Automatic Co-Synthesis of Hardware and Software Safety Monitors for Embedded Systems

Behnaz Rezvani

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Cameron Patterson, Chair
Leyla Nazhand-Ali
Tam Chantem
Binoy Ravindran
Mazen Farhood

August 12, 2024
Blacksburg, Virginia

Keywords: Runtime verification, Monitor, Functional requirements, Timing requirements, Real-time embedded systems.

Copyright 2024, Behnaz Rezvani

Automatic Co-Synthesis of Hardware and Software Safety Monitors for Embedded Systems

Behnaz Rezvani

(ABSTRACT)

Embedded systems have become pervasive and increasingly complex, especially in modern applications such as self-driving vehicles, where safety requires both accurate functionality and real-time guarantees. However, the complexity and the integration of artificial intelligence and machine learning algorithms in autonomous systems challenge conventional test-based verification methods. Given the continuous evolution and deployment of these systems, verification must keep pace to ensure their reliability and safety. Runtime verification is a promising approach for validating system behaviors during execution using monitors derived from formal system specifications. The adoption of runtime monitoring has historically been limited to experts, primarily due to the esoteric formal notations and verification processes. To overcome this barrier, this dissertation presents GROOT, a novel methodology and framework designed to automate synthesis of hardware and/or software monitors from pseudo-English statements. The automatic steps include translating English properties to formalisms, converting the formalisms into monitor automata, and formally verifying the monitors. GROOT addresses the distinction between functional and timing requirements inherent in real-time embedded systems by providing distinct pseudo-English languages and synthesis flows. This dual approach allows customized verification processes for each category.

To make the monitor structure simple, monitor inputs and responses are handled in separate external modules, allowing formal analysis methods to be used. The synthesized monitors can assist system development and be retained in fielded systems. Their lightweight nature

enables the deployment of multiple monitors, each focusing on specific circumstances independently and concurrently. Monitor implementations can range from sequential software to parallel hardware, allowing for flexibility in meeting various system constraints. By eliminating the need for manual code generation and verification, GROOT allows practitioners to synthesize monitors without requiring a formal methods background.

Automatic Co-Synthesis of Hardware and Software Safety Monitors for Embedded Systems

Behnaz Rezvani

(GENERAL AUDIENCE ABSTRACT)

An embedded system is a computer system that is *embedded* into a device to perform specific tasks. These systems are designed to function within the device they are built into and are often found in everyday technologies such as cars, household appliances and medical devices. They are programmed to carry out particular functions or operations automatically, without needing human intervention. These systems rely on precise timing and flawless functionality to operate safely and efficiently. However, ensuring that they work as intended, especially as they grow more complex and interconnected, presents significant hurdles. Traditional methods of verifying embedded systems rely on manual testing, which can be time-consuming and prone to errors. To address this issue, this dissertation introduces GROOT, a novel methodology and framework designed to automate the process of monitoring these systems in real-time. GROOT simplifies this complex task by automatically generating monitors from simple English statements. These monitors act as vigilant watchdogs, continuously checking whether the system behaves as intended and responds appropriately to its environment. This automation makes the verification process more efficient and less error-prone. This framework handles both functional requirements (ensuring the system performs its intended tasks) and timing requirements (ensuring tasks are completed within specific time frames). GROOT also offers flexibility in how monitors are implemented, allowing them to be tailored to specific hardware or software configurations. This flexibility allows GROOT's monitors to adapt to various applications, from small-scale prototypes to large-scale deployments.

Dedication

To my loving husband and best friend, Morteza. And to my parents Maman and Baba.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Cameron Patterson, for his invaluable guidance and insightful feedback. I am truly grateful to have had the opportunity to learn from him. I would also like to extend my sincere thanks to my first advisor, Dr. William Diehl and to all my committee members, Dr. Leyla Nazhand-Ali, Dr. Tam Chantem, Dr. Binoy Ravindran, and Dr. Mazen Farhood. I am grateful for the time and effort they invested in my research.

I am deeply appreciative of the financial support provided by GE Global Research and the National Science Foundation (NSF) under Grant No. 2123550. Their generous funding has made it possible for me to conduct my research and focus on my studies without financial burden. This dissertation would not have been possible without their support.

I want to thank my friends for their support, understanding, and encouragement. Their companionship and positive energy have been a source of strength during challenging times. I am grateful for the moments of laughter that have helped me maintain a balanced perspective. Here is a short list: Maede Hemmat and Ali Ansari, Fatemeh Lotfi, Hossein Rajoli and Rayan Rajoli, Keyvan Ramezanpour, Nastaran Khalili and Farshad Farkhondeh, Mohammad Karimzadeh, Ayda Kianmehr and Hadi Rahmani, Arian Zargami, Ali Hosseini, Esmat Mohammadi and Mohsen Hosseini, Mohammad Hossein Bahonar and Zahra Jandaghian, Somayeh Yarahmadi and Amirsina Torfi, Nooshin Omidvar, Kiana Moghimi and Ali Talebzadeh, Maedeh Maftouni and Ali Safiabadi, and Fatemeh Vafaee.

I would like to express my heartfelt gratitude to my family. To my parents, Maman and Baba, thank you for your unconditional love, support, and belief in me. Your sacrifices and encouragement have provided me with the foundation to pursue my dreams. To Amin and Fatemeh, Maman joon, Baba Joon, Samaneh and Behnam, Iliya, Aghajoon and Madar, thank you for your support and for always being there for me.

I want to close this section by a heartfelt thanks to my lovely husband Morteza and my dear friend Leyla, without whom I would have been lost so many times for the past five years of my life. Morteza, thank you for all the love, patience, and unwavering support you have given me throughout these past eleven years we have been together.

Contents

List of Figures **xiv**

List of Tables **xix**

1 Introduction **1**

1.1 Motivation and Background 1

1.1.1 Embedded Systems 1

1.1.2 Introduction to Runtime Verification 3

1.2 Problem Statement 4

1.3 Dissertation Contributions 5

1.4 Dissertation Organization 7

2 Background and Related Work **8**

2.1 Verification Approaches 8

2.1.1 Testing vs. Formal Verification 8

2.1.2 Theorem Proving vs. Model Checking 9

2.1.3 Runtime Verification 11

2.1.4 Verification Technique Challenges 12

2.2 Properties and Specifications 15

2.2.1 Formal Specification Languages 15

2.2.2 Pnueli's Linear Temporal Logic 18

2.2.3 Linear Temporal Logic over Finite Paths 19

2.2.4 Real-Time Properties 23

2.2.5 Monitorable Properties 24

2.2.6 Property Specification Patterns 25

2.3 Runtime Verification Monitors 26

2.3.1 Online vs. Offline Monitoring 26

2.3.2 Synchronous vs. Asynchronous Monitoring 27

2.3.3 Inline vs. Outline Monitoring 27

2.3.4 Monitoring via Büchi Automaton 28

2.3.5 Monitoring via Timed Automaton 30

2.3.6 Monitor Implementation 32

2.4 Existing Runtime Verification Frameworks 33

2.4.1 Software-based Monitoring Frameworks 34

2.4.2 Hardware-based Monitoring Frameworks 36

2.4.3 Comparison of Existing Monitoring Tools 38

2.5 Summary 39

3 Preliminary Work **41**

3.1 Phase 1: Intel Aero Drone Monitoring . . . 41
3.2 Phase 2: Virtual Drone Monitoring . . . 45
3.3 Phase 3: Monitor Synthesis Automation . . . 49
3.4 Summary . . . 52

4 GROOT Methodology 53

4.1 Methodology Steps . . . 53
4.1.1 Formalization via Specification Patterns . . . 54
4.1.2 Automated Monitor Generation . . . 55
4.1.3 Monitor Verification . . . 55
4.2 Monitor Structure in GROOT . . . 56
4.3 GROOT Workflow . . . 58
4.4 Monitor Synthesis in a HDL . . . 60
4.5 Trusted Software and Hardware Components . . . 60
4.6 Monitor Implementation in Embedded Systems . . . 62
4.7 Summary . . . 62

5 Functional Requirements Monitor Synthesis in GROOT 64

5.1 Overview . . . 64
5.2 Synthesis Front-end . . . 65
5.2.1 Handling Finite Traces in FRET . . . 68

5.3 Synthesis Back-end 69
5.3.1 Spot API 69
5.3.2 LTL2C Parser 70
5.3.3 LTL2C Code Generator 73
5.3.4 Monitor Formal Analysis 76
5.4 Summary 79

6 Timing Requirements Monitor Synthesis in GROOT 80

6.1 Overview 80
6.2 TIMESPEC 81
6.2.1 Pulse Duration 83
6.2.2 Causality 84
6.2.3 Example: Slave Configuration Timings 85
6.3 Synthesis Workflow 87
6.3.1 TS2C Parser 88
6.3.2 TS2C TA Generator 88
6.3.3 TS2C Code Generator 91
6.3.4 Monitor Formal Analysis 93
6.4 Summary 97

7 Evaluation 98

7.1 Overview 98

7.2 Case 1: UAS Flying within a Geofence 100

7.2.1 Monitor Synthesis 100

7.2.2 Monitor Formal Analysis 103

7.2.3 Monitor Simulation 106

7.2.4 Productivity Comparison 107

7.3 Case 2: Lightweight Cryptographic System 108

7.3.1 Monitor Synthesis 109

7.3.2 Monitor Formal Analysis 110

7.3.3 Monitor Simulation 114

7.3.4 Monitor Resource Utilization 117

7.4 Case 3: Adaptive Cruise Control System 118

7.4.1 Monitor Synthesis 119

7.4.2 Monitor Formal Analysis 121

7.4.3 Monitor Simulation 126

7.4.4 Comparison of Copilot and GROOT Monitors 128

7.5 Summary 134

8 Conclusions **136**

8.1 Contributions 136

8.2 Future Work 140

Bibliography **141**

Appendices **161**

Appendix A Monitors for the ACC System **162**

A.1 GROOT Monitors 162

A.2 Copilot Monitors 168

List of Figures

2.1 Schematic view of model checking process. 10

2.2 FSM of microwave oven model [47]. 11

2.3 Block diagram of RV process. 12

2.4 State machine diagram of traffic light, which S_0 is the initial state. 16

2.5 Graphical representation of LTL modalities for atomic propositions p and q. The values in gray states are unimportant. 18

2.6 BA for thermostat property $\Box(stop \ \mathtt{U} \ temp_dropped)$. 29

2.7 Property PT illustrated by a TA. 31

3.1 Adding isolated safety monitors to the UAS [153]. 42

3.2 AP values for different flight scenarios [152]. 43

3.3 The first prototype of monitor synthesis flow. 43

3.4 BA for the max_alt requirement [153]. 44

3.5 HIL simulation with RTA system in place. Details of this architecture can be found in [110]. 46

3.6 A flight plan example for BVLoS [110]. 47

3.7 A flight plan example for BVLoS [110]. 47

3.8 RTA design architecture [110]. 48

3.9 Complex geofence including two exclusive areas. 50

3.10 Generated BA for the *geofence_breach* requirement. 51

3.11 The developed monitor synthesis tool flow. 51

4.1 Visual representation of the methodological flow of GROOT. 54

4.2 Structure of GROOT monitoring process. 57

4.3 Three steps of GROOT for monitor synthesis of FRETish requirements. . . . 59

4.4 Three steps of GROOT for monitor synthesis of TIMESPEC requirements. . 59

4.5 Hardware monitor synthesis in GROOT. 60

5.1 GROOT monitor synthesis flow for functional requirements. 65

5.2 FRETish pattern for LTL formulas [68]. The starred fields are mandatory. . 67

5.3 FRET's explanation using diagram in the HELP tab. 68

5.4 A complex geofence with two no-fly zones, which are shown as gray areas. . 72

5.5 NuSMV simulation screenshot of the RTA-R4 requirement. 74

5.6 The generated header file, Input_Handler, Output_Handler and Main functions (from top to bottom on the left), along with the Monitor function (on the right) for RTA-R4. 76

5.7 Code-Gen generated assertions for the RTA-R4 requirement. 77

5.8 The RTA-R4 monitor automaton provided by Spot. 78

6.1 GROOT monitor synthesis flow for timing requirements. 81

6.2 Timing diagram of TIMESPEC causality template for specific values for ACTION and SIGNAL fields. 85

6.3 Timing diagram of OPT9221 slave configuration [155]. 86

6.4 The TA structure used for between, after and within values. Solid lines are shared, while the dotted lines are chosen based on the requirement's timing constraint. 90

6.5 TA constructed by TA-Gen for requirement RC3. 90

6.6 The generated header, Input_Handler, Output_Handler and Main functions (from top to bottom on the left), along with the Monitor function (on the right) for RC3. 94

6.7 ACSL contracts for proving the correctness of the RC3 monitor by Frama-C. 95

7.1 Visualizations of the original (left) and second (right) versions of RTA-R0. . 102

7.2 Estimation of the UAS future position. 103

7.3 Monitor FSM for the original version of RTA-R0. 104

7.4 Monitor FSM for the second version of RTA-R0. 104

7.5 EBMC verification results for the RTA-R0 monitor. 105

7.6 Simulation results for the RTA-R0 monitor. 106

7.7 GE approach for geofence monitor generation. 108

7.8 GIFT-R0 monitor FSM provided by LTL2C. 110

7.9 GIFT-R1 monitor FSM generated by TS2C. 111

7.10 EBMC verification results for GIFT-R0. 114

7.11 EBMC verification results for GIFT-R1. 115

7.12 Simulation results of GIFT and its monitors when there is no fault/attack. . . 116

7.13 Simulation results of a fault or HT attack on the *start* signal, triggering the GIFT-R0 monitor. 116

7.14 Simulation results of a fault or HT attack on the *done* signal, triggering the GIFT-R1 monitor. 117

7.15 Monitor overhead for GIFT-COFB in terms of LUTs, FFs, and power. 118

7.16 ACC modes of operation. 119

7.17 BA built by the Spot tool for ACC_R0. 120

7.18 TA built by TA-Gen for ACC_R1. 121

7.19 Assertions provided by Code-Gen for ACC_R0. 122

7.20 ESBMC verification results for ACC_R0. 123

7.21 ACSL statements provided by Code-Gen for ACC_R1. 124

7.22 Frama-C verification results for ACC_R1. 125

7.23 Simulation results for ACC_R0 monitor. 127

7.24 Simulation results for ACC_R1 monitor. 128

7.25 Monitor generation steps using FRET, Ogma and Copilot tools [127]. 130

7.26 Code snippet for the guard functions of Copilot ACC monitors. 131

7.27 Test case examples for executing ACC monitors. 132

7.28 Execution time of ACC monitors generated by GROOT (left) in comparison with monitors generated by Copilot (right). 133

List of Tables

2.1 Comparison of verification methods in RTES. 13

2.2 A brief comparison of select RV tools and GROOT. 38

3.1 The *geofence_inclusion* requirement written in English, FRETish and LTL. 50

3.2 The *geofence_breach* requirement written in English, FRETish and LTL. . . 50

5.1 Supported options for each field of a FRETish template. 67

5.2 The RTA-R1 requirement written in FRETish and LTL. 69

5.3 The RTA-R4 geofence requirement written in FRETish and LTL. 72

5.4 Modified FRETish and LTL statements for RTA-R4 along with its list of APs. 73

6.1 Possible reasons for timing violations in practice. 82

6.2 TIMESPEC pulse duration template. 83

6.3 TIMESPEC causality template. 84

6.4 Timing characteristics of OPT9221 slave configuration [155]. 86

6.5 Timing requirements of OPT922 using pulse duration and causality templates. 87

7.1 Geofence requirements for the RTA collaboration project. 101

7.2 Second version of FRETish and LTL for the RTA-R0 requirement. 102

7.3 Generated SVAs for the RTA-R0 hardware monitor. 105

7.4 End-to-end V&V process metrics for a requirement with only Boolean expressions. 108

7.5 GIFT properties in FRETish and TIMESPEC. 109

7.6 Generated SVAs for the GIFT-R0 monitor. 112

7.7 Generated SVAs for the GIFT-R1 monitor. 113

7.8 Implementation results for GIFT-COFB and its monitors on Artix-7. 117

7.9 ACC system requirements. 120

7.10 ACC model parameters default values. 126

7.11 Breaking the ACC_R1 requirement into two FRETish statements. 129

Chapter 1

Introduction

Embedded systems are specialized computer systems integrated into larger devices to handle specific tasks. They are commonly found in everyday devices such as smartphones, digital cameras, automotive control systems, and medical devices, playing critical roles in safety and public services. For example, in automotive applications, embedded systems control engine functions, deploy airbags and manage anti-lock braking systems, directly impacting passenger's safety. Ensuring the correctness of these systems is paramount, requiring a mathematical degree of certainty in both functionality and timing. Traditional verification methods, usually applied during the design phase, may not identify every possible error or fault. Consequently, a verification approach that dynamically functions during the system's runtime is essential. This method allows for continuous and thorough checks to detect and address software errors and hardware malfunctions as they occur, confirming the system consistently operates within its critical safety parameters.

1.1 Motivation and Background

1.1.1 Embedded Systems

Embedded systems are devices tailored for specific tasks, often constrained by factors such as cost, power consumption, available resources, and timing requirements due to their dedi-

cated functionality [123]. Real-time systems such as flight controllers, continuously interact with their environment and are required to meet precise timing constraints across various operational scenarios to ensure safety and efficiency [93]. Real-time embedded systems combine the characteristics of embedded systems and real-time systems. They are specialized devices designed to efficiently perform dedicated functions while operating within strict timing constraints. For example, a car's anti-lock braking system (ABS) is a real-time embedded system that monitors wheel speed and applies brakes to prevent wheel lock-up during sudden braking. This system ensures quick response and optimal braking performance, thereby enhancing vehicle safety and stability.

The applications of these systems span a wide range of domains, from everyday uses such as home appliances and video game controllers to highly critical applications such as avionics, medical devices, and nuclear reactor controllers. These systems often operate autonomously, without continuous human oversight. The environment can be complex, and despite advancements in safety-aware programming including artificial intelligence (AI) algorithms, there remains a risk of system failures due to unforeseen conditions. Such malfunctions, particularly in safety-critical applications, can lead to significant financial and even human losses. For example, in June 1996, the unmanned Ariane 5 rocket exploded 37 seconds after launch due to a software bug during conversion of a 64-bit floating-point to a 16-bit signed integer and resulted in $370 million loss [53]. More recently, in 2018, a pedestrian was fatally struck by a self-driving Uber vehicle in autonomous mode, which failed to recognize the individual as a pedestrian due to her location being away from a crosswalk [106].

Considering the critical role of real-time embedded systems, ensuring correct functionality and timely responses is paramount. These systems, which typically integrate microprocessors with specialized hardware, must meet a mixture of high-level functional requirements and hardware-specific timing constraints. The integration of complex AI algorithms adds layers

of unpredictability which introduces a higher risk of errors and failures. Verification, which is the process of confirming a system adheres to specified requirements, demands significant investment of time, effort, and cost. Studies have shown that approximately 60–70% of system development time is dedicated to the verification processes [83]. This emphasizes the importance of effective verification methods for ensuring safety and performance. While various standards and verification methodologies are available to reduce software defects, the growing complexity of modern embedded systems, combined with fast-paced market demands, limit the efficiency of conventional verification approaches.

1.1.2 Introduction to Runtime Verification

Verification methods fall into two primary categories based on when they are applied to the system. Static verification involves checking system before deployment, ensuring system correctness through exhaustive analysis and testing of code and logic. Conversely, dynamic verification happens while the system is running. Runtime verification (RV) is a dynamic verification approach, which actively observes system behaviors during execution to ensure they conform to predefined properties set by the system designer [102, 135]. For example, in automotive systems, RV can continuously monitor sensor inputs and vehicle control outputs to verify they comply with safety protocols during operation. Unlike traditional testing methods that involve running predetermined test cases and verifying system behavior against expected outcomes, RV provides real-time monitoring and instantly reports issues as they occur. This capability is particularly valuable because deviations may only become apparent during actual system execution [84].

RV can be employed for both phases of a computer system's operation. During the development phase, when the system is undergoing frequent changes and debugging, RV detects

bugs in software and identifies deficiencies in hardware for prompt resolution. While pre-deployment verification is vital for ensuring safety in critical applications, it may not be sufficient due to the possibility of unforeseen hardware and software issues [20]. In such cases, RV complements traditional verification methods by providing continuous monitoring in real-time, enhancing safety and minimizing the risk of catastrophic failures [129]. RV is also beneficial for regularly-updated fielded products to verify that each new version maintains compliance with safety standards and requirements across various scenarios.

1.2 Problem Statement

Runtime verification (RV) of real-time embedded systems presents significant challenges due to their specialized architecture and stringent operational constraints [135]. Unlike conventional computing systems, which may operate across a variety of hardware platforms, each embedded system is specifically optimized to function with its unique hardware. The lack of standard input/output interfaces further complicates the direct observation of their processes, as these systems are deeply integrated within larger devices and interact closely with them. They also operate under strict resource constraints and must adhere to real-time operational requirements which limits the feasibility of runtime monitoring. Consequently, most existing RV frameworks are primarily designed for software applications, not adequately addressing the specific needs of embedded systems.

Embedded systems have a diverse set of requirements, ranging from high-level system functionalities to more detailed low-level aspects such as signal generation and clock timing. These systems must satisfy both functional requirements, which dictate the specific tasks they must perform, and non-functional requirements, which are concerned with how these tasks are performed, particularly under strict timing constraints. For example, a pacemaker

must not only stimulate the heart at regular intervals (a functional requirement) but also do so with critical timing precision (a non-functional and timing requirement) to ensure patient safety. Despite the availability of many RV tools designed for various applications, as noted by Falcone et al. [62], only a few have an explicit notion of time, making them suitable for real-time system monitoring.

The RV process utilizes monitors—special software or hardware components that check if a running system adheres to certain desired properties. RV monitors are typically derived from high-level system specifications expressed in formal languages. These formalisms use mathematically precise rules and symbols to specify system behaviors clearly and without ambiguity. However, a major drawback of RV is the lack of standardized formalism for capturing system properties [140]. This leads to each RV tool adopting its own syntax and notation, which complicates the verification process. Consequently, engineers are generally required to have a deep understanding of formal methods to effectively use these tools.

This steep learning curve and the need for specialized expertise often restrict RV to experts in the field. This limitation is especially significant in real-time embedded systems, where verifying both software and hardware components usually requires multiple verification tools. Since RV is not well integrated into the overall process of testing and validating systems, its accessibility and usability are limited. Hence, there is a need for a more unified and user-friendly monitoring approach to enhance RV's utility across various stages of system development and operation.

1.3 Dissertation Contributions

The primary goal of this dissertation is to make RV more accessible and practical for verification engineers who may not have expertise in formal methods. To achieve this goal,

it introduces GROOT (Generalized Runtime mOnitOring Tool), a novel methodology and framework that automates the process of converting system behaviors, described in pseudo-English, into concise monitoring code. This automation creates monitors that operate efficiently with minimal consumption of system resources, making them suitable for embedded applications. Their lightweight design allows multiple monitors to operate simultaneously, each focused on detecting a specific condition. The structure of these monitors is kept simple to facilitate the automation of rigorous verification, which ensures they are correctly implemented and comply with the original specifications.

GROOT's monitors are designed to operate independently of the underlying software or hardware, treating the application as a black box. This independence allows them to thoroughly analyze system inputs and outputs, verifying adherence to defined requirements without needing detailed knowledge of the internal workings. During the development phase, these monitors run autonomously within simulation models and prototypes, assisting in fault detection and system improvement. In deployment, they continue to vigilantly monitor the system, promptly identifying any deviation from expected behavior and taking necessary actions to maintain operational safety if required.

Given that practicing engineers often distinguish between functionality and timing, similar to the information provided in component datasheets, GROOT addresses monitoring of functional and timing requirements separately. This tailored approach allows the monitors to be customized according to the specific needs of each requirement type, thus enhancing the effectiveness of the verification process. In line with the common practices of embedded system engineers, GROOT utilizes pseudo-English to specify desired behaviors and implements the monitors as finite state machines in ANSI standard C code. This approach simplifies RV for practitioners by concealing the complex mathematical notations typically used in formal languages and automating the entire process of monitor generation and verification.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents background knowledge and an overview of related work in RV. The developed framework is described in Chapter 3. Chapter 4 introduces GROOT's methodology, with Chapter 5 and Chapter 6 focusing on functional requirements and timing requirements flows, respectively. Chapter 7 illustrates the practical applications of GROOT's monitors across various embedded systems and includes an evaluation of monitor performance and overhead. Finally, Chapter 8 concludes with a discussion of potential future work.

Chapter 2

Background and Related Work

This chapter first discusses conventional verification methods and then provides an overview of runtime verification (RV) to better understand the subsequent chapters. Basic ideas are described, followed by what is needed from expressing system properties to generating monitoring modules. Finally, the main runtime monitoring frameworks are described.

2.1 Verification Approaches

2.1.1 Testing vs. Formal Verification

Testing is widely used in both software and hardware development to ensure quality and reliability especially before the product is released. It can be applied across different levels of system abstraction and can detect straightforward errors by subjecting the system to various input sequences. Engineers often create detailed test cases and then compare the actual system outputs against the expected ones to identify inconsistencies or bugs. However, achieving complete coverage of all execution paths by testing is often impractical [120]. Despite its considerable consumption of time and resources in the development process, even the most thorough testing may fail to detect subtle flaws in complex systems.

Formal verification is another approach that provides high assurance by mathematically analyzing system models that contain detailed sets of states and transitions. This can identify

issues such as inconsistencies and undefined behaviors, which require system specification to be precisely defined using formal mathematical language. Historical examples such as the Ariane 5 rocket failure [53] indicate that major failures might have been avoided with formal verification. The most commonly used formal verification techniques are theorem proving and model checking, which are usually employed during system's design phase and categorized as static verification (SV) methods.

2.1.2 Theorem Proving vs. Model Checking

The theorem proving technique methodically constructs formal proofs to confirm system properties by transforming them into mathematical logic formulas, and is adaptable to systems of any complexity [63, 69]. For example, theorem proving can be applied to verify an n-bit full adder (for any integer n) using mathematical induction. The prover first validates a 1-bit adder by establishing its functionality through axioms that define the sum function, i.e., ($b_1 + b_2 + c_{\text{in}} = 2 \times c_{\text{out}} + s$), where b_1 and b_2 are the two summand bits, c_{in} and c_{out} are the carry-in and carry-out bits, and s is the sum result. Next, it addresses the $(n-1)$-th case as part of the induction step. Building on this, the theorem prover infers the correctness of the n-bit case by principles of mathematical induction.

One limitation of theorem proving is its lack of full automation which requires verification engineers to have a deep understanding of translating specifications into a mathematical language, implementations into a mathematical model, and analysis into mathematical proofs. This requirement can create a substantial barrier in terms of time and resource investment. For example, formal verification of the seL4® microkernel [58], despite involving only 8,700 lines of C code, took nearly 20 person-years of effort by formal methods experts who designed the microkernel with formal verification in mind [90]. Another drawback is the inability to

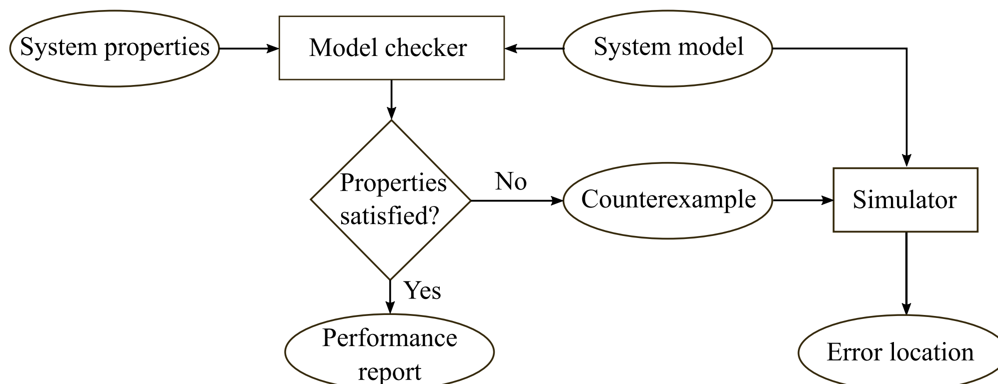

Figure 2.1: Schematic view of model checking process.

directly identify and pinpoint errors within the system [91]. That is, additional methods are often necessary to diagnose the root cause of any discovered inconsistency.

In contrast to theorem proving, model checking is a fully automated and fast algorithmic approach. It verifies whether a system complies with its specified formal properties by analyzing all possible system states [14, 45]. As depicted in Figure 2.1, this technique examines each transition within the model to identify any deviation from the established properties. Upon detecting a violation, the model checker generates a counterexample that illustrates the sequence of events leading to the problematic state. However, model checking faces a significant hurdle known as the *state explosion problem* [159]. This limitation is inevitable in large and complex systems, where the vast number of potential states and transitions can exceed the tool's ability to perform a thorough exploration within reasonable time and memory constraints [44].

Figure 2.2 illustrates a microwave oven modeled as a finite state machine (FSM) with multiple transitions. One property for this model, labeled as 'PO1', asserts that the oven should only start when its door is closed. To verify this property using model checking, both the FSM and 'PO1' are input to the model checker. This tool explores all the FSM's transitions to identify any potential violation of 'PO1'. If it finds a case where oven starts while its door is still

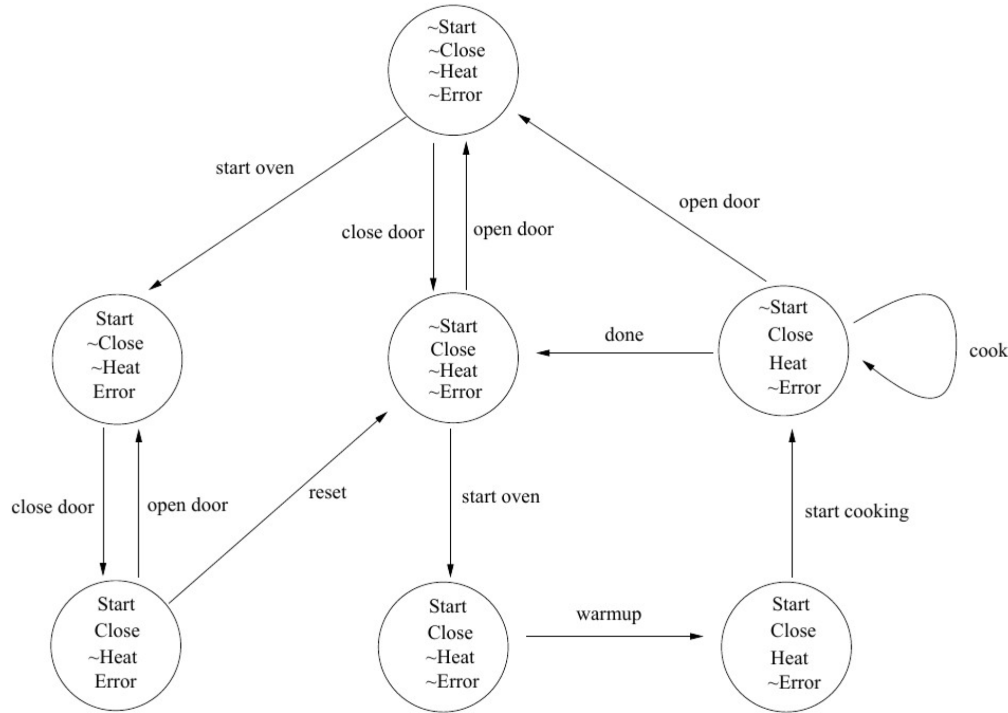

Figure 2.2: FSM of microwave oven model [47].

open, a counterexample is generated, signifying a verification failure. Model checking is an effective method for ensuring a system's compliance with its formally established properties, especially for FSMs of manageable size and complexity. When faced with overly complex or large FSMs, however, theorem proving may be necessary.

2.1.3 Runtime Verification

RV is a dynamic verification approach that utilizes monitors derived from formal system requirements to determine whether the real-time behaviors of a system adhere to its specifications [20, 26, 61, 77, 102, 161]. RV can be employed both during system simulation and actual execution, making it well-suited for systems that interact dynamically with their environment such as machine learning-based applications that require iterative training and adaptation. While model checking and theorem proving analyze system models to verify

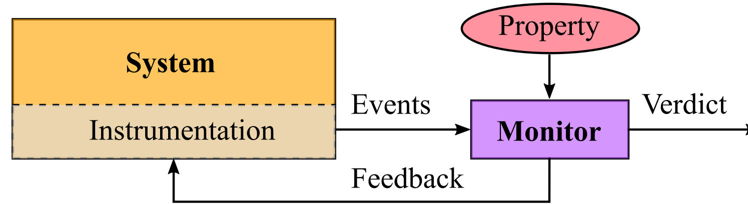

Figure 2.3: Block diagram of RV process.

theoretical properties, RV offers a practical complement. It is employed to provide real-time confirmation of system properties, bridging the gap between theoretical verification and practical operational assurance.

RV involves continuous evaluation of system correctness throughout its operation, as illustrated in Figure 2.3. The process begins with capturing a system property using a formal specification language, leading to the generation of a monitor based on this formalism. The monitor's role is to scrutinize system events such as individual runs, in relation to the defined property. It then provides a verdict indicating whether the system meets or fails the specified property. A crucial aspect of RV, known as *instrumentation*, involves the extraction or recording of data necessary for monitoring activities. If the monitor detects a violation, it can send feedback to the system, triggering corrective actions such as a system reset or activating safety protocols. While this feedback mechanism is optional during the development stage, it may be retained in deployed systems for enhancing safety and reliability. This chapter provides information regarding each stage of the RV process.

2.1.4 Verification Technique Challenges

Understanding the strengths and limitations of various verification approaches is important for choosing the most effective strategy tailored to a specific application. Table 2.1 provides a comparative analysis of different techniques with a focus on their applicability in real-time

Table 2.1: Comparison of verification methods in RTES.

Aspect \ Method	Testing	Theorem proving	Model checking	Runtime verification
Methodology type	execution monitoring	static analysis	static analysis	execution monitoring
Automation level	manual/automated	manual	automated	automated
Real-time capability[1]	low	low	moderate	high
Computer resources[2]	high	very high	high	moderate
Manual effort[3]	moderate	high	high	low
Complexity management[4]	moderate	high	high	moderate
Error detection[5]	pre-facto	pre-facto	pre-facto	post-facto
Applicability	development deployment	development	development	development deployment

[1] Method's ability to handle real-time behavior and timing constraints.
[2] The required level of computational resources such as CPU and memory.
[3] The required level of manual effort such as human labor and expertise.
[4] Method's ability to handle complex system designs and requirements.
[5] Detecting errors before they occur (pre-facto) and detecting errors as they occur or afterward (post-facto).

embedded systems (RTES). These systems are used across various domains and require verification methods that can address a wide range of requirements at different levels. Testing, while a fundamental aspect of system development, requires separate approaches to cover the varying characteristics of RTES [33]. This dynamic approach is often costly and time-consuming, especially for applications where extensive test cases are needed [81]. Another issue is the risk of undiscovered bugs in complex scenarios due to its limited coverage [10]. Even with advances in automated test case generation, considerable manual effort is needed in test design and analysis. These limitations reduce the efficacy of testing in identifying rare or complex bugs, necessitating additional verification techniques for complex RTES.

Theorem proving thoroughly validates foundational properties of a system, however, its application to temporal properties can be quite taxing and labor-intensive [124]. This method can deal with complex theoretical constructs during the development phase but often results

in extensive and intricate proofs that can be difficult to interpret and manage [87]. One of its most significant limitations is the requirement for substantial effort and expertise, making it a resource-intensive approach [9]. Theorem proving, although being powerful and accurate, is often complemented by other methods to fully address the diverse and practical challenges encountered in RTES development [158].

Model checking exhaustively verifies state-based properties and may require significant time and memory to explore the entire state space of a model. While it can handle temporal properties, its effectiveness is often constrained by the complexity and size of the system under analysis. Although model checking is widely used in industry due to the automatic tools available [166], the state explosion problem inhibits its applicability in complex systems. This limitation is a key consideration when choosing model checking for RTES, as it best suits control-oriented rather than data-intensive applications.

Unlike testing and SV approaches, RV analyzes system behavior during execution, making it valuable for high-assurance and safety-critical applications where real-time monitoring is vital. RV monitors are usually lightweight to conform to the strict time and resource constraints of RTES [26]; however, they still require computational resources for their continuous operation. The effectiveness of RV heavily relies on the accuracy and completeness of the properties it monitors [74]. Incomplete or inaccurate monitoring logic may result in missed errors or false alarms, undermining the reliability of the verification process. By focusing solely on the current system execution, RV offers partial assurance, making it a cost-effective method suited to corner cases, environmental complexities, and subtle anomalies in large and complex RTES [61]. Notably, for critical systems requiring high levels of assurance, RV may need to be complemented with other verification techniques during the development stage to ensure comprehensive verification and validation.

2.2 Properties and Specifications

The first step of the RV process is capturing the desired system property in a formal specification language such that a monitor can be derived from it. The diverse range of formalisms applicable to RV is described, with a selection made and justified. Not all system properties are *monitorable*, which is also discussed.

2.2.1 Formal Specification Languages

A system's property is characterized by its behavioral transformations over time, which update system internal states through various internal or external actions. A single system execution is termed a *trace*, which consists of a finite sequence of events and states. The act of expressing a system's property in textual form to define a set of traces is known as *system specification* [20]. Figure 2.4 depicts a simple state machine for a traffic light where the Boolean variables *green*, *yellow* and *red* represent the light status. The system property asserts that the traffic light should always transition to yellow after being green and before turning red. This property can be stated in English as: "PT1: The light must turn yellow before turning red and after being green." Based on this specification, a trace similar to $\tau_0 = green, yellow, red, green$ would satisfy 'PT1' as it adheres to the specified sequence of color changes. In contrast, trace $\tau_1 = green, green, red, green, yellow$ violates 'PT1', as it indicates a direct transition from green to red without the intermediate yellow state.

Specifications articulated in natural languages like English often suffer from vagueness and ambiguity, as natural languages inherently lack precision. In the above example, it is unclear whether the light eventually turns red or stays yellow after being green. In contrast, formal specification languages offer succinctness and precision, making them suitable for capturing system properties with mathematical notations. Temporal logic (TL) [98] is a well-known

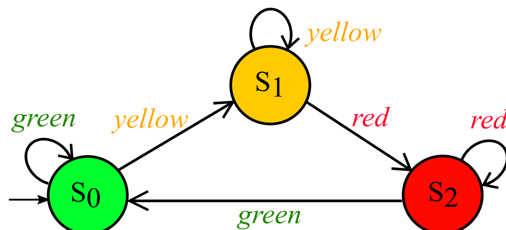

Figure 2.4: State machine diagram of traffic light, which S_0 is the initial state.

formal specification language that captures properties by describing behaviors over time using temporal modalities. For example, 'PT1' can be formally expressed in TL as (2.1), where `G` (`Globally`) and `U` (`Until`) are temporal operators.

$$\mathrm{G}((green\ \mathrm{U}\ yellow)\ \mathrm{U}\ red) \tag{2.1}$$

This TL formula specifies that whenever the light is green, it must inevitably turn red, but only after remaining yellow for some time. Based on (2.1), the transition to red is a mandatory future event during execution to satisfy this requirement. However, the formula also allows for the possibility that the light might stay yellow for an indefinite period before turning red, as indicated by the `Until` operator (will be elaborated in Section 2.2.2). This subtlety shows that effectively capturing properties in a formal language requires a thorough understanding of the principles of formal methods and also a deep familiarity with the specific formal specification language being utilized.

Formal languages are comprised of a finite set of symbols or tokens, and are characterized by three essential components: (1) a syntax, which dictates the allowed notations and grammar rules within the language, (2) semantics, which specify the meanings of the language elements, and (3) pragmatics, which interpret the expressions in the context they are used [133]. There exists a variety of formal specification languages designed to capture system behaviors with their main difference being the paradigm they rely on. For example, TL is a history-

based specification language that characterizes the acceptable sequences of events over time. On the other hand, Z [150] is a state-based language, which denotes the system's state and the operations that modifies it. A complete classification of different formal specification languages is provided by Lamsweerde [99].

Since RV monitors are often derived from high-level system specifications, choosing a concise and expressive language is crucial for efficient monitor generation. Languages permitting efficient monitor generation are also needed. Commonly used specification languages in RV include TL [25, 57], regular expressions [117, 144] and state machines [46, 71]. While regular expressions and state machines are more expressive than TL, regular expressions and TL make concise specifications. Each language has its strengths depending on the specific property being captured. For example, regular expressions are usually suitable for properties that are better expressed in a negated form. Havelund and Reger provide a thorough comparison of these languages in the context of RV [74]. They suggest that although the choice of specification language highly depends on the application, TL is generally more effective for a diverse range of properties. Other specification languages like those based on context-free grammars (CFGs) [116], rule-based languages [16, 18, 72] or stream-based languages [8, 103] are also utilized in RV. This dissertation focuses on linear temporal logic (LTL), the basic variant of TL, to capture functional properties without timing constraints due to its balance of simplicity and expressiveness.

One drawback of formal verification is the multitude of language dialects and notations employed by different tools. This diversity requires verification engineers to become familiar with each tool's unique syntax and semantics, which often needs significant time and effort. To circumvent this issue, this dissertation hides the formalisms by using structured English statements to express functional and timing properties.

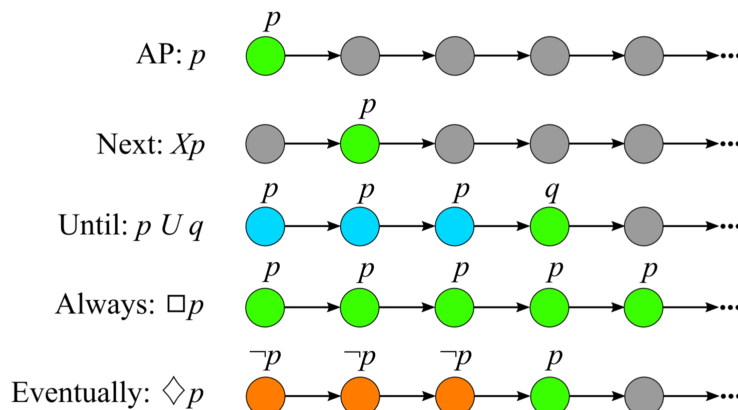

Figure 2.5: Graphical representation of LTL modalities for atomic propositions p and q. The values in gray states are unimportant.

2.2.2 Pnueli's Linear Temporal Logic

Pnueli's LTL is a widely recognized formalism for specifying properties across infinite traces of events [131]. In LTL, each point in time is succeeded by exactly one other moment, emphasizing the sequence rather than the specific timing of events. An LTL formula is constructed using a combination of: atomic propositions (APs) which are indivisible statements about the system; logical operators such as $\neg$ (negation) and $\wedge$ (conjunction) to build complex expressions; and temporal operators such as `U` (until) and `X` (next) to describe the ordering and timing of these propositions in relation to one another.

A simple intuitive meaning of LTL operators for propositions p and q are shown in Figure 2.5. The proposition p can be any single proposition such as a Boolean variable, say *reset*. This AP is true iff (if and only if) *reset* is true at the initial state. The `Next` operator (`X`p) specifies that for the statement to be true, p must hold in the next state. The `Until` operator (p `U` q) requires that p remains true until q becomes true; the state of p beyond this point is insignificant. The temporal operators `Always` and `Eventually` are duals of each other. `Always` ($\Box p$) implies p must hold all the time, whereas `Eventually` ($\Diamond p$) denotes p will become true at some point in the future.

A set of LTL formulas for proposition $p \in \mathcal{AP}$ and specifications φ and ψ is defined by the following syntax (also known as a grammar):

$$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \wedge \psi \mid \mathtt{X}\varphi \mid \varphi \ \mathtt{U} \ \psi \mid \Box\varphi \mid \Diamond\varphi \tag{2.2}$$

Assume $\Sigma = 2^{\mathcal{AP}}$ is a finite alphabet, and let $\omega = a_0 a_1 a_2 a_3 ... \in \Sigma^\omega$ be an infinite word (trace), where $a_i \in \Sigma$, $i \in \mathbb{N}$ is a set of APs at time position i, and Σ^ω is the set of infinite words over Σ. The LTL semantics are described in (2.3), where $\omega, i \models \varphi$ means infinite word ω at position i satisfies formula φ.

$$\begin{aligned}
&\omega, i \models true && \\
&\omega, i \models p && \text{iff} \quad p \in a_i \\
&\omega, i \models \neg\varphi && \text{iff} \quad \omega, i \not\models \varphi \\
&\omega, i \models \varphi \wedge \psi && \text{iff} \quad \omega, i \models \varphi \text{ and } \omega, i \models \psi \\
&\omega, i \models \mathtt{X}\varphi && \text{iff} \quad \omega, i+1 \models \varphi \\
&\omega, i \models \varphi \ \mathtt{U} \ \psi && \text{iff} \quad \exists j \geqslant i : (\omega, j \models \psi) \text{ and } \forall i \leqslant l < j : (\omega, l \models \varphi) \\
&\omega, i \models \Box\varphi && \text{iff} \quad \forall j \geq i : (\omega, j \models \varphi) \\
&\omega, i \models \Diamond\varphi && \text{iff} \quad \exists j \geq i : (\omega, j \models \varphi)
\end{aligned} \tag{2.3}$$

2.2.3 Linear Temporal Logic over Finite Paths

The LTL semantics described in (2.3) are defined over infinite words (Σ^ω). However, RV only focuses on monitoring current system execution, which is a finite prefix of potentially infinite traces. The direct application of traditional LTL in the context of RV encounters practical limitations, particularly due to the `Next` operator which the `Eventually` and `Always` operators are also derived from. In what follows, the limitations and potential adaptations and

solutions are discussed.

Consider the property "there is always no error," which can be expressed in LTL as $\square\neg e$ where e is an AP denoting an error occurrence. If an error occurs at the second moment of the execution ($u = \neg e, e$), any (finite or infinite) trace starting with the prefix $u = \neg e, e$ would contradict the property, making monitor verdict as false. Conversely, the prefix $v = \neg e, \neg e$ yields an inconclusive outcome since it can be extended to a path that satisfies $(\neg e, \neg e, \neg e, ...)$ or violates $(\neg e, \neg e, e, ...)$ the property. In other words, the monitor's decisive output for such cases depends on the trace continuation. The negation of this property can be stated as "an error eventually occurs" or $\lozenge e$ in LTL terms, demonstrating how the interpretation of LTL properties adjusts in the context of finite versus infinite traces.

The above example demonstrates that if a current execution either satisfies or violates a specified property, then all infinite extensions of that prefix will lead to the same verdict, making the prefix extendable to any infinite trace without changing its outcome. Such prefixes are categorized as *good* if they satisfy the property or *bad* if they contradict it [96]. In some cases, however, the monitor cannot issue a conclusive decision based on the current execution alone, which makes the result *unknown* and the trace *ugly*. For a language $L \subseteq \Sigma^\omega$ consisting of infinite words (ω), a finite word $u \in \Sigma^*$ is defined as good, bad or ugly as shown in (2.4), where Σ^ω is the set of all infinite words and Σ^* is the set of all finite words. This classification assists in managing finite traces in RV and acknowledges the inherent limitations when extending from finite to infinite behavior.

$$u \in \Sigma^* = \begin{cases} \text{a good prefix for } L, & \text{if } \forall \omega \in \Sigma^\omega : u\omega \in L \\ \text{a bad prefix for } L, & \text{if } \forall \omega \in \Sigma^\omega : u\omega \notin L \\ \text{an ugly prefix for } L, & \text{otherwise} \end{cases} \tag{2.4}$$

Since traces are finite length in practice, it is necessary to adopt LTL semantics to handle trace termination. One development is two-valued LTL semantics tailored for finite traces [50, 57, 67]. Eisner et al. introduce $\text{LTL}^{\mp}$ semantics over truncated paths based on three operational views: weak (LTL^{-}), neutral (null) and strong (LTL^{+}) [57]. In ambiguous scenarios within the current trace, the specification is considered true under the weak view, false under the strong view, and depends on the event in the neutral view. Under the weak view, $\Diamond\varphi$ is satisfied by any finite path, while $\Box\varphi$ holds if φ is consistently true throughout the execution. In the strong view, $\Diamond\varphi$ is true if φ occurs at any point in the trace, whereas $\Box\varphi$ is never satisfied in finite paths. The neutral view specifies that $\Diamond\varphi$ is true if φ occurs at some state during execution, and $\Box\varphi$ holds if φ remains true for the entire execution. The formal semantics for `Eventually` and `Always` of $\text{LTL}^{\mp}$ are shown in (2.5) and (2.6), respectively, where $|u|$ is the length of finite word u.

$$\begin{aligned} u &\models^{-} \Diamond\varphi && \text{iff } \exists j : u\omega, j \models \varphi \\ u &\models \Diamond\varphi && \text{iff } \exists j < |u| \colon u, j \models \varphi \\ u &\models^{+} \Diamond\varphi && \text{iff } \exists j < |u| \colon u, j \models \varphi \end{aligned} \tag{2.5}$$

$$\begin{aligned} u &\models^{-} \Box\varphi && \text{iff } \forall j < |u| \colon u, j \models \varphi \\ u &\models \Box\varphi && \text{iff } \forall j < |u| \colon u, j \models \varphi \\ u &\models^{+} \Box\varphi && \text{iff } \forall j : u\omega, j \models \varphi \end{aligned} \tag{2.6}$$

Bauer et al. present LTL_3 semantics by expanding the truth domain to three values: $\mathbb{B}_3 = \{\top, \bot, ?\}$ [23, 26]. Based on this approach, a monitor evaluates a good prefix to true ($\top$) and a bad prefix to false ($\bot$). If the current execution provides insufficient information to determine a satisfaction or violation, monitor output is inconclusive (?) for this ugly trace. The truth value of an LTL_3 formula φ is represented as $[u, i \models \varphi]_3$ and defined as:

$$[u \models \varphi]_3 = \begin{cases} \top & \text{if } \forall \omega \in \Sigma^\omega : u\omega \models \varphi \\ \bot & \text{if } \forall \omega \in \Sigma^\omega : u\omega \not\models \varphi \\ ? & \text{otherwise} \end{cases} \tag{2.7}$$

For a more specific evaluation of inconclusive cases, Bauer et al. introduce RV-LTL, which extends the truth domain to four values: $\mathbb{B}_4 = \{\top, \top^p, \bot^p, \bot\}$ [24, 25]. Monitor decision for good or bad prefixes is similar to LTL_3. However, for ugly traces, the monitor leads to a possibly true ($\top^p$) or possibly false ($\bot^p$) verdict based on the weak or strong versions of the formula, respectively. The truth value of RV-LTL formula φ can be written as:

$$[u \models \varphi]_{RV} = \begin{cases} \top & \text{if } [u \models \varphi]_3 = \top \\ \bot & \text{if } [u \models \varphi]_3 = \bot \\ \top^p & \text{if } [u \models \varphi]_3 =? \text{ and } [u \models \varphi]_{\mp} = \top \\ \bot^p & \text{if } [u \models \varphi]_3 =? \text{ and } [u \models \varphi]_{\mp} = \bot \end{cases} \tag{2.8}$$

Past-time LTL (ptLTL) is a variant of LTL with modalities referring to the past events and is widely used in RV [40, 75, 135, 147]. For example, the modalities `Next` and `Until` are replaced with `Previously` and `Since` in ptLTL, respectively. Since ptLTL only focuses on the execution history and not its future, it can be used over finite traces. Consequently, ptLTL is highly applicable in RV because it allows monitors to make decisions based on the recorded trace up to the current moment [76]. However, basic LTL, called future-time LTL (ftLTL), is crucial for applications such as avionics, which require assessment based on future states [19]. This forward-looking nature of ftLTL is also beneficial during the design and pre-deployment stages where engineers are interested in proving system properties over all possible future executions, not just those that have already occurred. Due to GROOT's

versatility goal, LTL_3 is chosen for monitoring of functional requirements.

2.2.4 Real-Time Properties

Real-time systems often require precise quantitative temporal constraints including specific timeouts and delays. However, LTL represents qualitative temporal properties such as eventually occurrences. To overcome this limitation, various extensions of LTL have been developed to include temporal aspects explicitly. Metric temporal logic (MTL) and signal temporal logic (STL) are two variants that are widely used in RV [112, 113]. MTL integrates real-time constraints and specifies temporal distances between events [94]. For example, consider a temperature control system in a greenhouse which requires "the temperature to remain below 30°C within the first 60 minutes after the cooling system starts." This property can be expressed in MTL as shown in (2.9). The `Always` ($\Box$) operator ensures the condition is constantly monitored when the cooling system is activated, while the `Eventually` ($\Diamond_{[a,b]}$) operator indicates the time constraint for this condition. Note that this example assumes temperature readings are discrete and taken at one-minute intervals.

$$\Box(cooling_activated \rightarrow \Diamond_{[0,60]}(temp < 30)) \tag{2.9}$$

STL enhances MTL by focusing on continuous signals, which makes it suitable for real-time systems processing continuous data streams [111]. STL provides a powerful method for capturing and monitoring of complex temporal dynamics within cyber-physical systems [82]. In the above example, if the greenhouse temperature is considered as a continuous real-valued signal, the property can be expressed in STL as follows:

$$\Box(cooling_activated \rightarrow \Diamond_{[0,60]}(temp(t) < 30)) \tag{2.10}$$

2.2.5 Monitorable Properties

In general, there are two different types of properties: *safety* and *liveness* [6, 97]. Safety properties ensure that something bad never happens during system operation, while liveness properties confirm that something good eventually occurs. For example, “there is always no error” is a safety property, whereas “for every request, there will be an acknowledgment” is a liveness property. Peled and Havelund redefine temporal properties in the context of RV and introduce *co-safety* and *morbidity* properties as duals of safety and liveness properties, respectively [125]. Co-safety properties can be satisfied by a finite (good) prefix, meaning they can detect the satisfaction of certain conditions in a finite context. On the other hand, morbidity properties refer to conditions that cannot be satisfied within any finite path of execution as their fulfillment might always be pending further action or event.

Property $\Box p$ (`Always` p) is a combination of safety and morbidity properties, indicating p must always hold throughout the trace. A bad prefix such as $u = p, \neg p$ can refute this property, but there is no finite trace to verify it. Conversely, property $\Diamond q$ (`Eventually` q) includes both co-safety and liveness aspects, stating q is expected to occur at some future point. While a bad prefix cannot disprove such properties, a good prefix such as $v = \neg q, q$ can verify them. As explained before, some properties cannot reach a decisive assessment within a finite trace. For example, consider property $\Box(p \rightarrow \Diamond q)$, expressing whenever p is true, q eventually holds. No finite trace can invalidate this property since q could always occur just beyond the observed sequence. Similarly, no finite observation can verify the occurrence of q for each p. This leads to the classification of such traces as ugly, with their main issue being their ambiguity as they hinder the monitor yielding a conclusive verdict.

A property is called *monitorable* if the monitor can lead to a definitive outcome based on the presence of solely good or bad prefixes within a trace [73]. Consequently, properties associ-

ated with ugly prefixes are labeled as *non-monitorable* [26]. Safety and co-safety properties are monitorable by identifying either a bad or a good prefix. Thus, this dissertation only addresses safety and co-safety monitorable properties.

2.2.6 Property Specification Patterns

Despite the critical role of formal methods in developing autonomous and safety-critical systems, generating accurate formal requirements remains a significant obstacle, hindering the practical application of these methods [140]. In the late '90s, Dwyer et al. proposed a set of property specification patterns (PSP) for various formalisms including LTL [56]. Although this aids practitioners in recognizing familiar specifications, it still requires them to be familiar with the underlying formal language semantics.

Expanding on this, Konrad and Cheng added MTL patterns to address real-time requirements and presented these patterns in structured English to enhance their accessibility [92]. Autili et al. further evolved this concept by scaling real-time properties and providing the PSPWizard framework, which offers a unified PSP catalogue and maps structured English to diverse formalisms [12], with application in various research projects [27, 28, 115]. In a recent study, Czepa and Zdun evaluated the understandability of PSP against LTL using 216 participants [49]. The results showed that PSP-based requirements are easier and faster to learn by non-experts, highlighting PSP's practical utility.

The effectiveness of RV highly depends on the precision of specifications, as they form the basis for monitor generation. While PSP have found applications in RV [36, 100, 149], their use is not as widespread as in other domains. One example of a PSP-based RV framework is RuSTL that translates Autili et al.'s patterns into STL and then generates offline monitors suitable for log file analysis [89]. To simplify RV for practitioners, this dissertation leverages

PSP to structure both functional and timing specifications within the GROOT framework.

2.3 Runtime Verification Monitors

Monitors are computational entities derived from user-defined, high-level system specifications expressed in a formalism. Similar to a state machine, a monitor observes system execution at each step and processes its inputs and transitions to produce an output [60]. This output usually belongs to a binary truth domain, $\mathbb{B} = \{true,\ false\}$. However, monitors can be more informative about the trace and the nature of any violation encountered. This section discusses the major aspects of monitor generation and deployment.

2.3.1 Online vs. Offline Monitoring

Monitoring operation can be either online (concurrently with system operation) or offline (at a later time) [20, 143]. Online monitoring processes events as they occur and allows for real-time verification with incrementally obtained data. In contrast, offline monitoring analyzes system behaviors using recorded traces and log files after data collection is complete. While an online monitor is adaptable for offline use, the reverse scenario is typically infeasible due to the real-time data processing capability of online monitors.

The decision to employ online or offline monitoring depends on the specific requirements of the application and system constraints. Offline monitoring minimizes intrusion by operating on existing logs, with the primary overhead related to data recording. However, its main drawback is detecting the problems after they occur. Conversely, online monitoring can identify the violation during system execution, albeit at the expense of increased overhead. In safety-critical and real-time applications such as autonomous vehicles or flight control

systems, online monitoring is better suited since timely error detection is critical [21, 165]. This dissertation focuses on online monitoring as it targets RTES.

2.3.2 Synchronous vs. Asynchronous Monitoring

Online monitoring can be synchronous or asynchronous depending on the level of monitor interference [37, 38]. Synchronous monitoring integrates closely with the system execution and halts operation upon each event generation until the monitor has processed the event and provided feedback. This tight coupling ensures immediate response to events at the cost of increased system interruption and potential performance degradation. On the other hand, asynchronous monitoring allows the system to operate independently without waiting for the monitor response. This approach reduces the impact on system performance, however, the drawback is a potential delay in violation detection. An intermediate approach, known as partially synchronized monitoring, offers a balance by applying synchronous monitoring selectively to certain critical events while remaining asynchronous to others [62].

2.3.3 Inline vs. Outline Monitoring

Monitors can be deployed in two distinct manners: inline (integrated within the system) or outline (deployed externally) [20, 101]. Inline monitors have direct access to system events but consume the same resources as the system itself. In contrast, outline monitors utilize independent resources to maintain operational integrity even in scenarios where the primary system encounters unrecoverable faults. However, implementing outline monitors might require interfaces or modifications to the system under scrutiny (SUS) to enable comprehensive monitoring. The choice between inline and outline monitoring should be guided by the specific requirements and constraints of the SUS.

2.3.4 Monitoring via Büchi Automaton

Monitors serve as observers that assess system executions against predefined properties. Most specification languages, including LTL, require transforming specifications into a form that can be employed during execution. A common monitoring algorithm for LTL properties is translating the LTL formula into a deterministic Büchi automaton (BA) [67]. Similar to an FSM, a BA [34] consists of a finite set of states, inputs, transitions, and accepted conditions.

Definition 2.1 (Büchi automaton)**.** A BA is defined by a tuple $\mathcal{B} = \langle \Sigma, Q, Q_0, \delta, F \rangle$, where

- $\Sigma = 2^{\mathcal{AP}}$ is the finite alphabet where $\mathcal{AP}$ is the set of APs,
- Q is the finite set of non-empty states,
- $Q_0 \subseteq Q$ is the set of initial states,
- $\delta : Q \times \Sigma \rightarrow 2^Q$ is the transition function,
- $F \subseteq Q$ is the set of accepting states.

The set of infinite words over Σ is denoted by Σ^ω with $\omega = a_0 a_1 a_2 ... \in \Sigma^\omega$ representing an infinite word and each $a_i \in \Sigma$ corresponding to APs holding true simultaneously at position i. In BA, there are no final states as it is defined over infinite words. Instead, it utilizes accepting states F, indicating states where the property modeled by the BA is satisfied. A *run* of automaton $\mathcal{B}$ over ω is a sequence of states $\rho = q_0, q_1, q_2...$ with $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2$ ... such that $q_0 \in Q_0$, $q_i \in Q$, and each transition $q_i \xrightarrow{a_i} q_{i+1}$ follows the transition relation based on the corresponding elements of ω. Let $\text{Inf}(\rho)$ be the set of states visited infinitely often in the run. The automaton $\mathcal{B}$ accepts an infinite word ω iff there exists a run ρ on ω that visits some accepting states belongs to F infinitely many times. The language recognized by BA

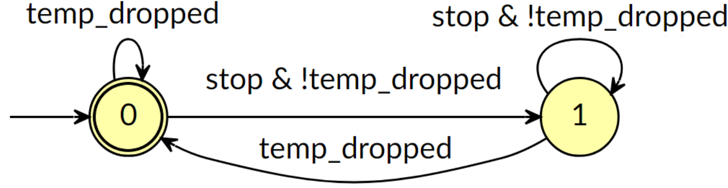

Figure 2.6: BA for thermostat property $\Box(stop \ \mathtt{U} \ temp_dropped)$.

is the set of all words accepted by $\mathcal{B}$, which is formally defined as:

$$L(\mathcal{B}) = \{\omega \in \Sigma^{\omega} | \exists \rho \text{ over } \omega : \text{Inf}(\rho) \cap F \neq \emptyset\} \tag{2.11}$$

A heating system typically ceases operation once the temperature reaches a set threshold, conserving energy until heating is needed again. This behavior can be expressed by the LTL formula $\Box(stop \ \mathtt{U} \ temp_dropped)$. Figure 2.6 illustrates the corresponding BA containing states $Q = \{0, 1\}$, initial state $Q_0 = \{0\}$, and accepting state $F = \{0\}$ (denoted by double circles). The alphabet Σ represents sequences of conditions over $\mathcal{AP} = \{stop, temp_dropped\}$, and the transition function δ determines the next state based on the current state and the input conditions. The automaton moves to and remains in state 1 whenever the temperature is above the threshold. Consequently, an infinite sequence $\omega_0 = (stop \wedge \neg temp_dropped), (stop \wedge \neg temp_dropped), ...$ satisfies the property, while sequence $\omega_1 = (stop \wedge \neg temp_dropped), (\neg stop \wedge \neg temp_dropped), ...$ does not belong to this BA, indicating a flaw in thermostat's logic as it has been activated $(\neg stop)$ without any noticeable temperature reduction $(\neg temp_dropped)$. To represent this BA as a monitor, an *ERROR* state is required to identify any deviation from the acceptable paths.

To generate monitors from LTL specifications using BA, an infinite run is essential, which can be achieved by indefinitely repeating the last state [67]. Several tools facilitate the translation of LTL formulas into BA including Spot [55], LTL2BA [66], and LTL3BA [13].

In this dissertation, the Spot open-source tool is selected due to its performance, robustness and support, which makes it well-suited for industrial applications as well [141]. Spot also offers C++ and Python APIs, which will be discussed further in Chapter 5.

2.3.5 Monitoring via Timed Automaton

While a BA is suitable for synthesizing monitors from functional requirements, its application does not extend to timing specifications which require quantitative analysis. A timed automaton (TA) is a common approach for generating monitors tailored to timing constraints by equipping a finite automaton with a finite set of real-valued clocks for modeling timed behaviors of real-time systems [130]. Clock constraints guard states and transitions to restrict the behavior of the automaton. A TA can be considered as a directed graph such that the nodes and edges are the states and transitions, respectively.

Definition 2.2 (Timed automaton)**.** Let Σ be the finite set of actions (or events), C be the finite set of real-valued clock variables, and $B(C)$ be the set of clock constraints. A TA is a tuple $\mathcal{A} = \langle Q, q_0, F, I, E\rangle$, where

- Q is the finite set of states,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the finite set of accepting states,
- $I : Q \rightarrow B(C)$ is the location invariant that assigns a clock guard g to a state $q \in Q$,
- $E \subseteq Q \times B(C) \times \Sigma \times 2^C \times Q$ is the finite set of edges $e = \langle q, a, Y, g, q'\rangle$, where $q, q' \in Q$ are the current and next states, $a \in \Sigma$ are the actions occurring in state q, $Y \subseteq C$ is the set of clocks to be reset after taking the transition, and g is the guard, which is the clock constraint on the current clock variable.

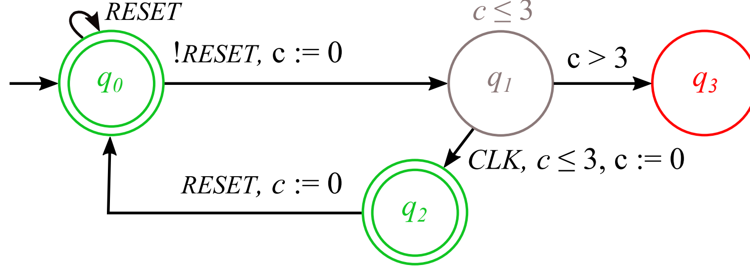

Figure 2.7: Property PT illustrated by a TA.

Definition 2.3 (Deterministic timed automaton)**.** A TA $\mathcal{A}$ is called deterministic iff:

- There is only one initial state.
- For every pair of transitions $\langle q, a, Y_1, g_1, q_1 \rangle \in E$ and $\langle q, a, Y_2, g_2, q_2 \rangle \in E$, the clock constraints g_1 and g_2 are mutually exclusive.

TA semantics defined as a timed transition system that specifies how the automaton behaves over time in response to events and clock constraints [7]. As an example, consider a timing property such as "PT: system clock must start within 3 us after reset is deasserted." Figure 2.7 represents a TA for this requirement, where $Q = \{q_0, q_1, q_2, q_3\}$ is the set of states, q_0 is the initial state, and $F = \{q_0, q_2\}$ are the accepting states (denoted as double green circles). $\Sigma = \{e_0, e_1, e_2\}$ is the set of events, where e_0 (e_2) means "deasserting (asserting) $RESET$" and e_1 declares "starting the clock signal (CLK)". This TA has one clock c with a constraint of $c \leq 3$ and a location invariant of $c \leq 3$ assigned to the non-accepting state q_1 to limit its duration in this state. The TA's clock is assumed to increase every 1 us.

When the $RESET$ signal is deasserted, the TA resets the clock ($c := 0$) and moves to state q_1, where it can stay for a maximum of 3 time units. If event e_1 does not occur before passing 3 units, the TA moves to the non-accepting state q_3 (denoted as a red circle), which indicates a behavior violation. If event e_1 does occur within 3 units, the TA moves to state q_2 and the clock is reset. The accepting state q_2 has no location invariant, which allows the TA to

remain in this state until event e_2 occurs. This TA is deterministic as there is at most one transition for any given state and set of clock values. This example demonstrates that a TA is a straightforward and expressive structure for capturing the timing aspects of a system.

2.3.6 Monitor Implementation

RV monitors can be purely software-based, hardware-based, or a hybrid. Software monitors benefit from the expressiveness of programming languages and can directly access system states and interfaces. However, adding extra software tasks for monitors may significantly impact the system performance [161]. In some applications such as RTES that are constrained in timing and resources, it is essential to minimize interference with system's temporal behavior. Nevertheless, monitor overheads can be manageable during subsystem development prior to integration and deployment.

Hardware monitors have been explored to reduce the time overheads of monitoring on the software stack. Reinbacher et al. present a non-intrusive online monitoring framework for RTES, which creates parallel hardware monitors from past-time MTL (ptMTL) specifications [135]. They also develop a hardware RV framework designed for safety-critical embedded systems such as unmanned aerial systems (UAS) and satellites [136]. This framework utilizes paired synchronous and asynchronous observers to assess the system status without altering the certified onboard software. While these monitors are primarily implemented on field programmable gate arrays (FPGAs), they could also be deployed on an application-specific integrated circuit (ASIC).

Solet et al. extend the application of hardware monitors to embedded systems within system on a programmable chip (SoPC) that integrates both a micro-controller unit (MCU) and an FPGA [146, 147]. Their methodology employs minimal software instrumentation to send

events to monitors generating from ptLTL properties. Mettler et al. introduce a distributed hardware monitoring system designed in a multi-processor system-on-chip (MPSoC) [118]. This architecture uses an integer linear programming-based algorithm for optimal requirement assignment to monitors in order to minimize resource utilization. Martino and Fey propose a framework for generating and optimizing monitors from LTL specifications for FPGA systems [114]. Their approach allows for efficient prediction of system behaviors, offering early warnings for potential specification violations.

2.4 Existing Runtime Verification Frameworks

Several RV frameworks exist for generating monitors from system specifications while differing in aspects such as formalism, monitoring algorithm, placement, deployment, and application domain. Among these, Temporal Rover (TR) is a pioneering commercial RV tool used for applications written in C, C++, Java, Verilog and VHDL [54]. TR allows users to embed LTL or MTL properties directly within the source code through a specialized syntax used for comments. It generates a new version of source file in which these specified properties are transformed into executable segments embedded within the application's code. Similar to assertions, these properties are checked at their designated points in the code.

In a comprehensive survey conducted by Falcone et al., 60 monitoring frameworks are evaluated, offering insights into the current landscape of RV tools [62]. The majority of these frameworks interpret system specifications articulated in TL (mostly a variant of LTL) and generate monitors across various programming languages, with Java, C and C++ being the most common examples. Despite the presence of hardware-based monitoring frameworks, their application remains comparatively less widespread, with a focus mainly on augmenting software monitoring efforts. The decision procedures in monitoring algorithms are gen-

erally categorized into automata-based [67], rewriting-based [142], stream-based [8], and rule-based [72] techniques. These methodologies offer various mechanisms for analysis of system properties along with their unique advantages and application domains. Automata-based methods employ finite automata for property verification, rewriting-based techniques simplify the verification process through property transformation, stream-based approaches handle data streams for real-time analysis, and rule-based systems apply sets of rules for monitoring compliance. This segmentation allows optimized RV approaches based on factors such as application domain, performance, scalability, and ease of integration. The following sections describe and compare select software and hardware RV frameworks to gauge their efficacy and applicability across different domains.

2.4.1 Software-based Monitoring Frameworks

JavaMOP

JavaMOP (Java monitoring oriented programming) is a MOP instance developed for monitoring Java programs' execution [117]. MOP is an approach that integrates system implementation with specifications by embedding monitors directly within the system, eliminating the need for preliminary system verification [40]. JavaMOP automatically generates monitors from specifications expressed in formalisms such as LTL, ptLTL, CFG, FSM, and extended regular expression (ERE).

RV-Monitor

RV-Monitor is based on JavaMOP, but developed to reduce the overhead associated with simultaneous monitoring of multiple properties in Java applications [109]. The inline monitoring tool is effective at observing complex behavioral patterns and supports parametric

properties. This functionality allows for dynamic tracking of property instances through parameters derived from monitored events.

RULER

RULER simplifies the monitoring process by compiling various LTL specifications into a more efficiently executable rule-based system [17]. It utilizes a set of named rules, where each rule is defined by a condition (antecedent) and a body (consequent) to evaluate against the system's runtime behavior. By dynamically activating or deactivating rules based on current observations and system states, RULER can lead to enhanced performance and smoother integration with existing systems.

EgMon

EgMon is an efficient runtime monitoring tool designed for real-time systems [86]. It operates by incrementally taking system state inputs and an MTL formula, then eagerly checks the trace for violations using a dynamic rewriting approach to reduce the input formula as soon as possible. This allows for early detection of violations, which is critical in applications with strict timing constraints.

Copilot

Copilot is a Haskell-based language and monitoring framework designed for ultra-critical embedded systems, which translates ptLTL and ptMTL specifications into C monitors [128, 129]. This framework adopts a stream-based technique, allowing users to define streams using constant values, conditional expressions, and operations over existing streams. The generated monitor periodically samples global variables from the target program and evaluates these

streams against the specified properties.

RiTHM

RiTHM (runtime time-triggered heterogeneous monitoring) specializes in generating monitors for real-time systems by processing C code and LTL properties [122]. To minimize the monitors' area and time overhead, it employs a time-triggered RV approach, which pauses system operations to evaluate properties before resuming execution [30]. Despite the risk of missing violations due to periodic checks, RiTHM suggests several strategies to mitigate this limitation, such as analyzing variables' history, integrating in-line monitors, and employing path prediction techniques to enhance monitoring accuracy.

NuRV

NuRV is an extension of the NUXMV model checker [39] that generates standalone monitors compatible with multiple programming languages including Java, C and C++ [42, 43]. It processes LTL properties, a model of the system, and a finite sequence of system execution, producing either verification results or the monitors themselves. NuRV supports both online and offline monitoring, however, its real-time monitoring functionality is primarily intended for debugging and manual testing purposes due to its operational constraints.

2.4.2 Hardware-based Monitoring Frameworks

P2V

PSL-to-Verilog (P2V) is a tool that facilitates online monitoring of C programs by translating assertions stated in property specification language (PSL) into Verilog-based hardware

monitors [108]. PSL is an LTL extension that enhances its expressiveness by using regular expressions [132]. These assertions are implemented on a softcore processor, specifically the eMIPS (extensible MIPS) processor running on an FPGA. This approach allows for exploiting the information produced by the C compiler with no additional overhead on the software as no software instrumentation is required.

BusMOP

BusMOP is another instance of MOP that targets online monitoring of commercial-off-the-shelf (COTS) peripherals connected via the peripheral component interconnect (PCI) bus [117, 126]. It supports various property specifications including LTL, ptLTL, FSM, and ERE. The monitors are synthesized on an FPGA connected to the system PCI bus to scrutinize bus activities. The goal is to verify communication integrity among peripherals without imposing any execution overhead on the software.

LAOCOÖN

LAOCOÖN automatically generates hardware monitors from PSL or LTL properties to detect hardware Trojan horse (HTH) and fault attacks for Verilog-based systems [51]. This framework also provides a symbolic test bench to instantiate the system under attack along with monitors and verify it against possible attacks. However, its effectiveness diminishes when dealing with large-scale programs or complex property scenarios.

R2U2

R2U2 (realizable responsive unobtrusive unit) was initially developed for real-time monitoring of security threats on UAS and implemented on FPGAs to ensure efficiency and

Table 2.2: A brief comparison of select RV tools and GROOT.

Tool	Formalism	Monitor procedure	Deployment		Application	
			Placement	Instrumentation	Language	Domain
TR [54]	LTL, MTL	automata-based	inline	software, hardware	C, C++, Java, HDL	protocols, reactive systems
JavaMOP [40]	LTL, ptLTL, ERE, FSM, CFG	automata-based	inline	software	Java	object-oriented systems
RV-Monitor [109]	LTL, ERE, FSM	automata-based	inline, outline	software	Java	Java packages
RULER [17]	LTL	rule-based	outline	software	Java	software development
EgMon [86]	MTL	rewriting-based	outline	software	C++	embedded systems
Copilot [128]	LTL, MTL	stream-based	outline	software	C (embedded)	aerospace, automotive
RiTHM [122]	LTL	automata-based	inline	software	C	embedded systems
NuRV [42]	LTL	automata-based	inline, outline	software	Java, C, C++, Dot, Lisp.	software development
P2V [108]	PSL	autoamata-based	outline	hardware	C	general purpose
BusMOP [117]	LTL, ptLTL, ERE, FSM	automata-based	outline	hardware	HDL	COTS peripherals
LAOCOÖN [51]	PSL, LTL	autoamta-based	outline	hardware	Verilog	hardware security
R2U2 [85]	LTL, MTL MLTL	automata-based	outline	hardware	Verilog	aerospace, military
GROOT [137, 138]	LTL, TA	automata-based	outline	software, hardware	C, HDL	real-time embedded systems

[1] The Generalized Runtime mOnitOring Tool introduced in this dissertation.

isolation [119]. A newer version is enhanced to target both hardware (FPGA) and software (C/C++) environments [85]. This framework supports a range of TL specifications including LTL, MTL, and mission-time LTL (MLTL) [107]. An improved version of the R2U2 algorithm has been effectively implemented on the onboard systems of Robonaut2, which is a resource-constrained humanoid robot designed for assisting astronauts [88].

2.4.3 Comparison of Existing Monitoring Tools

Table 2.2 compares the reviewed RV frameworks based on the main metrics discussed in this chapter. As noted before, LTL emerges as the common formalism for capturing system specifications due to its efficiency in RV. Inline monitoring requires source code access, making it unsuitable for treating the system as a black box. While embedding monitors within

the code provides internal state access, this approach is impractical for legacy products or systems requiring platform-independent monitoring. TR, JavaMOP, and RiTHM, although notable in RV, are less suited to embedded systems with strict timing or resource constraints, and are more relevant for software verification during development and testing. In the context of application domains, RV-Monitor and RULER are tailored for Java environments, whereas EgMon and Copilot target embedded systems. NuRV has limitations in real-time monitoring scenarios, although it extends the capabilities of the nuXmv model checker.

P2V and BusMOP utilize FPGAs for monitor implementation to ensure software application timing remains unaffected. P2V requires a dynamically extensible soft-core processor on an FPGA, which may not be ideal for industrial applications. Conversely, BusMOP is designed for monitoring COTS peripherals through bus transactions, having a more focused application scope. LACOÖN is dedicated to hardware applications, specifically targeting simple fault attack detection, while R2U2 provides non-intrusive real-time monitoring for both functional and timing properties of ultra-critical systems.

Choosing the right runtime monitoring tool depends on the application's specific needs, such as the choice between software and hardware verification, the complexity of properties to be monitored, and familiarity with the tool's formalism. As it will be explained in Chapter 4, GROOT monitors treat target applications as black boxes, eliminating the need for software instrumentation. GROOT also provides co-synthesis of software and hardware monitors, focusing on both functional and timing requirements of RTES.

2.5 Summary

This chapter explains essential RV concepts, including selecting specification languages for formalizing system properties, generating executable monitors from specifications, and eval-

uating tools for these purposes. While the choice of formalism is application-dependent, LTL stands out for its efficiency and conciseness across varied property types. Most tools are designed for software applications, either through software or hardware monitor implementations. A comparative analysis of these tools is presented to motivate developing a framework that supports both software and hardware monitors for real-time embedded systems, which automatically generates monitors using pseudo-English requirements.

Chapter 3

Preliminary Work

This dissertation builds on a collaborative project initiated in Spring 2018 with GE Global Research, which focused on applying formal methods to monitor UAS behaviors. Over three one-year phases, the project evolved from generating monitors from hand-written LTL formulas for various hardware platforms to aiming for automation in monitor synthesis during the third phase. This chapter outlines the progression and identifies the advancements and limitations encountered through each phase, setting the stage for this dissertation's focus on automating monitor synthesis.

3.1 Phase 1: Intel Aero Drone Monitoring

The first phase of the project was part of a Master's thesis and involved monitoring an Intel Aero drone [80] and enforcing safety measures during flight [152, 153]. The drone behaviors are in line with ASTM F3269-17 standard (*Standard Practice for Methods to Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions*) [11]. This standard describes a runtime assurance (RTA) architecture for flight safety of a UAS containing untrusted complex functions. This phase utilized drone's onboard MAX 10 FPGA for monitor implementation to ensure isolation from the software and maintain application performance. The monitors were purely additive, meaning they can be added to existing systems without requiring any modification. Figure 3.1 depicts the RTA architecture where

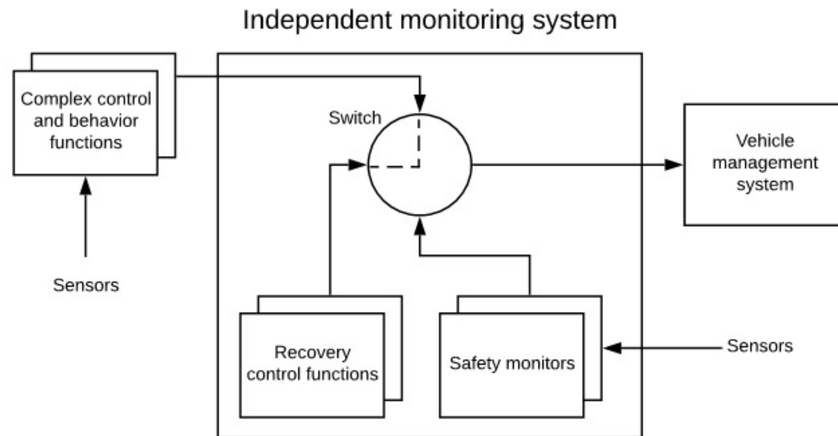

Figure 3.1: Adding isolated safety monitors to the UAS [153].

the monitors and recovery control functions (RCFs) are isolated from the application.

The monitors enforced a 3D virtual cage around the drone and tracked its movements by using indoor ultrasonic positioning or outdoor GPS. Each boundary was hard-coded into the FPGA, and each fence was monitored independently by one of the five parallel monitors. These monitors concurrently tracked the drone's position and velocity to ensure it remained within each boundary. A designated slowdown zone anticipated and prevented boundary breaches by stopping the drone's motion. The specific AP values corresponding to different flight scenarios are illustrated in Figure 3.2. For example, in Scenarios 4 and 5 where the UAS is in the slowdown zone, if the monitors determine that the current UAS speed would eventually put it outside the cage, a recovery action (such as immediate landing) is activated and relayed to the UAS management system.

The process for creating monitors in this phase is depicted in Figure 3.3. Initially, LTL specifications are manually defined and then transformed into BA using the Spot tool. A C++ script, named LTL2C, utilizes the Spot library to convert these automata into C code. The resulting code is then manually completed as a C function, which is subsequently converted

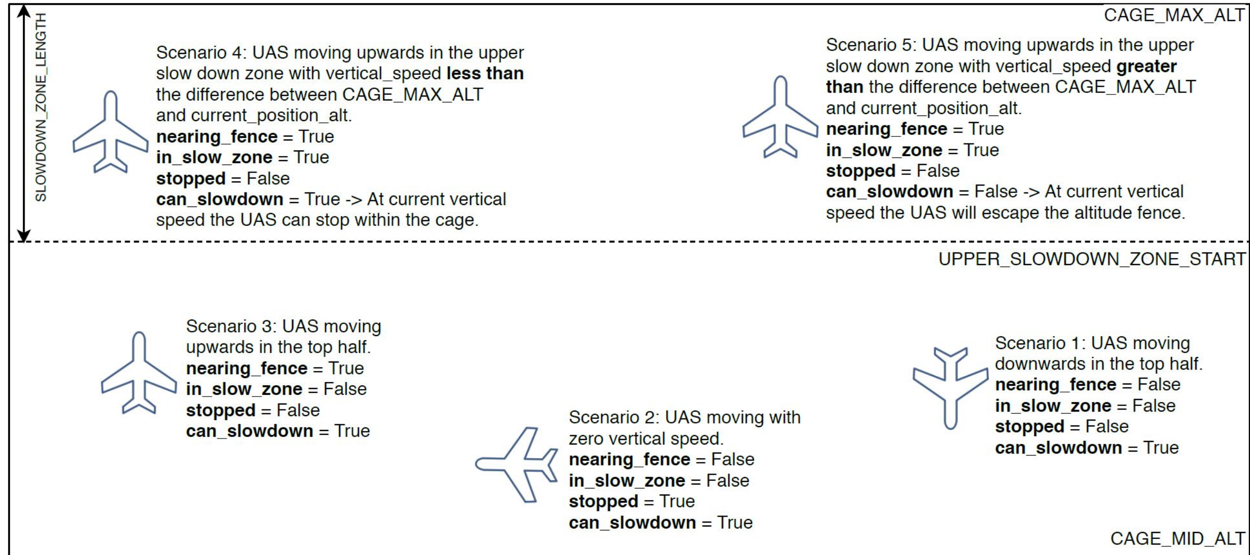

Figure 3.2: AP values for different flight scenarios [152].

into HDL (both Verilog and VHDL) via the Xilinx high-level synthesis (HLS) tool [162]. Finally, the EBMC model checker [95] verifies the Verilog translation against the formally specified requirements. Similar to LTL, SystemVerilog assertions (SVAs) for model checking are manually crafted and integrated into the source code before applying EBMC.

For example, to ensure the UAS remains below a maximum altitude, property *max_alt* specifies that as the UAS ascends into the slowdown zone, it must reduce its vertical speed until it ceases upward movement to avoid collision with the fence. This specification was formally captured in LTL as shown in (3.1). The AP values in this equation are determined by monitoring UAS position and calculating its vertical velocity. If the GPS data shows the altitude surpasses a predefined threshold, proposition *in_slow_zone* becomes true. Figure 3.4 shows

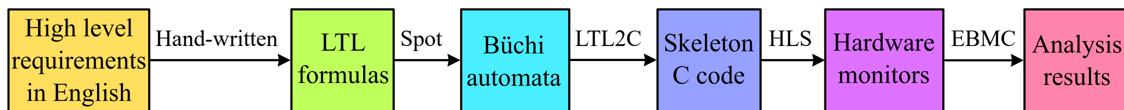

Figure 3.3: The first prototype of monitor synthesis flow.

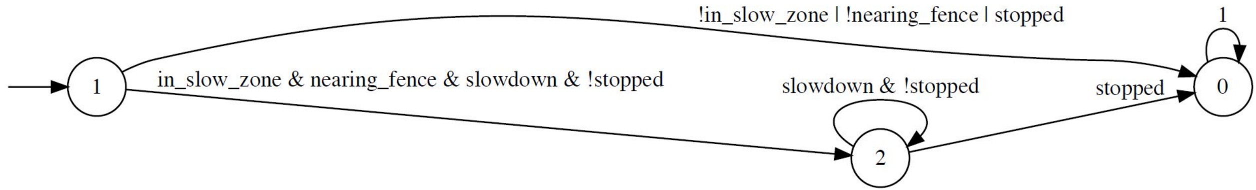

Figure 3.4: BA for the *max_alt* requirement [153].

the automaton generated by Spot for the LTL requirement:

$$(nearing_fence \ \& \ in_slow_zone) \implies (can_slowdown \ \mathtt{U} \ stopped) \tag{3.1}$$

Considering the importance of monitor latency and overhead, various implementations for the Aero monitors were explored including sequential and parallel approaches in both software and hardware, with parallel hardware implementation having the minimum execution time. This led to encapsulating all five monitors in the *within_cage* function as shown below.

Listing 3.1: C function for the *within_cage* monitor wrapper.

```
bool within_cage(int alt, int ver_speed, int lat, int lat_speed,
                 int lon, int lon_sped) {
    return (within_cage_max_alt(alt, ver_speed) &
            within_cage_min_lat(lat, lat_speed) &
            within_cage_max_lat(lat, lat_speed) &
            within_cage_min_lon(lon, lon_speed) &
            within_cage_max_lon(lon, lon_speed)); }
```

Due to the manual creation of LTL formulas and the use of several tools for generating hardware-encoded monitors, the EBMC model checker validates the final HDL code using several hand-written SVAs for each monitor. For example, (3.2) presents the SVA for the *max_alt* monitor, which is an anticipated failure according to Scenario 5 in Figure 3.2. EBMC catches this violation, indicating that the UAS cannot slow down and stop within

the cage. This is due to the velocity of the UAS exceeding the threshold and approaching the perimeter of the fence, implying insufficient time for deceleration and timely stop.

$$(nearing_fence \,\&\, in_slow_zone) \mid\!\rightarrow (can_slowdown \text{ until } stopped) \tag{3.2}$$

Integrating isolated monitors into the Intel Aero drone's FPGA demonstrated the RTA architecture's adaptability to existing systems without software modifications or performance impact. However, the static coordinates of the virtual cage required FPGA reprogramming for new limits, which precludes commercial use. Furthermore, the five monitors for position and speed fully utilized the MAX 10 FPGA's resources. To provide comprehensive UAS safety assurance, additional monitors are needed. As Intel discontinued support for the Aero drone, Phase 2 switched to the Xilinx Zynq UltraScale+ ZCU104 evaluation board [163] for monitor implementation.

3.2 Phase 2: Virtual Drone Monitoring

Phase 2 was also done by MS students, synthesizing more monitors to cover all essential sensor data for autonomous flight beyond visual line of sight (BVLoS), focusing on drone applications for delivery and data recording. Unlike Phase 1's static geofence, this phase considered a dynamic flight plan to eliminate the need for FPGA reprogramming with each new plan. A virtual drone in the Gazebo simulation environment [145] implemented dozens of safety monitors using hardware-in-the-loop (HIL) simulation [110, 134]. As shown in Figure 3.5, the drone's functionalities, sensory systems, and environmental interactions were modeled in the simulator, while the safety monitors and RCFs were implemented in hardware.

Figure 3.6 illustrates a flight plan where a drone takes off from Point 1, the starting point,

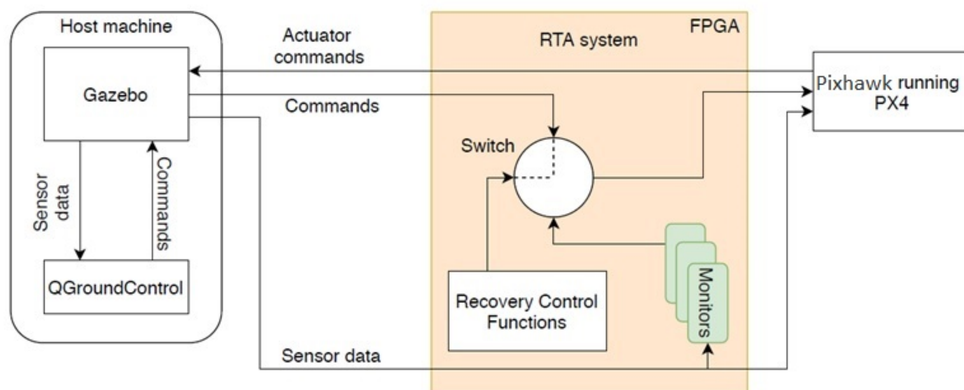

Figure 3.5: HIL simulation with RTA system in place. Details of this architecture can be found in [110].

cruises towards its destination and lands at Point 4, the final point. Different monitors are activated for each phase of the flight such as parallel monitors scrutinizing speed and altitude during takeoff. In the cruise stage, monitors for runtime position and speed are activated to ensure the UAS does not deviate excessively from the flight plan and also flies within the permitted range. Additional monitors for altitude verify the UAS remains within safe flying limits. When the UAS reaches the landing point, specific position and speed monitors are triggered to check the UAS status until it ultimately lands. Landing also involves critical over-speed and under-speed monitoring to prevent the UAS from landing too fast or stalling. The monitor synthesis process used the same tool flow as in Figure 3.3.

A collision avoidance monitor was generated from the LTL formula shown in (3.3) to protect the UAS against potential collisions with both mobile and stationary objects. This monitor activates upon receiving obstacle details (for example, its distance and convergence rate) from an obstacle detection system. It alerts the UAS about nearby or approaching obstacles and prompts suitable safety actions depending on the situation. As depicted in Figure 3.7, the operator can either hold the UAS in place or reposition it based on the obstacle type and proximity. Once the obstacle no longer poses a threat, a *continue* command is issued for

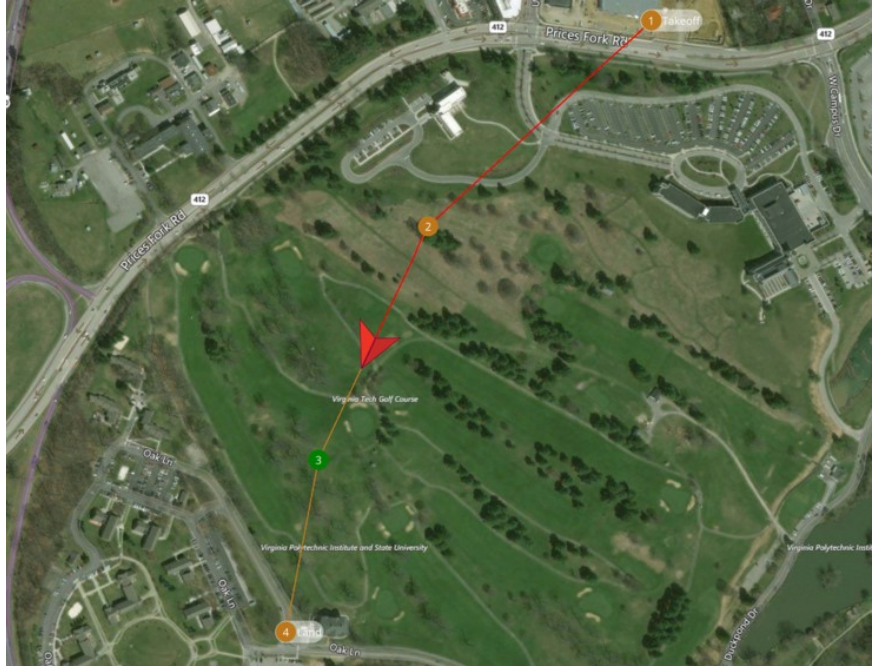

Figure 3.6: A flight plan example for BVLoS [110].

the UAS to resume its mission. The position and speed monitors may require adjustments if the UAS is relocated to avoid collisions.

$$(\mathit{obstacle_in_vicinity}\ \&\ \mathit{approaching}) \implies (\mathit{can_avoid}\ \mathtt{U}\ \mathit{obstacle_avoided}) \tag{3.3}$$

Phase 2 implemented several independent position and speed monitors for assessing UAS behavior during flight as shown in Figure 3.8. Comparing the initialization process between software and hardware implementations revealed that software initialization is faster due to

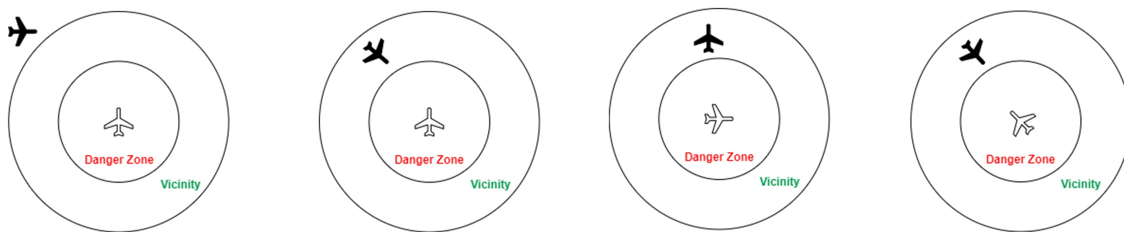

Figure 3.7: A flight plan example for BVLoS [110].

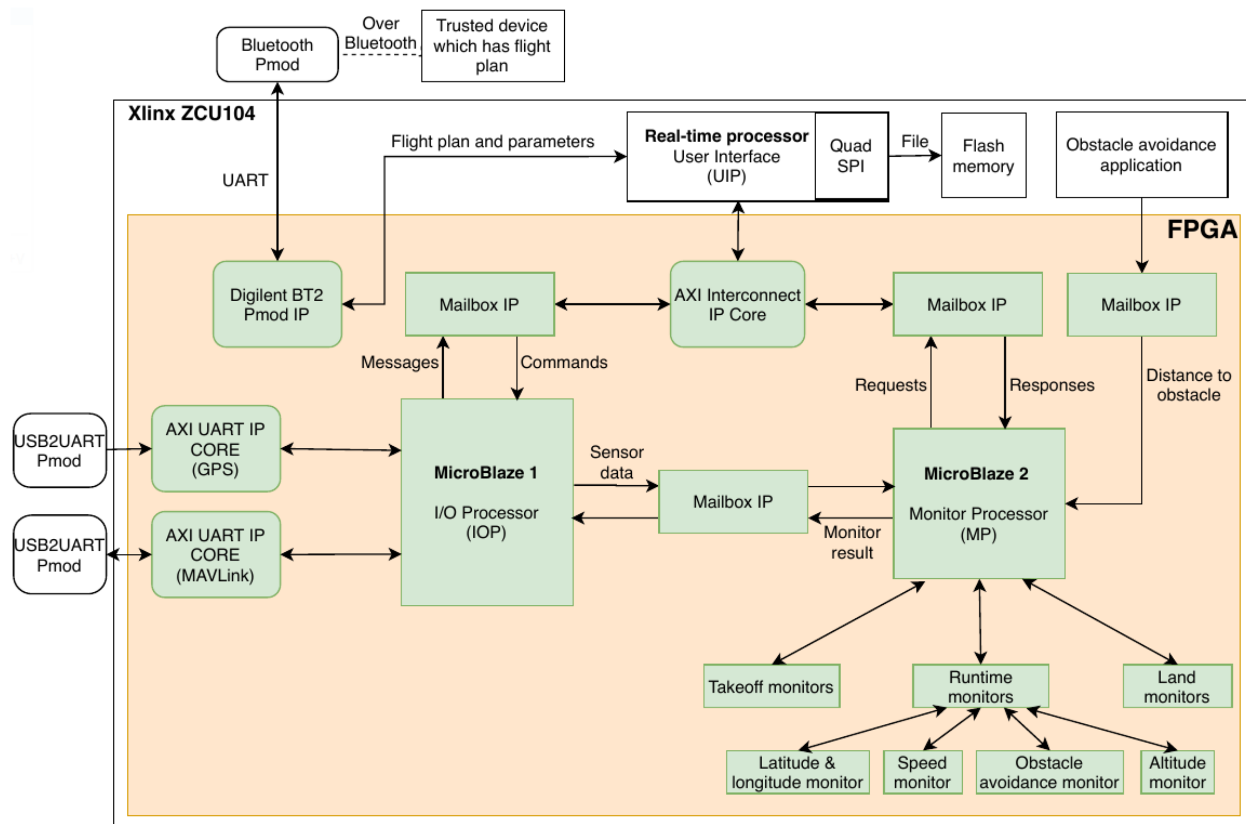

Figure 3.8: RTA design architecture [110].

invoking a function to set multiple variables, while hardware initialization requires writing to registers. However, given that monitors are triggered multiple times during flight while initialization is a one-time event, the execution time of monitors holds greater significance. Evaluation of execution times across different implementations showed that position and speed monitors benefit from hardware parallelism, leading to concurrent execution. Conversely, the single monitor for collision avoidance was more efficiently implemented in software due to communication overheads in hardware. To minimize resource usage and latency, software and hardware monitors were combined with communication mechanisms provided by the Xilinx software development kit (XSDK) [2].

Switching the RTA architecture to a more advanced FPGA for the virtual drone allowed the evaluation of both single and parallel monitors across various flight stages in software

and hardware. HIL simulation also confirmed monitor functionalities and RCF activation upon violations. However, the process of generating monitors remained partially manual, requiring hand-written LTL formulas and manual completion of generated C code including adding headers and defining the function name, arguments and return value. Consequently, the focus of Phase 3 was to improve monitor synthesis automation.

3.3 Phase 3: Monitor Synthesis Automation

The third phase, which is a part of this dissertation, involved collaboration with GE Global Research and GE Aviation on the NASA System-Wide Safety program. Unlike the manual LTL formulation of previous phases, this phase integrated FRET (Formal Requirements Elicitation Tool) [68, 121] to the front-end of the synthesis flow. FRET generates LTL formulas from system specifications expressed in a pseudo-English language called FRETish to facilitate formalization without the need for practicing engineers to have a deep understanding of formal syntax and semantics. Chapter 5 demonstrates FRET workflow in details.

Figure 3.9 depicts a geofence which requires a UAS to remain within the permitted green zone ($in_fence = true$) while avoiding the gray exclusion areas ($in_ex1_zone = false$ and $in_ex2_zone = false$). Table 3.1 shows this requirement in English, its FRETish representation (FRET's input) and the generated LTL formula. Comparing the FRETish and the derived LTL statements reveals the advantage of FRETish in human understandability and its efficacy in reducing manual errors in LTL specifications.

Consider a safety requirement that predicts if the UAS will breach the geofence within two time units based on its current trajectory. This involves estimating future positions from the current speed and position and triggering a corrective action if a breach is anticipated in two time units ($breach_in_2 = true$) but not the next ($breach_in_1 = false$). Table 3.2 and

Table 3.1: The *geofence_inclusion* requirement written in English, FRETish and LTL.

Requirement	RTA should set a geofence error if the UAS is inside either of the two exclusive zones or if not inside the inclusive fence.
FRET input	if ((in_ex1_zone | in_ex2_zone)| !in_fence) RTA shall immediately satisfy geofence_error
LTL formula	((LAST V (((!(((in_ex1_zone | in_ex2_zone) | ! in_fence))) & ((! LAST) & (X(((in_ex1_zone | in_ex2_zone) | ! in_fence))))) → (X(geofence_error)))) & ((((in_ex1_zone | in_ex2_zone) | ! in_fence)) → (geofence_error)))

Table 3.2: The *geofence_breach* requirement written in English, FRETish and LTL.

Requirement	RTA should set a corrective action if the UAS breaches the fence in the next two time units but not in the next time unit.
FRET input	if (breach_in_2 & !breach_in_1) UAS shall until !breach_in_2 satisfy corrective_action
LTL formula	((LAST V (((! ((breach_in_2 & ! breach_in_1))) & ((! LAST) & (X ((breach_in_2 & ! breach_in_1))))) → (X (((! breach_in_2) V (((corrective_action)) | (! breach_in_2))) | (LAST V ((corrective_action))))))) & (((breach_in_2 & ! breach_in_1)) → (((! breach_in_2) V (((corrective_action)) | (! breach_in_2))) | (LAST V ((corrective_action))))))

Figure 3.10 illustrate the details of this requirement and the corresponding BA generated by Spot, respectively. For immediate breaches ($breach_in_1 = true$), a different requirement triggers urgent maneuvers to ensure the UAS remains within bounds. The comparison between manual LTL formulas ((3.1) and (3.2)) and those generated by FRET demonstrates FRET's ability to simplify the articulation of complex properties. This capability assists the generation of multiple monitors from detailed specifications, while making sure that monitors match intentions.

Phase 3 developed the LTL2C tool to further automate the generation of C code, which prompts only for monitor name and variable types. Since Spot only accepts Boolean APs,

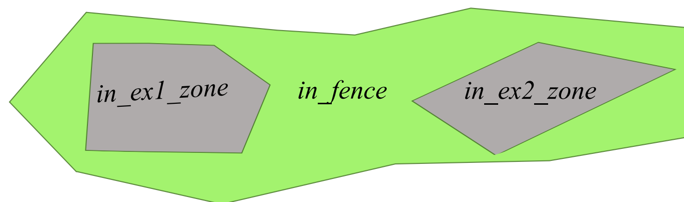

Figure 3.9: Complex geofence including two exclusive areas.

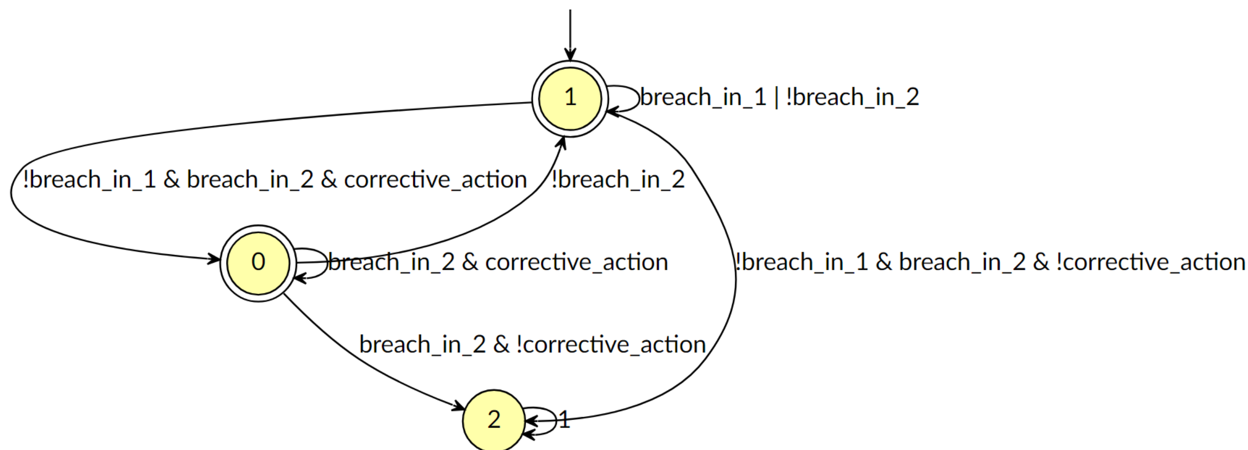

Figure 3.10: Generated BA for the *geofence_breach* requirement.

a parser was developed to support integer propositions within LTL formulas and translates them into Boolean expressions suitable for Spot. The enhanced tool flow is depicted in Figure 3.11, demonstrating the project's progression towards fully automated monitor generation. Further details of this flow will be discussed in Chapter 5.

In the final year of collaboration with GE Global Research and GE Aviation, hardware-implemented monitors for complex geofence requirements targeted GE Aviation's Skynode™ RTA flight controller for commercial UAS operating BVLoS [3]. GE Aviation's conventional method involved using Simulink [52] for algorithm development and sanity checks, generating VHDL code from Simulink model, and finally conducting FPGA simulations with ModelSim [1]. The development time required to implement monitors with the FRET-enhanced flow accelerated development time by 2.3x [78].

This phase improved monitor synthesis automation, yet the flow requires further development

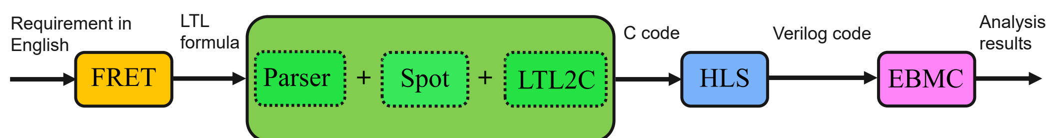

Figure 3.11: The developed monitor synthesis tool flow.

in specific areas such as C code verification through software model checkers and support for time-bound constraints within system specifications. Ensuring that monitor deployment does not impact system performance or lead to resource conflicts is essential, especially in fielded products. This dissertation addresses these goals, aiming to refine and extend the monitoring tool to incorporate these enhancements.

3.4 Summary

This chapter outlines the evolution of the monitor synthesis flow over three years, covering three distinct phases. Phase 1 demonstrated that monitors could be retrofitted to an existing commercial drone, Phase 2 showed that a large number of monitors could be created for a virtual drone, and Phase 3 focused on making it easier for practitioners to synthesize monitors without needing to learn a formal language. To enhance this flow further, this dissertation integrates timing constraints and automates the formal analysis of the generated monitors.

Chapter 4

GROOT Methodology

The adoption of RV has historically been limited to experts, primarily due to the inherent complexities associated with formal notations and the verification process. Recognizing this challenge, GROOT methodology is designed to make RV more accessible and understandable for practitioners in the field. GROOT achieves this goal by facilitating the expression of system requirements in structured English, effectively concealing the intricacies of formal languages and automating the monitor generation process. This methodology enhances the efficiency of the verification workflow by adopting a dual approach that addresses both timing and functional requirements. This deliberate segmentation enables GROOT to tailor its verification processes to the unique demands of each requirement type, ensuring a more precise and robust validation of system behavior during runtime. This chapter introduces GROOT methodology and provides detailed explanations of its main concepts.

4.1 Methodology Steps

GROOT methodology consists of three steps, each contributing uniquely to the generation of monitors for RV. As illustrated in Figure 4.1, these sequential steps are formalization, monitor generation, and monitor verification. First, an English property is transformed into its formal form. Given the executable nature of monitors, the formal representation then transitions into monitor automaton. Finally, the monitor structure undergoes formal

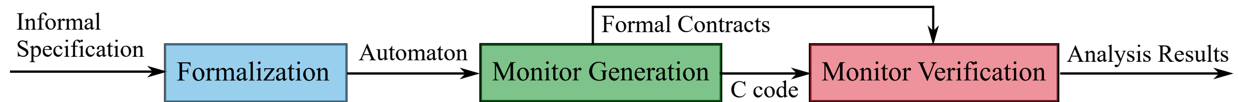

Figure 4.1: Visual representation of the methodological flow of GROOT.

verification to ensure its correctness. It is worthwhile to note that GROOT adopts a dual approach to address both timing and functional requirements. This dual focus introduces subtle distinctions in the logic of each step, while maintaining a cohesive concept and process. To aid practitioners, the entire process is mostly automated. Verification engineers are only required to provide the initial properties and offer occasional guidance throughout the process to ensure the generated monitors align with their expectations.

4.1.1 Formalization via Specification Patterns

The formalization step aims to reduce ambiguity by automating the translation of structured English requirements into formal representations, or more precisely, *automata*. This process is more than a mere syntactic conversion; it significantly enhances precision and clarity in expressing system behaviors. GROOT methodology also offers a visual model of the monitor automaton to encourage an intuitive understanding of the relationships and transitions within the system. The output of this step forms the basis for subsequent stages to establish an interconnected workflow.

This critical step relies on the presentation of various structured English patterns tailored for expressing functional and timing requirements. Specification patterns which are inherently amenable to automation make GROOT a very efficient method by significantly reducing the manual effort required for formalization. The use of such patterns ensures consistent representation of similar concepts across specifications and makes the monitor synthesis less susceptible to errors. These patterns facilitate the expression of a wide range of system

requirements and address the evolving needs of complex embedded systems.

4.1.2 Automated Monitor Generation

In GROOT methodology, the formal representation generated in the formalization step undergoes an automated transformation into FSM during the monitor generation stage. The limited number of states and transitions in FSMs align with GROOT's goal of providing a clear and understandable monitor implementation. Moreover, the familiarity of practicing engineers with FSMs, compared to other formal languages, adds a practical feature to GROOT [70]. FSMs excel in capturing sequences of events, making them well-suited for monitors that detect violations by observing system transitions based on input and output. Although FSMs may not capture every little detail of system behavior, their simplicity ensures a level of comprehensibility crucial for effective runtime monitoring.

Instead of being tightly integrated into the system, GROOT monitors are designed to be independent. These monitors operate as standalone entities alongside the system and consider it as a black box. This approach allows monitors to execute either on the same platform as the system or on separate hardware. This intentional design choice provides a high degree of flexibility and scalability. GROOT's monitors can be used both at the design stage and for systems that have already been deployed. The monitors can integrate into various environments, which makes GROOT a versatile and adaptable framework for RV.

4.1.3 Monitor Verification

Establishing trust in synthesized monitors is paramount. In GROOT, this is achieved through a rigorous process of static formal verification techniques such as model checking and theorem proving, which are explained in Section 2.1.2. This verification process

probes for subtle bugs like array bounds violations and deadlocks, all without necessitating any modification to the source code. GROOT's specification pattern foundation sets an expectation for each monitor structure to exhibit specific behavior. Thus, GROOT automatically generates formal specifications that evaluate interactions between monitor states. This approach not only ensures that the monitors meet their specifications individually, but also guarantees their compatibility and coherence when operating collectively.

By embedding formal analysis as an integral part of GROOT methodology, we establish a robust mechanism which systematically validates the correctness and compliance of the synthesized monitors with the specified system requirements. The result is an RV process characterized by heightened dependability that provides practitioners with confidence in the reliability and trustworthiness of the monitored software systems.

4.2 Monitor Structure in GROOT

Ensuring optimal performance in real-time embedded systems, especially when resource constraints are a critical concern, demands a meticulous approach to the integration of monitors. In GROOT methodology, this is addressed through a careful execution strategy where monitors operate externally, treating the application as a black box. This intentional separation occurs for two reasons: (i) to minimize overhead for efficient utilization of monitors, and (ii) to maintain isolation, such that monitors do not interfere with the system functionality. Handling inputs and monitoring responses are employed using separate external entities. Notably, these entities can be shared among monitors, which would effectively shrink overhead and add a layer of simplicity suiting automated formal analysis. A structural depiction of the GROOT monitor code appears in Figure 4.2, showing the configuration of the automatically generated Main block alongside its fundamental components.

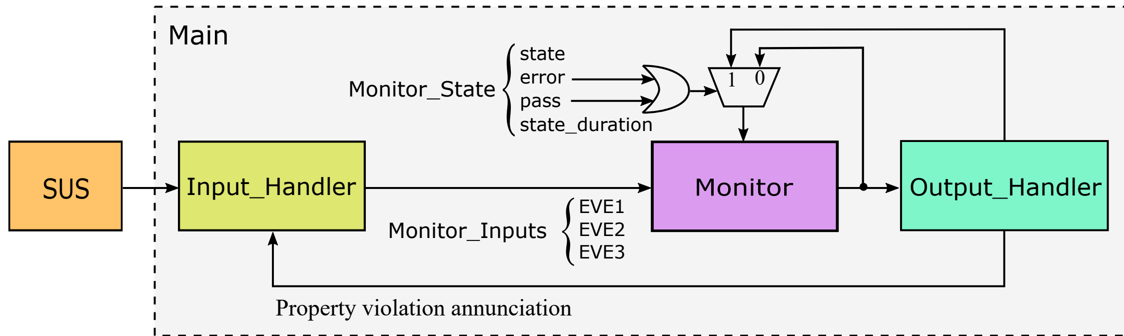

Figure 4.2: Structure of GROOT monitoring process.

The architecture of the monitoring structure involves several key components that collectively ensure an effective RV process. The Input_Handler module acts as an intermediary between the system under scrutiny (SUS) and the monitoring framework. It receives essential information from the SUS and translates these inputs into Boolean APs, which form the basis for subsequent monitoring activities. The Monitor_Inputs structure captures events represented by the Boolean APs. This structure is exclusively updated by Input_Handler to strongly separate responsibilities within the monitoring architecture.

At the core of monitoring process is the Monitor component, which is periodically invoked in synchrony with the time base of the SUS signals. The current status of the Monitor module is stored in the Monitor_State structure, providing a snapshot of the ongoing monitoring state. The error and pass variables denote the occurrence of a violation or satisfaction in the current trace and provide a clear indication of the monitoring outcome. The state_duration variable keeps track of self-transitions from the current state, which offers insights into the temporal aspects of the SUS behavior. When Monitor reaches a conclusive verdict regarding the compliance or violation of a specified requirement, it triggers Output_Handler to notify SUS. This communication acts as a passive alert to SUS about the observed behavior, and subsequently, Monitor_State is reset in preparation for the next set of observations.

This monitoring structure is designed to be adaptable and applicable to a diverse range of

requirements. The flexibility lies in the modular architecture, with Monitor being the only component that varies based on the specific requirement under consideration. This modular design facilitates the reuse of the underlying monitoring framework and allows seamless integration into various monitoring scenarios.

4.3 GROOT Workflow

This section provides a high-level overview of the monitor workflow within GROOT for both timing and functional requirements. GROOT methodology provides monitor synthesis for both types of requirements, and the upcoming chapters will delve into each branch, offering a more detailed exploration of their functionalities and processes.

The primary objective of this research is to encourage practitioners to use RV without needing specialized training in formal methods. GROOT is designed to achieve this aim by automatically generating executable monitors from properties expressed in a structured English format. Functional requirements, which lack metric time constraints, can be translated to LTL (discussed in Section 2.2.2). On the other hand, timing requirements typically refer to events indicating when a change has occurred and naturally map to a formalism such as TA (discussed in Section 2.3.5). GROOT recognizes the distinct treatment of time in LTL and TA. LTL operates with an indefinite notion of time, while TA employs precise time measurement. In our practical experience, employing a single logic or type of automaton for both functional and timing requirements introduces complications that may adversely impact formal analysis outcomes. To address this challenge, GROOT adopts a bifurcated approach, with the LTL-based flow tailored for functional properties and the TA-based flow dedicated to handling timing constraints. This division aligns with the accustomed practice in hardware engineering, where the documentation and design of functionality and timing

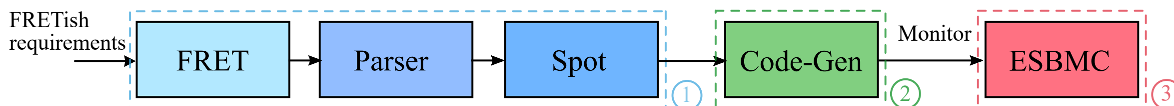

Figure 4.3: Three steps of GROOT for monitor synthesis of FRETish requirements.

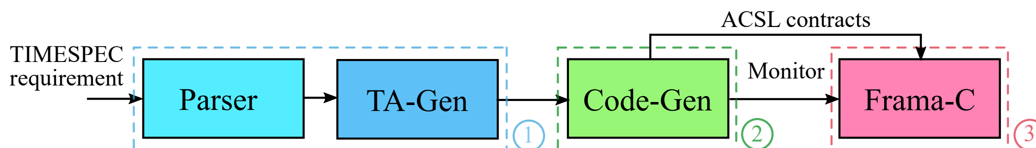

Figure 4.4: Three steps of GROOT for monitor synthesis of TIMESPEC requirements.

aspects are distinctly separated.

Figure 4.3 illustrates the monitor synthesis flow for functional requirements in GROOT. The first step, i.e., the formalization step, is comprised of three modules. Initially, the requirements are captured in a structured English language called FRETish [68]. The FRET module translates these FRETish expressions into their corresponding LTL formulas. Then, a parser prepares the LTL formulas to facilitate their transformation into Büchi automata executed by the Spot tool [55]. In the second step, the automata are translated into monitors expressed in C through the Code-Gen module. Finally in the last step, the ESBMC model checker [65] ensures the monitor implementation satisfies the specification. A detailed description of this process will be presented in Chapter 5.

Figure 4.4 depicts the GROOT monitor flow dedicated to handling timing specifications. The TIMESPEC language is purposefully crafted to capture requirements that involve metric time constraints. It consists of several structured English patterns to cover various timing properties. In the formalization stage, TA-Gen converts TIMESPEC requirements into TA. A slightly modified version of Code-Gen is then employed to interpret TA into C-based monitors. To automate the formal analysis process, Code-Gen generates formal contracts for verifying monitor structures using the Frama-C theorem prover [48]. More details on the timing branch is presented in Chapter 6.

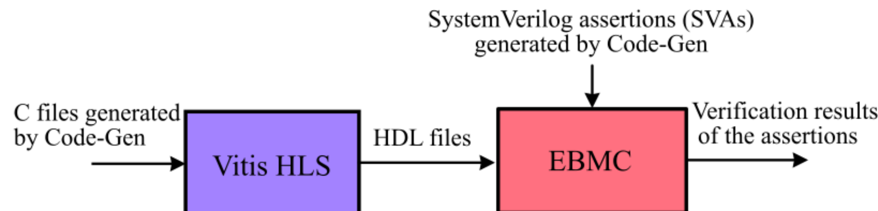

Figure 4.5: Hardware monitor synthesis in GROOT.

4.4 Monitor Synthesis in a HDL

The transformation of C-based monitors into HDL follows the process explained in Section 3.1, which involves converting C code into HDL using a commercial HLS tool, and then validating the HDL against SystemVerilog assertions (SVAs) using the EBMC model checker [95], as shown in Figure 4.5. Unlike the manual creation of these assertions described earlier, the current Code-Gen module automates the generation of SVAs to detect potential errors during the conversion of automata to HDL. This procedure is optional for those requiring only software monitors and is further explained in Chapter 7.

4.5 Trusted Software and Hardware Components

As explained earlier, GROOT ensures the correctness of both software and hardware translations of monitors. However, it relies on the assumption that other tools used in the process function correctly. This section explicitly defines and enumerates the *trust base* which comprises the set of software and hardware tools that are assumed to be trustworthy in their operations. Some of these tools are integrated into the synthesis process for verifying soundness of the translation, while others are employed for testing the monitor implementations.

- **FRET [121]:** A tool developed by NASA for capturing system requirements in structured English. It automatically converts these requirements into LTL and MTL for-

mulas and provides an interactive environment for simulating these formulas.

- **Spot** [151]**:** A tool for converting LTL formulas into Büchi automata (BA). It is commonly used in formal verification to generate automata-based monitors and verify system functional properties.
- **ESBMC** [59]**:** A model checker for C and C++ programs. It is widely used for verifying the correctness of embedded software and supports user-defined assertions.
- **Frama-C** [64]**:** A theorem prover for static analysis of C programs. It allows users to define formal specifications using ACSL (ANSI/ISO C Specification Language) [22] and verify that the code adheres to these specifications.
- **Vitis HLS** [162]**:** A tool that allows designers to generate HDL directly from C or C++. It automates the process of converting high-level algorithms into hardware designs for FPGAs and SoCs.
- **EBMC** [95]**:** A model checker designed for hardware verification designs against system assertions, ensuring that hardware behaves as specified.
- **Vivado** [4]**:** A comprehensive design suite by Xilinx for designing, synthesizing, and implementing FPGA and SoC projects. It includes tools for designing, simulating, and validating hardware logic designs.
- **Simulink** [52]**:** A graphical programming environment for modeling, simulating, and analyzing multi-domain dynamic systems. It is commonly used for designing embedded systems and can generate code from models for deployment on hardware.
- **Visual Studio** [5]**:** An integrated development environment (IDE) developed by Microsoft. It supports a wide range of programming languages like C, C++, C#, Python, and more. It is widely used for developing applications across various platforms.

4.6 Monitor Implementation in Embedded Systems

When integrating monitors into a microprocessor-based embedded system running a real-time operating system (RTOS), careful consideration is required to maintain the system's real-time performance. As discussed in Section 4.2, GROOT focuses on lightweight event-driven monitors with a simple structure. These monitors are designed to perform quick checks to minimize processing overhead and prevent delays that could impact other tasks. Many RTOS setups include an idle task that runs when no other tasks are ready. Consequently, incorporating these non-intrusive monitors into the idle task can ensure that critical tasks are prioritized and deadlines are still met.

For timing constraints or complex functional requirements, monitors may need to run as separate, dedicated tasks. In such cases, the priority of these tasks is critical. If a monitor checks a timing constraint, it must be treated as a real-time task with a priority that allows it to preempt less critical tasks while still respecting the system's overall scheduling. For example, the monitor can be scheduled periodically or triggered by hardware timer interrupts. Ensuring that these monitors do not disrupt existing deadlines requires thorough load analysis, testing, and optimization to balance responsiveness with overall system performance.

4.7 Summary

This chapter unfolds GROOT methodology in three interconnected steps: formalization, monitor generation, and monitor verification, each playing a unique role in the development of monitors for RV. In the formalization step, structured English requirements are automatically translated into precise formal representations such as automata. This process is facilitated by various structured English patterns to enhance precision and clarity

in expressing system behaviors. Notably, GROOT adopts a dual approach to address both timing and functional requirements, ensuring a distinct treatment of each type. The monitor generation step transforms monitors into finite state machines, which are chosen for their simplicity, ease of comprehension and stand-alone execution alongside the system. The verification step employs static formal techniques to ensure the correctness and compliance of synthesized monitors with specified system requirements.

Chapter 5

Functional Requirements Monitor Synthesis in GROOT

This chapter explains the monitor synthesis process within the functional requirements branch of GROOT. Similar to a compiler, GROOT has a front-end and back-end architecture. The front-end utilizes NASA's FRET tool to capture system specifications expressed in structured English. FRET then translates these requirements into corresponding LTL formulas, which precisely formalize the desired system behaviors. Subsequently, the back-end converts these LTL formulas into BA and ultimately generates executable monitors. In this chapter, each step of this process is described to better understand GROOT's functionality and practical application.

5.1 Overview

Figure 5.1 depicts the overall GROOT flow for functional requirements. First, FRET [121] captures system specifications phrased in a structured English language and transforms them into corresponding LTL formulas. The generated expressions can be evaluated against arbitrary inputs using the built-in NuSMV simulator [41]. Subsequently, the LTL formulas are parsed by the tailored LTL2C parser and translated into BA by the Spot tool [55]. These automata are then transitioned into FSMs expressed in the C programming language

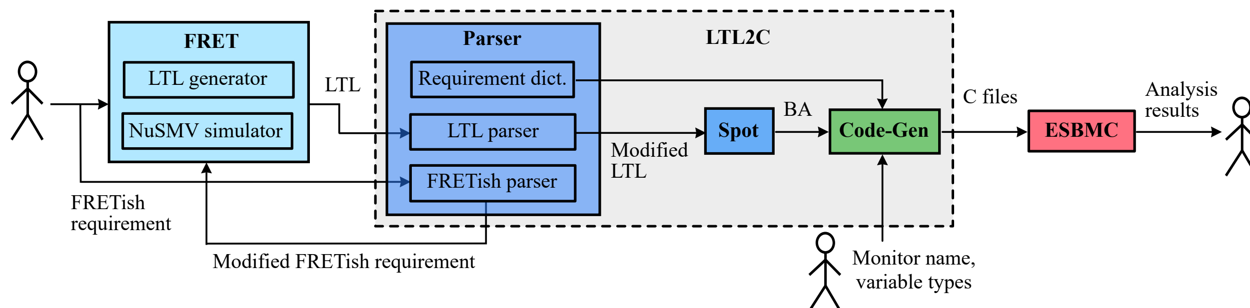

Figure 5.1: GROOT monitor synthesis flow for functional requirements.

by the Code-Gen module developed by this dissertation. Human intervention is minimal in this approach, which only requires the verification engineer to assign monitor name and describe monitor variable types. The ESBMC model checker [65] is employed to rigorously confirm the consistency of the synthesized monitors with the specified requirements. The monitor synthesis process for functional requirements encompasses FRET, Spot, Code-Gen, and ESBMC, and results in automatic generation of monitors capable of inspecting the behavior of complex embedded systems.

5.2 Synthesis Front-end

As mentioned earlier, capturing formal requirements in an efficient manner is paramount for the accurate generation of monitors, as they originate from the higher-level specifications of a system. Achieving a precise understanding of the system model, its inherent properties, and the specific syntax and semantics of the preferred formal logic is crucial when crafting a robust formal specification. While monitoring tools offer automated monitor generation from formal requirements, the manual formulation of formal specifications can be arduous and error-prone. To illustrate the complexity of this task, consider the LTL formula presented in (5.1). This formula is originated from an elevator requirement defined by Dwyer et al. as follows: "Between the time an elevator is called at a floor and the time it opens its

doors at that floor, the elevator can arrive at that floor at most twice" [56]. This example demonstrates the need for a deep understanding of LTL semantics to accurately compose such an intricate specification.

$$\Box((call \wedge \Diamond open) \rightarrow ((\neg atfloor \wedge \neg open) \ \mathrm{U} \ (open \vee ((atfloor \wedge \neg open) \ \mathrm{U} \ (open \vee ((\neg atfloor \wedge \neg open) \ \mathrm{U} \ (open \vee ((atfloor \wedge \neg open) \ \mathrm{U} \ (open \vee ((\neg atfloor \ \mathrm{U} \ open)))))))))) \quad (5.1)$$

One way to enhance the accessibility of formal methods for practitioners involves the definition of a set of property patterns tailored to requirements sharing similarities [35]. These patterns serve as templates for addressing common scenarios by leveraging solutions from analogous, previously solved situations. However, manually selecting the appropriate pattern without a deep familiarity with the formal logic can lead to errors that are difficult to identify. To mitigate this challenge, NASA's FRET tool introduces a structured English language known as FRETish to mask the LTL patterns [68]. FRET also offers a graphical representation that interprets the meaning of the generated logic and provides a visualization of the requirement using the integrated NuSMV simulator.

Figure 5.2 illustrates the structure of a FRETish requirement consisting of six sequential fields: SCOPE, CONDITIONS, COMPONENT, SHALL, TIMING, and RESPONSES. While COMPONENT, SHALL and RESPONSES are mandatory, the remaining fields are optional. COMPONENT is an arbitrary string name referring to the component for which the requirement applies. For instance, if the requirement is about a UAS, the component field could be filled with UAS. SHALL contains a constant string that indicates the component behavior must adhere to the requirement. The current version of FRET supports only one value for the RESPONSES field: satisfy BEXP (Boolean-valued expression). The SCOPE field signifies whether a requirement is applicable within a specific mode of the system; when cleared, the

Figure 5.2: FRETish pattern for LTL formulas [68]. The starred fields are mandatory.

Table 5.1: Supported options for each field of a FRETish template.

SCOPE	CONDITIONS	TIMING	RESPONSES
before MODE, **only before** MODE, **in** MODE, **not in** MODE, **only in** MODE, **after** MODE, **only after** MODE	**if** BEXP, **where** BEXP, **when** BEXP, **upon** BEXP **unless** BEXP	**always**, **never**, **at the next timepoint**, **immediately**, **eventually**, **within** DURATION, **after** DURATION, **for** DURATION, **until** STOP_CONDITION, **before** STOP_CONDITION	**satisfy** BEXP

component must conform to the requirement throughout the entire execution. The CONDITIONS field contains a Boolean condition that triggers the requirement when the condition is true. Finally, the TIMING field specifies the point in the execution when the requirement must hold, with default value of Eventually if omitted. Table 5.1 enumerates the currently supported values for FRET. Combining different values of SCOPE, CONDITIONS, TIMING, and RESPONSES creates a specific FRETish template that enables FRET to select the appropriate LTL formula from its library. FRET generates both ptLTL and ftLTL formulas for each requirement. GROOT strategically utilizes ftLTL formulas, a decision which is explained in Section 2.2.3. This choice is also influenced by Spot (explained in Section 5.3.1), which is tailored to support ftLTL formulas.

Consider requirement RTA-R0 from Table 3.2 stating "if (breach_in_2 & !breach_in_1) RTA shall until !breach_in_2 satisfy corrective_action". In this requirement, RTA is the designated component, and the corresponding LTL formula is generated based on the FRETish template [null, if BEXP, until STOP_CONDITION, satisfy BEXP]. Since no specific value is assigned to the SCOPE field, the requirement is applicable to all possible inputs. The FRET diagram explanation illustrating this requirement is presented in Figure 5.3. According to this figure, (*breach_in_*2 & !*breach_in_*1) and !*breach_in_*2 act as a trigger condition (TC) and a stop condition (SC), respectively. The response, denoted as *corrective_action*,

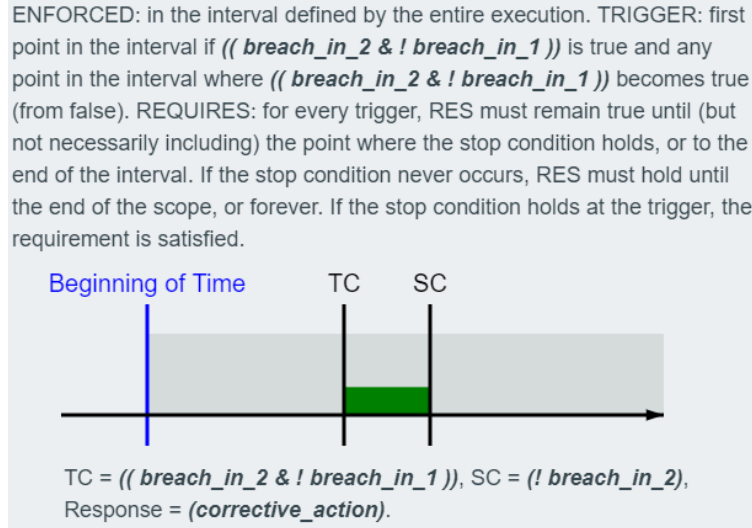

Figure 5.3: FRET's explanation using diagram in the HELP tab.

must occur between TC and SC to fulfill the requirement.

5.2.1 Handling Finite Traces in FRET

FRET utilizes a variable called `LAST` to generate LTL formulas suitable for finite paths. Table 5.2 presents requirement RTA-R1 along with its corresponding LTL formula generated by FRET. Before delving into this formula, it is important to understand the `Release` modality. This operator, denoted as $p \; \mathtt{V} \; q$, is the dual of `Until` ($p \; \mathtt{U} \; q$), indicating that once p becomes true, it releases the obligation for q to be true in the subsequent state. Unlike `Until`, the `Release` operator does not require p to eventually become true. Formally, the `Release` operator for specifications φ and ψ over word ω at position i can be expressed as:

$$\omega, i \models \varphi \; V \; \psi \quad \text{iff} \quad \forall j \geqslant i, \text{iff } (\omega, j \not\models \psi), \text{then } \exists i \leqslant l < j : (\omega, l \models \varphi) \tag{5.2}$$

The LTL formula shown in Table 5.2 indicates that the response should remain valid when the TC holds true, unless `LAST` becomes true and makes the correctness of the response

Table 5.2: The RTA-R1 requirement written in FRETish and LTL.

RTA-R1	If the UAS projected altitude is below an altitude floor, a warning should be issued by the RTA system (except in landing stage).
FRET input	except in landing RTA shall always satisfy ((future_alt < alt_floor) → alt_warning)
LTL formula	(LAST V ((! landing) → (((future_alt < alt_floor) → alt_warning))))

irrelevant. Given that LAST pertains to the final state of the trace, this LTL formula is suitable for handling the finite traces encountered in RV. However, GROOT omits this variable (assuming it is consistently false) during monitor synthesis and opts for LTL_3 by employing the error and pass variables instead, as discussed in Section 4.2.

5.3 Synthesis Back-end

Translating English specifications into LTL avoids burdening the verification engineer with formalizing the requirements. As outlined in Section 2.3.4, BA are a standard means of generating monitor automata from LTL requirements. Hence, the formalization step ends with the creation of BA. These automata are subsequently transformed into FSMs in GROOT's monitor generation step. To facilitate this conversion process, the LTL2C module has been developed and consists of three core elements: (i) the Spot API, (ii) a parser, and (iii) CodeGen. In the final stage, LTL2C invokes ESBMC to ensure the monitor's correctness and compliance with the specified system requirement.

5.3.1 Spot API

Spot is a mature, open-source tool able to translate future-time LTL or PSL formulas into BA [55]. Spot's robustness and efficiency makes it suitable for industrial applications [141]. It also offers a Python interface that suits the development of LTL2C, which is implemented

in Python as well. The primary role of LTL2C is to take LTL formulas as input and employ Spot's API to generate and visualize the corresponding BA. The resulting BA, accompanied by a list of APs used in the LTL requirement, is subsequently utilized by the CodeGen module to produce the FSM for the monitor. This joint operation between Spot and LTL2C ensures a seamless translation process, contributing to the overall efficiency and effectiveness of GROOT's monitor synthesis workflow.

5.3.2 LTL2C Parser

FRET allows requirements to be specified with numeric variables, and basic arithmetic and comparison operations. However, an issue arises during the subsequent monitor synthesis process using NuSMV, as it exclusively supports Boolean APs. Furthermore, Spot accepts LTL formulas featuring Boolean propositions only. To accommodate these restrictions, LTL2C incorporates a specialized parser to generate formulas suited for integration with Spot and NuSMV as depicted in Figure 5.1 and elaborated in Algorithm 1. The parser dissects the requirement into a sequence of tokens using the LEXER module. The tokens are processed by the SYNTAX_ANALYZER module, which returns an abstract syntax tree (AST). The PARSER module then uses this AST to transform LTL and FRETish requirements involving arithmetic or comparison operations into specifications containing only Boolean APs.

To gain a deeper insight into the functionality of this parser, consider an example involving a geofence requirement aimed at confining a UAS within a complex polygon, as illustrated in Figure 5.4. To enforce this restriction, algorithms like the winding number (WN) algorithm are quite useful in assessing the UAS position relative to the geofence [79]. The WN algorithm analyzes the position of an object relative to the geofence edges. When observing these edges in an anticlockwise direction, a horizontal line extending from the desired point causes the

Algorithm 1 LTL2C parser script

```
procedure PARSER(text)          ▷ The input is either a FRETish or an LTL requirement.
    Declare string req and initialize it with text
    tokens ← LEXER(text)                                  ▷ Perform lexical analysis
    AST ← SYNTAX_ANALYZER(tokens)
    Declare dictionary Req_dict
    if AST is valid then
        if there is any comparison operation then
            if it is not already in APs then
                Assign a name to the operation and add it to APs as a new AP
            end if
            Update req based on the AP's name
        end if
        Add names to inputs
        Add req to Req_dict                 ▷ req contains only Boolean propositions.
    else
        Print "Parsing failed"
    end if
    return Req_dict
end procedure
procedure LEXER(text)
    Build LTL alphabet (tokens)      ▷ Including LTL, numeric, and comparison operators
    Build FRETish alphabet (tokens) ▷ Including FRET fields and comparison operators
    Split text into tokens based on the relevant alphabet
    return tokens
end procedure
procedure SYNTAX_ANALYZER(tokens)
    Define LTL grammar rules                      ▷ Including name, number, equation
    Define FRETish grammar rules          ▷ Including name, number, scope, condition
    Construct Abstract Syntax Tree (AST) using tokens and proper rules
    if AST is invalid then
        Print "Syntax error"
    end if
    return AST
end procedure
```

WN value to increase (decrease) if the point lies on the left-hand (right-hand) side of an edge with upward (downward) crossing [154]. Applying this algorithm to Figure 5.4, the UAS is inside the geofence, while the airplane lies outside. This geofence requirement, derived

Table 5.3: The RTA-R4 geofence requirement written in FRETish and LTL.

<table>
<tr><td>RTA-R4</td><td>For a specific edge, RTA decides whether the UAS is on the correct side.</td></tr>
<tr><td>FRET input</td><td>if ((((y1 <= 0) & (y2 > 0)) & (((x1 * y2) - (x2 * y1)) > 0)) | (((y1 >= 0) & (y2 < 0)) & (((x1 * y2) - (x2 * y1)) < 0))) RTA shall immediately satisfy edge_detect</td></tr>
<tr><td>LTL formula</td><td>((LAST V (((! ((((((y1 <= 0) & (y2 > 0)) & (((x1 * y2) - (x2 * y1)) > 0)) | (((y1 >= 0) & (y2 < 0)) & (((x1 * y2) - (x2 * y1)) < 0))))) & ((! LAST) & (X (((((y1 <= 0) & (y2 > 0)) & (((x1 * y2) - (x2 * y1)) > 0)) | (((y1 >= 0) & (y2 < 0)) & (((x1 * y2) - (x2 * y1)) < 0))))))) → (X (edge_detect)))) & ((((((y1 <= 0) & (y2 > 0)) & (((x1 * y2) - (x2 * y1)) > 0)) | (((y1 >= 0) & (y2 < 0)) & (((x1 * y2) - (x2 * y1)) < 0)))) → (edge_detect)))</td></tr>
</table>

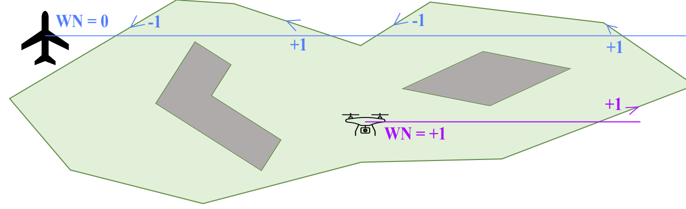

Figure 5.4: A complex geofence with two no-fly zones, which are shown as gray areas.

from [78], aims to aid WN calculations in GE Aviation's RTA unit [3].

This intricate RTA requirement consists of three simpler parts: determining whether the UAS is on the left-hand (right-hand) side of an upward (downward) edge; increasing the WN counter if the previous requirement is true; and considering the UAS inside if the counter is odd, otherwise outside. Table 5.3 outlines the first requirement (RTA-R4) in English, FRETish, and LTL. The edge ending points $P1(x1, y1)$ and $P2(x2, y2)$ are considered with respect to the UAS position. The generated LTL formula could be quite complex and error-prone to capture manually. FRET simplifies the formalization process by masking the LTL notation and enabling formula creation with the FRETish language. This formula involves numeric and relational operations, which are not supported by NuSMV and Spot.

The LTL2C parser substitutes each arithmetic or comparison operation with a Boolean variable and creates an LTL formula and a FRETish requirement compatible with Spot and NuSMV, respectively. Table 5.4 shows the modified FRETish and LTL expressions of RTA-R4 along with a roster of its APs identified by the parser. Note that in the event of an

Table 5.4: Modified FRETish and LTL statements for RTA-R4 along with its list of APs.

<table>
<tr><th>List of APs</th><th>FRET input (modified)</th></tr>
<tr><td rowspan="3">AP_0 → y1 <= 0
AP_1 → y2 > 0
AP_2 → ((x1*y2) - (x2*y1)) > 0
AP_3 → y1 >= 0
AP_4 → y2 < 0
AP_5 → ((x1*y2) - (x2*y1)) < 0</td><td>If ((((AP_0) & (AP_1)) & (AP_2)) | (((AP_3) & (AP_4)) & (AP_5))) RTA shall immediately satisfy edge_detect</td></tr>
<tr><th>LTL formula (modified)</th></tr>
<tr><td>((LAST V (((! (((((AP_0) & (AP_1)) & (AP_2)) | (((AP_3) & (AP_4)) & (AP_5))))) & ((! LAST) & (X (((((AP_0) & (AP_1)) & (AP_2)) | (((AP_3) & (AP_4)) & (AP_5))))))) → (X (edge_detect)))) & ((((((AP_0) & (AP_1)) & (AP_2)) | (((AP_3) & (AP_4)) & (AP_5)))) → (edge_detect)))</td></tr>
</table>

error during the creation of these APs by the engineer, rather than having to resynthesize the monitor from scratch, only the definition of the associated APs need to be updated in the Input_Handler module, thus simplifying the monitor synthesis process for practitioners. As mentioned earlier, the adapted FRETish statement allows FRET's simulator to visualize the functionality of the generated LTL formula. This visualization offers a dynamic view of the temporal behavior, enabling verification engineers to interactively explore the impact of inputs on the final evaluation of the requirement. The simulation snapshot for the modified requirement, presented in Figure 5.5, illustrates the successful capture of the RTA failure at $t = 12$ due to invalidity of the $edge_detect$ variable.

5.3.3 LTL2C Code Generator

Algorithm 2 illustrates the monitor code generation process of Code-Gen, which translates the abstract automaton generated by Spot into a concrete FSM implemented in C. As illustrated in Figure 5.1, human intervention is only required to assign the monitor name and specify the types of monitor arguments. For example, when processing the RTA-R4 requirement, the parser sends a variable list to Code-Gen such as $\{x1, x2, y1, y2, edge_detect\}$. Code-Gen asks the verification engineer to specify the variable types to synthesize monitors with minimal manual input. Upon completion of this phase, the source code and header files for Main, Input_Handler, Output_Handler, and Monitor (as expressed in Section 4.2) are

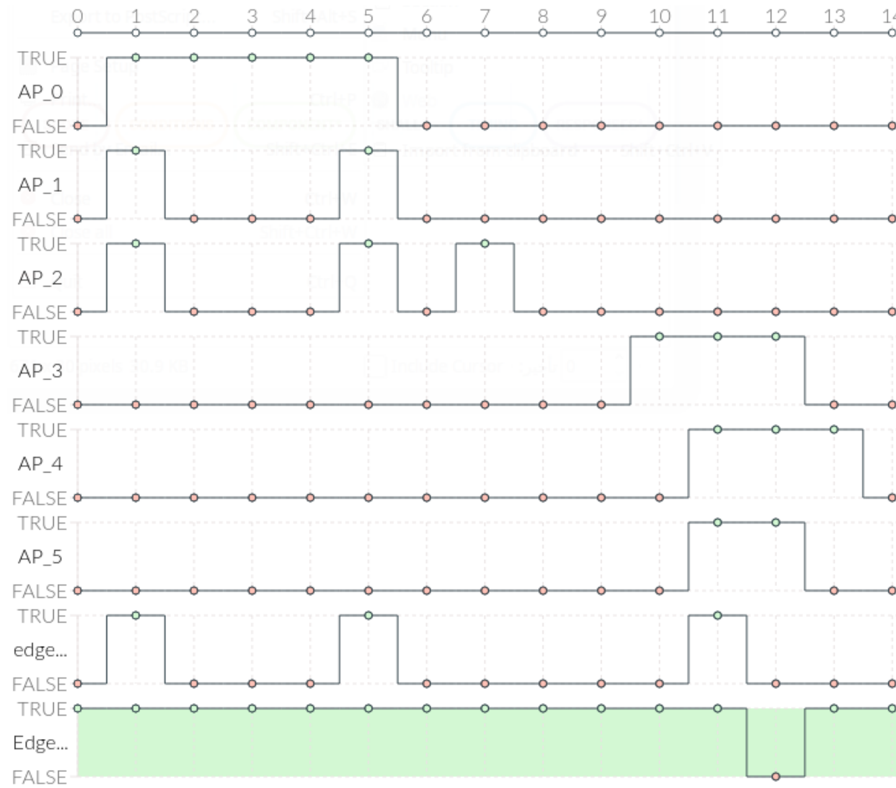

Figure 5.5: NuSMV simulation screenshot of the RTA-R4 requirement.

automatically generated.

Figure 5.6 shows the header and source files generated by LTL2C for the RTA-R4 requirement. The header file includes necessary headers, enumerates the monitor's states, and defines essential function declarations and the Monitor_State and Monitor_Inputs structures. In the Input_Handler function, the monitoring events are defined to simplify the monitor code structure. As explained before, the Output_Handler function informs the SUS of the final verdict and resets the monitor status for the subsequent observations.

The Monitor code encompasses the user-specified name, designated arguments and the structured automaton state machine represented using a `switch` statement. This structured approach enhances the readability and maintainability of the monitor code. The Main source code corresponds to a task specifically tailored for embedded systems utilizing a real-time

Algorithm 2 LTL2C code generator script

```
procedure CODE-GEN(BA, Req_dict)                                ▷ BA is received from Spot
    Get the monitor name from the user
    Get the type of each input (Req_dict['inputs']) from the user
    Create files as header, Main, Input_Handler, Output_Handler, and Monitor blocks
    // In the header file
    Define an enum based on states in BA['states']
    Define Monitor_State struct and Monitor_Inputs struct
    for each AP in Req_dict['APs'] do
        Define a Boolean variable for AP in Monitor_Inputs
    end for
    Define Monitor, Input_Handler, and Output_Handler functions
    // In the Main file
    Instantiate monitor_state and monitor_inputs
    Create a while loop with condition 1
    Call Input_Handler, Monitor, and Output_Handler functions in the while loop
    // In the Input_Handler file
    for each AP, def in Req_dict['APs'] do
        Assign def to AP in monitor_inputs
    end for
    // In the Output_Handler file
    if monitor_state.error is true then
        Print "Violation is detected" and reset monitor_state
    end if
    if monitor_state.pass is true then
        Print "Response is received" and reset monitor_state
    end if
    // In the Monitor file
    Create a switch-case
    for each state in BA['states'] do
        Create a case for state
        for each tran in state['transitions'] do
            Create if-else statement with condition tran['condition']
            Update monitor_state.state with tran['destination']
        end for
    end for
    Output
end procedure
```

```c
#include <stdbool.h>
#include <assert.h>
#include <stdio.h>
enum States {STATE0, STATE1, STATE2};
struct Monitor_State {
    enum States state;
    bool error, pass;
    unsigned int state_duratio; };
struct Monitor_Inputs {
    bool AP0, AP1, AP2, AP3, AP4, AP5, AP6; };
void RTA_R4_monitor(struct Monitor_State* monitor_state,
                    struct Monitor_Inputs* monitor_inputs);
void input_handler(double y1, double y2, double x1, double x2,
    bool edge_detect, struct Monitor_Inputs* monitor_inputs);
void output_handler(struct Monitor_State* monitor_state);
```

```c
void input_handler(double y1, double y2, double x1, double x2,
              bool edge_detect, struct Monitor_Inputs* monitor_inputs){
    monitor_inputs->AP0 = (y1 <= 0);
    monitor_inputs->AP1 = (y2 > 0);
    monitor_inputs->AP2 = (((x1 * y2) - (x2 * y1)) > 0);
    monitor_inputs->AP3 = (y1 >= 0);
    monitor_inputs->AP4 = (y2 < 0);
    monitor_inputs->AP5 =  (((x1 * y2) - (x2 * y1)) < 0);
    monitor_inputs->AP6 = edge_detect; }
```

```c
void output_handler(struct Monitor_State* monitor_state){
    if (monitor_state->error || monitor_state->pass) {
        monitor_state->state = STATE1;
        monitor_state->error = false;
        monitor_state->pass = false;
        monitor_state->state_duration = 0;
        if ((monitor_state->error) { printf("Violation is detected!"); }
        else { printf("Response is received!"); }}}
```

```c
void main(int argc, char** argv) {
    struct Monitor_State monitor_state =
    { STATE1, false, false, 0 };
        struct Monitor_Inputs monitor_inputs =
    { false,false,false,false,false,false,false };
        while (1) {
            input_handler(&monitor_inputs);
            RTA_R4_monitor(&monitor_state, &monitor_inputs);
            output_handler(&monitor_state); }}
```

```c
void RTA_R4_monitor(struct Monitor_State* monitor_state,
                    struct Monitor_Inputs* monitor_inputs){
    enum States state = monitor_state->state;
    bool error = monitor_state->error;
    bool pass = monitor_state->pass;
    unsigned int state_duration =
        monitor_state_func->state_duration;

    bool AP0 = monitor_inputs->AP0;
    bool AP1 = monitor_inputs->AP1;
    bool AP2 = monitor_inputs->AP2;
    bool AP3 = monitor_inputs->AP3;
    bool AP4 = monitor_inputs->AP4;
    bool AP5 = monitor_inputs->AP5;
    bool AP6 = monitor_inputs->AP6;

    switch (state) {
        case STATE0:
            state_duration += 1;
            pass = true;                                    {
            if ((AP0 && AP1 && AP2) || (AP3 && AP4 && AP5)){
                state = STATE0;}
            else {
                state = STATE1;}
            break;
        case STATE1:
            state_duration += 1;
            if ((AP0 && AP1 && AP2 && AP6) ||
                (AP3 && AP4 && AP5 && AP6)) {
                state = STATE0;}
            else if ((AP0 && AP1 && AP2 && !AP6) ||
                     AP3 && AP4 && AP5 && !AP6)) {
                state = STATE2;}
            break;
        case STATE2:
            error = true;
            break;
        default:
            error = false;
            pass = false;
            break;}

    monitor_state_func->state = state;
    monitor_state_func->error = error;
    monitor_state_func->pass = pass;
    monitor_state_func->state_duration = state_duration;}
```

Figure 5.6: The generated header file, Input_Handler, Output_Handler and Main functions (from top to bottom on the left), along with the Monitor function (on the right) for RTA-R4.

operating system (RTOS). This task is designed to wrap the monitor blocks and is typically executed by the RTOS scheduler with the lowest priority, ensuring efficient monitoring without disrupting critical system operations.

5.3.4 Monitor Formal Analysis

As explained in Section 2.1.2, model checking is an efficient verification method for FSMs with reasonable size. To complete the synthesis process, the ESBMC model checker is utilized

```
    // state validity
(1) assert(monitor_state.state == STATE0 || monitor_state.state == STATE1 || monitor_state.state == STATE2);
(2) assert(!(monitor_state.error && monitor_state.pass));   // error and pass mutual exclusivity
(3) assert(monitor_state.state_duration >= 0);              // state_duration validity
(4) assert(monitor_state.error == (monitor_state.state == STATE2)); // Error condition consistency
(5) assert(monitor_state.pass == (monitor_state.state == STATE0));  // Response condition consistency

    // Transitions validity
(6) if (monitor_state.state == STATE1) {
        assert(((!monitor_inputs.AP0 && !monitor_inputs.AP3) || (!monitor_inputs.AP0 && !monitor_inputs.AP4) ||
            (!monitor_inputs.AP0 && !monitor_inputs.AP5) || (!monitor_inputs.AP1 && !monitor_inputs.AP3) ||
            (!monitor_inputs.AP1 && !monitor_inputs.AP4) || (!monitor_inputs.AP1 && !monitor_inputs.AP5) ||
            (!monitor_inputs.AP2 && !monitor_inputs.AP3) || (!monitor_inputs.AP2 && !monitor_inputs.AP4) ||
            (!monitor_inputs.AP2 && !monitor_inputs.AP5)) == (monitor_state.state == STATE1));
        assert(((monitor_inputs.AP0 && monitor_inputs.AP1 && monitor_inputs.AP2 && monitor_inputs.AP6) ||
            (monitor_inputs.AP3 && monitor_inputs.AP4 && monitor_inputs.AP5 && monitor_inputs.AP6))
            == (monitor_state.state == STATE0));
        assert(((monitor_inputs.AP0 && monitor_inputs.AP1 && monitor_inputs.AP2 && !monitor_inputs.AP6) ||
            (monitor_inputs.AP3 && monitor_inputs.AP4 && monitor_inputs.AP5 && !monitor_inputs.AP6))
            == (monitor_state.state == STATE2));
    }
(7) if (monitor_state.state == STATE0) {
        assert(((!monitor_inputs.AP0 && !monitor_inputs.AP3) || (!monitor_inputs.AP0 && !monitor_inputs.AP4) ||
            (!monitor_inputs.AP0 && !monitor_inputs.AP5) || (!monitor_inputs.AP1 && !monitor_inputs.AP3) ||
            (!monitor_inputs.AP1 && !monitor_inputs.AP4) || (!monitor_inputs.AP1 && !monitor_inputs.AP5) ||
            (!monitor_inputs.AP2 && !monitor_inputs.AP3) || (!monitor_inputs.AP2 && !monitor_inputs.AP4) ||
            (!monitor_inputs.AP2 && !monitor_inputs.AP5)) == (monitor_state.state == STATE1));
        assert(((monitor_inputs.AP0 && monitor_inputs.AP1 && monitor_inputs.AP2) ||
            (monitor_inputs.AP3 && monitor_inputs.AP4 && monitor_inputs.AP5)) == (monitor_state.state == STATE0));
    }
```

Figure 5.7: Code-Gen generated assertions for the RTA-R4 requirement.

to ensure the generated FSM satisfies the specified requirement. ESBMC is an open-source bounded model checker for validating C and C++ programs against established properties such as pointer validity and deadlock prevention, all achieved without modifications to the source code [65]. It also equipped with a Python interface and allows for user-defined assertions. The Code-Gen module generates the necessary assertions to ensure the synthesis process did not introduce any errors. As discussed in Section 4.5, this work assumes the correctness of the trust base.

Figure 5.7 provides a visual representation of the assertions embedded into the monitor code by the Code-Gen module. These assertions validate that the transitions within the monitor's `switch-case` structure accurately adhere to the logical conditions specified by the BA. To grasp the significance of each assertion, consider Figure 5.8 and Figure 5.6, which depict the BA and the monitor structure for the RTA-R4 requirement, respectively. Asser-

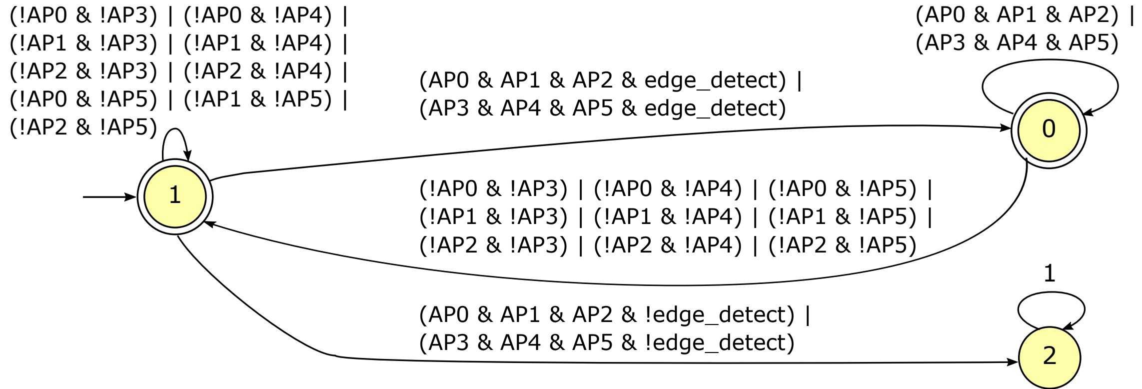

Figure 5.8: The RTA-R4 monitor automaton provided by Spot.

tion (1) pertains to state validity, ensuring that the monitor contains the necessary states for proper functionality. Assertion (2) confirms the exclusivity of the monitor_state.error and monitor_state.pass variables, as only one should be true at any given time (as discussed in Section 2.2.5). Assertion (3) ensures that the monitor_state.state_duration counter stays within acceptable bounds. This counter serves to track time within the monitor, making it essential to verify that it consists solely of non-negative integers.

Assertions (4) and (5) verify that the monitor_state.error and monitor_state.pass variables accurately reflect the monitor's current status. Given the critical importance of trusting the monitor's verdict at each iteration, it is essential to validate these variables. Referencing Figure 5.8 where state 2 is non-accepting, assertion (4) ensures that monitor_state.error is true when the monitor is in this state. It also confirms that whenever this variable is true, the monitor is in the error state 2. Conversely, assertion (5) verifies that monitor_state.pass is true only when the monitor is in the accepting state 0, signifying the reception of the *edge_detect* assertion response.

Assertions (6) and (7) ensure that the conditions within each `case` statement correspond to the specified transitions outlined in the BA. For example, in the scenario where the monitor is in state 0, assertion (7) checks that the monitor remains in state 0 under certain conditions,

transitioning to state 1 only if specific events do not hold true. These automatically generated assertions aid practitioners in verifying the integrity of the monitor implementation. They also provide reassurance that the monitor continues to function as intended even if engineers make modifications to the FSM to meet specific requirements or preferences.

5.4 Summary

This chapter provides a detailed overview of GROOT's monitor synthesis process for functional requirements and highlights its efficiency and accessibility for practitioners by automating the whole process. Initially, the process converts system requirements expressed in structured English into LTL formulas. It then translates these LTL expressions into automata and generates monitors written in C with minimal and straightforward manual intervention. For the automated verification step, a model checker is employed to ensure that the synthesized monitors correctly adhere to the transitions specified in the automata.

Chapter 6

Timing Requirements Monitor Synthesis in GROOT

This chapter introduces a structured English language called TIMESPEC to express time limits, which are then translated into automata. TIMESPEC offers practitioners a practical way to capture timing requirements without needing a formal methods background. GROOT's framework fully automates the monitor generation process from TIMESPEC requirements. This automation minimizes the reliance on manual intervention, thereby enhancing efficiency and reducing potential errors introduced through manual steps. GROOT also provides an automated mechanism for verifying the correctness of each generated monitor. Each phase of monitor synthesis for timing requirements is described in this chapter.

6.1 Overview

Real-time embedded systems often combine software programming with hardware elements such as FPGAs, requiring a monitoring framework to support both system-level functional requirements and low-level hardware-oriented timing requirements. Figure 6.1 illustrates the monitor synthesis process for handling timing requirements. The process begins with reading the TIMESPEC statement. A parser extracts the essential information encapsulated within the TIMESPEC requirement. The parsed information is read by the TA-Gen

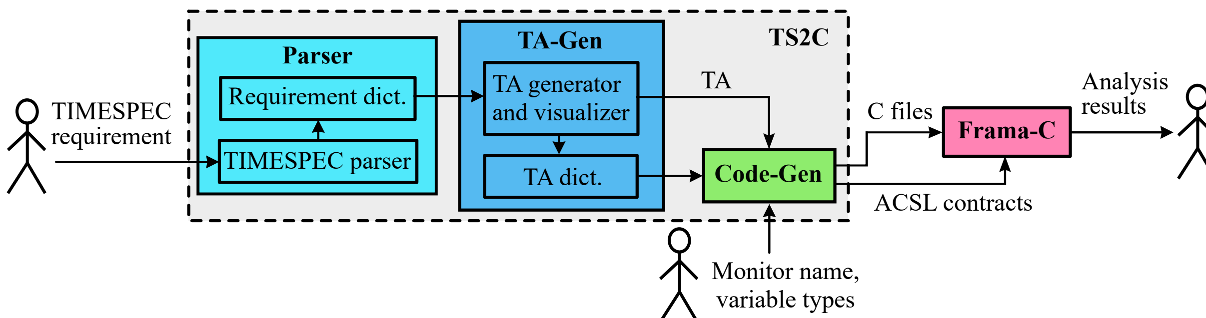

Figure 6.1: GROOT monitor synthesis flow for timing requirements.

module to generate the corresponding TA (stated in Section 2.3.5). This TA serves as a formal representation of the specified timing requirement, which captures the temporal details outlined in the original TIMESPEC statement. The generated TA is then translated into executable C code via the Code-Gen module, which also provides formal expressions to facilitate formal analysis step. Similar to the FRET-based flow, human intervention is only required to assign the monitor's name and describe monitor variable types. At the last step, the Frama-C tool [48] is used to verify the created monitors. The formal statements generated by Code-Gen serve as the basis for Frama-C's verification process to inspect the synthesized monitors against predefined correctness criteria. The interplay of parser, TA-Gen, Code-Gen, and Frama-C contributes to the creation of efficient and formally verified monitors for timing requirements.

6.2 TIMESPEC

Ensuring correct behavior in real-time systems is challenging as they must satisfy both functional and non-functional requirements containing timing constraints. Even if schedulability analysis shows that all tasks in these systems meet their timing constraints, various factors can still cause timing violations in practice. For example, the Therac-25 radiation therapy

Table 6.1: Possible reasons for timing violations in practice.

Reason	Explanation
Priority inversion	If a low-priority task holds a resource that a high-priority task needs, it can cause delays for the high-priority task.
Clock drift	Variations in clock speed due to environmental factors (like temperature) could cause slight timing errors to accumulate over time, affecting task scheduling.
Network delays	In systems with distributed components, communication delays or network congestion can impact timing.
Software updates	Changes in software behavior or timing after updates may introduce bugs, delays, or resource conflicts.
Dynamic reconfiguration	Real-time systems that allow dynamic loading of new tasks or reconfiguration could experience unpredictable interactions that were not accounted for during initial analysis.
Environment variations	Environmental factors like temperature, power supply fluctuations, or electromagnetic interference could alter system behavior, leading to timing violations.
Unforeseen interactions	Complex systems with many interacting tasks can behave unpredictably, especially under edge cases or rare conditions not considered during analysis.

machine tragically delivered lethal doses to patients due to timing errors in the control software, allowing the machine to enter an invalid state [105]. Another significant example is the 1990 AT&T long-distance network crash, where a software update introduced a bug in AT&T's network switches [157]. The bug caused the switches to crash and restart when they received a specific message during a particular time window. This bug spread across the network like a virus, leading to a nine-hour outage that impacted millions of customers. These examples highlight how even well-analyzed systems can experience timing issues due to unforeseen factors like software bugs, hardware faults or communication failures. Table 6.1 summarizes some potential causes of timing violations.

Given the critical role of timing requirements in safety-critical applications, several formal specification languages have been developed to address these needs. Formalisms such as metric temporal logic (MTL) [94], signal temporal logic (STL) [111], or TeSSLa [104] are specifically designed to capture and verify timing constraints. However, the absence of a standardized language to define system specifications remains a significant impediment. This gap requires verification engineers to possess formal methods expertise and familiarize themselves with the syntax of each domain-specific monitoring tool. Studies have shown

Table 6.2: TIMESPEC pulse duration template.

Constraint Type	Pulse Duration
Template	TYPE of SIGNAL should be TIMING_CONSTRAINT.
TYPE	active_pulse_width, period, duty_cycle
SIGNAL	Source of events such as: CLK, RESET
TIMING_CONSTRAINT	TL, >= TL_{min} and <= TL_{max}, >= TL_{min}, <= TL_{max}
Time limits (value + time unit)	TL_{min} (lower bound), TL_{max} (upper bound), TL (duration)
Time unit	%, ns, us, ms, s, min, h

that this learning curve can be overwhelming and is one of the key obstacles to adopting formal methods in industry [56].

Timing requirements differ fundamentally from functional requirements and are often repetitive in nature, making it possible to reuse or adapt variants of the same monitor. Furthermore, resource overheads may be minimized by using distinct formalisms for functional and timing specifications (will be further explained in Section 6.3). To overcome the challenges associated with expressing timing constraints, GROOT provides TIMESPEC, a structured English dialect specifically designed to articulate timing requirements. This language simplifies the process of capturing time limits to make it more accessible for practitioners without a deep background in formal methods. TIMESPEC supports properties containing metric time bounds by defining a set of property patterns for specifications sharing similarities.

6.2.1 Pulse Duration

The precise timing of clocks and signals in digital systems and interfaces plays a critical role in ensuring the correct operation of the overall design. In particular, the pulse width of certain signals must fall within a specific range to trigger the desired events. To address this type of requirement, a pulse duration template is employed to specify the pulse width of a signal or the period and duty cycle of a clock signal. The supported values for each field

Table 6.3: TIMESPEC causality template.

Constraint Type	Causality
Template	If ACTION SIGNAL, ACTION SIGNAL TIMING_CONSTRAINT.
ACTION	assert, deassert, start
SIGNAL	Source of events such as: CLK, START
TIMING_CONSTRAINT	between TL_{min} and TL_{max}, after TL_{min}, after-exactly TL, within TL_{max}, for TL
Time limits (value + time unit)	TL_{min} (lower bound), TL_{max} (upper bound), TL (duration or a specific time)
Time unit	ns, us, ms, s, min, h, cycle

of this template are illustrated in Table 6.2. TYPE indicates the nature of pulse duration requirement. SIGNAL identifies the clock signal or the source of events to be monitored. The TIMING_CONSTRAINT value varies based on the specification constraint. Each time limit consists of a numeric value and a time unit. The duty cycle bound is typically expressed as a percentage, distinguishing it from other TYPEs that use units such as nanoseconds (ns). To demonstrate this template, consider a clock signal (clk) with specific period (T_{clk}), say 50 ms. If this clock has any glitches to be detected, GROOT may create a monitor using a TIMESPEC requirement similar to "P1: Period of clk should be 50 ms".

6.2.2 Causality

In communication protocols such as handshaking, events often have causal relationships, where one event may trigger another with a constraint on the elapsed time between the two. To capture this behavior, TIMESPEC uses a causality template described in Table 6.3. For example, the requirement "P2: If deassert RESET, start CLK within 3 us" indicates that the clock signal in a digital system must start within a specific time after the reset signal is deasserted. In this context, ACTION includes asserting/deasserting signals or starting a clock, while SIGNAL identifies events associated with these actions. TIMING_CONSTRAINT expresses the temporal relationship between events based on defined time bounds. To provide practitioners with a more intuitive understanding, a causality relationship expressed in

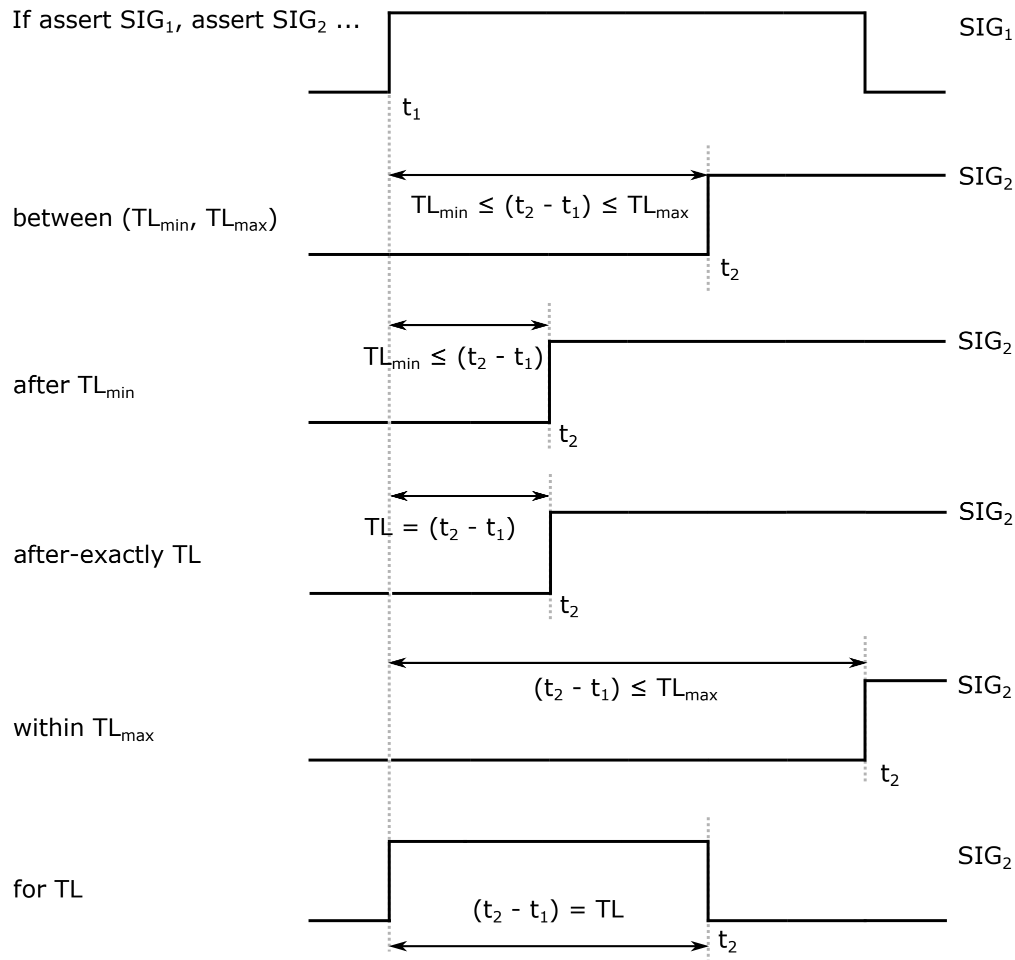

Figure 6.2: Timing diagram of TIMESPEC causality template for specific values for ACTION and SIGNAL fields.

TIMESPEC can be visually represented using a timing diagram, as shown in Figure 6.2.

6.2.3 Example: Slave Configuration Timings

The first step of monitor generation is choosing a suitable TIMESPEC template. This section selects several timing requirements extracted from an industrial sensor, OPT9221, to illustrate the versatility of TIMESPEC templates. OPT9221 is a time-of-flight (ToF) controller that measures the depth of digitized sensor data and is often used to create 3D

Table 6.4: Timing characteristics of OPT9221 slave configuration [155].

PARAMETER		MIN	MAX	UNIT
t_{CZ}	$\overline{\text{TIC_CONFIG}}$ low pulse duration	500	- -	ns
t_{CR}	Delay from $\overline{\text{TIC_CONFIG}}$ falling edge to TIC_CONF_DONE falling edge	- -	500	ns
t_{SZ}	$\overline{\text{TIC_STATUS}}$ low pulse duration	45	230	us
t_{DC}	$\overline{\text{TIC_STATUS}}$ rising edge to configuration clock's first rising edge	- -	2	us
t_{CCK}	Configuration clock period	15	- -	ns
DC	Configuration clock duty cycle	40%	60%	- -
t_{INIT}	End of configuration to start of firmware execution	300	650	us

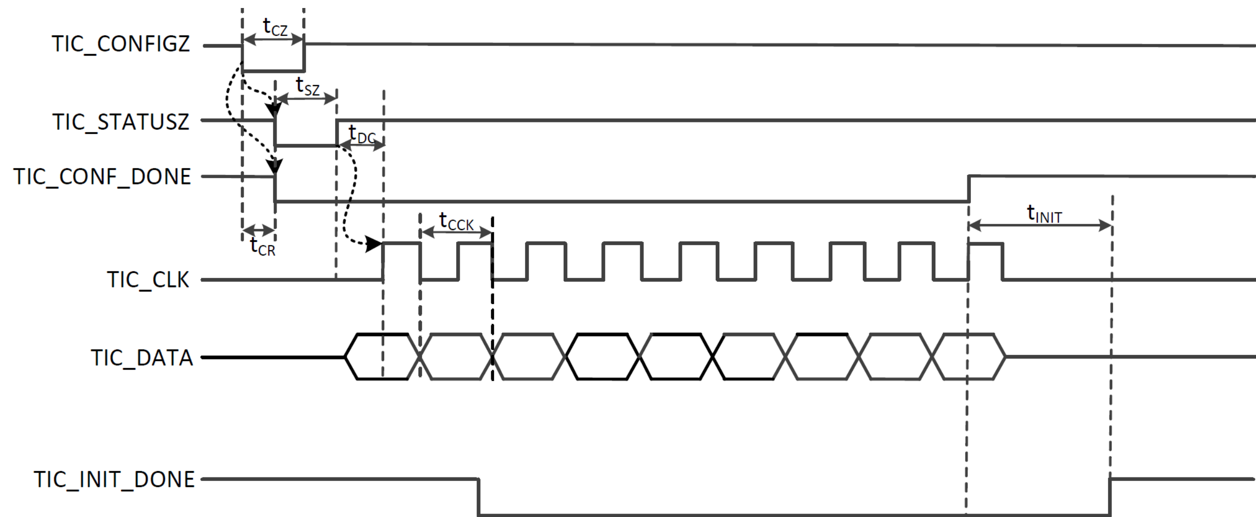

Figure 6.3: Timing diagram of OPT9221 slave configuration [155].

cameras [155]. Table 6.4 and Figure 6.3 show timing characteristics and timing diagram of the OPT9221 slave configuration signals, respectively. Variable t_{CZ} is the pulse width of active-low signal TIC_CONFIGZ, which has a minimum duration for proper functionality. Prameter t_{CCK} refers to the period of the TIC_CLK clock with a specific duty cycle. The pulse width of these signals should fall within defined ranges to ensure the optimal operation. To capture these temporal requirements, a TIMESPEC duration template is employed.

Selecting the appropriate timing constraint is critical as causal relationships between events are essential to system behavior. For example, the assertion of TIC_CONF_DONE causes TIC_INIT_DONE (denoted by t_{INIT}) to be asserted. This behavior should happen within a

Table 6.5: Timing requirements of OPT922 using pulse duration and causality templates.

Pulse Duration	RP1	t_{CZ}	Active_pulse_width of TIC_CONFIGZ should be ≥ 500 ns.
	RP2	t_{SZ}	Active_pulse_width of TIC_STATUSZ should be ≥ 45 us and ≤ 230 us.
	RP3	t_{CCK}	Period of TIC_CLK should be ≥ 15 ns.
	RP4	DC	Duty_cycle of TIC_CLK should be $\geq 40\%$ and $\leq 60\%$.
Causality	RC1	t_{CR}	If assert TIC_CONFIGZ, deassert TIC_CONF_DONE within 500 ns.
	RC2	t_{DC}	If deassert TIC_STATUSZ, start TIC_CLK within 2 us.
	RC3	t_{INIT}	If assert TIC_CONF_DONE, assert TIC_INIT_DONE between 300 us and 650 us.

certain time bound, captured by between value for TIMING_CONSTRAINT. Another instance is t_{DC}, where the deassertion of TIC_STATUSZ starts the clock signal TIC_CLK within a specific time frame. These distinct timing specifications are listed in Table 6.5.

6.3 Synthesis Workflow

FRET can manage a subset of timing requirements and express them as MTL formulas. However, its effectiveness is limited by two main factors: (i) there is no dedicated framework to convert these formulas into automata capable of handling timing constraints effectively, (ii) it does not support all common types of timing constraints. Adding timing requirements into the FRET-based flow would significantly complicate the synthesis process and reduce its efficiency. Furthermore, while model checking is well-suited for functional verification, it is not the best fit for timing verification. To address these limitations, GROOT uses a separate monitor synthesis flow specifically tailored to meet the unique needs of timing requirements.

As discussed in Section 2.3.5, adopting TA is well-suited for capturing the temporal behaviors of a system. Given GROOT's focus on minimizing the complexity of automata for formal verification and monitor efficiency, it automates the conversion of TIMESPEC expressions into TA. This formalization process is executed through TS2C, a tool designed for the automatic generation of C-based monitors from TIMESPEC statements. TS2C con-

sists of three modules: (i) the Parser module, which interprets TIMESPEC expressions; (ii) TA-Gen, which generates TA based on the parsed expressions; and (iii) Code-Gen, which transforms the generated automata into executable C code. TS2C also supports the formal analysis of the synthesized monitors.

6.3.1 TS2C Parser

Algorithm 3 details the parsing process. This parser is developed to handle TIMESPEC requirements and extract essential information from temporal requirements. It starts its operation with lexical analysis through the LEXER module. This initial phase segments the input characters into a sequence of tokens. The SYNTAX_ANALYZER module then defines grammar rules and constructs the abstract syntax tree (AST) using the generated tokens and established rules. The PARSER module uses the AST to build a requirement dictionary. This dictionary contains elements including Boolean events (APs), input variables, timing constraint type, and associated time values and units. The parser processes arbitrarily complex SIGNAL expressions containing logical, comparison and arithmetic operations on Boolean and integer variables and constants.

6.3.2 TS2C TA Generator

Following the parsing phase, the TA-Gen module maps each TIMESPEC statement to a specific TA based on the parsed data. A template-based flow minimizes the number of distinct TA variations needed and promotes consistency in the structure of the generated TAs. Given that each monitor supports one requirement, the resulting TA only needs one clock. In safety-critical systems, the TA's deterministic behavior ensures that it produces predictable responses to specific inputs or conditions, which eliminates any ambiguity in the

Algorithm 3 TIMESPEC parser script

```
procedure Parser(text)
    tokens ← Lexer(text)
    AST ← Syntax_Analyzer(tokens)
    Declare an empty dictionary (Req_dict)
    Add text to Req_dict['req']
    if AST is valid then
        if there is any comparison operation then
            if it is not already in APs then
                Assign a name to the operation and add it to APs as a new event
            end if
        end if
        Add names to Req_dict['inputs']                         ▷ variables
        Add TIMESPEC type to Req_dict['type']                   ▷ Pulse duration or causality
        Add time values and time units to Req_dict['time_constraints']
    else
        Print "Parsing failed"
    end if
    return Req_dict
end procedure
procedure Lexer(text)                   ▷ Perform lexical analysis on TIMESPEC requirement
    Build TIMESPEC alphabet (tokens)                          ▷ Including TIMESPEC fields
    Split text into tokens
    return tokens                                             ▷ List of Tokens
end procedure
procedure Syntax_Analyzer(tokens)
    Define TIMESPEC grammar rules                             ▷ Including ACTION, SIGNAL
    Construct abstract syntax tree (AST) using tokens and rules
    if AST is invalid then
        Print "Syntax error"
    end if
    return AST
end procedure
```

system's temporal behavior.

TA-Gen handles similar timing constraints by using a shared structure for certain types of requirements. For example, TA-Gen employs the structure illustrated in Figure 6.4 to address three variations of TIMING_CONSTRAINT in the causality template: between TL_{min}

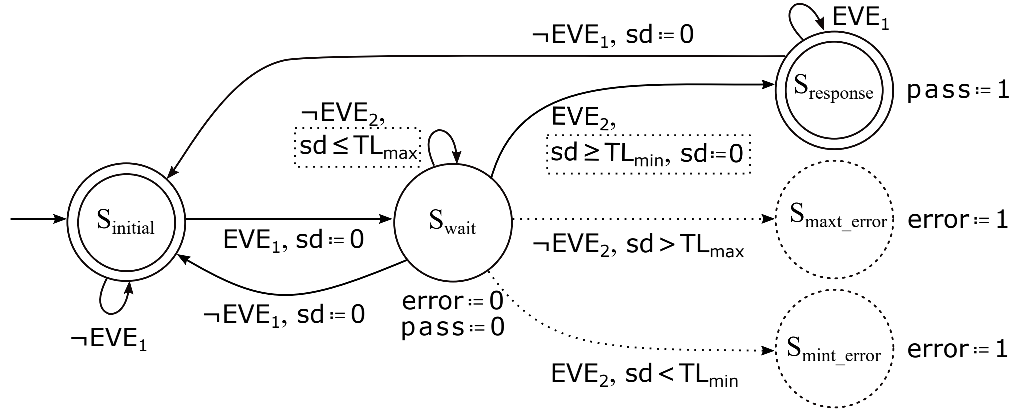

Figure 6.4: The TA structure used for between, after and within values. Solid lines are shared, while the dotted lines are chosen based on the requirement's timing constraint.

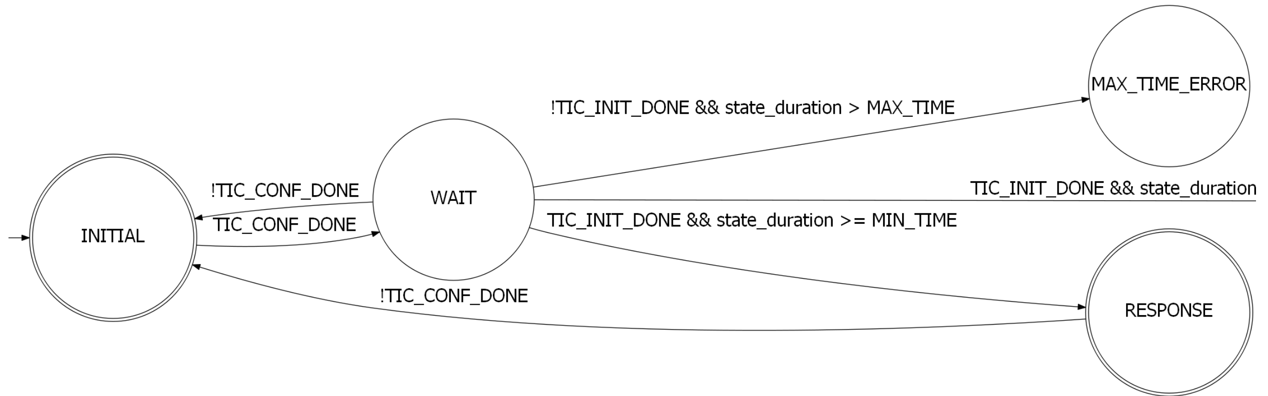

Figure 6.5: TA constructed by TA-Gen for requirement RC3.

and TL_{max}, after TL_{min}, and within TL_{max}. Transitions within TA are guarded by Boolean events (provided by Input_Handler) or restricted by temporal values, which depend on the monitor's invocation period. As indicated in Section 4.2, the state_duration (sd) parameter records the number of transitions spent in the present state to compare with the time bounds of the requirement every time the monitor is called. The purpose of sd variable is twofold: detecting potential deadline violations and providing useful information such as how long TA spends in a state. TA-Gen also provides visualization of the created automaton to help engineers comprehend the TA's behavior.

Figure 6.5 depicts a simplified version of the generated TA for RC3 extracted from Table 6.4.

As long as the monitor stays in the WAIT state, state_duration increments by one with each monitor invocation. This counter is compared to the MIN_TIME and MAX_TIME constants to ensure the expected event does not arrive sooner than the minimum time (300 us) but not later than the maximum time (650 us) either. In the event of a deviation from these temporal constraints, the monitor takes corrective action by setting the error variable. It also transits to the x_TIME_ERROR state and notifies Output_Handler. The monitor remains in this state until an external intervention from Output_Handler resets the monitor status.

Algorithm 4 outlines the steps involved in generating the TA. The **TA-Gen** module accepts the parsed requirement data from the previous phase and constructs the TA. A graph is built from this TA and displayed to provide a visual representation of the TA's states and transitions. Automaton creation involves iterating through the parsed data and adding states and transitions based on a predefined structure called the Generalized Timed Automaton (GTA), which is similar to Figure 6.4. The GTA is a TA with all possible states and transitions that address various types of TIMESPEC requirements. This structured approach helps to efficiently and consistently generate TAs from TIMESPEC requirements.

6.3.3 TS2C Code Generator

The Code-Gen module automatically translates the TA into an FSM implemented in C. The only human intervention required in this step is to assign a name to the monitor function and specify the argument types for Input_Handler. The generated code contains the source code and header files for the Main, Monitor, Input_Handler, and Output_Handler blocks, as illustrated in Figure 6.6 for the RC3 requirement. The header file enumerates TA states and defines time limit values, essential function declarations, and Monitor_State and Monitor_Inputs structures. It also declares a variable named monitor_period to specify the invocation

Algorithm 4 TA generator script

```
procedure TA-GEN(dictionary)
    TA_dict ← CREATE_AUTOMATON(dictionary)
    Build a graph from TA_dict
    Display the graph        ▷ TA's states and transitions create graph's nodes and edges
    return TA_dict
end procedure
procedure CREATE_AUTOMATON(dictionary)        ▷ Dictionary generated by parser
    Declare an empty dictionary (TA_dict)
    Set Boolean flags max and min based on data from dictionary
    gta ← GTA(min, max)                         ▷ Generalized TA
    for state in dictionary['states'] do
        if state in gta['states'] then
            Add state to TA_dict
            for dst, tran in dictionary['states'][state] do
                if dst in gta['states'][state] then
                    Add dst and tran to TA_dict
                end if
            end for
        end if
    end for
    return TA_dict
end procedure
procedure GTA(min, max)
    Define dictionary G based on all possible sources and destinations for TA
    Derive customized dictionary gta from G using min and max flags
    return gta
end procedure
```

period of monitor. This parameter assists the portability of monitors on different platforms, as it allows engineers to explicitly define the frequency at which the monitor is invoked.

The Input_Handler function processes the SIGNAL values extracted during parsing to construct Boolean events stored in Monitor_Inputs. This approach simplifies the overall monitor structure and reduces potential runtime overhead. The Output_Handler module provides the monitor's status by using data stored in Monitor_State. Monitor output depends on the states of the error and pass variables. These variables encode the following scenarios: (i) error

= 0 and pass = 0, which indicates an undetermined outcome meaning the monitor has not yet reached a verdict, (ii) error = 1, which implies the detection of an error needing corrective action, and (iii) pass = 1, which indicates the successful capture of the expected event. Note that in the latter scenario, monitoring for the current trace becomes unnecessary.

The Monitor code contains a `switch` statement representing the TA shown in Figure 6.5. To track time, the value of state_duration is compared to the automatically calculated time limit values. These values are derived based on the temporal constraints in the TIME-SPEC requirement and the user-defined monitor_period. This enables the reuse of the same monitor across systems with varying timebases, which requires only the adjustment of the monitor_period value. As discussed in Section 5.3.3, the Main source code is structured as a task within RTOS, responsible for encapsulating the monitoring modules.

6.3.4 Monitor Formal Analysis

As mentioned in Section 4.5, this work relies on the correctness of the established trust base. However, GROOT generates monitors using several transformations on the specified requirements. The correctness of these synthesized monitors is crucial as they are used to ensure system meets its specifications. To ascertain the trustworthiness of the monitors, a formal verification step is required. However, the inherent complexity arising from integer arithmetic calculations related to time measurements poses a challenge for traditional model checkers due to the well-known state explosion problem (as discussed in Section 2.1.2). Consequently, GROOT employs the Frama-C theorem prover for the formal verification of monitor implementations. The verification process involves the formulation of formal behavioral contracts expressed in the ANSI/ISO C specification language (ACSL) [22].

An advantage of using templates for timing requirements is automating this step by defining

```c
#include <stdbool.h>
#include <stdio.h>
#define monitor_period 1 // Change it based on the system frequency
#define min_time 300
#define max_time 650
enum States {INITIAL, WAIT, RESPONSE, MAX_TIME_ERROR, MIN_TIME_ERROR};
struct Monitor_State {
    enum States state;
    bool error, pass;
    unsigned int state_duration; };
struct Monitor_Inputs {
    bool EVE0, EVE1; };
void RC3_monitor(struct Monitor_State* monitor_state,
                struct Monitor_Inputs* monitor_inputs);
void input_handler(bool TIC_INIT_DONE, bool TIC_CONF_DONE,
                struct Monitor_Inputs* monitor_inputs);
void error_handler(struct Monitor_State* monitor_state);
```

```c
void input_handler(bool TIC_INIT_DONE, bool TIC_CONF_DONE,
    struct Monitor_Inputs* monitor_inputs) {
    monitor_inputs->EVE0 = TIC_CONF_DONE;
    monitor_inputs->EVE1 = TIC_INIT_DONE; }
```

```c
void error_handler(struct Monitor_State* monitor_state) {
    if (monitor_state->error || (monitor_state->pass) {
        monitor_state->state = INITIAL;
            monitor_state->error = false;
            monitor_state->pass = false;
            monitor_state->state_duration = 0;
            if (monitor_state->error) { printf("Response is received!"); }
            else { printf("Response is received!"); }}}
```

```c
void main(int argc, char** argv) {
    struct Monitor_State monitor_state =
    { INITIAL, false, false, 0 };
    struct Monitor_Inputs monitor_inputs = { false,false };
    while (1) {
        input_handler(&monitor_inputs);
        RC3_monitor(&monitor_state, &monitor_inputs);
        error_handler(&monitor_state); }}
```

```c
void RC3_monitor(struct Monitor_State* monitor_state,
                struct Monitor_Inputs* monitor_inputs) {
    enum States state = monitor_state->state;
    bool error = monitor_state->error;
    bool pass = monitor_state->pass;
    unsigned int state_duration = monitor_state->state_duration;
    bool EVE0 = monitor_inputs->EVE0;
    bool EVE1 = monitor_inputs->EVE1;
    unsigned int MAX_TIME = max_time / monitor_period;
    unsigned int MIN_TIME = min_time / monitor_period;
    switch (state) {
        case INITIAL:
            state_duration += 1;
            if (EVE0) {
                state = WAIT;
                state_duration = 0; }
            break;
        case WAIT:
            state_duration += 1;
            if (!EVE0) {
                state = INITIAL;
                state_duration = 0; }
            else if (EVE1 && state_duration >= MIN_TIME) {
                state = RESPONSE;
                state_duration = 0; }
            else if (!EVE1 && state_duration > MAX_TIME) {
                state = MAX_TIME_ERROR; }
            else if (EVE1 && state_duration < MIN_TIME) {
                state = MIN_TIME_ERROR; }
            break;
        case RESPONSE:
            state_duration += 1;
            pass = true;
            if (!EVE0) {
                state = INITIAL;
                state_duration = 0; }
            break;
        case MAX_TIME_ERROR:
            error = true;
            break;
        case MIN_TIME_ERROR:
            error = true;
            break;
        default:
            error = false;
            pass = false;
            break; }
    monitor_state->state = state;
    monitor_state->error = error;
    monitor_state->pass = pass;
    monitor_state->state_duration = state_duration; }
```

Figure 6.6: The generated header, Input_Handler, Output_Handler and Main functions (from top to bottom on the left), along with the Monitor function (on the right) for RC3.

templated ACSL contracts for each TIMESPEC statement. For example, Figure 6.7 shows the ACSL specifications generated by Code-Gen for the RC3 monitor. These specifications outline the function's preconditions (the *ensures* contracts), postconditions (the *requires* contracts), its expected behavior (the *behavior* blocks), and variations under different circumstances. In other words, these specifications verify that the monitor's `switch-case` structure depicted in Figure 6.6 conforms to the TA illustrated in Figure 6.5.

```
/*@
(1)  requires \valid(monitor_state) && \valid_read(monitor_inputs);
(2)  requires \separated(monitor_state, monitor_inputs);
(3)  requires 0 <= monitor_state->state_duration;
(4)  ensures 0  <= monitor_state->state_duration;
(5)  ensures (monitor_state->state\in{INITIAL, WAIT, RESPONSE,
                MAX_TIME_ERROR, MIN_TIME_ERROR});
(6)  ensures((monitor_state->error == 1)==>
                (monitor_state->state\in{MAX_TIME_ERROR, MIN_TIME_ERROR}));
(7)  ensures((monitor_state->pass == 1)==>
                (monitor_state->state == RESPONSE));
(8)  ensures (!(monitor_state->error && monitor_state->pass));
(9)  ensures unchanged_inputs: *monitor_inputs == \old(*monitor_inputs);
(10) behavior not_started:
         assumes monitor_state->state \in {INITIAL};
         assumes monitor_inputs->EVE0 == 0;
         ensures monitor_state->state == INITIAL;
(11) behavior begin:
         assumes monitor_state->state \in {INITIAL};
         assumes monitor_inputs->EVE0 == 1;
         ensures monitor_state->state == WAIT;
(12) behavior satisfaction_or_min_time_violation:
         assumes monitor_state->state \in {WAIT};
         assumes monitor_inputs->EVE1 == 1;
         ensures ((monitor_state->state_duration >= MIN_TIME ==>
                    monitor_state->state == RESPONSE) ||
                 (monitor_state->state_duration < MIN_TIME ==>
                    monitor_state->state == MIN_TIME_ERROR));
(13) behavior wait_or_max_time_violation:
         assumes monitor_state->state \in {WAIT};
         assumes monitor_inputs->EVE1 == 0;
         ensures ((monitor_state->state_duration <= MAX_TIME ==>
                    monitor_state->state == WAIT) ||
                 (monitor_state->state_duration > MAX_TIME ==>
                    monitor_state->state == MAX_TIME_ERROR));
(14) behavior reset_trigger:
         assumes monitor_state->state \in {RESPONSE};
         assumes monitor_inputs->EVE0 == 1;
         ensures monitor_state->state == RESPONSE;
(15) behavior reintitialize:
         assumes monitor_state->state \in {RESPONSE};
         assumes monitor_inputs->EVE0 == 0;
         ensures monitor_state->state == INITIAL;
(16) behavior stop:
         assumes monitor_state->state \in{MIN_TIME_ERROR,MAX_TIME_ERROR};
         ensures monitor_state->error == 1;
(17) disjoint behaviors;
*/
```

Figure 6.7: ACSL contracts for proving the correctness of the RC3 monitor by Frama-C.

Contract (1) ensures the validity of pointers monitor_state and monitor_inputs, confirming that they are appropriately set up for writing and reading operations, respectively. This is crucial for memory safety, as it prevents the function from accessing null or uninitialized pointers. Contract (2) asserts that the memory regions pointed to by monitor_state and monitor_inputs do not overlap, reducing the risk of unintended aliasing issues. Note that unlike ESBMC which automatically verifies these properties as explained in Section 5.3.4, Frama-C requires Code-Gen to generate ACSL specifications pertaining to memory properties.

Contracts (3) and (4) establish a valid range for the monitor_state.state_duration counter, guaranteeing its contents consist solely of non-negative integers. This constraint is crucial for precise timing and effective detection of responses. Contract (5) guarantees that the `switch-case` structure encompasses all possible monitor states defined by the TA. Contracts (6) and (7) validate the consistency between monitor_state.error and monitor_state.pass and the actual error and satisfaction states, ensuring that the monitor correctly reflects the system's behavior and provides reliable verdicts based on its observations. For example, the monitor_state.pass variable holds a true value when the monitor is in the RESPONSE state, as shown in the monitor source code illustrated in Figure 6.6. Contract (8) ensures the mutual exclusivity of the monitor's verdicts, which is critical for maintaining the trustworthiness of the monitor's results by preventing conflicting outcomes.

In contract (9), the state of monitor_inputs before and after the function execution is compared, confirming that the monitor code only reads the APs without altering them. This ensures that the APs, which represent the system's observable properties, are updated exclusively by the Input_Handler module, as discussed in Section 4.2. Contracts (10) through (16) verify that the conditions specified in each `case` statement of the monitor code comply with the transitions defined within the TA. For example, contract (10) guarantees that the monitor remains in the INITIAL state as long as trigger event TIC_CONF_DONE is deasserted.

These behaviors partition the function's specification based on different scenarios or assumptions, with each behavior having its own set of assumptions (*assumes*) and guarantees (*ensures*), effectively tailoring the specification for different conditions. Last, contract (17) confirms that these behaviors are mutually exclusive, indicating that the conditions for one behavior cannot be true for another at the same time. This is crucial for ensuring that the specification is clear and non-overlapping.

6.4 Summary

This chapter demonstrates GROOT's monitor synthesis process tailored for timing requirements. It introduces TIMESPEC, a structured English language designed for expressing time limits. TIMESPEC assists practitioners to capture specifications containing metric time constraints without requiring expertise in formal methods. In the synthesis process, the TIMESPEC requirements are automatically converted into automata, followed by the automatic generation of monitors coded in C. The correctness of these synthesized monitors is confirmed through automated verification using an established theorem prover.

Chapter 7

Evaluation

This chapter evaluates the capabilities of GROOT framework through a series of illustrative examples that demonstrate its application across different systems. The chapter is structured into three main parts, each addressing a specific set of challenges and environments where GROOT's monitors are utilized.

7.1 Overview

This chapter presents three case studies in the fields of aviation, embedded security systems and autonomous vehicles, which represent high-stakes industries where precision and reliability are paramount. These examples involve intricate interactions of hardware and software with significant implications for human safety and economic impact. They are chosen to cover a substantial portion of the modern landscape of safety-critical real-time embedded systems to show GROOT's application and performance across various scenarios.

The first case study provides insights into our collaboration with GE Global Research and GE Aviation, where GE engineers evaluate the effectiveness of the generated monitors. This collaboration primarily focused on automating monitor synthesis and verification in hardware. This section first outlines several specific requirements from GE that guide the creation of tailored monitors. Subsequent parts include formal analysis and simulation, which complement each other. The formal analysis enhances trust in the monitor's outcomes, while

simulation validates the monitor's accuracy against the system specifications. The case study concludes with a comparison of the time required to implement a specific monitor using GROOT versus GE's conventional method.

Embedded systems such as IoT devices and smart sensors often operate in environments where security is crucial. These systems frequently employ lightweight cryptographic algorithms to secure data and communications. However, the inherent resource constraints of embedded systems pose difficulties in ensuring both correct implementation and ongoing secure operation of these cryptographic solutions. Consequently, the second case study addresses the monitoring of an existing lightweight cryptographic system using GROOT's monitors. This section illustrates the entire process from the synthesis of monitors to their formal analysis and simulation. It also assesses the impact of these monitors on system resources and power consumption by comparing them to the original cryptographic system.

Autonomous vehicles, equipped with various sensors and processors, aim to enhance safety and reduce human error. However, the transition to fully autonomous driving requires a high level of safety assurance. For example, Tesla's autopilot system has been associated with several accidents including fatal crashes, due to the vehicle's inability to handle unexpected road conditions [31, 32, 164]. In the third example, an autonomous system is employed to demonstrate the applicability of GROOT's monitors across different phases of product development. This section begins by defining requirements and generating monitors based on these requirements. Subsequently, it provides a formal analysis of these monitors, followed by simulations that verify the functionality of monitors against an existing model. Finally, the study compares the code and performance of GROOT's monitors with the closest available monitor synthesis framework.

7.2 Case 1: UAS Flying within a Geofence

In collaboration with GE Global Research and GE Aviation on the NASA System-wide Safety program, hardware monitors were developed to ensure the UAS operation within a designated geofence consisting of complex polygonal boundaries with exclusion zones. The project applied requirements for GE Aviation's Skynode™ runtime assurance (RTA) flight controller [3] supporting commercial UAS flights, with monitors helping to provide the safety assurance needed for FAA certification. Monitor synthesis using GROOT was compared to GE Aviation's existing approach in terms of engineering time and effort.

7.2.1 Monitor Synthesis

Table 7.1 shows the RTA functional requirements in English, FRETish and LTL, with specific examples discussed in Section 3.3 and Section 5.3.2. GROOT treats the RTA system as a black box, with GE engineers assessing the adherence of generated monitors to predefined requirements. Various monitors were crafted, simulated, and tested using FRET's simulator and monitor implementation tools to select the optimal ones, without requiring any modification to GE Aviation's original system design. The monitor synthesis process follows the flow explained in Chapter 5.

While FRET simplifies the formalization of requirements, crafting a valid FRETish requirement needs a solid understanding of system specifications and functionality. In order to illustrate the subtleties in capturing correct LTL, Table 7.2 presents a possible alternative interpretation of the RTA-R0 requirement in both FRETish and LTL. This requirement is intended to ensure the UAS can operate near the geofence limits without violating them. Comparing the generated LTL formulas, it might be challenging for a practitioner unfamiliar with LTL semantics to distinguish between these versions. However, FRET's built-in

Table 7.1: Geofence requirements for the RTA collaboration project.

<table>
<tr><td>RTA-R0</td><td>RTA should set a corrective action if the UAS breaches the fence in the next two time units but not in the next time unit.</td></tr>
<tr><td>FRET input</td><td>if (breach_in_2 & !breach_in_1) RTA shall until !breach_in_2 satisfy corrective_action</td></tr>
<tr><td>LTL formula</td><td>((LAST V (((! ((breach_in_2 & ! breach_in_1))) & ((! LAST) & (X ((breach_in_2 & ! breach_in_1))))) → (X (((! breach_in_2) V (((corrective_action)) | (! breach_in_2))) | (LAST V ((corrective_action))))))) & (((breach_in_2 & ! breach_in_1)) → (((! breach_in_2) V (((corrective_action)) | (! breach_in_2))) | (LAST V ((corrective_action))))))</td></tr>
<tr><td>RTA-R1</td><td>If the UAS projected altitude is below an altitude floor, a warning should be issued (except in landing stage).</td></tr>
<tr><td>FRET input</td><td>except in landing RTA shall always satisfy ((future_alt < alt_floor) -> alt_warning)</td></tr>
<tr><td>LTL formula</td><td>(LAST V ((! landing) → (((future_alt < alt_floor) → alt_warning))))</td></tr>
<tr><td>RTA-R2</td><td>If the WN calculator shows the UAS is outside the polygon for three consecutive times, an error must be set to enable a safety action.</td></tr>
<tr><td>FRET input</td><td>if (out_cntr >= 3) RTA shall immediately satisfy error_UAS_out</td></tr>
<tr><td>LTL formula</td><td>((LAST V (((! ((out_cntr >= 3))) & ((! LAST) & (X ((out_cntr >= 3))))) → (X (error_UAS_out)))) & (((out_cntr >= 3)) → (error_UAS_out)))</td></tr>
<tr><td>RTA-R3</td><td>RTA should set a geofence error if the UAS is inside either of the two exclusive fences or if not inside the inclusive fence.</td></tr>
<tr><td>FRET input</td><td>if ((in_ex1_zone | in_ex2_zone)| !in_fence) RTA shall immediately satisfy geofence_error</td></tr>
<tr><td>LTL formula</td><td>((LAST V (((!(((in_ex1_zone | in_ex2_zone) | ! in_fence))) & ((! LAST) & (X(((in_ex1_zone | in_ex2_zone) | ! in_fence))))) → (X(geofence_error)))) & ((((in_ex1_zone | in_ex2_zone) | ! in_fence)) → (geofence_error)))</td></tr>
<tr><td>RTA-R4</td><td>For a specific edge, RTA decides whether the UAS is on the correct side.</td></tr>
<tr><td>FRET input</td><td>if ((((y1 <= 0) & (y2 > 0)) & (((x1 * y2) - (x2 * y1)) > 0)) | (((y1 >= 0) & (y2 < 0)) & (((x1 * y2) - (x2 * y1)) < 0))) RTA shall immediately satisfy edge_detect</td></tr>
<tr><td>LTL formula</td><td>((LAST V (((! (((((y1 <= 0) & (y2 > 0)) & (((x1 * y2) - (x2 * y1)) > 0)) | (((y1 >= 0) & (y2 < 0)) & (((x1 * y2) - (x2 * y1)) < 0))))) & ((! LAST) & (X (((((y1 <= 0) & (y2 > 0)) & (((x1 * y2) - (x2 * y1)) > 0)) | (((y1 >= 0) & (y2 < 0)) & (((x1 * y2) - (x2 * y1)) < 0))))))) → (X (edge_detect)))) & ((((((y1 <= 0) & (y2 > 0)) & (((x1 * y2) - (x2 * y1)) > 0)) | (((y1 >= 0) & (y2 < 0)) & (((x1 * y2) - (x2 * y1)) < 0)))) → (edge_detect)))</td></tr>
</table>

simulator can assist in evaluating the formulas with desired inputs.

Figure 7.1 illustrates the simulation outcomes for both versions of RTA-R0 concerning a UAS confined within the square geofence depicted in Figure 7.2. Assume that at time $t = 1$, the UAS is located at point B, and its trajectory suggests a potential breach within two

Table 7.2: Second version of FRETish and LTL for the RTA-R0 requirement.

FRET input (v2)	if (breach_in_2 & !breach_in_1) RTA shall immediately satisfy corrective_action
LTL formula (v2)	((LAST V (((! ((breach_in_2 & ! breach_in_1))) & ((!LAST) & (X ((breach_in_2 & ! breach_in_1))))) → (X ((corrective_action))))) & (((breach_in_2 & ! breach_in_1)) → ((corrective_action))))

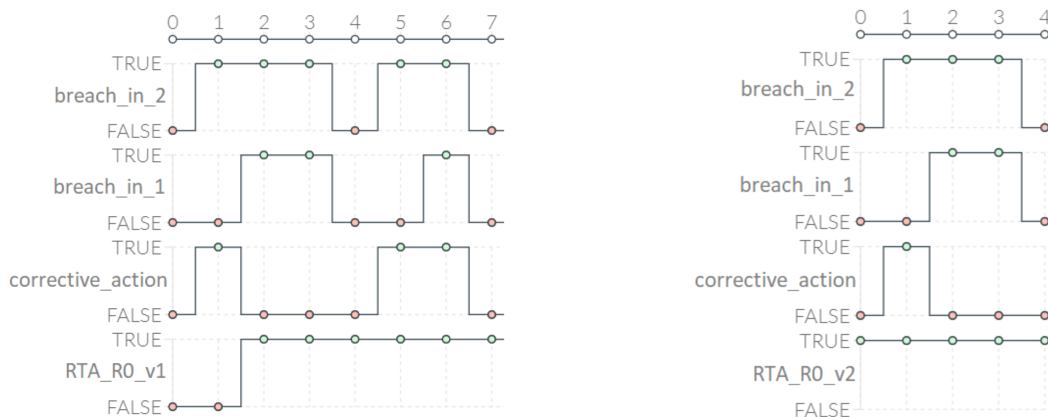

Figure 7.1: Visualizations of the original (left) and second (right) versions of RTA-R0.

subsequent GPS data points if it maintains its current velocity. In other words, by $t = 3$, the UAS is projected to reach point D under the current conditions. However, the trajectory also indicates that the UAS still remains within the geofence for the subsequent point, marked as C, implying that the breach is not imminent. Consequently, at point B, the RTA should initiate a corrective measure, such as stopping the UAS longitude velocity, to prevent it from reaching point D. If the RTA monitor operates as intended, it is expected that at point C, the RTA's intervention successfully averts the anticipated breach, allowing the UAS to hover near the boundary. Note that in the event of a breach within the next time unit, another monitor is tasked with initiating a safety action such as immediately landing the UAS.

Upon comparing the simulation results, the second version does not detect any error, while the original version identifies a violation. This discrepancy arises because *corrective_action* in the original version relies on *breach_in_*2, implying that activating corrective action for one cycle may not be adequate. Figure 7.3 and Figure 7.4 depict the monitor FSMs for

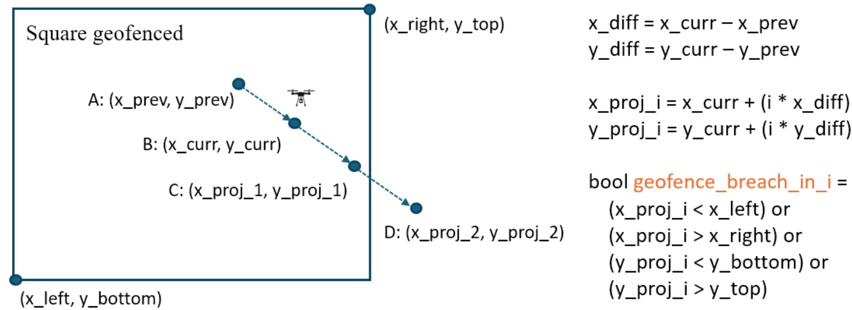

Figure 7.2: Estimation of the UAS future position.

these two versions. The trigger condition for both monitors is the same (transition from state 1 to state 0). However, if *corrective_action* is false while *breach_in_2* remains true, the original monitor moves to non-accepting state 2, whereas the second monitor disregards *corrective_action* status in this state. This example highlights that while GROOT verifies the C and HDL codes to validate the the synthesized monitors, determining which requirement version best fits the system ultimately rests with the engineer.

As explained in Section 4.4, GROOT's C-based monitors can be automatically transformed into hardware description language (HDL) using a commercial high-level synthesis (HLS) tool, which translates high-level descriptions such as those scripted in C and C++ into digital circuits. This transformation process significantly reduces the time and effort required to manually develop hardware source codes. Following the approach described in Section 3.1, this dissertation employs the Xilinx HLS tool [162] to automate this conversion.

7.2.2 Monitor Formal Analysis

In this project, the monitors undergo several stages of transformation or translation, requiring verification of the final HDL codes against the original automata. To achieve this, the EBMC model checker [95] is utilized to analyze the hardware modules against a set of specifications known as SystemVerilog assertions (SVAs) [160]. These assertions specify properties or

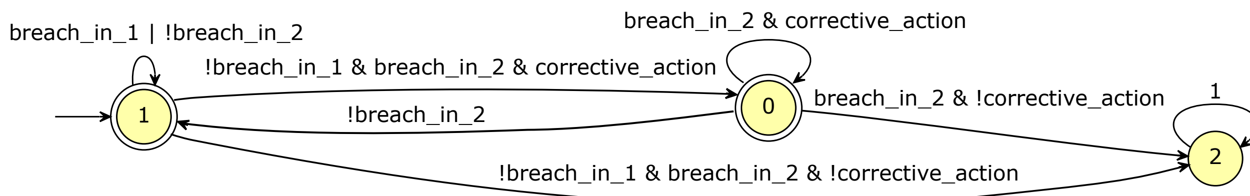

Figure 7.3: Monitor FSM for the original version of RTA-R0.

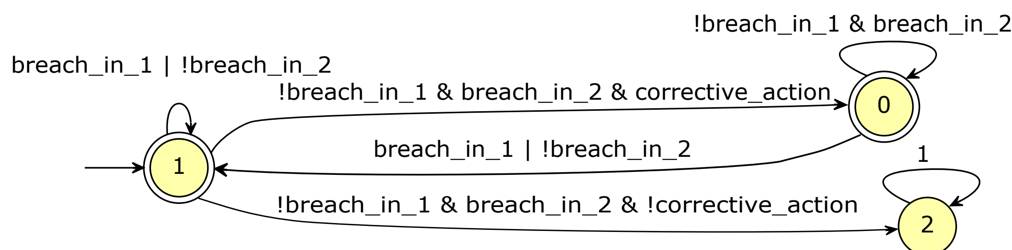

Figure 7.4: Monitor FSM for the second version of RTA-R0.

conditions that must hold in the hardware design. Section 2.1.2 provides an overview of the model checking process.

Table 7.3 shows various SVAs generated by GROOT for the RTA-R0 monitor. These SVAs validate that the monitor hardware design accurately adheres to the transitions specified by the BA depicted in Figure 7.3. For example, assertion (1) ensures that the monitor remains in state 1 as long as either *breach_in_*1 is true or *breach_in_*2 is false. This means the monitor stays in its initial state and does not activate if either an imminent breach is detected or if there is no prediction of a breach within the next two time units. As mentioned earlier, a separate monitor addresses situations where the breach occurs in the subsequent time unit.

Assertion (2) validates that upon the occurrence of the monitor's trigger condition, the *corrective_action* signal immediately becomes true. The monitor then transitions to state 0 and stays there while both *breach_in_*2 and *corrective_action* are true, a condition verified by assertion (4). In the event that *breach_in_*2 is deasserted while the monitor is in state 0, the monitor can reset and moves to the initial state, as discussed in the previous section and confirmed by assertion (5). If *corrective_action* fails to assert while *breach_in_*2 is true,

Table 7.3: Generated SVAs for the RTA-R0 hardware monitor.

<table>
<tr><th>ID</th><th>SystemVerilog Assertion (SVA)</th></tr>
<tr><td>(1)</td><td>(ap_start & (state == 2'd1) & (breach_in_1 | !breach_in_2) |=> (state == 2'd1))</td></tr>
<tr><td>(2)</td><td>(ap_start & (state == 2'd1) & (!breach_in_1 & breach_in_2 & corrective_action) |=> (state == 2'd0))</td></tr>
<tr><td>(3)</td><td>(ap_start & (state == 2'd1) & (!breach_in_1 & breach_in_2 & !corrective_action) |=> (state == 2'd2))</td></tr>
<tr><td>(4)</td><td>(ap_start & (state == 2'd0) & (breach_in_2 & corrective_action) |=> (state == 2'd0))</td></tr>
<tr><td>(5)</td><td>(ap_start & (state == 2'd0) & (!breach_in_2) |=> (state == 2'd1))</td></tr>
<tr><td>(6)</td><td>(ap_start & (state == 2'd0) & (breach_in_2 & !corrective_action) |=> (state == 2'd2))</td></tr>
</table>

```
** Results:
[breach_prediction.property.1] always (breach_prediction.ap_start && breach_prediction.state ==
2'b1) & (breach_prediction.breach_in_1 | !breach_prediction.breach_in_2) |=> breach_prediction.s
tate == 2'b1: SUCCESS
[breach_prediction.property.2] always (breach_prediction.ap_start && breach_prediction.state ==
2'b1) & !breach_prediction.breach_in_1 & breach_prediction.breach_in_2 & breach_prediction.corre
ctive_action |=> breach_prediction.state == 2'b0: SUCCESS
[breach_prediction.property.3] always (breach_prediction.ap_start && breach_prediction.state ==
2'b1) & !breach_prediction.breach_in_1 & breach_prediction.breach_in_2 & !breach_prediction.corr
ective_action |=> breach_prediction.state == 2'b10: SUCCESS
[breach_prediction.property.4] always (breach_prediction.ap_start && breach_prediction.state ==
2'b0) & breach_prediction.breach_in_2 & breach_prediction.corrective_action |=> breach_predictio
n.state == 2'b0: SUCCESS
[breach_prediction.property.5] always breach_prediction.ap_start && breach_prediction.state == 2
'b0 && !breach_prediction.breach_in_2 |=> breach_prediction.state == 2'b1: SUCCESS
[breach_prediction.property.6] always (breach_prediction.ap_start && breach_prediction.state ==
2'b0) & breach_prediction.breach_in_2 & !breach_prediction.corrective_action |=> breach_predicti
on.state == 2'b10: SUCCESS
```

Figure 7.5: EBMC verification results for the RTA-R0 monitor.

it indicates a violation, and the monitor should move to non-accepting state 2. Hence, assertions (3) and (6) ensure that this violation is correctly handled in state 1 and state 0, respectively. This is crucial because, based on the specification, once the monitor is activated, *corrective_action* must remain asserted until *breach_in_2* becomes false.

The *ap_start* variable referenced in these assertions denotes the initiation signal of the monitor module. During the conversion of C-based FSM to HDL by the HLS tool, additional control signals such as clock, reset, start, idle, and done are incorporated into the hardware description. These signals provide necessary control and status information, enabling proper synchronization between different hardware modules and the overall system. Figure 7.5 presents the successful verification results of EBMC for RTA-R0 against the discussed SVAs.

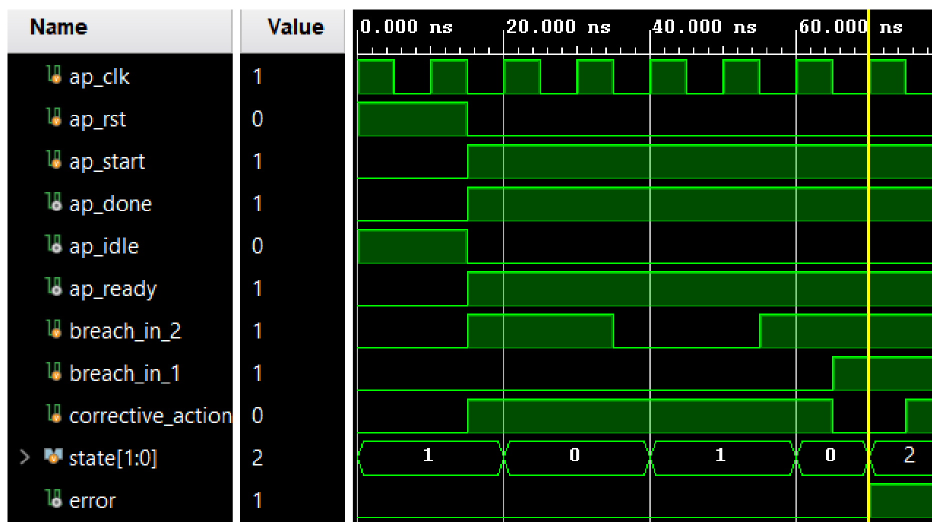

Figure 7.6: Simulation results for the RTA-R0 monitor.

7.2.3 Monitor Simulation

While formal analysis of monitors provides theoretical verification of their implementation, testing them in a simulation environment with arbitrary inputs offers an additional layer of assurance. These two complementary approaches increase confidence in the correctness and validation of the monitor implementation. To ensure that the system specification is correctly captured, the synthesized hardware monitor is implemented in a commercial logic simulator called Vivado [4]. This tool is widely used for synthesis and analysis of HDL designs targeting various hardware platforms.

Figure 7.6 provides an overview of the simulation outcomes for the RTA-R0 monitor with a clock duration set to 10 ns. After the reset (ap_rst) is deasserted and all the signals are initialized at $t = 15$ ns, the monitor begins its operation at the rising edge of the clock signal (ap_clk). By $t = 20$ ns, the monitor transitions from state 1 to state 0 as $breach_in_1$ and $breach_in_2$ are false and true, respectively, while $corrective_action$ is true. This transition indicates that the RTA has detected a potential breach event in the next two time units and initiated the appropriate corrective action.

At $t = 40$ ns, the monitor returns to initial state 1 as *breach_in_2* becomes false, indicating that the potential breach has been effectively addressed and no further monitoring is required. At $t = 60$ ns, the monitor is triggered to transition back to state 0. According to RTA-R0 presented in Table 7.1, *corrective_action* must remain true until *breach_in_2* becomes false. However, at $t = 70$ ns, *corrective_action* becomes false, resulting in the activation of the *error* flag and the transition of monitor to non-accepting state 2. These simulation results correspond with the expected behavior described by the BA shown in Figure 7.3, confirming that the monitor has been successfully implemented in hardware and its functionality conforms to the specified requirement.

7.2.4 Productivity Comparison

Figure 7.7 illustrates GE Aviation's conventional process for generating RTA monitors. Initially, algorithms are implemented in Simulink [52], which is a graphical environment commonly used for modeling and simulating dynamic systems. Subsequently, VHDL code is automatically generated from the Simulink model. Finally, ModelSim [1], a commonly used HDL simulation tool for testing and debugging digital circuits, is employed to simulate the generated code and validate its outputs against expected results using manually crafted test vectors. This approach solely relies on simulation, requiring comprehensive test cases to achieve high coverage and enhance trust in the generated code. Table 7.4 compares the verification and validation (V&V) time required to implement a monitor for RTA-R3 (containing only Boolean expressions) using GE Aviation's existing methodology and GROOT. Results show that GROOT is 2.3 times more time-efficient for the RTA-R3 requirement. The existing approach's time-consuming part is test case generation, which is not required for the formal approach as the monitors are thoroughly validated by the model checker.

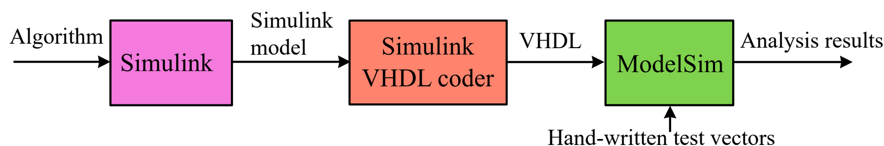

Figure 7.7: GE approach for geofence monitor generation.

Table 7.4: End-to-end V&V process metrics for a requirement with only Boolean expressions.

Formal (GROOT's FRET-based) approach (hours)			
Requirement articulation in FRET	Model checking with EBMC	Simulation in Vivado	Total
0.5	0.5	0.5	1.5
Traditional (GE) approach (hours)			
Simulink model generation	Test case generation	Simulation in ModelSim	Total
1	2.5	< 0.5	3.5

7.3 Case 2: Lightweight Cryptographic System

As ubiquitous devices become prevalent in the Internet of Things (IoT), there is a growing demand for lightweight cryptographic algorithms. The NIST Lightweight Cryptography (LWC) standardization process evaluates and standardizes algorithms suitable for resource-constrained devices [148]. Despite the mathematical security proofs, vulnerabilities in physical implementations can be exploited by adversaries, for example through hardware Trojan (HT) attacks, which manipulate a device's electronic circuits at the hardware level [29]. HTs can be introduced at various stages, from design to manufacturing, posing significant threats in safety-critical applications. Countermeasures against such attacks should impose minimal overhead, particularly for constrained devices. To evaluate the performance of GROOT's monitors in this scenario, two specific requirements are established for GIFT-COFB, a candidate in the LWC selection process, to detect HT threats. The SUS utilizes the hardware implementation of GIFT-COFB and its characteristics that are detailed in [139].

Table 7.5: GIFT properties in FRETish and TIMESPEC.

GIFT-R0	During the encryption process, the *start* signal should remain asserted until the *done* signal turns true.
FRET input	If start GIFT shall until done satisfy start.
LTL formula	((LAST V (((! (start)) & ((! LAST) & (X (start)))) → (X (((done) V ((start) | (done))) | (LAST V (start)))))) & ((start) → (((done) V ((start) | (done))) | (LAST V (start)))))
GIFT-R1	Once the encryption process is initiated, the *done* signal should be asserted exactly 40 ticks later.
TIMESPEC	If assert start, assert done after-exactly 40 cycles.

7.3.1 Monitor Synthesis

GIFT-COFB is a lightweight cryptographic algorithm with the GIFT block cipher as its underlying cipher [15]. Using a 128-bit key, GIFT encrypts a 128-bit plaintext into a 128-bit ciphertext through a series of 40 rounds. In the current implementation, the encryption procedure initiates upon receiving the *start* signal and concludes after 40 clock cycles (cc), indicated by the assertion of the *done* signal. To detect malfunctions or potential attacks, two monitors are developed based on the specifications expressed in Table 7.5. These monitors scrutinize the cipher, with particular emphasis on the control signals *start* and *done*. Chapter 5 and Chapter 6 discuss the developed flows for generating these monitors.

The first specification is a functional requirement, ensuring that GIFT encryption process is executed as expected from start to finish, without any unauthorized or unexpected interruption. For example, premature deassertion of the *start* signal could indicate a potential security breach. Figure 7.8 illustrates the monitor FSM for the GIFT-R0 requirement, with state 2 as the initial state and all states except for state 3 being accepting (discussed in Section 2.3.4). According to this diagram, if *start* is deasserted during an active encryption process without *done* being asserted, the monitoring system identifies this as a protocol breach, indicating an interruption or manipulation of the intended encryption sequence.

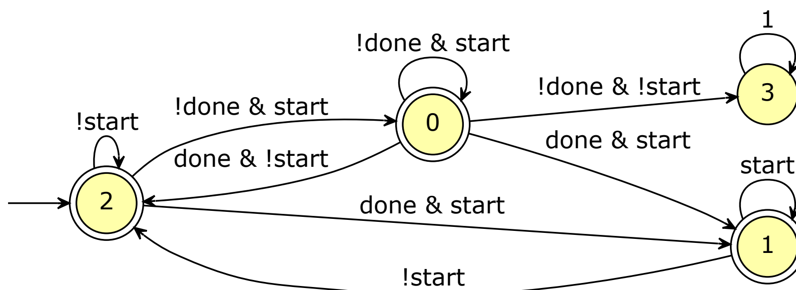

Figure 7.8: GIFT-R0 monitor FSM provided by LTL2C.

While the GIFT-R0 monitor can verify the integrity of the *start* and *done* signals, it lacks the capability to confirm the timing of the *done* signal assertion. For example, in Figure 7.8, if *done* is asserted at any point during the encryption process, the monitor transitions to either accepting state 1 or accepting state 2 based on the current status of *start*. Thus, the second requirement is established to ensure the correct progression of the encryption process through its 40 rounds and the proper assertion of *done* without premature termination or extension beyond the expected rounds.

Figure 7.9 depicts the GIFT-R1 monitor FSM, where the acceptance of the *done* signal occurs only when the value of monitor counter *monitor_state* reaches 40. If the *done* signal is manipulated to falsely indicate completion ahead of schedule or is intentionally delayed, suggesting tampering possibly due to an HT, the GIFT-R1 monitor triggers a security alert. According to this automaton, the monitor transitions to the RESPONSE state only if the *done* signal is received precisely 40 cc after the *start* signal is asserted.

7.3.2 Monitor Formal Analysis

Table 7.6 provides SVAs for the GIFT-R0 monitor. Assertions (1) to (9) validate the monitor's transitions according to the edge specifications illustrated in Figure 7.8. Note that the monitor module initiates upon the assertion of the ap_start signal. Each assertion verifies

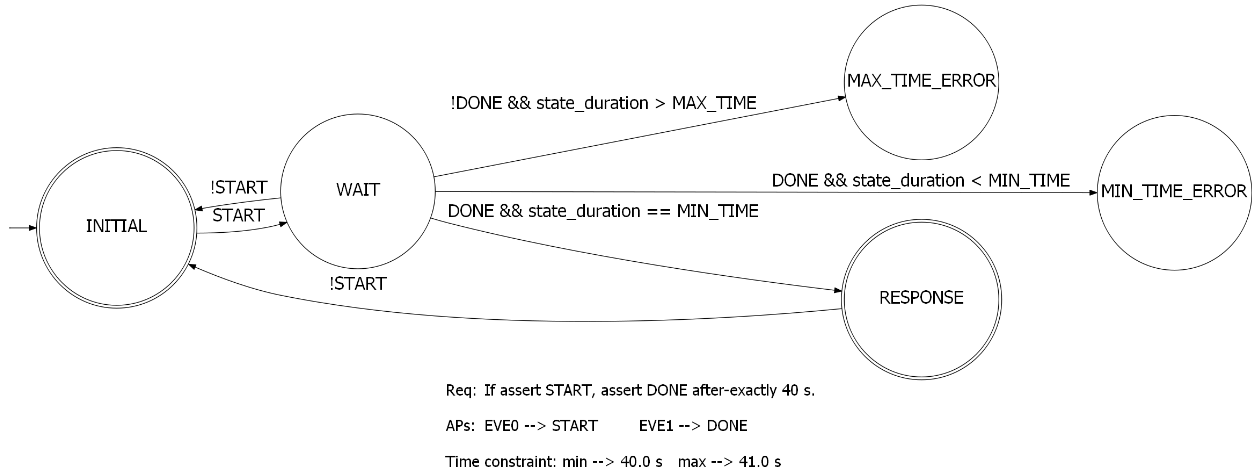

Figure 7.9: GIFT-R1 monitor FSM generated by TS2C.

that the monitor switches states correctly based on its current state and the conditions of its inputs. For example, assertion (1) ensures that the monitor only activates when the *start* signal is asserted. If this trigger condition in not met, the monitor remains in initial state 2. Upon the assertion of *start*, assertion (2) confirms that the monitor transitions to state 0 and waits for *done* to be asserted. Assertion (3) guarantees that the monitor correctly transitions to state 1 after receiving *done*, indicating that the expected response has been received.

Assertions (4) to (7) focus on the transitions from state 0, depending on the status of *start* and *done*. For example, assertion (6) ensures that if *start* becomes false before *done* is asserted, the monitor detects this error condition and transitions to error state 3. Assertion (8) verifies that in state 1, if *start* remains true, the monitor continues to stay in state 1. On the other hand, assertion (9) validates that the monitor returns to its initial state when *start* is deasserted, marking a reset condition for the monitor's ongoing monitoring process.

Assertion (10) guarantees that after error detection and asserting the *error* signal, the subsequent state of the monitor must be state 3. In this context, *error* denotes the current value of the `monitor_state.error` signal, while *state* corresponds to the monitor's registered status. The operator `##1` in the assertion indicates a one-cycle delay between the evaluation

Table 7.6: Generated SVAs for the GIFT-R0 monitor.

<table>
<tr><th>ID</th><th>SystemVerilog Assertion (SVA)</th></tr>
<tr><td>(1)</td><td>(ap_start & (state == 2) & (!start) |=> (state == 2))</td></tr>
<tr><td>(2)</td><td>(ap_start & (state == 2) & (!done & start) |=> (state == 0))</td></tr>
<tr><td>(3)</td><td>(ap_start & (state == 2) & (done & start) |=> (state == 1))</td></tr>
<tr><td>(4)</td><td>(ap_start & (state == 0) & (!done & start) |=> (state == 0))</td></tr>
<tr><td>(5)</td><td>(ap_start & (state == 0) & (done & !start) |=> (state == 2))</td></tr>
<tr><td>(6)</td><td>(ap_start & (state == 0) & (!done & !start) |=> (state == 3))</td></tr>
<tr><td>(7)</td><td>(ap_start & (state == 0) & (done & start) |=> (state == 1))</td></tr>
<tr><td>(8)</td><td>(ap_start & (state == 1) & (start) |=> (state == 1))</td></tr>
<tr><td>(9)</td><td>(ap_start & (state == 1) & (! start) |=> (state == 2))</td></tr>
<tr><td>(10)</td><td>(ap_start & (error == 1) |=> ## 1 (state == 3))</td></tr>
<tr><td>(11)</td><td>(ap_start & (state == 3) |=> (error == 1))</td></tr>
<tr><td>(12)</td><td>(ap_start & state_duration >= 0)</td></tr>
</table>

of conditions. This means that the monitor's current state and the error signal are evaluated at different clock cycles, distinguishing between the registered value of the monitor's state and the combinational evaluation of the error signal. This differentiation is crucial for accurately interpreting the monitor's behavior and ensuring that error detection triggers the appropriate state transition.

Assertion (11) ensures that once the monitor detects an error and transitions to state 3, the error condition persists until the monitor is reset. As described in Section 4.2, the Output_Handler module is responsible for resetting the monitor upon receiving the final verdict and new input signals. Thus, the error flag should continue to indicate the presence of an error until the monitor is reset. Last, assertion (12) guarantees that the value of monitor_state.state_duration is always non-negative, indicating that the monitor counter accurately measures time. While GIFT-R0 lacks explicit timing constraints, making this counter seem optional, it can offer helpful insights for debugging and testing purposes by tracking the time spent in each state.

Table 7.7: Generated SVAs for the GIFT-R1 monitor.

ID	SystemVerilog Assertion (SVA)
(1)	(ap_start & (state == INITIAL) & (!start) |=> (state == INITIAL))
(2)	(ap_start & (state == INITIAL) & (start) |=> (state == WAIT))
(3)	(ap_start & (state == WAIT) & (!start) |=> (state == INITIAL))
(4)	(ap_start & (state == WAIT) & (done) & (state_duration == 40)|=> (state == RESPONSE))
(5)	(ap_start & (state == WAIT) & (done) & (state_duration < 40)|=> (state == MIN_TIME_ERROR))
(6)	(ap_start & (state == WAIT) & (!done) & (state_duration > 40)|=> (state == MAX_TIME_ERROR))
(7)	(ap_start & (state == RESPONSE) & (!start) |=> (state == INITIAL))
(8)	(ap_start & (state == RESPONSE) & (start) |=> (state == RESPONSE))
(9)	(ap_start & (error == 1) |=> ## 1 (state == MIN_TIME_ERROR | state == MAX_TIME_ERROR))
(10)	(ap_start & (state == MIN_TIME_ERROR) |=> (error == 1))
(11)	(ap_start & (state == MAX_TIME_ERROR) |=> (error == 1))
(12)	(ap_start & (pass == 1 |=> ## 1 (state == RESPONSE))
(13)	(ap_start & (state == RESPONSE) |=> (pass == 1))
(14)	(ap_start & state_duration >= 0)

The SVAs for the GIFT-R1 monitor are described in Table 7.7. Similar to SVAs (1) to (9) for GIFT-R0, assertions (1) to (8) validate that the execution of conditions and transitions between states are consistent with the TA depicted in Figure 7.9. For example, assertions (4), (5) and (6) address the behavior of the monitor while it is in the WAIT state. They verify that whether the monitor correctly identifies the expected response, which is the assertion of *done*, within the specified time frame. According to the specification, the *done* signal should be asserted precisely 40 cc after the *start* signal is asserted. If this condition is met, the monitor transitions to the RESPONSE state, as stated in assertion 4. However, if the response occurs before the 40th clock cycle, indicating an early response, the monitor transitions to the MIN_TIME_ERROR state (assertion 5). Conversely, if *done* has not been asserted by the 40th clock cycle, signifying a delayed response, the monitor moves to the MAX_TIME_ERROR state (assertion 6).

```
** Results:
[gift_func.property.1] always gift_func.ap_start && gift_func.state == 2'b10 && !gift_func
.start |=> gift_func.state == 2'b10: SUCCESS
[gift_func.property.2] always (gift_func.ap_start && gift_func.state == 2'b10) & !gift_fun
c.done & gift_func.start |=> gift_func.state == 2'b0: SUCCESS
[gift_func.property.3] always (gift_func.ap_start && gift_func.state == 2'b10) & gift_func
.done & gift_func.start |=> gift_func.state == 2'b1: SUCCESS
[gift_func.property.4] always (gift_func.ap_start && gift_func.state == 2'b0) & !gift_func
.done & gift_func.start |=> gift_func.state == 2'b0: SUCCESS
[gift_func.property.5] always (gift_func.ap_start && gift_func.state == 2'b0) & gift_func.
done & !gift_func.start |=> gift_func.state == 2'b10: SUCCESS
[gift_func.property.6] always gift_func.ap_start && gift_func.state == 2'b0 && !gift_func.
done && !gift_func.start |=> gift_func.state == 2'b11: SUCCESS
[gift_func.property.7] always (gift_func.ap_start && gift_func.state == 2'b0) & gift_func.
done & gift_func.start |=> gift_func.state == 2'b1: SUCCESS
[gift_func.property.8] always (gift_func.ap_start && gift_func.state == 2'b1) & gift_func.
start |=> gift_func.state == 2'b1: SUCCESS
[gift_func.property.9] always gift_func.ap_start && gift_func.state == 2'b1 && !gift_func.
start |=> gift_func.state == 2'b10: SUCCESS
[gift_func.property.10] always gift_func.error == 1 |=> ##1 gift_func.state == 3: SUCCESS
[gift_func.property.11] always gift_func.state == 3 |=> gift_func.error == 1: SUCCESS
[gift_func.property.12] always gift_func.state_duration >= 0: SUCCESS
```

Figure 7.10: EBMC verification results for GIFT-R0.

Similar to assertion (10) for GIFT-R0, assertion (9) ensures that any detected violation causes the monitor to transition into an error state in the subsequent clock cycle. Following this, assertions (10) and (11) verify that the error signal remains asserted as long as the monitor resides in any of the error states, indicating the persistence of detected issue until the monitor is reset by the Output_Handler module. Equivalently, assertions (12) and (13) guarantee the alignment between the RESPONSE state and the monitor_state.pass flag, ensuring consistent behavior in detecting successful responses. Assertion (14) validates the accuracy of the monitor_state.state_duration range. This is paramount to precisely determine whether the monitor identifies satisfaction or violation of the requirement based on the time spent in each state. The outcomes shown in Figure 7.10 and Figure 7.11 demonstrate the correctness of the SVAs for both GIFT-R0 and GIFT-R1, respectively.

7.3.3 Monitor Simulation

Simulating the monitors alongside the main system provides valuable insights into the monitors' behaviors under various conditions and inputs. This integrated simulation approach is essential for verifying how effectively the monitors detect potential faults or attacks in

```
** Results:
[gift_time.property.1] always gift_time.ap_start && gift_time.state == 0 && !gift_time.sta
rt |=> gift_time.state == 0: SUCCESS
[gift_time.property.2] always (gift_time.ap_start && gift_time.state == 0) & gift_time.sta
rt |=> gift_time.state == 1: SUCCESS
[gift_time.property.3] always gift_time.ap_start && gift_time.state == 1 && !gift_time.sta
rt |=> gift_time.state == 0: SUCCESS
[gift_time.property.4] always (gift_time.ap_start && gift_time.state == 1) & gift_time.don
e & gift_time.state_duration == 40 |=> gift_time.state == 2: SUCCESS
[gift_time.property.5] always (gift_time.ap_start && gift_time.state == 1) & gift_time.don
e & gift_time.state_duration < 40 |=> gift_time.state == 3: SUCCESS
[gift_time.property.6] always gift_time.ap_start && gift_time.state == 1 && !gift_time.don
e && gift_time.state_duration > 40 |=> gift_time.state == 4: SUCCESS
[gift_time.property.7] always gift_time.ap_start && gift_time.state == 2 && !gift_time.sta
rt |=> gift_time.state == 0: SUCCESS
[gift_time.property.8] always (gift_time.ap_start && gift_time.state == 2) & gift_time.sta
rt |=> gift_time.state == 2: SUCCESS
[gift_time.property.9] always gift_time.error == 1 |=> ##1 gift_time.state == 3 || gift_ti
me.state == 4: SUCCESS
[gift_time.property.10] always gift_time.state == 3 |=> gift_time.error == 1: SUCCESS
[gift_time.property.11] always gift_time.state == 4 |=> gift_time.error == 1: SUCCESS
[gift_time.property.12] always gift_time.pass == 1 |=> ##1 gift_time.state == 2: SUCCESS
[gift_time.property.13] always gift_time.ap_start && gift_time.state == 2 |=> gift_time.pa
ss == 1: SUCCESS
[gift_time.property.14] always gift_time.state_duration >= 0: SUCCESS
```

Figure 7.11: EBMC verification results for GIFT-R1.

practical scenarios. The two monitors and the SUS are implemented using the Vivado implementation tool on a Xilinx Artix-7 FPGA. Under normal operations, GIFT produces the correct ciphertext 40 cc after the *start* signal is asserted, following by the assertion of the *done* signal. Figure 7.12 illustrates this scenario, displaying the Monitor_State values for both monitors, which confirms the successful operation by setting the monitor_state.pass flags. Now consider Figure 7.13, where *start* is abruptly deasserted, suggesting the presence of an HT intentionally inserted to disrupt the encryption operation or a system malfunction. As shown in Figure 7.9, the GIFT-R1 monitor transitions from the WAIT state to the INITIAL state since the trigger event is no longer active. However, GIFT-R0 successfully detects this violation as the deassertion occurs before receiving the *done* signal, prompting a transition to state 3 and setting the monitor_state.error variable (denoted as $r0_error$).

Figure 7.14 depicts a scenario where the *done* signal is received prematurely, resulting in an incorrect ciphertext at that point. In this situation, the GIFT-R0 monitor does not detect any violation and consequently sets the monitor_state.pass flag and moves to state 2. Conversely, the GIFT-R1 monitor registers an monitor_state.error and transitions to the

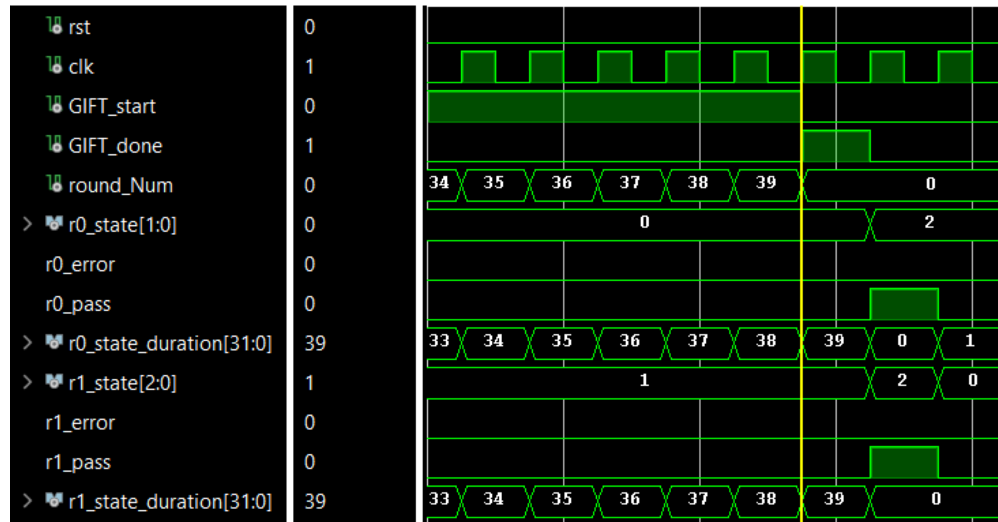

Figure 7.12: Simulation results of GIFT and its monitors when there is no fault/attack.

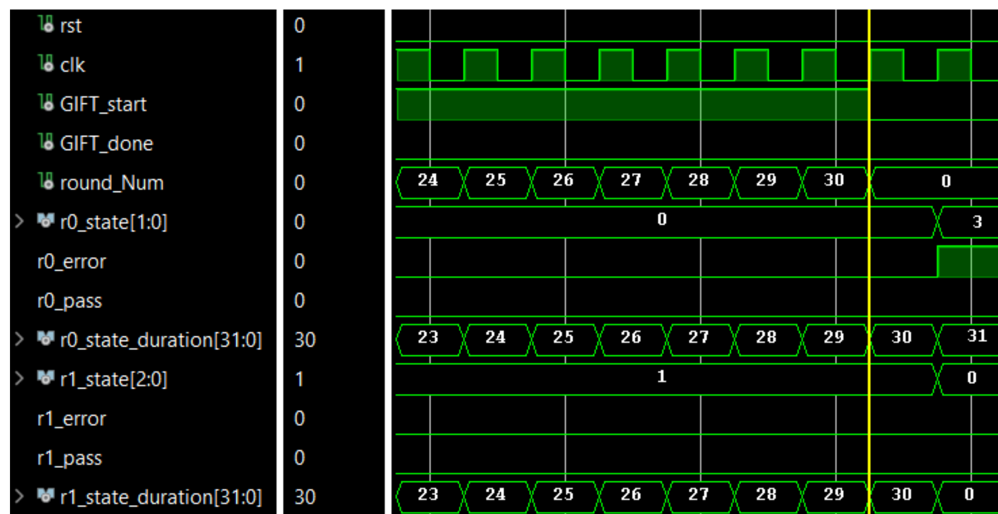

Figure 7.13: Simulation results of a fault or HT attack on the *start* signal, triggering the GIFT-R0 monitor.

MIN_TIME_ERROR state as its monitor_state.state_duration value is less than 40. In a scenario with a delayed response, the monitor would detect the issue as well. These simulation results underscore the importance of employing monitors that complement each other to address a variety of situations.

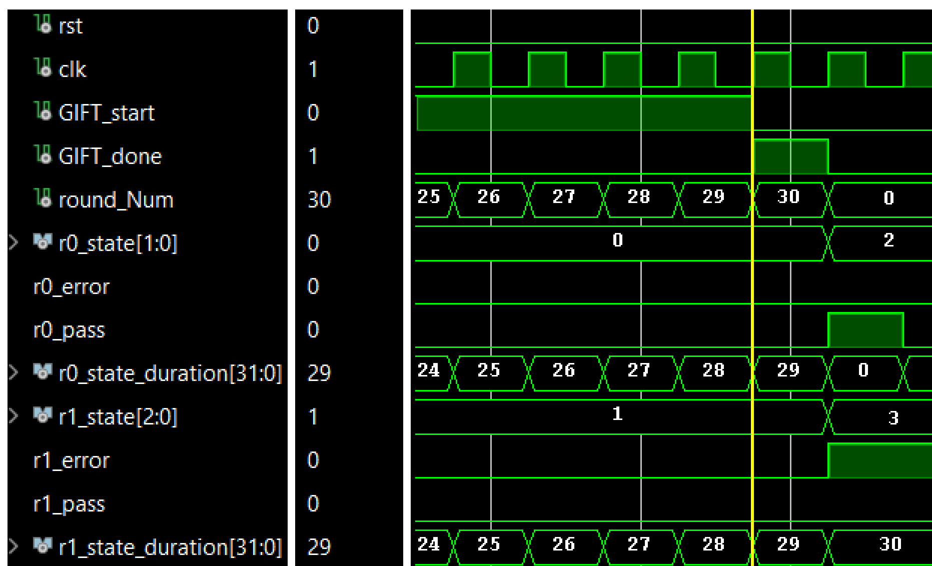

Figure 7.14: Simulation results of a fault or HT attack on the *done* signal, triggering the GIFT-R1 monitor.

Table 7.8: Implementation results for GIFT-COFB and its monitors on Artix-7.

Module / Parameter	SUS	SUS & GIFT-R0	SUS & GIFT-R1	SUS & both monitors
LUTs	1656	1668	1689	1714
Registers (FFs)	2037	2041	2085	2089
Power (mW)	112	115	122	122

7.3.4 Monitor Resource Utilization

To assess the overhead of incorporating monitors, four implementations are analyzed: the SUS alone, the SUS with the GIFT-R0 monitor, the SUS with the GIFT-R0 monitor, and all three modules combined. All implementations adhere to common conditions and a clock constraint of 250 MHz adapted from [139]. Table 7.8 shows the utilization of lookup tables (LUTs), flip flops (FFs), and power for each implementation. Considering the SUS implementation as baseline, the results indicate that including the GIFT-R0 monitor results in negligible overhead as depicted in Figure 7.15. The GIFT-R1 monitor demonstrates a slight increase in overhead, with a 3.5% increase in LUTs and an 8.9% increase in power consump-

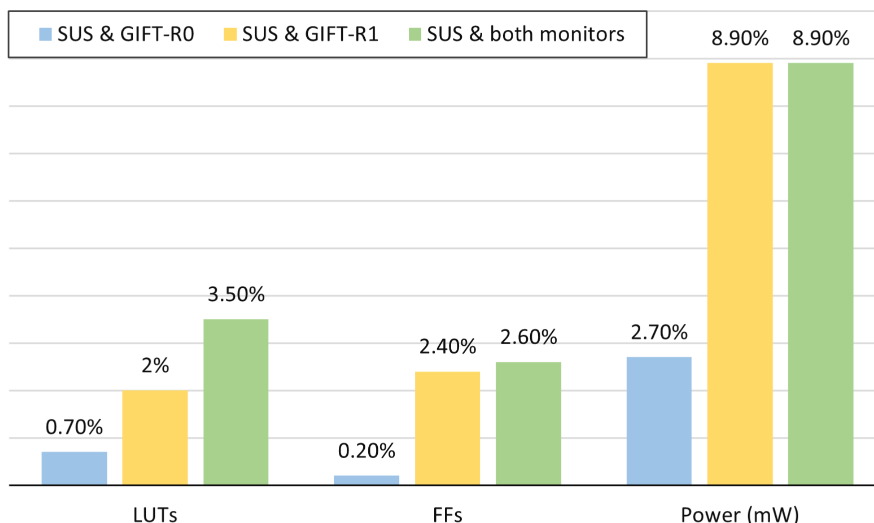

Figure 7.15: Monitor overhead for GIFT-COFB in terms of LUTs, FFs, and power.

tion. This discrepancy can be attributed to the additional comparator modules required by GIFT-R1 for timing constraint checks. As illustrated in Figure 7.9, the GIFT-R1 constraint involves both upper and lower limits, also requiring extra conditions and states compared to GIFT-R0. The GIFT-COFB implementation utilizes only 2.5% of available LUTs and 1.6% of available FFs on Artix-7. When both monitors are added, the utilization increases slightly to 2.7% and 1.65%, respectively. These findings underscore the minimal overhead of the monitors, which is crucial for the efficiency of this LWC algorithm.

7.4 Case 3: Adaptive Cruise Control System

As illustrated in Figure 7.16, an adaptive cruise control (ACC) system operates in two distinct modes: (i) speed control, which ensures the vehicle maintains a consistent speed set by the driver, and (ii) spacing control, which adjusts the velocity to keep a safe distance (D_{safe}) between the autonomous (ego) car and the lead vehicle. The ACC system autonomously determines the appropriate mode based on real-time radar data, specifically the relative dis-

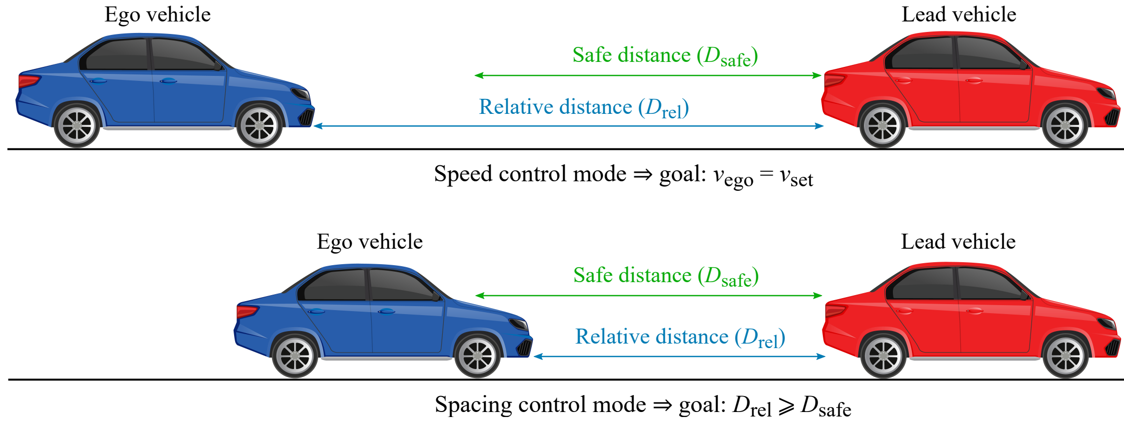

Figure 7.16: ACC modes of operation.

tance (D_{rel}) and relative velocity (v_{rel}) between the vehicles. A critical parameter in ACC operation is the headway time (τ_{headway}), which represents the constant time gap maintained between the ego and lead vehicles. This parameter confirms that the ego vehicle has sufficient time to react and decelerate if the lead vehicle suddenly stops. To calculate D_{safe}, it is assumed that the lead vehicle has stopped while the ego vehicle maintains its speed ($v_{\text{rel}} = -v_{\text{ego}}$). Considering a safety margin denoted as D_{offset}, D_{safe} is calculated as follows:

$$D_{\text{safe}} = v_{\text{ego}} \times \tau_{\text{headway}} + D_{\text{offset}} \tag{7.1}$$

7.4.1 Monitor Synthesis

The application of monitors in different development stages is demonstrated by two requirements of the ACC system outlined in Table 7.9. The first requirement, ACC_R0, is a functional requirement, which mandates that D_{rel} must never fall below D_{safe} throughout the system's operation, irrespective of its mode. Figure 7.17 depicts the generated BA, which transitions to non-accepting state 1 upon a violation and remains there until Output_Handler resets the monitor. This requirement is a fundamental safety property (discussed in

Table 7.9: ACC system requirements.

ACC_R0	Ego vehicle should always keep a safe distance from the lead vehicle.
FRETish	ACC shall always satisfy (Drel ≥ Dsafe)
LTL	(LAST V (Drel ≥ Dsafe))
ACC_R1	In speed control, ACC should reach set velocity between 1 and 3 seconds.
TIMESPEC	If assert (Drel ≥ Dsafe), assert (Vego ≥ Vset_n & Vego ≤ Vset_p) between 1 s and 3 s.

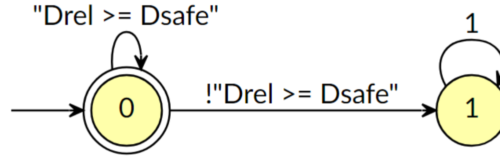

Figure 7.17: BA built by the Spot tool for ACC_R0.

Section 2.2.5), ensuring the ego vehicle always maintains a safe distance from the lead vehicle. Checking this requirement is crucial as it directly impacts the ACC's safety, preventing potentially hazardous situations such as collisions or unsafe driving conditions. The synthesis flow employed to generate the ACC_R0 monitor is discussed in Chapter 5.

The second requirement, ACC_R1, guarantees that the ego vehicle's velocity adjusts to the set target speed (v_{set}) within a specified time frame when operating in speed control mode. This ensures a smooth acceleration process, neither too slow nor too fast, for passenger's comfort. To validate compliance with this requirement, a velocity threshold (v_{th}) is established to verify that v_{ego} remains within an acceptable range, shown as follows:

$$v_{\text{set_n}} \leq v_{\text{ego}} \leq v_{\text{set_p}}, \tag{7.2}$$

$$\text{where} \quad v_{\text{set_n}} = v_{\text{set}} - v_{\text{th}} \quad \text{and} \quad v_{\text{set_p}} = v_{\text{set}} + v_{\text{th}}$$

The ACC_R1 requirement is formalized using a TIMESPEC causality template. Figure 7.18 displays a graphical version of the TA synthesized by TA-Gen. The monitor triggers when the ACC system transitions to speed control mode (the relative distance is safe) and enters

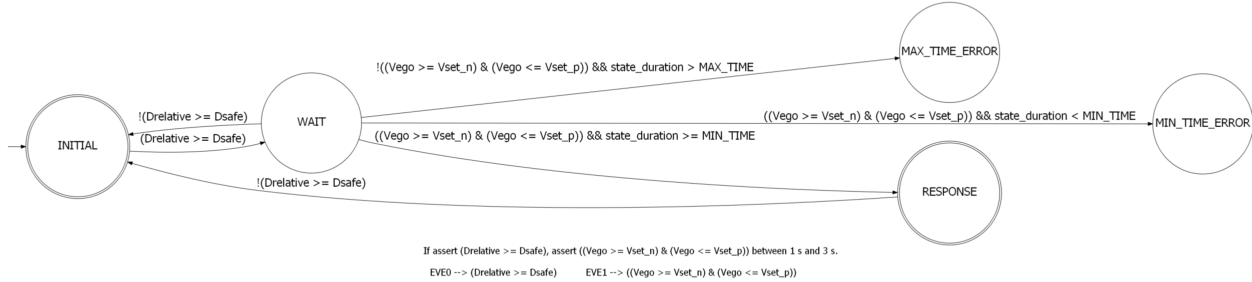

Figure 7.18: TA built by TA-Gen for ACC_R1.

the WAIT state. In this state, the monitor assesses variables such as v_{ego} and the elapsed time to determine its final decision. The ACC_R1 monitor is synthesized using the time-based synthesis flow described in Chapter 6.

Considering the ACC_R0 and ACC_R1 requirements presented in Table 7.9, the events ACC_EVE0 and ACC_EVE1 are defined according to (7.3). As discussed in Section 4.2, these events are managed by the Input_Handler module, where D_{safe} is calculated as specified in (7.1). Since ACC_EVE0 is utilized in both monitors, the Input_Handler component can be shared, reducing resource consumption.

$$\text{ACC_EVE0} : D_{\text{rel}} \geq D_{\text{safe}}, \qquad \text{ACC_EVE1} : v_{\text{ego}} \geq v_{\text{set_n}} \;\&\&\; v_{\text{ego}} \leq v_{\text{set_p}} \tag{7.3}$$

7.4.2 Monitor Formal Analysis

Similar to formal analysis section of previous examples, the assertions illustrated in Figure 7.19 validate different aspects of the ACC_R0 monitor's behavior and implementation. For example, assertion (1) checks that the monitor's state is either STATE0 or STATE1, ensuring that the monitor operates within its defined states. As explained in Section 5.3.4, assertion (2) confirms that the state duration counter is non-negative, which is important for tracking the amount of time the monitor spends in a particular state without errors

```
(1) assert(monitor_state_func->state == STATE0 || monitor_state_func->state == STATE1); // state validity
(2) assert(monitor_state_func->state_duration >= 0);                    // state_duration validity
(3) assert(!(monitor_state_func->error && monitor_state_func->pass));       // error and pass mutual exclusivity
(4) assert(monitor_state_func->error == (monitor_state_func->state == STATE1)); // Error condition consistency
(5) assert(monitor_state_func->pass == false);                      // pass validity

(6) if (monitor_state_func->state == STATE0) {                      // Transitions validity
        assert(monitor_inputs->EVE0 == (monitor_state_func->state == STATE0 && !monitor_state_func->error));
        assert((!monitor_inputs->EVE0) == (monitor_state_func->state == STATE1 && monitor_state_func->error));
    }
```

Figure 7.19: Assertions provided by Code-Gen for ACC_R0.

such as invalid values. Assertion (3) guarantees that the monitor's outcomes represented by monitor_state→pass and monitor_state→error are never simultaneously true throughout the execution. This is consistent with the principle that a requirement can either be satisfied or violated, but not both concurrently.

Assertion (4) verifies an error is only noted when the monitor is in a specific non-accepting state (STATE1), confirming that errors are recognized and reported accurately according to the expected behavior depicted in Figure 7.17. It is noteworthy that monitor_state→pass should never become true as no finite trace can validate ACC_R0, as discussed in Section 2.2.5. Hence, assertion (5) ensures that the monitor's verdict is inconclusive, meaning both pass and error flags are false, unless a violation occurs. Finally, assertion (6) checks for valid transitions based on the input event ACC_EVE0 within the specific case of STATE0, confirming the monitor reacts appropriately to input changes. These assertions provide assurance that the monitor's implementation behaves as intended and adheres to the BA's specifications. Figure 7.20 illustrates the successful outcome of the ESBMC model checker, verifying that these assertions hold under all possible execution scenarios.

As discussed in Section 6.3.4, TS2C generates ACSL contracts pertaining to the generated TA, validating the monitor's behavior comply with its intended function. Figure 7.21 illustrates the specifications generated for the ACC_R1 monitor. Contracts (1) and (2) ensure that the pointers monitor_state and monitor_inputs are properly initialized before respective

```
ESBMC version 7.4.0 64-bit x86_64 windows
Target: 64-bit little-endian x86_64-unknown-windows with esbmclibc
Parsing acc_monitor_func.c
Converting
Generating GOTO Program
GOTO program creation time: 0.396s
GOTO program processing time: 0.001s
Starting Bounded Model Checking
Symex completed in: 0.001s (40 assignments)
Slicing time: 0.000s (removed 12 assignments)
Generated 7 VCC(s), 7 remaining after simplification (28 assignments)
No solver specified; defaulting to z3
Encoding remaining VCC(s) using bit-vector/floating-point arithmetic
Encoding to solver time: 0.000s
Solving with solver Z3 v4.12.2
Runtime decision procedure: 0.001s
BMC program time: 0.015s

VERIFICATION SUCCESSFUL
```

Figure 7.20: ESBMC verification results for ACC_R0.

write and read operations. This is critical for preventing errors related to null or uninitialized pointers, thus enhancing memory safety. Note that the ESBMC model checker comes with a set of default properties already defined for scanning common memory issues and deadlocks, so assertions for functional requirements typically do not require these checks.

Precondition (3) and post-condition (4) define a valid range for the duration counter, which is essential for precise monitoring and timely identification of responses. Precondition (3) sets clear expectations on the function's input, indicating that this counter is initially assumed to be in a valid state (non-negative) for successful execution of Frama-C. Contract (5) asserts that all possible monitor states are covered by the `switch-case` logic, confirming that the implementation adheres closely to the TA specifications.

Contract (6) ensures that the monitor_state→error flag accurately represents the presence of an error condition within the system. Conversely, contract (7) verifies that the monitor_state→pass flag is set correctly when all conditions are met and the monitor transitions to state RESPONSE. These two contracts validate the functionality of the monitor's feedback mechanisms, guaranteeing that errors are truly reported and successful operations are acknowledged. Contract (8) emphasizes the mutual exclusivity of these verdicts, increasing the

```
/*@
(1)  requires \valid(monitor_state) && \valid_read(monitor_inputs);
(2)  requires \separated(monitor_state, monitor_inputs);
(3)  requires 0 <= monitor_state->state_duration;
(4)  ensures 0  <= monitor_state->state_duration;
(5)  ensures (monitor_state->state\in{INITIAL, WAIT, RESPONSE,
             MAX_TIME_ERROR, MIN_TIME_ERROR});
(6)  ensures((monitor_state->error == 1)==>
             (monitor_state->state\in{MAX_TIME_ERROR, MIN_TIME_ERROR}));
(7)  ensures((monitor_state->pass == 1)==>
             (monitor_state->state == RESPONSE));
(8)  ensures (!(monitor_state->error && monitor_state->pass));
(9)  ensures unchanged_inputs: *monitor_inputs == \old(*monitor_inputs);
(10) behavior not_started:
         assumes monitor_state->state \in {INITIAL};
         assumes monitor_inputs->EVE0 == 0;
         ensures monitor_state->state == INITIAL;
(11) behavior begin:
         assumes monitor_state->state \in {INITIAL};
         assumes monitor_inputs->EVE0 == 1;
         ensures monitor_state->state == WAIT;
(12) behavior satisfaction_or_min_time_violation:
         assumes monitor_state->state \in {WAIT};
         assumes monitor_inputs->EVE1 == 1;
         ensures ((monitor_state->state_duration >= MIN_TIME ==>
                 monitor_state->state == RESPONSE) ||
             (monitor_state->state_duration < MIN_TIME ==>
                 monitor_state->state == MIN_TIME_ERROR));
(13) behavior wait_or_max_time_violation:
         assumes monitor_state->state \in {WAIT};
         assumes monitor_inputs->EVE1 == 0;
         ensures ((monitor_state->state_duration <= MAX_TIME ==>
                 monitor_state->state == WAIT) ||
             (monitor_state->state_duration > MAX_TIME ==>
                 monitor_state->state == MAX_TIME_ERROR));
(14) behavior reset_trigger:
         assumes monitor_state->state \in {RESPONSE};
         assumes monitor_inputs->EVE0 == 1;
         ensures monitor_state->state == RESPONSE;
(15) behavior reintitialize:
         assumes monitor_state->state \in {RESPONSE};
         assumes monitor_inputs->EVE0 == 0;
         ensures monitor_state->state == INITIAL;
(16) behavior stop:
         assumes monitor_state->state \in{MIN_TIME_ERROR,MAX_TIME_ERROR};
         ensures monitor_state->error == 1;
(17) disjoint behaviors;
*/
```

Figure 7.21: ACSL statements provided by Code-Gen for ACC_R1.

reliability of the monitoring outcomes by preventing contradictory conditions. Contract (9) compares the pre and post-execution states of monitor_inputs, ensuring that the monitoring

```
[kernel] Parsing acc_monitor_time.c (with preprocessing)
[kernel] Parsing acc_monitor_time_wrapper.c (with preprocessing)
[rte:annot] annotating function acc_monitor_time
[rte:annot] annotating function acc_monitor_time_error_handler
[rte:annot] annotating function acc_monitor_time_input_handler
[rte:annot] annotating function wrapper
[wp] 137 goals scheduled
[wp] Proved goals:  143 / 143
  Terminating:       3
  Unreachable:       3
  Qed:              81
  Alt-Ergo 2.4.2:   56 (7ms-144ms-1.8s)
```

Figure 7.22: Frama-C verification results for ACC_R1.

logic only reads APs without modification, as they are solely updated by the Input_Handler module (discussed in Section 4.2).

Contracts (10) through (16) validate the conditions within each `case` statement of the monitor's `switch` structure against transitions defined by the TA. For example, contract (12) specifically focuses on the scenario where the monitor is in the WAIT state and detects event ACC_EVE1. If the monitor_state→state_duration, implying the duration for which the monitor has remained in the current state, falls between MIN_TIME (defined as 1 second in the header file) and MAX_TIME (set as 3 seconds in the header file), the monitor should transition to the RESPONSE state as the event occurred within the acceptable time range. However, if the event is detected before the counter reaches MIN_TIME, indicating a premature response, the monitor then transitions to the MIN_TIME_ERROR state.

Contract (13) addresses a different scenario where no response is received within the expected time frame. Here, the monitor continues to stay in the WAIT state until the time limit expires, after which it moves to the MAX_TIME_ERROR state. The inclusion of "assumes" statements is crucial as they guide the Frama-C tool in validating the behavior under specified conditions, thereby reducing the risk of encountering a timeout error. Lastly, contract (17) verifies that the behaviors within each case are clearly and distinctly defined, preventing any potential overlap in the specification. Figure 7.22 shows the Frama-C results, validating

Table 7.10: ACC model parameters default values.

Parameter	Description	Value
τ_{headway}	Headway time	1.5 s
D_{offset}	Offset distance	15 m
v_{set}	Set velocity	21.5 m/s
v_{th}	Velocity threshold	1 m/s

that the monitor correctly sequences through the states.

Comparing the generated SVAs, ESBMC assertions and ACSL specifications in the context of verifying a monitor implementation, it becomes clear that each of these languages, while different in syntax and semantics, aims to rigorously check that the monitor's behavior conform to the intended logical framework set by the automaton. However, as explained earlier, simulation is for confirming that the ACC requirements are accurately expressed in FRETish and TIMESPEC languages.

7.4.3 Monitor Simulation

The synthesized monitors are integrated into an ACC system modeled in the Simulink graphical environment to verify that a safe following distance is maintained and to ensure that the ACC system accelerates correctly. Table 7.10 presents the default parameter values used to determine the events ACC_EVE0 and ACC_EVE1 (defined in (7.3)) during the simulation, which are obtained from the Simulink model [156].

Simulation results for the ACC_R0 monitor are illustrated in Figure 7.23. Initially, the ego vehicle maintains a safe following distance (ACC_EVE0 = 1). Around $t = 12.5$ s, another vehicle cuts into its lane, causing D_{rel} to decrease. At $t = 16.6$ s, D_{rel} falls below D_{safe} (ACC_EVE0 = 0), triggering an error condition, which is promptly detected by the monitor. The *error* flag remains true until D_{rel} returns to the safe zone. In this specific

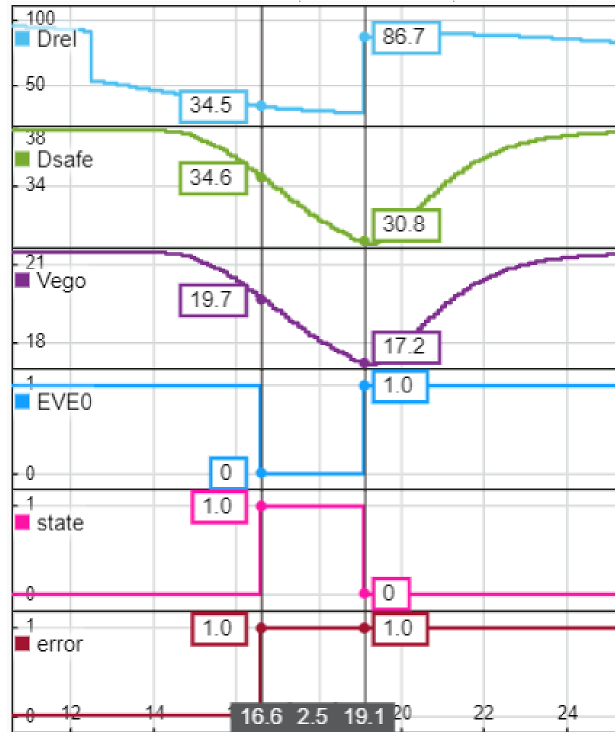

Figure 7.23: Simulation results for ACC_R0 monitor.

scenario, the monitor can be employed as a safety measure to activate an alert system, ensuring the driver is informed if the ACC system fails to respond as expected.

Figure 7.24 depicts the simulation outcomes for the ACC_R1 monitor. Given the simulation time step of 100 ms, the monitor_period variable (explained in Section 6.3.3) is set to 0.1, while MIN_TIME and MAX_TIME are automatically determined as 10 and 30, corresponding to the requirement's time limits of 1 and 3 seconds, respectively. At t = 19.1 s, the lead vehicle from the preceding scenario changes lanes, resulting in a noticeable increase in D_{rel} (ACC_EVE0 = 1). This prompts the monitor to start timing and verify whether v_{ego} falls within the specified range in the designated time frame. By t = 22.1 s, the velocity aligns with the target range (ACC_EVE1 = 1), indicating a successful response just before the maximum time deadline expires. Consequently, the monitor transitions to the RESPONSE state with the *pass* flag set to true. It remains in this state as long as the relative distance

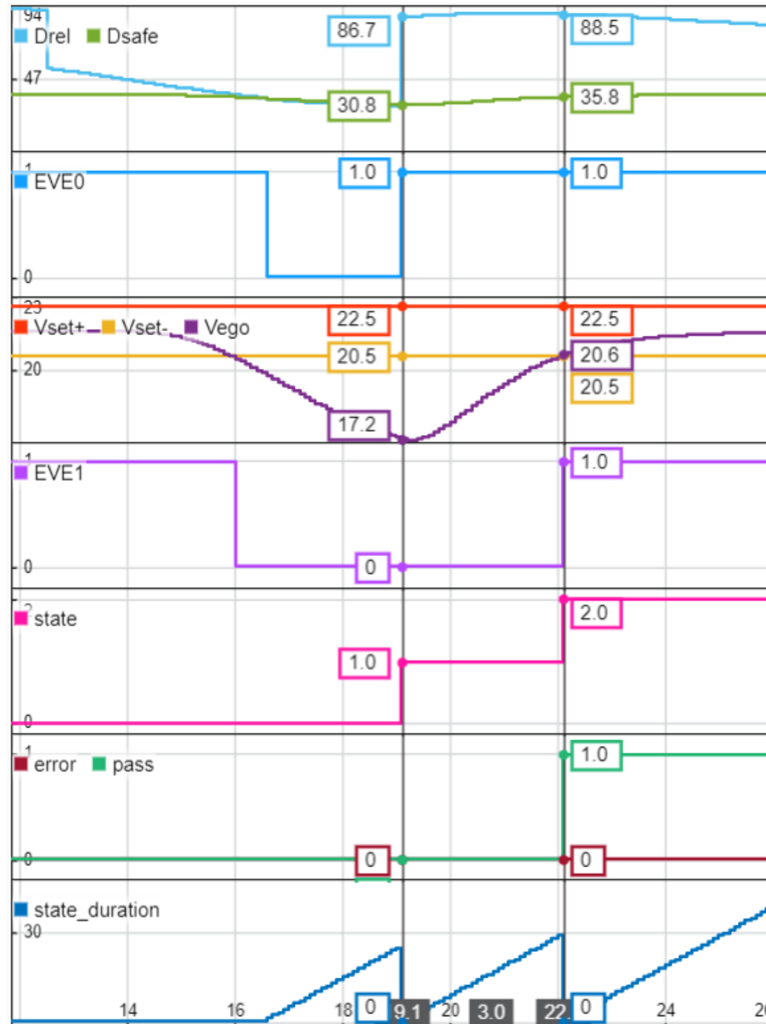

Figure 7.24: Simulation results for ACC_R1 monitor.

stays within the safe margin. Monitoring is particularly helpful during the debugging phase, providing a means to confirm the ACC system's ability to safely and promptly resume acceleration after an obstruction clears.

7.4.4 Comparison of Copilot and GROOT Monitors

As discussed in Section 2.4.1, Copilot is a stream-based RV framework focusing on LTL and MTL formulas. In [127], Perez et al. present a toolchain that generates C-based monitors

Table 7.11: Breaking the ACC_R1 requirement into two FRETish statements.

ACC_R1_min	If (Drel ≥ Dsafe) ACC shall after 1 sec satisfy (Vego ≥ Vset_n & Vego ≤ Vset_p).
MTL formula	((LAST V (((! ((Drel >= Dsafe))) & ((! LAST) & (X ((Drel >= Dsafe))))) → (X (((G[<=] (! ((Vego >= Vset_n & Vego <= Vset_p)))) | (LAST V (! ((Vego >= Vset_n & Vego <= Vset_p))))) & ((F[<=1+1] ((Vego >= Vset_n & Vego <= Vset_p))) | (F[<1+1] LAST)))))) & (((Drel >= Dsafe)) → (((G[<=1] (! ((Vego >= Vset_n & Vego <= Vset_p)))) | (LAST V (! ((Vego >= Vset_n & Vego <= Vset_p))))) & ((F[<=1+1] ((Vego >= Vset_n & Vego <= Vset_p))) | (F[<1+1] LAST)))))
ACC_R1_max	If (Drel ≥ Dsafe) ACC shall within 3 sec satisfy (Vego ≥ Vset_n & Vego ≤ Vset_p).
MTL formula	((LAST V (((! ((Drel >= Dsafe))) & ((! LAST) & (X ((Drel >= Dsafe))))) → (X ((F[<=3] ((Vego >= Vset_n & Vego <= Vset_p))) | (F[<3] LAST))))) & (((Drel >= Dsafe)) → ((F[<=] ((Vego >= Vset_n & Vego <= Vset_p))) | (F[<3] LAST))))

from structured English statements using FRET and Copilot. This toolchain is employed in the current work to create monitors for requirements ACC_R0 and ACC_R1. Section 6.3 states FRET's limitations in handling certain timing constraints, leading to the necessity of capturing ACC_R1 with two complementary FRETish statements, as shown in Table 7.11. Comparing these two MTL formulas, ACC_R1_min appears more complex due to the variety of temporal and logical operators to specify a broader range of conditions that the system must meet. This includes conditions that must be true in the next state ($\mathtt{X}$), continuously over a specified short duration ($\mathtt{G}[\leq 1]$), and by certain points in the future ($\mathtt{F}[\leq 1+1]$ and $\mathtt{F}[< 1+1]$), combined with detailed logical conditions related to the ACC system behavior. The presence of both $\mathtt{G}[\leq]$ and $\mathtt{F}[\leq]$ introduces layered time constraints, increasing the formula's complexity. On the contrary, ACC_R1_max simplifies the interplay between next ($\mathtt{X}$) and global ($\mathtt{G}[\leq]$) conditions, lowering the overall complexity. This formula focuses on what happens within certain future intervals ($\mathtt{F}[\leq 3]$ and $\mathtt{F}[< 3]$), which can be easier to manage than the stringent and overlapping conditions in ACC_R1_min.

Figure 7.25 illustrates the process of creating a Copilot monitor from a FRETish requirement. Initially, the requirement is captured by FRET. Next, the specific details of the generated LTL or MTL formula are exported from FRET and converted into a format that Copilot can interpret. Ogma serves as an intermediary tool for facilitating this conversion process [127].

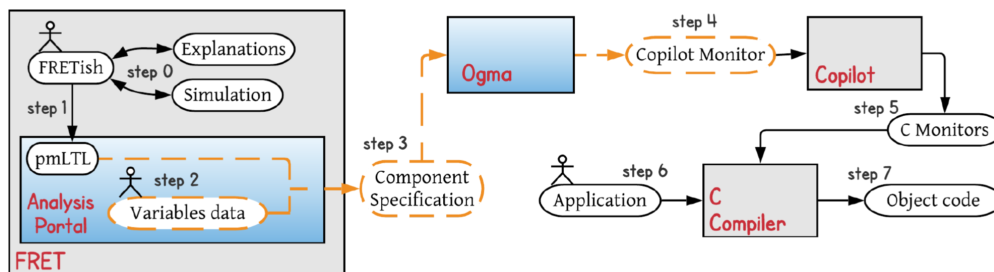

Figure 7.25: Monitor generation steps using FRET, Ogma and Copilot tools [127].

Finally, Copilot utilizes the transformed information to generate the monitor as specified in the FRETish statement. The process is largely automated, but the user is required to provide information regarding the arguments within the original requirement.

Copilot monitors evaluate a continuous stream of data against the MTL specifications. As depicted in Figure 7.26, these specifications are checked using designated *guard* functions, which determine whether the requirements are met or violated. When new data arrives, these guards update the status of each requirement based on predefined criteria, utilizing static variables to maintain system states or measurements. For example, `s0_get()` within `handlerpropACC_R0_guard()` retrieves the previous outcome of the ACC_R0 LTL formula by accessing the associated Boolean static variable $s0$, which is then updated after the execution of `handlerpropACC_R0_guard()`. In the event of a requirement violation, the corresponding guard triggers a designated handler function, which may reset the monitor to prepare it for subsequent data streams. However, the utilization of static variables and a sophisticated state management system implies that integrating or modifying this code requires a deeper understanding of its operational context.

To compare the performance of GROOT and Copilot monitors, various scenarios are examined, including both compliant and violation-inducing conditions. Figure 7.27 shows some of these test cases, each spanning four seconds to capture a complete behavior. For example, `case 1` depicts a scenario where relative distance is unsafe, resulting in a violation of

```c
bool handlerpropACC_R0_guard(void) {
  return !(((drel_cpy) >= (dsafe_cpy)) && ((s0_get)((0))));
}
bool handlerpropACC_R1_min_guard(void) {
    return !((((!(((((s1_get)((0))) - ((int64_t)(1))) <= ((s1_get)((0)))) && (((((s1_get)((0))) <= ((s1_get)((0)))) &&
        (((drel_cpy) >= (dsafe_cpy)) && (((s15_get)((0))) || ((s3_get)((0)))))) || (((((s1_get)((0))) - ((int64_t)(1))) <= ((s16_get)((0)))) &&
        ((((s16_get)((0))) <= ((s1_get)((0)))) && ((s17_get)((0)))))))) || (((!((drel_cpy) > (dsafe_cpy))) && ((s18_get)((0)))) ||
        (!(((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy)))))) && ((s19_get)((0)))) &&
        (((!(((((s1_get)((0))) - ((int64_t)(2))) <= ((s1_get)((0)))) && (((((s1_get)((0))) <= (((s1_get)((0))) - ((int64_t)(2)))) &&
        ((((drel_cpy) > (dsafe_cpy)) && (((s20_get)((0))) || ((s3_get)((0))))) && (!(((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy)))))) ||
        (((((s1_get)((0))) - ((int64_t)(2))) <= ((s21_get)((0)))) && (((((s21_get)((0))) <= (((s1_get)((0))) - ((int64_t)(2)))) && ((s22_get)((0)))) ||
        (((((s1_get)((0))) - ((int64_t)(2))) <= ((s23_get)((0)))) && ((((s23_get)((0))) <= (((s1_get)((0))) - ((int64_t)(2)))) && ((s24_get)((0))))))))) ||
        (((((s1_get)((0))) - ((int64_t)(1))) <= ((s1_get)((0)))) && (((((s1_get)((0))) <= ((s1_get)((0)))) && (((s3_get)((0))) ||
        (((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy))))) || (((((s1_get)((0))) - ((int64_t)(1))) <= ((s25_get)((0)))) &&
        ((((s25_get)((0))) <= ((s1_get)((0)))) && ((s26_get)((0)))))))) && ((s27_get)((0)))));
}
bool handlerpropACC_R1_max_guard(void) {
  return !(((!(((((s1_get)((0))) - ((int64_t)(3))) <= ((s1_get)((0)))) && (((((s1_get)((0))) <= (((s1_get)((0))) - ((int64_t)(3)))) &&
      ((((drel_cpy) >= (dsafe_cpy)) && (((s2_get)((0))) || ((s3_get)((0))))) && (!(((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy)))))) ||
      (((((s1_get)((0))) - ((int64_t)(3))) <= ((s4_get)((0)))) && (((((s4_get)((0))) <= (((s1_get)((0))) - ((int64_t)(3)))) && ((s5_get)((0)))) ||
      (((((s1_get)((0))) - ((int64_t)(3))) <= ((s6_get)((0)))) && (((((s6_get)((0))) <= (((s1_get)((0))) - ((int64_t)(3)))) && ((s7_get)((0)))) ||
      (((((s1_get)((0))) - ((int64_t)(3))) <= ((s8_get)((0)))) && ((((s8_get)((0))) <= (((s1_get)((0))) - ((int64_t)(3)))) && ((s9_get)((0))))))))))) ||
      (((((s1_get)((0))) - ((int64_t)(2))) <= ((s1_get)((0)))) && (((((s1_get)((0))) <= ((s1_get)((0)))) && (((s3_get)((0))) ||
      (((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy))))) || (((((s1_get)((0))) - ((int64_t)(2))) <= ((s10_get)((0)))) &&
      (((((s10_get)((0))) <= ((s1_get)((0)))) && ((s11_get)((0)))) || (((((s1_get)((0))) - ((int64_t)(2))) <= ((s12_get)((0)))) &&
      ((((s12_get)((0))) <= ((s1_get)((0)))) && ((s13_get)((0))))))))) && ((s14_get)((0))));
}
```

Figure 7.26: Code snippet for the guard functions of Copilot ACC monitors.

ACC_R0. Conversely, `case 2` demonstrates a situation where v_{ego} quickly falls within the expected range (less than one second) after the distance becomes safe, triggering a violation of ACC_R1 for GROOT's monitor and ACC_R1_min for Copilot. In each case, random values are assigned to v_{ego} and D_{rel}, while D_{safe} is computed using the parameters outlined in Table 7.10 and equation shown in (7.1). A total of 12 cases are examined, with some differing only in the range of values. For instance, the distinction between `case 1` and `case 4` is the v_{ego} value, one falls within the range while the other lies outside.

The monitors are executed on a Core i7 CPU running at 2.5 GHz, using the Visual Studio integrated development environment (IDE) for testing, which is a commonly employed platform for software development and testing [5]. Figure 7.28 illustrates the relationship between the average execution times of GROOT and Copilot monitors and the increasing number of iterations. These iterations encompass the test cases previously outlined, with each test being executed 1000 times to ensure a more accurate average measurement. The data reveals that GROOT consistently requires less execution time compared to Copilot, indicating that GROOT is more efficient in terms of execution time. Furthermore, as the load

```
for (int test_case = 0; test_case < n; test_case++) {
    for (int sec = 0; sec <= test_duration; sec++) {
        double Drel, Dsafe, Vego;
        switch (test_case) {
        case 0: // Normal operation -->  Vego within [Vset_n, Vset_p] after 2 sec
            if (sec >= 2) Vego = (Vset_n + (rand() % (int)(Vset_p - Vset_n)));
            else Vego = (Vset_n - 5);
            Dsafe = (Vego * Theadway) + Doffset;
            Drel = Dsafe + (rand() % (int)Doffset);
            break;
        case 1: // Violating (Drel >= Dsafe)
            Vego = Vset_n + (rand() % (int)(Vset_p - Vset_n));
            Dsafe = (Vego * Theadway) + Doffset;
            Drel = Dsafe - (rand() % (int)Doffset);
            break;
        case 2: // Violating Vego adjustment --> Vego within [Vset_n, Vset_p] right away
            Vego = (Vset_n + (rand() % (int)(Vset_p - Vset_n)));
            Dsafe = (Vego * Theadway) + Doffset;
            Drel = Dsafe + (rand() % (int)Doffset);
            break;
        case 3: // Normal operation -->  Vego within [Vset_n, Vset_p] after 1 sec
            if (sec == 0) Vego = (Vset_n - ((rand() % (int)(Vset_p - Vset_n)) + 1));
            else Vego = (Vset_n + (rand() % (int)(Vset_p - Vset_n)));
            Dsafe = (Vego * Theadway) + Doffset;
            Drel = Dsafe + (rand() % (int)Doffset);
            break;
        case 4: // Violating (Drel >= Dsafe)
            Vego = Vset_n - (rand() % (int)(Vset_p - Vset_n));
            Dsafe = (Vego * Theadway) + Doffset;
            Drel = Dsafe - (rand() % (int)Doffset);
            break;
        case 5: // Violating Vego adjustment --> Vego outside [Vset_n, Vset_p]
            Vego = Vset_n - ((rand() % (int)(Vset_p - Vset_n)) + 1);
            Dsafe = (Vego * Theadway) + Doffset;
            Drel = Dsafe + (rand() % (int)Doffset);
            break;
```

Figure 7.27: Test case examples for executing ACC monitors.

increases, the gap between GROOT and Copilot also increases, implying that GROOT may put less load on the system, potentially leaving more resources available for other processes and applications. While the significance of execution time differences may not be evident in use cases characterized by short and infrequent monitor runs, in environments demanding persistent and frequent monitoring, such gaps are likely to accumulate, thereby influencing overall system performance.

As depicted in Figure 7.26, Copilot monitor implementation uses an FSM where each state is represented by a boolean or integer value stored in single static arrays. The FSM computes

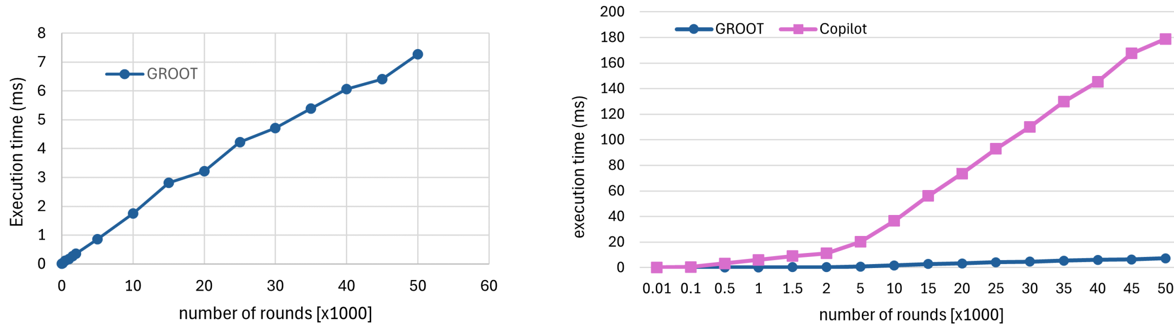

Figure 7.28: Execution time of ACC monitors generated by GROOT (left) in comparison with monitors generated by Copilot (right).

the next state based on the current state and other conditions (refer to Appendix A.2 for more information). The observed nonlinearity in Copilot execution time is likely due to the growing complexity of the conditional logic and more frequent state transitions triggering resets. As iterations progress, the conditions involving multiple state variables become increasingly complex to evaluate. Additionally, guard functions increasingly trigger resets, which despite being constant-time operations, introduce overhead when triggered frequently. This leads to a nonlinear increase in execution time, especially as more conditions are met and more branches are evaluated in later iterations, resulting in additional computational overhead.

To simulate time, each iteration of the loop is considered as one second. For finer time control, smaller steps such as 0.5 seconds per iteration, can be considered. When updating timing parameters, GROOT monitors only require modifying the monitor_period parameter in the header file, allowing for straightforward adjustments. However, Copilot monitors require resynthesis and updating the requirements with adjusted minimum and maximum timing constraints based on the number of required iterations. This is because FRET generates identical formulas for requirements differing only in time units. For example, three milliseconds, three seconds and three minutes are treated equally by FRET. Furthermore, Copilot utilizes past-time MTL (ptMTL) formulas, requiring static variables to retain past

states or values for decision-making based on historical data. Increasing the time limit results in a higher number of required static variables, leading to increased memory consumption.

As discussed in Section 4.1.2, practitioners are often familiar with FSMs, enabling them to comprehend the structures of GROOT's monitors and make modifications as needed. If engineers modify the monitor to extract specific information while ensuring the logical model remains unchanged, the generated formal specifications can serve to ensure the revised source code still adheres to the original requirement. GROOT also facilitates the reuse of the same monitor in similar contexts. For example, when encountering requirements with new events, an Input_Handler module can be developed to define these APs while utilizing the same Monitor component. Conversely, Copilot monitors offer broader capabilities in handling a wider range of conditions and states by considering the past behavior of the system, as discussed in Section 2.2.3. Appendix A presents the source codes for both GROOT and Copilot ACC monitors to enhance comprehension.

7.5 Summary

This chapter presents the application of GROOT's monitors through three different cases. The first case examines a comparative study on man-time efficiency between GROOT and a conventional method employed at GE Aviation. The findings show that adopting a formal approach like GROOT facilitates a 2.3 times quicker development of similar monitors when compared to the existing procedures at GE Aviation. In the second case, hardware monitors are developed to detect potential malicious attacks or faults in a lightweight encryption algorithm. These monitors undergo formal verification and simulation to uncover implemented attacks or faults. The implementation results reveal that the monitors impose minimal overhead on the main system, which is crucial in devices with limited resources.

The third case focuses on autonomous vehicles, emphasizing the significance of monitors across various stages of the product lifespan. It also compares how GROOT and another contemporary monitoring framework, Copilot, perform in generating monitors for the same requirements. The synthesized monitors are tested and evaluated in software, highlighting GROOT's efficiency in terms of execution-time. An analysis of the monitor code structure indicates that GROOT provides a more transparent depiction of monitor transitions by using state machines, a format familiar to practitioners. This clarity facilitates the automated formal analysis of monitors and increases trust in their functionality.

Chapter 8

Conclusions

Real-time embedded systems are employed in various critical applications such as avionics, robotics and autonomous systems, where both correct functionality and timely response are crucial. Despite thorough verification efforts during development, potential hardware and software faults may still arise, necessitating ongoing validation, especially as environmental interactions may only become apparent in fielded systems. Runtime verification (RV) is a dynamic verification approach that utilizes monitors derived from formal system requirements to assess whether the real-time behaviors of a system conform to its specifications. However, a lack of familiarity makes RV adoption challenging for practicing engineers without formal methods background. To help overcome this barrier, a framework is developed which automatically synthesizes monitors using pseudo-English statements, thereby avoiding the need for formal notations.

8.1 Contributions

To encourage practitioners to use RV without needing training in formal methods, this dissertation introduces GROOT (Generalized Runtime mOnitOring Tool), a novel methodology and framework designed to automate monitor synthesis from structured English requirements. The methodology consists of three sequential and automated steps: (i) conversion of English specifications into formal representations, (ii) transformation of these representations

into monitor code, and (iii) formal verification of monitor implementations. The last step ensures that the monitor's final implementation adheres to the expected behavior, providing assurance of the monitor's correctness. By offering a practical and accessible approach, GROOT helps to bridge the often daunting gap between formal verification techniques and real-world embedded systems development.

In real-time systems, where response time is as crucial as correct functionality, timing constraints play a critical role across both software and hardware levels. Software tasks must be executed within designated time frames to ensure system stability, while hardware components rely on precise timing of inputs and outputs for proper functioning. Consequently, monitors must verify compliance with these timing metrics as stated in the requirements. GROOT achieves this goal by using distinct pseudo-English languages, monitor implementations and formal analysis for functional and timing requirements. This involves tailored front ends, intermediate forms, and formal verification techniques to translate pseudo-English functional and timing specifications into monitors. For functional requirements, the NASA FRET tool is utilized to formalize structured English expressions. For timing requirements, this work introduces TIMESPEC, a structured English language specifically designed to capture various timing bounds.

Highly optimized embedded systems may integrate software with hardware components such as FPGAs to serve various specialized functions. Normally this would require dissimilar approaches to monitor specification, implementation and verification for software and hardware components. GROOT avoids this by supporting monitors in various implementation languages, including C and HDL for software and hardware, respectively. This flexibility allows hardware monitors to be isolated entirely from the software stack to mitigate the risk of software crashes or potential attack vectors aimed at the software.

GROOT monitors are designed to function autonomously and operating outside the soft-

ware/hardware systems they monitor. This approach treats the application as a black box, providing versatility and adaptability for RV. Input events to the monitors along with handling violations and responses are managed in separate external modules, maintaining a straightforward structure for monitors to facilitate formal analysis of the state transitions. These modules can also be shared among multiple monitors with similar triggering events to reduce required resources.

In the development phase, synthesized monitors significantly reduce the manual effort required to hard-code expected results for each test case. Instead of spending time on creating and maintaining individual test cases, testers can develop reusable monitors that comprehensively cover functional and safety requirements. Automated monitors ensure uniform application of rules across all scenarios, minimizing human errors and inconsistencies. As requirements evolve, updating monitors is easier and more maintainable than revising multiple hard-coded tests.

Once deployed, these monitors provide real-time checks during system operation. For example, in critical situations such as UAS operations, monitors can trigger immediate safety measures such as initiating an emergency landing to prevent catastrophic outcomes. Real-time feedback also allows operators to make informed decisions during mission-critical tasks. This approach reduces the time and resources required for troubleshooting and repairs.

Embedded systems often encounter resource constraints arising from cost and power limitations. Monitors are required to have minimal overhead and latency to avoid interfering with the system's real-time constraints. The extent and type of optimization necessary depend on factors including the desired level of monitor isolation, whether single or multiple monitors are employed for specific behaviors, implementation in software or hardware, and whether the target is a development prototype or a deployed product. GROOT monitors have minimal overhead in both software and hardware contexts. Furthermore, examples are presented

to illustrate single or multiple monitor configurations serving diverse purposes, ranging from debugging to providing backup safety in real-time scenarios.

The following is a listing of funded proposal and publications related to this work:

- Cameron Patterson and Behnaz Rezvani, "FMCloak: Practitioners Using Formal Methods Without Knowing It," NSF 2123550 Formal Methods in the Field: Track II, Virginia Tech, 2021.
- Behnaz Rezvani and Cameron Patterson, "Differentiated Monitor Generation for Real-Time Systems," *Proceedings of the 18th International Conference on Software Technologies (ICSOFT)*, pages 353-360, July 2023.
- Behnaz Rezvani and Cameron Patterson, "A Monitoring Methodology and Framework to Partition Embedded Systems Requirements," *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 561-568, April 2024.
- Behnaz Rezvani and Cameron Patterson, "Automatic Generation of Hardware and Software Monitors for Real-Time Embedded Systems," to be submitted to *Communications in Computer and Information Science (CCIS)*, August 2024.

Other publications:

- Behnaz Rezvani and William Diehl, "Hardware implementations of NIST lightweight cryptographic candidates: A first look," *Proceedings of the NIST Lightweight Cryptography Workshop*, November 2019.
- Flora Coleman, Behnaz Rezvani, Sachin Sachin, and William Diehl, "Side channel resistance at a cost: A comparison of ARX-based authenticated encryption," *Proceedings*

of the 30th International Conference on Field-Programmable Logic and Applications (FPL), pp. 193-199. August 2020.

- Behnaz Rezvani, Thomas Conroy, Luke Beckwith, Matthew Bozzay, Trevor Laffoon, David McFeeters, Yijia Shi, Minh Vu, William Diehl, "Efficient Simultaneous Deployment of Multiple Lightweight Authenticated Ciphers," Cryptology ePrint Archive, 2020.

8.2 Future Work

Future work involves developing a moderately portable benchmark suite of applications and their associated requirements, tailored for specific embedded platforms such as the Raspberry Pi. Unlike platforms that abstract applications from hardware details, like Windows hosts, embedded systems vary widely in their configurations and capabilities. Developing such a benchmark suite would contribute significantly to the research community by enabling comparative evaluations of different approaches in terms of efficiency and effectiveness.

To accurately evaluate the real-world impact of the GROOT framework beyond simulations and analysis, it is desirable to integrate the monitors within a live system under real-time conditions. This approach will involve utilizing drones as test subjects to comprehensively assess both timing and functional requirements. Given the necessity for multiple monitors to ensure safety in large and complex systems, a "monitor of monitors" will help to manage all the monitors and isolate system faults. The expansion of TIMESPEC templates is also planned to accommodate more complex scenarios involving multiple timing constraints, enhancing the ability to capture intricate system behaviors.

Bibliography

[1] ModelSim. URL https://eda.sw.siemens.com/en-US/ic/modelsim/.

[2] Xilinx Software Development Kit (XSDK). URL https://www.xilinx.com/products/design-tools/embedded-software/sdk.html.

[3] Auterion Skynode. URL https://auterion.com/drone-manufacturers/skynode/.

[4] Xilinx Vivado Design Suite. URL https://www.xilinx.com/products/design-tools/vivado.html.

[5] Visual Studio. URL https://visualstudio.microsoft.com/.

[6] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, Sep. 1987.

[7] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Comput. Sci.*, 126 (2):183–235, 1994.

[8] B. Angelo, S. Sankaranarayanan, C. Sanchez, and M. Others. LOLA: Runtime monitoring of synchronous systems. pages 166–174, Jul. 2005.

[9] M. Archer, B. DiVito, and C. A. Muñoz. Developing user strategies in PVS: A tutorial. 2003.

[10] A. Arcuri, M. Z. Iqbal, and L. Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *Testing Software and Systems*, pages 95–110, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[11] ASTM F3269-17. Standard Practice for Methods to Safely Bound Flight Behavior of Unmanned Aircraft Systems Containing Complex Functions. Standard, ASTM International, West Conshohocken, PA, 2017. URL www.astm.org.

[12] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar. *IEEE Transactions on Software Engineering*, 41(7):620–638, 2015.

[13] T. Babiak, M. Křetínský, V. Řehák, and J. Strejček. LTL to Büchi automata translation: Fast and more deterministic. In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 95–109, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[14] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, MA, USA, 2008.

[15] S. Banik, A. Chakraborti, T. Iwata, K. Minematsu, M. Nandi, T. Peyrin, Y. Sasaki, S. M. Sim, and Y. Todo. GIFT-COFB: An Authenticated Cipher Submission to the NIST LWC Competition, Mar. 2019. URL https://csrc.nist.gov/Projects/lightweight-cryptography/finalist.

[16] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 44–57. Springer, 2004.

[17] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From EAGLE to RULER. In *Runtime Verification*, pages 111–125, Berlin, Heidelberg, 2007. Springer.

[18] H. Barringer, K. Havelund, D. Rydeheard, and A. Groce. Rule systems for runtime

verification: A short tutorial. In S. Bensalem and D. A. Peled, editors, *Runtime Verification*, pages 1–24, Berlin, Heidelberg, 2009. Springer.

[19] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan. *Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications*, pages 135–175. Springer International Publishing, Cham, 2018.

[20] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. *Introduction to Runtime Verification*, pages 1–33. Feb. 2018.

[21] E. Bartocci, L. Bortolussi, M. Loreti, L. Nenzi, and S. Silvetti. Moonlight: A lightweight tool for monitoring spatio-temporal properties, 2021.

[22] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C specification language. 2008.

[23] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *Proceedings of the 26th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'06, page 260–272, Berlin, Heidelberg, 2006. Springer-Verlag.

[24] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Runtime Verification*, pages 126–138, Berlin, Heidelberg, 2007. Springer.

[25] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.

[26] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4), Sep. 2011.

[27] M. Bernaerts, B. Oakes, K. Vanherpen, B. Aelvoet, H. Vangheluwe, and J. Denil. Validating industrial requirements with a contract-based approach. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 18–27, 2019.

[28] V. Bertram, H. Kausch, E. Kusmenko, H. Nqiri, B. Rumpe, and C. Venhoff. Leveraging natural language processing for a consistency checking toolchain of automotive requirements. In *2023 IEEE 31st International Requirements Engineering Conference (RE)*, pages 212–222, 2023.

[29] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan. Hardware Trojan attacks: Threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, 2014.

[30] B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Time-triggered runtime verification. *Formal Methods in System Design (FMSD)*, 43:29–60, 2013.

[31] N. E. Boudette. Tesla says autopilot makes its cars safer. crash victims say it kills. *International New York Times*, 2021.

[32] N. E. Boudette. ‘it happened so fast’: Inside a fatal Tesla autopilot accident. *International New York Times*, 2021.

[33] B. Broekman and E. Notenboom. *Testing embedded software*. Pearson Education, 2003.

[34] J. R. Büchi. *On a Decision Method in Restricted Second Order Arithmetic*, pages 425–435. Springer New York, New York, NY, 1990.

[35] J. C. Campos and J. Machado. Pattern-based analysis of automated production sys-

tems. *IFAC Proceedings Volumes*, 42(4):972–977, 2009. 13th IFAC Symposium on Information Control Problems in Manufacturing.

[36] M. Carwehl, T. Vogel, G. Rodrigues, and L. Grunske. Runtime verification of self-adaptive systems with changing requirements. In *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 104–114, Los Alamitos, CA, USA, May 2023. IEEE Computer Society.

[37] I. Cassar and A. Francalanza. On synchronous and asynchronous monitor instrumentation for actor-based systems. *FOCLASA: 13th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems*, 175:54–68, Jan. 2014.

[38] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. *Electronic Proceedings in Theoretical Computer Science*, 254:15–28, Sep. 2017.

[39] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In A. Biere and R. Bloem, editors, *Computer Aided Verification*, pages 334–342, Cham, 2014. Springer International Publishing.

[40] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. *SIGPLAN Not.*, 42(10):569–588, Oct. 2007.

[41] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An open source tool for symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification*, pages 359–364, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[42] A. Cimatti, C. Tian, and S. Tonetta. NuRV: a nuXmv extension for runtime verification. In *International Conference on Runtime Verification*, pages 382–392. Springer, 2019.

[43] A. Cimatti, C. Tian, and S. Tonetta. NuRV 1.6. 0 user manual. 2020.

[44] E. Clarke, W. Klieber, M. Nováček, and P. Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Jan. 2012.

[45] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 2000.

[46] C. Colombo, G. J. Pace, and G. Schneider. LARVA — safer monitoring of real-time Java programs (tool paper). In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 33–37, 2009.

[47] U. Costa, S. Campos, N. Vieira, and D. Déharbe. Explicit-symbolic modelling for formal verification. *Electr. Notes Theor. Comput. Sci.*, 130:301–321, May 2005.

[48] P. Cuoq et al. Frama-C: A software analysis perspective. In *Proc. Int. Conf. Softw. Eng. and Formal Methods*, SEFM'12, page 233–247, Berlin, 2012. Springer.

[49] C. Czepa and U. Zdun. On the understandability of temporal properties formalized in linear temporal logic, property specification patterns and event processing language. *IEEE Transactions on Software Engineering*, 46(1):100–112, 2020.

[50] M. d'Amorim and G. Roşu. Efficient monitoring of ω-languages. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, page 364–378, Berlin, Heidelberg, 2005. Springer-Verlag.

[51] J.-L. Danger, L. Fribourg, U. Kühne, and M. Naceur. LAOCOÖN: A run-time monitoring and verification approach for hardware Trojan detection. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 269–276, 2019.

[52] Documentation, Simulink. Simulation and model-based design, 2020. URL https://www.mathworks.com/products/simulink.html.

[53] M. Dowson. The Ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84, Mar. 1997.

[54] D. Drusinsky. The temporal rover and the ATG rover. In *International SPIN Workshop on Model Checking of Software*, pages 323–330. Springer, 2000.

[55] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, Oct. 2016.

[56] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, page 411–420, New York, NY, USA, 1999. Association for Computing Machinery.

[57] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In W. A. Hunt and F. Somenzi, editors, *Computer Aided Verification*, pages 27–39, Berlin, Heidelberg, 2003. Springer.

[58] D. Elkaduwe, P. Derrin, and K. Elphinstone. Kernel design for isolation and assurance of physical memory. In *Proceedings of the 1st Workshop on Isolation and Integration*

in Embedded Systems, IIES '08, page 35–40. Association for Computing Machinery, 2008.

[59] ESBMC. An efficient SMT-based bounded model checker. https://ssvlab.github.io/esbmc/.

[60] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer*, 14, Jun. 2011.

[61] Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. *Engineering Dependable Software Systems*, 34:141–175, Jan. 2013.

[62] Y. Falcone, S. Krstić, G. Reger, and D. Traytel. A taxonomy for classifying runtime verification tools. *International Journal on Software Tools for Technology Transfer*, 23 (2):255–284, 2021.

[63] M. Fitting. *First-Order Logic and Automated Theorem Proving (2nd Ed.)*. Springer-Verlag, Berlin, Heidelberg, 1996.

[64] Frama-C. A platform to make C code safer and more secure. https://frama-c.com/index.html.

[65] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole. ESBMC 5.0: An industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 888–891, New York, NY, USA, 2018. Association for Computing Machinery.

[66] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, pages 53–65, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[67] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 412–416, 09 2001.

[68] D. Giannakopoulou, T. Pressburger, A. Mavridou, J. Rhein, J. Schumann, and N. Shi. Formal requirements elicitation with FRET. In *REFSQ Workshops*, 2020.

[69] J. Harrison, J. Urban, and F. Wiedijk. *History of Interactive Theorem Proving*, volume 9, pages 135–214. 12 2014.

[70] K. Havelund. Runtime verification of C programs. In K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, editors, *Testing of Software and Communicating Systems*, pages 7–22, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[71] K. Havelund. Runtime verification of C programs. In *Testing of Software and Communicating Systems*, pages 7–22. Springer, 2008.

[72] K. Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, 17(2):143–170, 2015.

[73] K. Havelund and D. Peled. Monitorability for runtime verification. In *Runtime Verification*, pages 447–460, Cham, 2023. Springer Nature Switzerland.

[74] K. Havelund and G. Reger. *Runtime Verification Logics A Language Design Perspective*, pages 310–338. Springer International Publishing, Cham, 2017.

[75] K. Havelund and G. Rosu. Java PathExplorer–a runtime verification tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, pages 18–21. Citeseer, 2001.

[76] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. TACAS '02, page 342–356, Berlin, Heidelberg, 2002. Springer-Verlag.

[77] K. Havelund, G. Reger, D. Thoma, and E. Zalinescu. Monitoring events that carry data. In E. Bartocci and Y. Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 61–102. Springer, 2018.

[78] H. Herencia-Zapana, J. Lopez, G. Gallagher, B. Meng, C. Patterson, and L. Maalolan. Formal verification tool evaluation for unmanned aircraft containing complex functions. In *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, pages 1–9, 2020.

[79] K. Hormann and A. Agathos. The point in polygon problem for arbitrary polygons. *Computational Geometry*, 20(3):131–144, 2001.

[80] Intel Aero. A computer board with a high performance Linux board. https://ardupilot.org/copter/docs/common-intel-aero-overview.html.

[81] M. Z. Iqbal, A. Arcuri, and L. Briand. Environment modeling and simulation for automated testing of soft real-time embedded software. *Softw. Syst. Model.*, 14(1): 483–524, Feb. 2015.

[82] S. Jakšić, E. Bartocci, R. Grosu, and D. Ničković. An algebraic framework for runtime verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2233–2243, 2018.

[83] S. Jakšić, E. Bartocci, R. Grosu, R. Kloibhofer, T. Nguyen, and D. Ničkovié. From signal temporal logic to FPGA monitors. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 218–227, 2015.

[84] S. Jalili and M. MirzaAghaei. RVERL: Run-time verification of real-time and reactive programs using event-based real-time logic approach. In *5th ACIS International Con-*

ference on Software Engineering Research, Management & Applications (SERA 2007), pages 550–557, 2007.

[85] C. Johannsen, P. Jones, B. Kempa, K. Y. Rozier, and P. Zhang. R2U2 version 3.0: Re-imagining a toolchain for specification, resource estimation, and optimized observer generation for runtime verification in hardware and software. In *Computer Aided Verification*, pages 483–497, Cham, 2023. Springer Nature Switzerland.

[86] A. Kane, O. Chowdhury, A. Datta, and P. Koopman. A case study on runtime monitoring of an autonomous research vehicle (ARV) system. In *Runtime Verification*, pages 102–117, Cham, 2015. Springer.

[87] M. Kaufmann and J. Moore. Some key research problems in automated theorem proving for hardware and software verification. *RACSAM*, 98, 01 2004.

[88] B. Kempa, P. Zhang, P. H. Jones, J. Zambreno, and K. Y. Rozier. Embedding online runtime verification for fault disambiguation on Robonaut2. In *Formal Modeling and Analysis of Timed Systems*, pages 196–214, Cham, 2020. Springer.

[89] W. Khan. RuSTL: Runtime verification using Signal Temporal Logic. Master's thesis, University of Waterloo, 2019. Available at: https://uwspace.uwaterloo.ca/handle/10012/14552.

[90] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. SeL4: Formal verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[91] P. Z. Kolano. Proof assistance for real-time systems using an interactive theorem

prover. *Theoretical Computer Science*, 282(1):53–99, 2002. Real-Time and Probabilistic Systems.

[92] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE'05, page 372–381, New York, USA, 2005. Association for Computing Machinery.

[93] H. Kopetz and W. Steiner. *Real-time systems: design principles for distributed embedded applications*. Springer Nature, 2022.

[94] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, Oct. 1990.

[95] D. Kroening and M. Purandare. EBMC: Enhanced bounded model checker, Mar. 2017. URL http://www.cprover.org/ebmc/.

[96] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[97] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[98] L. Lamport. What good is temporal logic? In *IFIP congress*, volume 83, pages 657–668, 1983.

[99] A. V. Lamsweerde. Formal specification: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, page 147–159, New York, NY, USA, 2000. Association for Computing Machinery.

[100] D. Leroy, P. Jeanjean, E. Bousse, M. Wimmer, and B. Combemale. Runtime monitoring for executable DSLs. *The Journal of Object Technology*, 19:6:1, 01 2020.

[101] M. Leucker. Teaching runtime verification. In S. Khurshid and K. Sen, editors, *Runtime Verification*, pages 34–48, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[102] M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[103] M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, and A. Schramm. TeSSLa: Runtime verification of non-synchronized real-time streams. SAC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[104] M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, and A. Schramm. TeSSLa: Runtime verification of non-synchronized real-time streams. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 1925–1933, New York, NY, USA, 2018. Association for Computing Machinery.

[105] N. Leveson and C. Turner. An investigation of the Therac-25 accidents. *Computer*, 26 (7):18–41, 1993.

[106] S. Levin and J. C. Wong. Self-driving Uber kills Arizona woman in first fatal crash involving pedestrian. *The Guardian*, Mar. 2018. URL https://www.theguardian.com/technology/2018/mar/19/uber-self-driving-car-kills-woman-arizona-tempe.

[107] J. Li, M. Y. Vardi, and K. Y. Rozier. Satisfiability checking for mission-time LTL. In *Computer Aided Verification*, pages 3–22, Cham, 2019. Springer International Publishing.

[108] H. Lu and A. Forin. The design and implementation of P2V, an architecture for zero-overhead online verification of software programs. Technical report, August 2007.

[109] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. ŞerbănuŢă, and G. Roşu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In B. Bonakdarpour and S. A. Smolka, editors, *Runtime Verification*, pages 285–300, Cham, 2014. Springer International Publishing.

[110] L. T. Maalolan. Trusted unmanned aerial system operations. Master's thesis, Virginia Tech, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, May 2020.

[111] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In Y. Lakhnech and S. Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[112] O. Maler and D. Nickovic. Monitoring properties of analog and mixed-signal circuits. *International Journal on Software Tools for Technology Transfer*, 15, Jun. 2012.

[113] K. Mamouras, A. Chattopadhyay, and Z. Wang. A compositional framework for quantitative online monitoring over continuous-time signals. In *Runtime Verification*, pages 142–163, Cham, 2021. Springer International Publishing.

[114] G. Martino and G. Fey. Runtime monitoring of c-LTL specifications on FPGAs using HLS. In *2022 18th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design*, pages 1–4, 2022.

[115] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger. Specification patterns for robotic missions. *IEEE Transactions on Software Engineering*, 47(10): 2208–2224, 2021.

[116] P. O. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *Automated Software Engineering*, 17(2):149–180, 2010.

[117] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Tools for Technology Transfer*, 14(3):249–289, 2012.

[118] M. Mettler, D. Mueller-Gritschneder, and U. Schlichtmann. A distributed hardware monitoring system for runtime verification on multi-tile MPSoCs. *ACM Trans. Archit. Code Optim.*, 18(1), Dec. 2021.

[119] P. Moosbrugger, K. Y. Rozier, and J. Schumann. R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. *Formal Methods in System Design*, 51 (1):31–61, 2017.

[120] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, Hoboken and N.J, 3rd ed edition, 2012.

[121] NASA Ames Research Center. FRET: formal requirements elicitation tool. https://www.nasa.gov/fret.

[122] S. Navabpour, Y. Joshi, W. Wu, S. Berkovich, R. Medhat, B. Bonakdarpour, and S. Fischmeister. RiTHM: A tool for enabling time-triggered runtime verification for C programs. ESEC/FSE 2013, page 603–606, New York, NY, USA, 2013. Association for Computing Machinery.

[123] R. Oshana and M. Kraeling. *Software engineering for embedded systems: Methods, practical techniques, and applications*. Newnes, second edition, 2019.

[124] J. S. Ostroff. Composition and refinement of discrete real-time systems. *ACM Trans. Softw. Eng. Methodol.*, 8(1):1–48, Jan. 1999.

[125] D. Peled and K. Havelund. *Refining the Safety–Liveness Classification of Temporal Properties According to Monitorability*, pages 218–234. Springer International Publishing, Cham, 2019.

[126] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In *2008 Real-Time Systems Symposium*, pages 481–491, 2008.

[127] I. Perez, A. Mavridou, T. Pressburger, A. Goodloe, and D. Giannakopoulou. Automated translation of natural language requirements to runtime monitors. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 387–395, Cham, 2022. Springer.

[128] L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A hard real-time runtime monitor. In *Runtime Verification*, pages 345–359, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[129] L. Pike, S. Niller, and N. Wegmann. Runtime verification for ultra-critical systems. In S. Khurshid and K. Sen, editors, *Runtime Verification*, pages 310–324, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[130] S. Pinisetty, T. Jéron, S. Tripakis, Y. Falcone, H. Marchand, and V. Preoteasa. Predictive runtime verification of timed properties. *Journal of Systems and Software*, 132: 353–365, 2017.

[131] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (SFCS 1977)*, pages 46–57, 1977.

[132] A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In

J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, pages 573–586, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[133] R. S. Pressman. *Software engineering: a practitioner's approach.* Palgrave macmillan, 2005.

[134] A. A. Rafeeq. A development platform to evaluate UAV runtime verification through hardware-in-the-loop simulation. Master's thesis, Virginia Tech, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, May 2020.

[135] T. Reinbacher, M. Függer, and J. Brauer. Runtime verification of embedded real-time systems. *Formal Methods in System Design*, 44:203–239, June 2014.

[136] T. Reinbacher, K. Y. Rozier, and J. Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–372, Berlin, Heidelberg, 2014. Springer.

[137] B. Rezvani and C. Patterson. Differentiated monitor generation for real-time systems. In *Proceedings of the 18th International Conference on Software Engineering*, volume 1, pages 353–360. SCITEPRESS, 2023.

[138] B. Rezvani and C. Patterson. A monitoring methodology and framework to partition embedded systems requirements. In *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 561–568. SCITEPRESS, 2024.

[139] B. Rezvani, F. Coleman, S. Sachin, and W. Diehl. Hardware implementations of NIST lightweight cryptographic candidates: A first look. *Cryptology ePrint Archive*, 2019.

[140] K. Y. Rozier. Specification: The biggest bottleneck in formal methods and autonomy. In S. Blazy and M. Chechik, editors, *Verified Software. Theories, Tools, and Experiments*, pages 8–26, Cham, 2016. Springer International Publishing.

[141] K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In D. Bošnački and S. Edelkamp, editors, *Model Checking Software*, pages 149–167, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[142] G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.

[143] C. Sánchez. Online and offline stream runtime verification of synchronous systems. In *International Conference on Runtime Verification*, pages 138–163. Springer, 2018.

[144] K. Sen and G. Roşu. Generating optimal monitors for extended regular expressions. *Electronic Notes in Theoretical Computer Science*, 89:226–245, 07 2003.

[145] G. Simulation. A powerful 3D simulation environment for autonomous robots. https://gazebosim.org/home.

[146] D. Solet, J.-L. Béchennec, M. Briday, S. Faucou, and S. Pillement. Hardware runtime verification of embedded software in SoPC. pages 1–6, May 2016.

[147] D. Solet, S. Pillement, J.-L. Béchennec, M. Briday, and S. Faucou. HW-based architecture for runtime verification of embedded software on SoPC systems. In *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 249–256, 2018.

[148] M. Sönmez Turan, K. McKay, Ç. Çalık, D. Chang, and L. Bassham. Status report on the first round of the NIST lightweight cryptography standardization process. Technical report, National Institute of Standards and Technology, 2019.

[149] G. Spanoudakis, C. Kloukinas, and K. Androutsopoulos. Towards security monitoring patterns. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, page 1518–1525, New York, NY, USA, 2007. Association for Computing Machinery.

[150] J. M. Spivey and J. Abrial. *The Z notation*, volume 29. Prentice Hall Hemel Hempstead, 1992.

[151] Spot. Application for rewriting LTL formula or translating them into automata. https://spot.lre.epita.fr/app/.

[152] J. Stamenkovich, L. Maalolan, and C. Patterson. Formal assurances for autonomous systems without verifying application software. In *2019 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED UAS)*, pages 60–69, 2019.

[153] J. A. Stamenkovich. Enhancing trust in autonomous systems without verifying software. Master's thesis, Virginia Tech, Bradley Department of Electrical and Computer Engineering, Blacksburg, VA, May 2019.

[154] D. Sunday. Inclusion of a point in a polygon, 2001. URL https://web.archive.org/web/20130126163405/http://geomalgorithms.com/a03-_inclusion.html.

[155] Texas Instruments. *OPT9221 Time-of-Flight Controller*. Dallas, TX, USA, Jun. 2015. https://www.ti.com/product/OPT9221.

[156] The MathWorks Inc. Adaptive cruise control with sensor fusion, 2024. https://www.mathworks.com/help/mpc/ug/adaptive-cruise-control-with-sensor-fusion.html.

[157] P. Travis. Why the AT&T network crashed. *Telephony*, 218(4):11–12, 1990.

[158] T. E. Uribe. Combinations of model checking and theorem proving. In H. Kirchner and C. Ringeissen, editors, *Frontiers of Combining Systems*, pages 151–170, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[159] A. Valmari. *The state explosion problem*, pages 429–528. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[160] S. Vijayaraghavan and M. Ramanathan. *A practical guide for SystemVerilog assertions*. Springer Science & Business Media, 2005.

[161] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *Software, IET*, 1:172 – 179, Nov. 2007.

[162] Xilinx. Amd vitis hls. https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html.

[163] Xilinx. ZCU104 Evaluation Board User Guide. https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf.

[164] D. Yadron and D. Tynan. Tesla driver dies in first fatal crash while using autopilot mode. *The Guardian*, Jun. 2016. URL https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk.

[165] E. Zapridou, E. Bartocci, and P. Katsaros. Runtime verification of autonomous driving systems in CARLA. Oct. 2020.

[166] T. Zhang and Y. Wang. An industrial software model checking method based on machine learning and its application in education. In M. A. Jan and F. Khan, editors, *Application of Big Data, Blockchain, and Internet of Things for Education Informatization*, pages 53–61, Cham, 2023. Springer Nature Switzerland.

Appendices

Appendix A

Monitors for the ACC System

A.1 GROOT Monitors

The below listings provide the ACC_R0 and ACC_R1 monitors and their dependencies generated by GROOT.

Listing A.1: The header file for the ACC_R0 and ACC_R1 monitors.

```
#include <stdbool.h>
#include <assert.h>
#include <stdio.h>
#define monitor_period 1
#define min_time 1
#define max_time 3
enum States_func { STATE0, STATE1 };
enum States_time { INITIAL, WAIT, RESPONSE, MAX_TIME_ERROR, MIN_TIME_ERROR };
struct Monitor_State_Func {
  enum States_func state;
  bool error;
  bool pass;
  unsigned int state_duration;
};
struct Monitor_State_Time {
  enum States_func state;
  bool error;
```

```
18    bool pass;
19    unsigned int state_duration;
20  };
21  struct Monitor_Inputs {
22    bool EVE0;
23    bool EVE1;
24  };
25  void acc_monitor_input_handler(double drel, double dsafe, double vego, double
        vset_n, double vset_p, struct Monitor_Inputs* monitor_inputs);
26  void acc_monitor_func(struct Monitor_State_Func* monitor_state_func, struct
        Monitor_Inputs* monitor_inputs);
27  void acc_monitor_time(struct Monitor_State_Time* monitor_state_time, struct
        Monitor_Inputs* monitor_inputs);
28  void acc_monitor_output_handler_func(struct Monitor_State_Func*
        monitor_state_func);
29  void acc_monitor_output_handler_time(struct Monitor_State_Time*
        monitor_state_time);
```

Listing A.2: The Input_Handler and Output_Handler modules.

```
1   #include "acc_monitor.h"
2   void acc_monitor_input_handler(double drel, double dsafe, double vego, double
        vset_n, double vset_p, struct Monitor_Inputs* monitor_inputs) {
3     monitor_inputs->EVE0 = (drel >= dsafe);
4     monitor_inputs->EVE1 = ((vego >= vset_n) && (vego <= vset_p));
5   }
6   void acc_monitor_output_handler_func(struct Monitor_State_Func*
        monitor_state_func) {
7     if (monitor_state_func->error) {
8       printf("Monitor ACC_R0 has been violated.\n");
9       monitor_state_func->state = STATE0;
10      monitor_state_func->error = false;
```

```
      monitor_state_func->pass = false;
      monitor_state_func->state_duration = 0;
    }
}
void acc_monitor_output_handler_time(struct Monitor_State_Time*
     monitor_state_time) {
    if (monitor_state_time->error) {
      printf("Monitor ACC_R1 has been violated.\n");
      monitor_state_time->state = INITIAL;
      monitor_state_time->error = false;
      monitor_state_time->pass = false;
      monitor_state_time->state_duration = 0;
    }
    if (monitor_state_time->pass) {
      printf("Monitor ACC_R1 has been satisfied.\n");
    }
}
```

Listing A.3: The ACC_R0 monitor.

```
#include "acc_monitor.h"
void acc_monitor_func(struct Monitor_State_Func* monitor_state_func, struct
     Monitor_Inputs* monitor_inputs)
{
    enum States_func state = monitor_state_func->state;
    bool error = monitor_state_func->error;
    unsigned int state_duration = monitor_state_func->state_duration;
    bool EVE0 = monitor_inputs->EVE0;
    switch (monitor_state_func->state) {
      case STATE0:
        state_duration += 1;
        if (!EVE0) {
```

```
12          error = true;
13          state = STATE1;
14        }
15        break;
16      case STATE1:
17        error = true;
18        break;
19      default:
20        error = true;
21        break;
22    }
23    monitor_state_func->state = state;
24    monitor_state_func->error = error;
25    monitor_state_func->state_duration = state_duration;
26
27    assert(monitor_state_func->state == STATE0 || monitor_state_func->state ==
       STATE1); // state validity
28    assert(monitor_state_func->state_duration >= 0);          // state_duration
       validity
29    assert(!(monitor_state_func->error && monitor_state_func->pass));   // error
        and pass mutual exclusivity
30    assert(monitor_state_func->error == (monitor_state_func->state == STATE1));
       // Error condition consistency
31    assert(monitor_state_func->pass == false);            // pass validity
32
33    if (monitor_state_func->state == STATE0) {            // Transitions
       validity
34      assert(monitor_inputs->EVE0 == (monitor_state_func->state == STATE0 && !
       monitor_state_func->error));
35      assert((!monitor_inputs->EVE0) == (monitor_state_func->state == STATE1 &&
       monitor_state_func->error));
```

```
36   }
37 }
```

Listing A.4: The ACC_R1 monitor.

```
1  #include "acc_monitor.h"
2  void acc_monitor_time(struct Monitor_State_Time* monitor_state_time, struct
       Monitor_Inputs* monitor_inputs)
3  {
4    enum States_time state = monitor_state_time->state;
5    bool error = monitor_state_time->error;
6    bool pass = monitor_state_time->pass;
7    unsigned int state_duration = monitor_state_time->state_duration;
8    bool EVE0 = monitor_inputs->EVE0;
9    bool EVE1 = monitor_inputs->EVE1;
10   unsigned int MAX_TIME = max_time/monitor_period;
11   unsigned int MIN_TIME = min_time/monitor_period;
12   switch (state) {
13     case INITIAL:
14       state_duration += 1;
15       if (EVE0 & !EVE1) {
16         state = WAIT;
17         state_duration = 1;
18       }
19       else if (EVE0 & EVE1) {
20         state = MIN_TIME_ERROR;
21         error = true;
22       }
23       break;
24     case WAIT:
25       state_duration += 1;
26       if (!EVE0) {
```

```
27            state = INITIAL;
28            state_duration = 0;
29          }
30          else if (EVE1 && state_duration >= MIN_TIME) {
31            state = RESPONSE;
32            pass = true;
33            state_duration = 0;
34          }
35          else if (!EVE1 && state_duration > MAX_TIME) {
36            state = MAX_TIME_ERROR;
37            error = true;
38          }
39          else if (EVE1 && state_duration < MIN_TIME) {
40            state = MIN_TIME_ERROR;
41            error = true;
42          }
43          break;
44        case RESPONSE:
45          state_duration += 1;
46          if (!EVE0) {
47            state = INITIAL;
48            pass = false;
49            state_duration = 0;
50          }
51          break;
52        case MAX_TIME_ERROR:
53          error = true;
54          break;
55        case MIN_TIME_ERROR:
56          error = true;
57          break;
```

```
    default:
      error = true;
      pass = false;
      break;
  }
  monitor_state_time->state = state;
  monitor_state_time->error = error;
  monitor_state_time->pass = pass;
  monitor_state_time->state_duration = state_duration;
}
```

A.2 Copilot Monitors

The monitors generated by Copilot are shown in the following listings.

Listing A.5: The header file for the ACC_R0, ACC_R1_min and ACC_R1_max monitors.

```
extern double drel;
extern double dsafe;
extern double vego;
extern double vset_n;
extern double vset_p;
void handlerpropACC_R0(void);
void handlerpropACC_R1_max(void);
void handlerpropACC_R1_min(void);
void step(void);
void reset_monitor(void);
```

Listing A.6: The ACC_R0, ACC_R1_min and ACC_R1_max monitors.

```
#include <stdint.h>
```

```
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include "fret_acc_all.h"
static double drel_cpy;
static double dsafe_cpy;
static double vego_cpy;
static double vset_n_cpy;
static double vset_p_cpy;
static bool s0[(1)] = {(true)};
static int64_t s1[(1)] = {((int64_t)(0))};
static bool s2[(1)] = {(false)};
static bool s3[(1)] = {(true)};
static int64_t s4[(1)] = {((int64_t)(0))};
static bool s5[(1)] = {(false)};
static int64_t s6[(1)] = {((int64_t)(0))};
static bool s7[(1)] = {(false)};
static int64_t s8[(1)] = {((int64_t)(0))};
static bool s9[(1)] = {(false)};
static int64_t s10[(1)] = {((int64_t)(0))};
static bool s11[(1)] = {(false)};
static int64_t s12[(1)] = {((int64_t)(0))};
static bool s13[(1)] = {(false)};
static bool s14[(1)] = {(true)};
static bool s15[(1)] = {(false)};
static int64_t s16[(1)] = {((int64_t)(0))};
static bool s17[(1)] = {(false)};
static bool s18[(1)] = {(true)};
static bool s19[(1)] = {(true)};
```

```
static bool s20[(1)] = {(false)};
static int64_t s21[(1)] = {((int64_t)(0))};
static bool s22[(1)] = {(false)};
static int64_t s23[(1)] = {((int64_t)(0))};
static bool s24[(1)] = {(false)};
static int64_t s25[(1)] = {((int64_t)(0))};
static bool s26[(1)] = {(false)};
static bool s27[(1)] = {(true)};
static size_t s0_idx = (0);
static size_t s1_idx = (0);
static size_t s2_idx = (0);
static size_t s3_idx = (0);
static size_t s4_idx = (0);
static size_t s5_idx = (0);
static size_t s6_idx = (0);
static size_t s7_idx = (0);
static size_t s8_idx = (0);
static size_t s9_idx = (0);
static size_t s10_idx = (0);
static size_t s11_idx = (0);
static size_t s12_idx = (0);
static size_t s13_idx = (0);
static size_t s14_idx = (0);
static size_t s15_idx = (0);
static size_t s16_idx = (0);
static size_t s17_idx = (0);
static size_t s18_idx = (0);
static size_t s19_idx = (0);
static size_t s20_idx = (0);
static size_t s21_idx = (0);
static size_t s22_idx = (0);
```

```
64 static size_t s23_idx = (0);
65 static size_t s24_idx = (0);
66 static size_t s25_idx = (0);
67 static size_t s26_idx = (0);
68 static size_t s27_idx = (0);
69
70 bool s0_get(size_t x) {
71   return (s0)[((s0_idx) + (x)) % (1)];
72 }
73 int64_t s1_get(size_t x) {
74   return (s1)[((s1_idx) + (x)) % (1)];
75 }
76 bool s2_get(size_t x) {
77   return (s2)[((s2_idx) + (x)) % (1)];
78 }
79
80 bool s3_get(size_t x) {
81   return (s3)[((s3_idx) + (x)) % (1)];
82 }
83 int64_t s4_get(size_t x) {
84   return (s4)[((s4_idx) + (x)) % (1)];
85 }
86 bool s5_get(size_t x) {
87   return (s5)[((s5_idx) + (x)) % (1)];
88 }
89 int64_t s6_get(size_t x) {
90   return (s6)[((s6_idx) + (x)) % (1)];
91 }
92 bool s7_get(size_t x) {
93   return (s7)[((s7_idx) + (x)) % (1)];
94 }
```

```
int64_t s8_get(size_t x) {
  return (s8)[((s8_idx) + (x)) % (1)];
}
bool s9_get(size_t x) {
  return (s9)[((s9_idx) + (x)) % (1)];
}
int64_t s10_get(size_t x) {
  return (s10)[((s10_idx) + (x)) % (1)];
}
bool s11_get(size_t x) {
  return (s11)[((s11_idx) + (x)) % (1)];
}
int64_t s12_get(size_t x) {
  return (s12)[((s12_idx) + (x)) % (1)];
}
bool s13_get(size_t x) {
  return (s13)[((s13_idx) + (x)) % (1)];
}
bool s14_get(size_t x) {
  return (s14)[((s14_idx) + (x)) % (1)];
}
bool s15_get(size_t x) {
  return (s15)[((s15_idx) + (x)) % (1)];
}
int64_t s16_get(size_t x) {
  return (s16)[((s16_idx) + (x)) % (1)];
}
bool s17_get(size_t x) {
  return (s17)[((s17_idx) + (x)) % (1)];
}
bool s18_get(size_t x) {
```

```
  return (s18)[((s18_idx) + (x)) % (1)];
}
bool s19_get(size_t x) {
  return (s19)[((s19_idx) + (x)) % (1)];
}
bool s20_get(size_t x) {
  return (s20)[((s20_idx) + (x)) % (1)];

int64_t s21_get(size_t x) {
  return (s21)[((s21_idx) + (x)) % (1)];
}
bool s22_get(size_t x) {
  return (s22)[((s22_idx) + (x)) % (1)];
}
int64_t s23_get(size_t x) {
  return (s23)[((s23_idx) + (x)) % (1)];
}
bool s24_get(size_t x) {
  return (s24)[((s24_idx) + (x)) % (1)];
}
int64_t s25_get(size_t x) {
  return (s25)[((s25_idx) + (x)) % (1)];
}
bool s26_get(size_t x) {
  return (s26)[((s26_idx) + (x)) % (1)];
}
bool s27_get(size_t x) {
  return (s27)[((s27_idx) + (x)) % (1)];
}
bool s0_gen(void) {
  return ((drel_cpy) >= (dsafe_cpy)) && ((s0_get)((0)));
```

```
157 }
158 int64_t s1_gen(void) {
159   return ((s1_get)((0))) + ((int64_t)(1));
160 }
161 bool s2_gen(void) {
162   return !((drel_cpy) >= (dsafe_cpy));
163 }
164 bool s3_gen(void) {
165   return false;
166 }
167 int64_t s4_gen(void) {
168   return (s1_get)((0));
169 }
170 bool s5_gen(void) {
171   return (((drel_cpy) >= (dsafe_cpy)) && (((s2_get)((0))) || ((s3_get)((0)))))
        && (!(((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy))));
172 }
173 int64_t s6_gen(void) {
174   return (s4_get)((0));
175 }
176 bool s7_gen(void) {
177   return (s5_get)((0));
178 }
179 int64_t s8_gen(void) {
180   return (s6_get)((0));
181 }
182 bool s9_gen(void) {
183   return (s7_get)((0));
184 }
185 int64_t s10_gen(void) {
186   return (s1_get)((0));
```

```
187 }
188 bool s11_gen(void) {
189   return ((s3_get)((0))) || (((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (
      vset_p_cpy)));
190 }
191 int64_t s12_gen(void) {
192   return (s10_get)((0));
193 }
194 bool s13_gen(void) {
195   return (s11_get)((0));
196 }
197 bool s14_gen(void) {
198   return ((!(((((s1_get)((0))) - ((int64_t)(3))) <= ((s1_get)((0)))) && (((((
      s1_get)((0))) <= (((s1_get)((0))) - ((int64_t)(3)))) && ((((drel_cpy) >= (
      dsafe_cpy)) && (((s2_get)((0))) || ((s3_get)((0))))) && (!(((vego_cpy) > (
      vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy)))))) || (((((s1_get)((0))) -
      ((int64_t)(3))) <= ((s4_get)((0)))) && (((((s4_get)((0))) <= (((s1_get)
      ((0))) - ((int64_t)(3)))) && ((s5_get)((0)))) || (((((s1_get)((0))) - ((
      int64_t)(3))) <= ((s6_get)((0)))) && (((((s6_get)((0))) <= (((s1_get)((0))
      ) - ((int64_t)(3)))) && ((s7_get)((0)))) || (((((s1_get)((0))) - ((int64_t
      )(3))) <= ((s8_get)((0)))) && ((((s8_get)((0))) <= (((s1_get)((0))) - ((
      int64_t)(3)))) && ((s9_get)((0))))))))))) || (((((s1_get)((0))) - ((
      int64_t)(2))) <= ((s1_get)((0)))) && (((((s1_get)((0))) <= ((s1_get)((0)))
      ) && (((s3_get)((0))) || (((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (
      vset_p_cpy))))) || (((((s1_get)((0))) - ((int64_t)(2))) <= ((s10_get)((0))
      )) && (((((s10_get)((0))) <= ((s1_get)((0)))) && ((s11_get)((0)))) ||
      (((((s1_get)((0))) - ((int64_t)(2))) <= ((s12_get)((0)))) && ((((s12_get)
      ((0))) <= ((s1_get)((0)))) && ((s13_get)((0)))))))))) && ((s14_get)((0)));
199 }
200 bool s15_gen(void) {
201   return !((drel_cpy) >= (dsafe_cpy));
```

```
202 }
203 int64_t s16_gen(void) {
204   return (s1_get)((0));
205 }
206 bool s17_gen(void) {
207   return ((drel_cpy) >= (dsafe_cpy)) && (((s15_get)((0))) || ((s3_get)((0))));
208 }
209 bool s18_gen(void) {
210   return (!((drel_cpy) >= (dsafe_cpy))) && ((s18_get)((0)));
211 }
212 bool s19_gen(void) {
213   return ((!(((((s1_get)((0))) - ((int64_t)(1))) <= ((s1_get)((0)))) && (((((
      s1_get)((0))) <= ((s1_get)((0)))) && (((drel_cpy) >= (dsafe_cpy)) && (((
      s15_get)((0))) || ((s3_get)((0)))))) || (((((s1_get)((0))) - ((int64_t)(1)
      )) <= ((s16_get)((0)))) && ((((s16_get)((0))) <= ((s1_get)((0)))) && ((
      s17_get)((0)))))))) || (((!((drel_cpy) > (dsafe_cpy))) && ((s18_get)((0)))
      ) || (!(((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy)))))) &&
       ((s19_get)((0)));
214 }
215 bool s20_gen(void) {
216   return !((drel_cpy) >= (dsafe_cpy));
217 }
218 int64_t s21_gen(void) {
219   return (s1_get)((0));
220 }
221 bool s22_gen(void) {
222   return (((drel_cpy) >= (dsafe_cpy)) && (((s20_get)((0))) || ((s3_get)((0))))
      ) && (!(((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy))));
223 }
224 int64_t s23_gen(void) {
225   return (s21_get)((0));
```

```
226 }
227 bool s24_gen(void) {
228   return (s22_get)((0));
229 }
230 int64_t s25_gen(void) {
231   return (s1_get)((0));
232 }
233 bool s26_gen(void) {
234   return ((s3_get)((0))) || (((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (
      vset_p_cpy)));
235 }
236 bool s27_gen(void) {
237   return ((!(((((s1_get)((0))) - ((int64_t)(2))) <= ((s1_get)((0)))) && (((((
      s1_get)((0))) <= (((s1_get)((0))) - ((int64_t)(2)))) && ((((drel_cpy) >= (
      dsafe_cpy)) && (((s20_get)((0))) || ((s3_get)((0))))) && (!(((vego_cpy) >
      (vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy)))))) || (((((s1_get)((0))) -
      ((int64_t)(2))) <= ((s21_get)((0)))) && (((((s21_get)((0))) <= (((s1_get)
      ((0))) - ((int64_t)(2)))) && ((s22_get)((0)))) || (((((s1_get)((0))) - ((
      int64_t)(2))) <= ((s23_get)((0)))) && ((((s23_get)((0))) <= (((s1_get)((0)
      )) - ((int64_t)(2)))) && ((s24_get)((0))))))))) || (((((s1_get)((0))) -
      ((int64_t)(1))) <= ((s1_get)((0)))) && (((((s1_get)((0))) <= ((s1_get)((0)
      ))) && (((s3_get)((0))) || (((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <=
      (vset_p_cpy))))) || (((((s1_get)((0))) - ((int64_t)(1))) <= ((s25_get)((0)
      ))) && ((((s25_get)((0))) <= ((s1_get)((0)))) && ((s26_get)((0)))))))) &&
      ((s27_get)((0)));
238 }
239 bool handlerpropACC_R0_guard(void) {
240   return !(((drel_cpy) >= (dsafe_cpy)) && ((s0_get)((0))));
241 }
242 bool handlerpropACC_R1_max_guard(void) {
243   return !(((!(((((s1_get)((0))) - ((int64_t)(3))) <= ((s1_get)((0)))) &&
```

```
    (((((s1_get)((0))) <= (((s1_get)((0))) - ((int64_t)(3)))) && ((((drel_cpy)
     >= (dsafe_cpy)) && (((s2_get)((0))) || ((s3_get)((0))))) && (!(((vego_cpy
    ) > (vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy)))))) || (((((s1_get)((0))
    ) - ((int64_t)(3))) <= ((s4_get)((0)))) && (((((s4_get)((0))) <= (((s1_get
    )((0))) - ((int64_t)(3)))) && ((s5_get)((0)))) || (((((s1_get)((0))) - ((
    int64_t)(3))) <= ((s6_get)((0)))) && (((((s6_get)((0))) <= (((s1_get)((0))
    ) - ((int64_t)(3)))) && ((s7_get)((0)))) || (((((s1_get)((0))) - ((int64_t
    )(3))) <= ((s8_get)((0)))) && ((((s8_get)((0))) <= (((s1_get)((0))) - ((
    int64_t)(3)))) && ((s9_get)((0))))))))))) || (((((s1_get)((0))) - ((
    int64_t)(2))) <= ((s1_get)((0)))) && (((((s1_get)((0))) <= ((s1_get)((0)))
    ) && (((s3_get)((0))) || (((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (
    vset_p_cpy))))) || (((((s1_get)((0))) - ((int64_t)(2))) <= ((s10_get)((0))
    )) && (((((s10_get)((0))) <= ((s1_get)((0)))) && ((s11_get)((0)))) ||
    (((((s1_get)((0))) - ((int64_t)(2))) <= ((s12_get)((0)))) && ((((s12_get)
    ((0))) <= ((s1_get)((0)))) && ((s13_get)((0))))))))) && ((s14_get)((0))))
    ;
244 }
245 bool handlerpropACC_R1_min_guard(void) {
246   return !((((!(((((s1_get)((0))) - ((int64_t)(1))) <= ((s1_get)((0)))) &&
    (((((s1_get)((0))) <= ((s1_get)((0)))) && (((drel_cpy) >= (dsafe_cpy)) &&
    (((s15_get)((0))) || ((s3_get)((0)))))) || (((((s1_get)((0))) - ((int64_t)
    (1))) <= ((s16_get)((0)))) && ((((s16_get)((0))) <= ((s1_get)((0)))) && ((
    s17_get)((0)))))))) || (((!((drel_cpy) > (dsafe_cpy))) && ((s18_get)((0)))
    ) || (!(((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy)))))) &&
     ((s19_get)((0)))) && (((!(((((s1_get)((0))) - ((int64_t)(2))) <= ((s1_get
    )((0)))) && (((((s1_get)((0))) <= (((s1_get)((0))) - ((int64_t)(2)))) &&
    ((((drel_cpy) > (dsafe_cpy)) && (((s20_get)((0))) || ((s3_get)((0))))) &&
    (!(((vego_cpy) > (vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy)))))) ||
    (((((s1_get)((0))) - ((int64_t)(2))) <= ((s21_get)((0)))) && (((((s21_get)
    ((0))) <= (((s1_get)((0))) - ((int64_t)(2)))) && ((s22_get)((0)))) ||
    (((((s1_get)((0))) - ((int64_t)(2))) <= ((s23_get)((0)))) && ((((s23_get)
```

```
    ((0))) <= (((s1_get)((0))) - ((int64_t)(2)))) && ((s24_get)((0))))))))))
    || (((((s1_get)((0))) - ((int64_t)(1))) <= ((s1_get)((0)))) && (((((s1_get
    )((0))) <= ((s1_get)((0)))) && (((s3_get)((0))) || (((vego_cpy) > (
    vset_n_cpy)) && ((vego_cpy) <= (vset_p_cpy))))) || (((((s1_get)((0))) - ((
    int64_t)(1))) <= ((s25_get)((0)))) && ((((s25_get)((0))) <= ((s1_get)((0))
    )) && ((s26_get)((0)))))))) && ((s27_get)((0)))));
247 }
248 // *label handler*
249 // Function called by Copilot to report a property violation.
250 void handlerpropACC_R0(void) {
251     printf("%s\n", "ACC_R0 has been violated");
252 }
253 void handlerpropACC_R1_max(void) {
254     printf("%s\n", "ACC_R1_max has been violated");
255 }
256 void handlerpropACC_R1_min(void) {
257     printf("%s\n", "ACC_R1_min has been violated");
258 }
259 void reset_monitor(void) {
260     memset(s0, true, sizeof(s0));
261     memset(s1, 0, sizeof(s1));
262     memset(s2, false, sizeof(s2));
263     memset(s3, true, sizeof(s3));
264     memset(s4, 0, sizeof(s4));
265     memset(s5, false, sizeof(s5));
266     memset(s6, 0, sizeof(s6));
267     memset(s7, false, sizeof(s7));
268     memset(s8, 0, sizeof(s8));
269     memset(s9, false, sizeof(s9));
270     memset(s10, 0, sizeof(s10));
271     memset(s11, false, sizeof(s11));
```

```
272     memset(s12, 0, sizeof(s12));
273     memset(s13, false, sizeof(s13));
274     memset(s14, true, sizeof(s14));
275     memset(s15, false, sizeof(s15));
276     memset(s16, 0, sizeof(s16));
277     memset(s17, false, sizeof(s17));
278     memset(s18, true, sizeof(s18));
279     memset(s19, true, sizeof(s19));
280     memset(s20, false, sizeof(s20));
281     memset(s21, 0, sizeof(s21));
282     memset(s22, false, sizeof(s22));
283     memset(s23, 0, sizeof(s23));
284     memset(s24, false, sizeof(s24));
285     memset(s25, 0, sizeof(s25));
286     memset(s26, false, sizeof(s26));
287     memset(s27, true, sizeof(s27));
288     s0_idx = (0);
289     s1_idx = (0);
290     s2_idx = (0);
291     s3_idx = (0);
292     s4_idx = (0);
293     s5_idx = (0);
294     s6_idx = (0);
295     s7_idx = (0);
296     s8_idx = (0);
297     s9_idx = (0);
298     s10_idx = (0);
299     s11_idx = (0);
300     s12_idx = (0);
301     s13_idx = (0);
302     s14_idx = (0);
```

```
303     s15_idx = (0);
304     s16_idx = (0);
305     s17_idx = (0);
306     s18_idx = (0);
307     s19_idx = (0);
308     s20_idx = (0);
309     s21_idx = (0);
310     s22_idx = (0);
311     s23_idx = (0);
312     s24_idx = (0);
313     s25_idx = (0);
314     s26_idx = (0);
315     s27_idx = (0);
316 }
317 void step(void) {
318   bool s0_tmp;
319   int64_t s1_tmp;
320   bool s2_tmp;
321   bool s3_tmp;
322   int64_t s4_tmp;
323   bool s5_tmp;
324   int64_t s6_tmp;
325   bool s7_tmp;
326   int64_t s8_tmp;
327   bool s9_tmp;
328   int64_t s10_tmp;
329   bool s11_tmp;
330   int64_t s12_tmp;
331   bool s13_tmp;
332   bool s14_tmp;
333   bool s15_tmp;
```

```
334    int64_t s16_tmp;
335    bool s17_tmp;
336    bool s18_tmp;
337    bool s19_tmp;
338    bool s20_tmp;
339    int64_t s21_tmp;
340    bool s22_tmp;
341    int64_t s23_tmp;
342    bool s24_tmp;
343    int64_t s25_tmp;
344    bool s26_tmp;
345    bool s27_tmp;
346    (drel_cpy) = (drel);
347    (dsafe_cpy) = (dsafe);
348    (vego_cpy) = (vego);
349    (vset_n_cpy) = (vset_n);
350    (vset_p_cpy) = (vset_p);
351    if ((handlerpropACC_R0_guard)()) {
352        reset_monitor();
353        (handlerpropACC_R0)();
354    };
355    if ((handlerpropACC_R1_max_guard)()) {
356        reset_monitor();
357        (handlerpropACC_R1_max)();
358    };
359    if ((handlerpropACC_R1_min_guard)()) {
360        reset_monitor();
361        (handlerpropACC_R1_min)();
362    };
363    (s0_tmp) = ((s0_gen)());
364    (s1_tmp) = ((s1_gen)());
```

```
365   (s2_tmp) = ((s2_gen)());
366   (s3_tmp) = ((s3_gen)());
367   (s4_tmp) = ((s4_gen)());
368   (s5_tmp) = ((s5_gen)());
369   (s6_tmp) = ((s6_gen)());
370   (s7_tmp) = ((s7_gen)());
371   (s8_tmp) = ((s8_gen)());
372   (s9_tmp) = ((s9_gen)());
373   (s10_tmp) = ((s10_gen)());
374   (s11_tmp) = ((s11_gen)());
375   (s12_tmp) = ((s12_gen)());
376   (s13_tmp) = ((s13_gen)());
377   (s14_tmp) = ((s14_gen)());
378   (s15_tmp) = ((s15_gen)());
379   (s16_tmp) = ((s16_gen)());
380   (s17_tmp) = ((s17_gen)());
381   (s18_tmp) = ((s18_gen)());
382   (s19_tmp) = ((s19_gen)());
383   (s20_tmp) = ((s20_gen)());
384   (s21_tmp) = ((s21_gen)());
385   (s22_tmp) = ((s22_gen)());
386   (s23_tmp) = ((s23_gen)());
387   (s24_tmp) = ((s24_gen)());
388   (s25_tmp) = ((s25_gen)());
389   (s26_tmp) = ((s26_gen)());
390   (s27_tmp) = ((s27_gen)());
391   ((s0)[s0_idx]) = (s0_tmp);
392   ((s1)[s1_idx]) = (s1_tmp);
393   ((s2)[s2_idx]) = (s2_tmp);
394   ((s3)[s3_idx]) = (s3_tmp);
395   ((s4)[s4_idx]) = (s4_tmp);
```

```
396 ((s5)[s5_idx]) = (s5_tmp);
397 ((s6)[s6_idx]) = (s6_tmp);
398 ((s7)[s7_idx]) = (s7_tmp);
399 ((s8)[s8_idx]) = (s8_tmp);
400 ((s9)[s9_idx]) = (s9_tmp);
401 ((s10)[s10_idx]) = (s10_tmp);
402 ((s11)[s11_idx]) = (s11_tmp);
403 ((s12)[s12_idx]) = (s12_tmp);
404 ((s13)[s13_idx]) = (s13_tmp);
405 ((s14)[s14_idx]) = (s14_tmp);
406 ((s15)[s15_idx]) = (s15_tmp);
407 ((s16)[s16_idx]) = (s16_tmp);
408 ((s17)[s17_idx]) = (s17_tmp);
409 ((s18)[s18_idx]) = (s18_tmp);
410 ((s19)[s19_idx]) = (s19_tmp);
411 ((s20)[s20_idx]) = (s20_tmp);
412 ((s21)[s21_idx]) = (s21_tmp);
413 ((s22)[s22_idx]) = (s22_tmp);
414 ((s23)[s23_idx]) = (s23_tmp);
415 ((s24)[s24_idx]) = (s24_tmp);
416 ((s25)[s25_idx]) = (s25_tmp);
417 ((s26)[s26_idx]) = (s26_tmp);
418 ((s27)[s27_idx]) = (s27_tmp);
419 (s0_idx) = (((s0_idx) + (1)) % (1));
420 (s1_idx) = (((s1_idx) + (1)) % (1));
421 (s2_idx) = (((s2_idx) + (1)) % (1));
422 (s3_idx) = (((s3_idx) + (1)) % (1));
423 (s4_idx) = (((s4_idx) + (1)) % (1));
424 (s5_idx) = (((s5_idx) + (1)) % (1));
425 (s6_idx) = (((s6_idx) + (1)) % (1));
426 (s7_idx) = (((s7_idx) + (1)) % (1));
```

```
427   (s8_idx) = (((s8_idx) + (1)) % (1));
428   (s9_idx) = (((s9_idx) + (1)) % (1));
429   (s10_idx) = (((s10_idx) + (1)) % (1));
430   (s11_idx) = (((s11_idx) + (1)) % (1));
431   (s12_idx) = (((s12_idx) + (1)) % (1));
432   (s13_idx) = (((s13_idx) + (1)) % (1));
433   (s14_idx) = (((s14_idx) + (1)) % (1));
434   (s15_idx) = (((s15_idx) + (1)) % (1));
435   (s16_idx) = (((s16_idx) + (1)) % (1));
436   (s17_idx) = (((s17_idx) + (1)) % (1));
437   (s18_idx) = (((s18_idx) + (1)) % (1));
438   (s19_idx) = (((s19_idx) + (1)) % (1));
439   (s20_idx) = (((s20_idx) + (1)) % (1));
440   (s21_idx) = (((s21_idx) + (1)) % (1));
441   (s22_idx) = (((s22_idx) + (1)) % (1));
442   (s23_idx) = (((s23_idx) + (1)) % (1));
443   (s24_idx) = (((s24_idx) + (1)) % (1));
444   (s25_idx) = (((s25_idx) + (1)) % (1));
445   (s26_idx) = (((s26_idx) + (1)) % (1));
446   (s27_idx) = (((s27_idx) + (1)) % (1));
447 }
```