      ↪ tag.inject(1))
467
468 lemma  "OCMToOCM (v) preserves R1 under LRE_Beh_inv"
469   apply (zpog_full add: prod_var_decomp)
470   by (metis (no_types, opaque_lifting) Nat.lessE One_nat_def St.simps(6)
      ↪ Suc_lessI diff_Suc_1 nth_append tag.inject(1))
471
472 lemma  "MOMToOCM_2() preserves R1 under LRE_Beh_inv"
473   apply (zpog_full add: prod_var_decomp)
474   by (metis (no_types, opaque_lifting) Nat.lessE One_nat_def St.simps(12)
      ↪ Suc_lessI diff_Suc_1 nth_append tag.inject(1))
475
476 lemma  "HCMToCAM() preserves R1 under LRE_Beh_inv"
477   apply (zpog_full add: prod_var_decomp)
478   by (metis (no_types, lifting) Nat.lessE Suc_less_eq Suc_pred nth_append
      ↪ nth_append_length zero_less_Suc)
479
480 lemma  "MOMToCAM() preserves  R1 under LRE_Beh_inv"
481   apply (zpog_full add: prod_var_decomp)
482   by (metis (no_types, lifting) Nat.lessE Suc_less_eq Suc_pred nth_append
      ↪ nth_append_length zero_less_Suc)
483
484 lemma  "CAMToOCM () preserves R1 under LRE_Beh_inv"
485   apply (zpog_full add: prod_var_decomp)
486   by (metis (no_types, lifting) Nat.lessE Suc_less_eq Suc_pred nth_append
      ↪ nth_append_length zero_less_Suc)
487
488 lemma  "CAMToOCM_1() preserves  R1 under LRE_Beh_inv"
489   apply (zpog_full add: prod_var_decomp)
490   by (metis (no_types, lifting) Nat.lessE Suc_less_eq Suc_pred nth_append
      ↪ nth_append_length zero_less_Suc)
491
492 zexpr R2  is
493 "st = MOM ⟶ (tr ! (length tr-2) = Event advVel ∧ (xvel^2 + yvel^2 < 0.36) ∧
    ↪ (xvel^2 + yvel^2 >0.16) ) "
494
495 lemma  "Init establishes R2"
496   by  zpog_full
497
498 lemma  "InitialToOCM () preserves  R2  under LRE_Beh_inv"
499   by  zpog_full
500
501 lemma "OCMToMOM() preserves  R2  under LRE_Beh_inv"
502   apply  zpog_full
503   apply (metis One_nat_def Suc_eq_plus1 nth_Cons_0 nth_Cons_Suc
      ↪ nth_append_length_plus)
504   apply (simp add: power_divide)
505   apply (simp add: nth_append power_divide)
506   by (simp add: nth_append power_divide)
507
508 lemma  "MOMToOCM() preserves R2  under LRE_Beh_inv"
509   by  zpog_full
510
511 lemma  "MOMToOCM_1() preserves  R2  under LRE_Beh_inv"
512   by  zpog_full
513
514 lemma  "MOMToOCM_2() preserves R2  under LRE_Beh_inv"
515   by  zpog_full
```

```
516
517 lemma  "HCMToOCM () preserves R2  under LRE_Beh_inv"
518   by  zpog_full
519
520 lemma  "HCMToOCM_1() preserves R2  under LRE_Beh_inv"
521   by  zpog_full
522
523 lemma  "MOMToHCM() preserves R2  under LRE_Beh_inv"
524   by  zpog_full
525
526 lemma  "HCMToMOM() preserves R2  under LRE_Beh_inv"
527   apply  zpog_full
528   apply (simp add: power_divide)
529   apply (simp add: power_divide)
530   apply (simp add: power_one_over)
531   by (simp add: power_one_over)
532
533 lemma  "OCMToOCM (v) preserves R2  under LRE_Beh_inv"
534   by  zpog_full
535
536 lemma  "HCMToCAM() preserves R2  under LRE_Beh_inv"
537   by  zpog_full
538
539 lemma  "HCMToCAM_1() preserves R2  under LRE_Beh_inv"
540   by  zpog_full
541
542 lemma  "MOMToCAM() preserves  R2  under LRE_Beh_inv"
543   by  zpog_full
544
545 lemma  "MOMToCAM_1() preserves  R2  under LRE_Beh_inv"
546   by  zpog_full
547
548 lemma  "CAMToCAM() preserves R2  under LRE_Beh_inv"
549   by  zpog_full
550
551 lemma  "CAMToCAM_1() preserves R2  under LRE_Beh_inv"
552   by  zpog_full
553
554 lemma  "CAMToOCM () preserves R2  under LRE_Beh_inv"
555   by  zpog_full
556
557 lemma  "CAMToOCM_1() preserves  R2  under LRE_Beh_inv"
558   by  zpog_full
559
560 lemma "Move() preserves R2  under LRE_Beh_inv"
561   by  zpog_full
562
563 zexpr R3 is
564 "st ≠ OCM⟶reqOCM ∈ triggers"
565
566 lemma  "Init establishes  R3"
567   by  zpog_full
568
569 lemma  "InitialToOCM () preserves  R3 under LRE_Beh_inv"
570   by  zpog_full
571
572 lemma "OCMToMOM() preserves  R3 under LRE_Beh_inv"
573   by  zpog_full
574
```

```
lemma  "MOMToOCM() preserves R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "MOMToOCM_1() preserves  R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "MOMToOCM_2() preserves R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "HCMToOCM () preserves R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "HCMToOCM_1() preserves R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "MOMToHCM() preserves R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "HCMToMOM() preserves R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "OCMToOCM (v) preserves R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "HCMToCAM() preserves R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "HCMToCAM_1() preserves R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "MOMToCAM() preserves  R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "MOMToCAM_1() preserves  R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "CAMToCAM() preserves R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "CAMToCAM_1() preserves R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "CAMToOCM () preserves R3 under LRE_Beh_inv"
  by  zpog_full

lemma  "CAMToOCM_1() preserves  R3 under LRE_Beh_inv"
  by  zpog_full

lemma "Move() preserves R3 under LRE_Beh_inv"
  by  zpog_full

end
```

References

[1] S. Marvin, A. While, M. Kovacic, A. Lockhart, and R. Macrorie, *Urban robotics and automation: Critical challenges, international experiments and transferable lessons for the UK*. United Kingdom: EPSRC UK Robotics and Autonomous Systems (RAS) Network, 2018.

[2] P. Pop, D. Scholle, I. Šljivo, H. Hansson, G. Widforss, and M. Rosqvist, "Safe cooperating cyber-physical systems using wireless communication: The SafeCOP approach," *Microprocessors and Microsystems*, vol. 53, no. 2017, pp. 42–50, 2017.

[3] M. Trapp and D. Schneider, "Safety Assurance of Open Adaptive Systems – A Survey," *Models@ run. time*, vol. 8378, pp. 279–318, 2014.

[4] J. Reich, D. Schneider, I. Sorokos, Y. Papadopoulos, and T. Kelly, "Engineering Digital Dependability Identities for Dynamic Safety Assurance of Cyber-Physical Systems at Runtime," no. June, 2019.

[5] D. Schneider, M. Becker, and M. Trapp, "Approaching runtime trust assurance in open adaptive systems," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 196–201, 2011.

[6] ISO, "ISO 26262 Road vehicles–Functional Safety, Version 1," 2011.

[7] D. Standard, "Standard 00-55 part 1 (1997)," *Requirements for safety related software in defence equipment, Part*, vol. 1, 1997.

[8] OMG, "Structured Assurance Case Metamodel Version: 2.3."

[9] N. Jakobi, P. Husbands, and I. Harvey, "Noise and the reality gap: The use of simulation in evolutionary robotics," in *Advances in Artificial Life: Third European Conference on Artificial Life Granada, Spain, June 4–6, 1995 Proceedings 3*, pp. 704–720, Springer, 1995.

[10] R. Calinescu, R. Mirandola, D. Perez-Palacin, and D. Weyns, "Understanding uncertainty in self-adaptive systems," in *2020 IEEE international conference on autonomic computing and self-organizing systems (ACSOS)*, pp. 242–251, IEEE, 2020.

[11] J. B. Dabney and T. L. Harman, *Mastering simulink*, vol. 230. Pearson/Prentice Hall Upper Saddle River, 2004.

[12] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (AADL): An introduction," tech. rep., Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

[13] R. D. Hawkins, I. Habli, and T. Kelly, "The need for a weaving model in assurance case automation," *Ada User Journal*, pp. 187–191, 2015.

[14] R. Hawkins, I. Habli, D. Kolovos, R. Paige, and T. Kelly, "Weaving an Assurance Case from Design: A Model-Based Approach," in *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, pp. 110–117, IEEE, 2015.

[15] E. Denney and S. Trac, "A software safety certification tool for automatically generated guidance, navigation and control code," in *2008 IEEE Aerospace Conference*, pp. 1–11, IEEE, 2008.

[16] Y. Prokhorova, L. Laibinis, and E. Troubitsyna, "Facilitating construction of safety cases from formal models in Event-B," *Information and Software Technology*, vol. 60, pp. 51–76, 2015.

[17] R. Calinescu, D. Weyns, S. Gerasimou, M. U. Iftikhar, I. Habli, and T. Kelly, "Engineering trustworthy self-adaptive software with dynamic assurance cases," *IEEE Transactions on Software Engineering*, vol. 44, pp. 1039–1069, Nov 2018.

[18] E. Jee, I. Lee, and O. Sokolsky, "Assurance cases in model-driven development of the pacemaker software," in *Leveraging Applications of Formal Methods, Verification, and Validation: 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part II 4*, pp. 343–356, Springer, 2010.

[19] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock, "RoboChart: modelling and verification of the functional behaviour of robotic applications," *Software and Systems Modeling*, vol. 18, no. 5, pp. 3097–3149, 2019.

[20] E. Denney and G. Pai, "Tool support for assurance case development," *Automated Software Engineering*, vol. 25, no. 3, pp. 435–499, 2018.

[21] A. Gacek, J. Backes, D. Cofer, K. Slind, and M. Whalen, "Resolute: an assurance case language for architecture models," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 19–28, 2014.

[22] F. Yan, S. Foster, and I. Habli, "Safety case generation by model-based engineering: State of the art and a proposal," in *The Eleventh International Conference on Performance, Safety and Robustness in Complex Systems and Applications*, pp. 4–7, IARIA, 2021.

[23] OMG, "Goal Structuring Notation Community Standard. Version 3," tech. rep., OMG, 2021.

[24] R. E. Bloomfield, P. G. Bishop, C. Jones, and P. K. D. Froome, "Ascad—adelard safety case development manual," *Adelard, 1998. ISBN 0-9533771-0*, vol. 5, 1998.

[25] T. P. Kelly, *Arguing safety: a systematic approach to managing safety cases*. PhD thesis, University of York York, UK, 1999.

[26] M. Tokoro, *Open systems dependability: dependability engineering for ever-changing systems*. CRC press, 2015.

[27] OMG, "Structured Assurance Case Metamodel (SACM), Version 2.1 beta," 2020.

[28] R. Wei, T. P. Kelly, X. Dai, S. Zhao, and R. Hawkins, "Model based system assurance using the structured assurance case metamodel," *Journal of Systems and Software*, vol. 154, pp. 211–233, Aug 2019.

[29] T. P. Kelly and J. A. McDermid, "Safety case construction and reuse using patterns," in *Safe Comp 97*, pp. 55–69, Springer, 1997.

[30] E. Denney and G. Pai, "A formal basis for safety case patterns," in *International Conference on Computer Safety, Reliability, and Security*, pp. 21–32, Springer, 2013.

[31] M. Szczygielska and A. Jarzębowicz, "Assurance case patterns on-line catalogue," in *Advances in Dependability Engineering of Complex Systems*, pp. 407–417, Springer, 2017.

[32] Y. Luo, Z. Li, and M. Van Den Brand, "A categorization of GSN-based safety cases and patterns," *MODELSWARD 2016 - 4th International Conference on Model-Driven Engineering and Software Development, Doctoral Consortium*, vol. 2016-Febru, pp. 509–516, 2016.

[33] R. Hawkins, T. Kelly, and I. Habli, "Developing Assurance Cases for D-MILS Systems," in *MILS@ HiPEAC*, 2015.

[34] A. Ayoub, B. Kim, I. Lee, and O. Sokolsky, "A safety case pattern for model-based development approach," in *NASA Formal Methods Symposium*, pp. 141–146, Springer, 2012.

[35] I. Šljivo, G. J. Uriagereka, S. Puri, and B. Gallina, "Guiding assurance of architectural design patterns for critical applications," *Journal of Systems Architecture*, vol. 110, p. 101765, 2020.

[36] N. Basir, E. Denney, and B. Fischer, "Deriving safety cases for hierarchical structure in model-based development," in *Computer Safety, Reliability, and Security: 29th International Conference, SAFECOMP 2010, Vienna, Austria, September 14-17, 2010. Proceedings 29*, pp. 68–81, Springer, 2010.

[37] E. Denney and G. Pai, "Automating the assembly of aviation safety cases," *IEEE Transactions on Reliability*, vol. 63, no. 4, pp. 830–849, 2014.

[38] Adelard, "ASCE." `https://www.adelard.com/asce/`, 2023. Accessed: June 21, 2023.

[39] A. Agrawal, S. Khoshmanesh, M. Vierhauser, M. Rahimi, J. Cleland-Huang, and R. Lutz, "Leveraging Artifact Trees to Evolve and Reuse Safety Cases," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1222–1233, IEEE, Aug 2019.

[40] C. L. Lin, W. Shen, and S. Drager, "A Framework to Support Generation and Maintenance of an Assurance Case," in *Proceedings - 2016 IEEE 27th International Symposium on Software Reliability Engineering Workshops, ISSREW 2016*, pp. 21–24, IEEE, 2016.

[41] M. D. Del Fabro, J. Bézivin, and P. Valduriez, "Weaving models with the eclipse amw plugin," in *Eclipse Modeling Symposium, Eclipse Summit Europe*, vol. 2006, pp. 37–44, 2006.

[42] A. Retouniotis, Y. Papadopoulos, I. Sorokos, D. Parker, N. Matragkas, and S. Sharvia, "Model-connected safety cases," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10437 LNCS, pp. 50–63, 2017.

[43] B. Gallina and M. Nyberg, "Pioneering the creation of ISO 26262-compliant OSLC-based safety cases," in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 325–330, IEEE, 2017.

[44] Oasis, "Open Services for Lifecycle Collaboration.." `http://open-services.net/`, 2021. Online; accessed 15th August, 2021.

[45] F. Manola, E. Miller, B. McBride, *et al.*, "Rdf primer," *W3C recommendation*, vol. 10, no. 1-107, p. 6, 2004.

[46] E. Denney and G. Pai, "Evidence arguments for using formal methods in software certification," in *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 375–380, IEEE, 2013.

[47] E. W. Denney and G. J. Pai, "Safety case patterns: theory and applications," tech. rep., NASA, 2015.

[48] N. Basir, E. Denney, and B. Fischer, "Constructing a safety case for automatically generated code from formal program verification information," in *Computer Safety, Reliability, and Security: 27th International Conference, SAFECOMP 2008 Newcastle upon Tyne, UK, September 22-25, 2008 Proceedings 27*, pp. 249–262, Springer, 2008.

[49] S. Foster, Y. Nemouchi, M. Gleirscher, R. Wei, and T. Kelly, "Integration of formal proof into unified assurance cases with Isabelle/SACM," *Formal Aspects of Computing*, vol. 33, no. 6, pp. 855–884, 2021.

[50] S. Foster, Y. Nemouchi, C. O'Halloran, K. Stephenson, and N. Tudor, "Formal model-based assurance cases in Isabelle/SACM: An autonomous underwater vehicle case study," in *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*, pp. 11–21, 2020.

[51] J. Rushby, "Formalism in safety cases," *Making Systems Safer - Proceedings of the 18th Safety-Critical Systems Symposium, SSS 2010*, no. February, pp. 3–17, 2010.

[52] Z. Diskin, T. Maibaum, A. Wassyng, S. Wynn-Williams, and M. Lawford, "Assurance via model transformations and their hierarchical refinement," in *Proceedings - 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018*, pp. 426–436, Association for Computing Machinery, Inc, 2018.

[53] M. Gleirscher, S. Foster, and Y. Nemouchi, "Evolution of formal model-based assurance cases for autonomous robots," in *International Conference on Software Engineering and Formal Methods*, pp. 87–104, Springer, 2019.

[54] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, and J. Woodcock, "RoboChart: modelling and verification of the functional behaviour of robotic applications," *Software & Systems Modeling*, vol. 18, no. 5, pp. 3097–3149, 2019.

[55] K. Ye, A. Cavalcanti, S. Foster, A. Miyazawa, and J. Woodcock, "Probabilistic modelling and verification using RoboChart and PRISM," *Software and Systems Modeling*, vol. 21, no. 2, pp. 667–716, 2022.

[56] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe, "FDR3: a parallel refinement checker for CSP," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 2, pp. 149–167, 2016.

[57] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic symbolic model checker," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pp. 200–204, Springer, 2002.

[58] S. Foster, J. Baxter, A. Cavalcanti, A. Miyazawa, and J. Woodcock, "Automating verification of state machines with reactive designs and Isabelle/UTP," in *International Conference on Formal Aspects of Component Software*, pp. 137–155, Springer, 2018.

[59] L. Zou, N. Zhany, S. Wang, M. Fränzle, and S. Qin, "Verifying simulink diagrams via a hybrid hoare logic prover," in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pp. 1–10, IEEE, 2013.

[60] L. Zou, N. Zhan, S. Wang, and M. Fränzle, "Formal verification of simulink/stateflow diagrams," in *International Symposium on Automated Technology for Verification and Analysis*, pp. 464–481, Springer, 2015.

[61] J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou, "A calculus for hybrid CSP," in *Asian Symposium on Programming Languages and Systems*, pp. 1–15, Springer, 2010.

[62] M. Hause *et al.*, "The sysml modelling language," in *Fifteenth European Systems Engineering Conference*, vol. 9, pp. 1–12, 2006.

[63] A. S. A. Hadad, C. Ma, and A. A. O. Ahmed, "Formal verification of AADL models by event-B," *IEEE Access*, vol. 8, pp. 72814–72834, 2020.

[64] 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*, vol. 54, no. 5, pp. 754–775, 2011.

[65] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, pp. 410–425, 2000.

[66] J.-P. Katoen, M. Khattri, and I. Zapreevt, "A Markov reward model checker," in *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pp. 243–244, IEEE, 2005.

[67] Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, and J.-P. Talpin, "From AADL to timed abstract state machines: A verified model transformation," *Journal of Systems and Software*, vol. 93, pp. 42–68, 2014.

[68] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis, "Translating AADL into BIP-application to the verification of real-time systems," in *Models in Software Engineering: Workshops and Symposia at MODELS 2008, Toulouse, France, September 28-October 3, 2008. Reports and Revised Selected Papers 11*, pp. 5–19, Springer, 2009.

[69] O. Sokolsky, I. Lee, and D. Clarke, "Schedulability analysis of AADL models," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pp. 8–pp, IEEE, 2006.

[70] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Dal Zilio, M. Filali, and F. Vernadat, "Formal verification of AADL specifications in the Topcased environment," in *Reliable Software Technologies–Ada-Europe 2009: 14th Ada-Europe International Conference, Brest, France, June 8-12, 2009. Proceedings 14*, pp. 207–221, Springer, 2009.

[71] P. C. Ölveczky, A. Boronat, and J. Meseguer, "Formal semantics and analysis of behavioral AADL models in Real-Time Maude," in *Formal Techniques for Distributed Systems: Joint*

12th IFIP WG 6.1 International Conference, FMOODS 2010 and 30th IFIP WG 6.1 International Conference, FORTE 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings, pp. 47–62, Springer, 2010.

[72] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens, "Virtual execution of AADL models via a translation into synchronous programs," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pp. 134–143, 2007.

[73] M. Soeken, C. B. Harris, N. Abdessaied, I. G. Harris, and R. Drechsler, "Automating the translation of assertions using natural language processing techniques," in *Proceedings of the 2014 Forum on Specification and Design Languages (FDL)*, vol. 978, pp. 1–8, IEEE, 2014.

[74] T. Zhang, D. Saab, and J. A. Abraham, "Automatic assertion generation for simulation, formal verification and emulation," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 471–476, IEEE, 2017.

[75] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pp. 626–629, IEEE, 2010.

[76] M. L. Bolton, N. Jiménez, M. M. van Paassen, and M. Trujillo, "Automatically generating specification properties from task models for the formal verification of human–automation interaction," *IEEE Transactions on Human-Machine Systems*, vol. 44, no. 5, pp. 561–575, 2014.

[77] H. Bourbouh, M. Farrell, A. Mavridou, I. Sljivo, G. Brat, L. A. Dennis, and M. Fisher, "Integrating Formal Verification and Assurance: An Inspection Rover Case Study," in *NASA Formal Methods Symposium*, vol. 12673 LNCS, pp. 53–71, Springer, 2021.

[78] D. Giannakopoulou, A. Mavridou, J. Rhein, T. Pressburger, J. Schumann, and N. Shi, "Formal requirements elicitation with FRET," in *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020)*, 2020.

[79] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner, "Arsenal: automatic requirements specification extraction from natural language," in *NASA Formal Methods: 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings 8*, pp. 41–46, Springer, 2016.

[80] D. Elenius, E. Yeh, S. Graham-Lengrand, S. Ghosh, P. Lincoln, and N. Shankar, "Deriving Formal Specifications from Natural Language Requirements using ARSENAL 2," in *High Confidence Software and Systems Conference Series 2019*, 2019.

[81] W. Lindoso, S. C. Nogueira, R. Domingues, and L. Lima, "Visual Specification of Properties for Robotic Designs," in *Brazilian Symposium on Formal Methods*, pp. 34–52, Springer, 2021.

[82] A. Miyazawa, P. Ribeiro, K. Ye, A. Cavalcanti, W. Li, J. Woodcock, and J. Timmis, "RoboChart Reference Manual," 2021.

[83] J. Baxter, A. Miyazawa, P. Ribeiro, and K. Ye, "RoboTool Manual," 2023.

[84] "PRISM Lab Session, Part B: Mail Delivery Robot." `http://www.prismmodelchecker.org/courses/aims1617/deliveryRobot.php`, 2017. online; accessed 5th May, 2022.

[85] Y. Murray, M. Sirevåg, P. Ribeiro, D. A. Anisi, and M. Mossige, "Safety assurance of an industrial robotic control system using hardware/software co-verification," *Science of Computer Programming*, p. 102766, 2022.

[86] F. Yan, S. D. Foster, I. Habli, and R. Wei, "Model-based generation of hazard-driven arguments and formal verification evidence for assurance cases," in *10th International Conference on Model-Driven Engineering and Software Development*, pp. 252–263, SciTePress, 2022.

[87] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The Epsilon transformation language," in *International Conference on Theory and Practice of Model Transformations*, pp. 46–60, Springer, 2008.

[88] D. S. Kolovos, R. F. Paige, and F. A. Polack, "The epsilon object language (EOL)," in *European conference on model driven architecture-foundations and applications*, pp. 128–142, Springer, 2006.

[89] D. D. Kolovos, L. Rose, R. Paige, and A. García-domínguez, "The Epsilon book," tech. rep., 2010.

[90] "GSN Metamodel Specification Version 2.2," 2021.

[91] A. W. Roscoe, *Understanding concurrent systems*. Springer Science & Business Media, 2010.

[92] "FDR 4.2 Documentation." `https://cocotec.io/fdr/manual/index.html`, 2020. Online; accessed 04th May, 2022.

[93] A. W. Roscoe, *The theory and practice of concurrency*. Prentice Hall, 1998.

[94] "PRISM Manual." `https://www.prismmodelchecker.org/manual`, 2017. Online; accessed 04th May, 2022.

[95] F. Yan, S. D. Foster, and I. Habli, "Automated Compositional Verification for Robotic State Machines using Isabelle/HOL," in *27th International Conference on Engineering of Complex Computer Systems*, IEEE, 2023.

[96] H. Kausch, M. Pfeiffer, D. Raco, and B. Rumpe, "Model-based design of correct safety-critical systems using dataflow languages on the example of sysml architecture and behavior diagrams," in *Software Engineering (Satellite Events)*, 2021.

[97] S. Böhme and T. Nipkow, "Sledgehammer: judgement day," in *International Joint Conference on Automated Reasoning*, pp. 107–121, Springer, 2010.

[98] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*, vol. 2283. Springer Science & Business Media, 2002.

[99] M. Spivey, *The Z-Notation - A Reference Manual*. Englewood Cliffs, N. J.: Prentice Hall, 1989.

[100] J.-R. Abrial, *The B-Book: assigning programs to meanings*. Cambridge University Press, 1996.

[101] R.-J. Back and R. Kurki-Suonio, "Decentralization of process nets with centralized control," *Distributed Computing*, vol. 3, pp. 73–87, June 1989.

[102] J. A. Hilder, N. D. Owens, M. J. Neal, P. J. Hickey, S. N. Cairns, D. P. Kilgour, J. Timmis, and A. M. Tyrrell, "Chemical detection using the receptor density algorithm," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 6, pp. 1730–1741, 2012.

[103] C. A. R. Hoare and J. He, *Unifying Theories of Programming*. Prentice-Hall, 1998.

[104] M. Wenzel, "Interaction with formal mathematical documents in Isabelle/PIDE," in *CICM*, LNCS 11617, pp. 1–15, Springer, 2019.

[105] A. Riazanov and A. Voronkov, "The design and implementation of VAMPIRE," *AI communications*, vol. 15, no. 2-3, pp. 91–110, 2002.

[106] S. Foster, C.-K. Hur, and J. Woodcock, "Formally verified simulations of state-rich processes using interaction trees in Isabelle/HOL," in *32nd Intl. Conf. on Concurrency Theory (CONCUR)*, vol. 203 of *LIPIcs*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[107] K. Ye, S. Foster, and J. Woodcock, "Formally Verified Animation for RoboChart using Interaction Trees," in *International Conference on Formal Engineering Methods*, pp. 404–420, Springer, 2022.

[108] K. Johnson, R. Calinescu, and S. Kikuchi, "An incremental verification framework for component-based software systems," in *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, pp. 33–42, 2013.

[109] A. L. C. Cavalcanti, A. C. A. Sampaio, A. Miyazawa, P. Ribeiro, M. S. C. Filho, A. Didier, W. Li, and J. Timmis, "Verified simulation for robotics," *Science of Computer Programming*, vol. 174, pp. 1–37, 2019.