Runtime Monitoring for Safety-Critical Embedded Systems

Submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Aaron Kane

B.S., Computer Engineering, Georgia Institute of Technology

M.S., Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University

Pittsburgh, PA 15213

February 2015

The first step in fixing something is getting it to break.

- Tracy Kidder, The Soul of a New Machine

Abstract

The trend towards more commercial-off-the-shelf (COTS) components in complex safety-critical systems is increasing the difficulty of verifying system correctness. Runtime verification (RV) is a lightweight technique to verify that certain properties hold over execution traces. RV is usually implemented as runtime monitors that can be used as runtime fault detectors or test oracles to analyze a system under test for bad behaviors. Most existing RV methods utilize some form of system or code instrumentation and thus are not designed to monitor potentially black-box COTS components.

This thesis presents a suitable runtime monitoring framework for monitoring safety-critical embedded systems with black-box components. We provide an end-to-end framework including proven correct monitoring algorithms, a formal specification language with semi-formal techniques to map the system onto our formal system trace model, specification design patterns to aid translating informal specifications into the formal specification language, and a safety-case pattern example showing the argument that our monitor design can be safely integrated with a target system. We utilized our monitor implementation to check test logs from several system tests. We show the monitor being used to check system test logs offline for interesting properties. We also performed real-time replay of logs from a system network bus, demonstrating the feasibility of our embedded monitor implementation in real-time operation.

Acknowledgments

First and foremost, I'd like to thank and dedicate this thesis to Stephanie Goldfein for her support and patience putting up with me throughout this entire ordeal. I'd also like to thank my parents, Judy and Jerry Kane for instilling an unending curiousity in me and providing me every opportunity to succeed. I'd also like to thank my brother Gary for his friendship and drive which keeps me on my toes. I also must thank all my friends in Pittsburgh without whom I would not have survived so many winters. I'd like to thank Malcolm Taylor, Milda Zizyte, Felix Huchinson, Jon Filleau, Chris Szilagyi, and Justin Ray for their thoughts, friendship, and necessary distractions throughout this entire process.

I thank my advisor Phil Koopman for his guidance and collaboration. Thank you for helping me get this far. I would also like to thank my thesis committee members Andupam Datta, Andre Platzer, and Chuck Weinstock for their time, feedback, and patience. I owe a special thanks to Omar Chowdhury for helping me get up to speed on the necessary formal math background and guiding me through writing the algorithms and proofs in an acceptable way.

I'd also like to thank the researchers at General Motors R&D and NREC for their feedback and help obtaining systems to test. In particular, I thank Tom Fuhrman, Max Osella, Mike Wagner, and the rest of the ASTAA team.

Finally, I thank my research sponsors for funding this work. This work was funded in part by the General Motors-Carnegie Mellon University Vehicular Information Technology Collaborative Research Lab (CMU VIT-CRL).

Contents

1 Introduction **1**

1.1 Problem and Scope 3

1.2 Thesis Contributions 5

2 Background and Related Work **7**

2.1 Overview 7

2.2 Safety-Critical Systems 7

2.2.1 Safety Standards 8

2.2.2 Hazard Analysis 10

2.2.3 Embedded networks 12

2.3 Requirements and Specifications 13

2.3.1 Safety Cases 15

2.4 System Verification 17

2.4.1 Runtime Verification 17

2.4.2 Monitors 20

2.4.3 Formal Methods 26

2.4.4 Test Oracles 28

2.4.5 Enforcement Techniques 29

3 Monitoring Architecture **31**

3.1 Motivation 32

3.2 Monitor Architecture 34

3.2.1 Use Cases 36

3.3 Test Oracle Example 39

3.3.1 Robustness Testing 40

3.3.2 Target Feature 42

3.3.3 Safety Specification 43

3.3.4 Testing Results and Lessons Learned 48

3.4 External Bus monitoring 49

3.4.1 Observability 51

3.4.2 Sampling Based Monitoring 54

3.5 Semi-formal Monitoring 56

3.5.1 Semi-Formal Interface Design 59

3.6 Usability Concerns 61

3.6.1 Specification Patterns 62

3.6.2 Safety Case Templates 63

4 Formal Monitoring **73**

4.1 Preliminaries 73

4.1.1 Time Model 74

4.1.2 Specifications 75

4.2 Practical Issues 76

4.3 Monitoring Algorithms . . . 79
4.3.1 Definitions . . . 79
4.4 Correctness of the algorithms . . . 90
4.4.1 Definitions . . . 90
4.4.2 Proof of agmon Correctness . . . 93

5 Monitor Implementation **103**
5.1 Implementation Overview . . . 103
5.1.1 Embedded Limitations . . . 104
5.1.2 System Specifications . . . 107
5.1.3 Optimizations . . . 109
5.2 Implementations . . . 113
5.2.1 PC-based Monitor . . . 113
5.2.2 Embedded ARM Monitor . . . 115

6 Monitor Evaluation **121**
6.1 Analysis . . . 121
6.1.1 Artificial Traces . . . 123
6.2 Offline Vehicle Logs . . . 129
6.2.1 Rule Elicitation . . . 130
6.2.2 Monitoring Results . . . 133
6.2.3 Exploratory Example . . . 134
6.3 Embedded Monitor . . . 137
6.3.1 Rule Elicitation . . . 139
6.3.2 Monitoring results . . . 141

6.4 Lessons Learned 145

7 Discussion **149**

7.1 Design Issues 149

7.1.1 Monitor Correctness 150

7.1.2 Monitor Consolidation 153

7.1.3 Semi-Formal Interface 154

7.2 Time Model 156

7.3 Future Work 157

7.3.1 System Recovery 157

7.3.2 Semi-Formal DSL 158

8 Conclusion **161**

8.1 Thesis Contributions 161

8.1.1 Identifying suitable runtime verification architecture 161

8.1.2 Monitoring Framework 162

8.1.3 Feasibility of real-time monitoring 163

A Acronyms **165**

B Specification Patterns **167**

Bibliography **181**

List of Figures

2.1 Principal elements of goal structuring notation, from [1] 16

3.1 External monitor architecture outline 35
3.2 HIL feature instrumentation diagram 41
3.3 FSRACC module IO signals 42
3.4 Monitor architecture showing multi-part specification 58
3.5 Partially instantiated safety case pattern 71

4.1 Aggressive Monitoring Algorithm 85

5.1 Static global formula representation 106
5.2 Example of iterative reduce execution 108
5.3 Comparison of history structure list and interval representations 110
5.4 Hybrid monitor ideal task schedule 119
5.5 Oscilloscope capture of embedded monitor task execution 120

6.1 **agmon** Evaluation loop 122
6.2 Execution time per step for simple formula 124
6.3 Execution time per step of specifications with increasing number of policies 125
6.4 Execution time for formulas with increasing temporal durations 126

6.5 Execution time for nested temporal formulas 126
6.6 Comparison of intervals and lists 127
6.7 Comparison of full and restricted logic 128
6.8 CAN replay setup 138
6.9 Heartbeat counter values over time 142
6.10 Bad heartbeat counter values 144

List of Tables

3.1 Monitor specification for HIL simulator 45

3.2 Specification pattern example 64

3.3 Safety case pattern 70

5.1 ARM hybrid monitor task allocation 118

6.1 Offline log monitoring specification 131

6.2 Offline log monitoring propositions 132

6.3 Propositions for headway specification 136

6.4 CAN replay monitoring specification 139

6.5 CAN replay propositions 141

B.1 Pattern 1.a Bounded Response 168

B.2 Pattern 1.b Bounded Response with Duration 169

B.3 Pattern 1.c Bounded Response with Cancel 170

B.4 Pattern 1.d Bounded Response with Duration and Cancel 171

B.5 Pattern 2.a Conflicting State 172

B.6 Pattern 2.b Conflicting State with Duration 173

B.7 Pattern 2.c Past-Time Conflicting State with Duration 174

B.8 Pattern 3.a No Instantaneous Transition 175
B.9 Pattern 3.b No Transition within Duration 176
B.10 Pattern 4.a Always 176
B.11 Pattern 4.b Guarded Always (Implies) 177
B.12 Pattern 5.a Periodic State 178
B.13 Pattern 5.b Periodic State with Duration 179

Chapter 1

Introduction

Embedded systems, from home appliances to automobiles, are becoming increasingly complex due to the addition of new advanced features. Even traditionally non-critical systems are becoming safety- or mission-critical due to the addition of connectivity, complex autonomy and software reliant control (e.g., X-by-wire [2]. This is a risk both for obviously critical systems such as automobiles as well as more subtly critical systems such as thermostats [3].

As more embedded systems become safety-critical, it is imperative that developers have methods to ensure that these systems are correct. The software engineering methods used to ensure correctness include model-based and code-based methods [4]. We focus on model-based methods in this thesis due to the complexities of modern system integration including multi-vendor subsystems and black-box components.

Traditionally, three main techniques are used for system verification: *theorem proving*, *model checking*, and *testing* [5]. Theorem proving and model checking are formal methods which can provide guarantees about a model of the system. Safety critical systems are gen-

erally real-time systems and most also are distributed due to the redundancy employed to reach their high system reliability requirements. These additional complexities increase the complexity of the system's models which helps lead to the state space explosion problem which affects the scalability of these methods. Testing scales better with increasing system complexity, although it's well known that complete testing is infeasible for high-reliability systems [6]. Since testing does scale better with increasing system complexity, we look to build upon it.

Runtime verification (RV) is a more lightweight method aimed at verifying that a specific execution of a system satisfies or violates a given critical property [5]. Runtime verification can provide a formal analysis while avoiding many of the pitfalls that traditional model-based methods have such as state space explosion and model abstractions. Because RV scales with the trace rather than the system model, it can provide formal assurances on systems with greater complexity. RV essentially provides the scalability of testing with the stronger assurances of formal methods, giving up the guaranteed coverage that other formal methods provide. Typically, runtime verification is implemented in the form of *monitors*, which are devices or programs that observe the behavior of a system and detect whether it is consistent with a given specification [5].

It is important to note that runtime verification is a supplementary technique to other common methods. One way RV techniques can complement traditional methods is by checking that the system satisfies a proven formal model's assumptions and abstractions at runtime [7, 8]. On the supplementary side, runtime monitoring is a useful way to find design and run-time defects which can occur in software, hardware or even system requirements. Even if a systems has a "perfect" specification and implementation, RV detectable faults can still arise from unanticipated operating conditions, maintenance errors, runtime

faults, malicious attacks, and other sources [10].

Though runtime verification is a growing field, there has not been much work focusing on current trends in safety critical embedded systems, especially as industry moves towards more commercial-off-the-shelf (COTS) components and faster design cycles. Being able to verify that a system made up of diverse components from multiple suppliers is safe and correct is difficult even without the lack of design information that is inherent with COTS black-box components. This has led us to the need for a runtime monitor which can check these types of systems. We present an external, isolated bus-monitor for this purpose. By monitoring a system for high level properties at a system network/bus level (i.e., above the black boxes), the monitor can be used to detect system behaviors that violate a specification envelope regardless of the underlying cause (including hardware faults, software bugs and design errors).

This type of monitoring has applications across the system development cycle. During development, runtime monitoring can be used as a debugging monitor or test oracle for system simulation, prototyping, or tests [9]. This type of monitoring ensures testers are notified of all specification violations, even if they are not easily noticable by testers (such as short transient violations). In deployed systems, monitors can be used as a fault detector to trigger recovery mechanisms such as safety shutdowns or switching to degraded performance [11].

1.1 Problem and Scope

This thesis addresses the problem of performing runtime monitoring on a safety critical embedded real-time system. The primary research questions are:

- How do the properties and design constraints of safety-critical embedded systems affect the use of runtime monitoring?
- What monitoring architectures fit the constraints of these systems?
- Can we build a runtime monitor that fits existing constraints and can detect important specification violations?

This thesis focuses on monitoring a distributed system by listening on the system's broadcast bus which connects (potentially black-box) distributed nodes. This design is a common system architecture, especially for modern automobiles and other ground vehicles which are the primary focus of this thesis.

System Requirements Perhaps the most important, and one of the most difficult aspects of safety critical system design is defining the right set of requirements. While obtaining meaningful benefits from runtime monitoring relies on having a good system specification to check, requirements elicitation is beyond the scope of this thesis. This thesis does, however, propose specification patterns which ease the translation from traditional informal system requirements to formal logic-based requirements [12].

System Testing Although this thesis is motivated by the long-term goal of practical real-time monitors performing live fault detection and recovery on deployed systems, much of this work has been performed from a testing perspective. Some of this was influenced by practicality (e.g., restricted system access and ease of analysis), but incorporating runtime verification into system testing has the opportunity to provide very useful benefits on its own [13]. We discuss these benefits by exploring the use of runtime monitors as test oracles and exploratory diagnostic tools for system under development.

Assumptions This thesis focuses on issues directly related to monitoring safety-critical embedded systems. There are many practical engineering issues we assume can be adequately handled in this work. For example, although we use a proven correct monitoring algorithm, we assume that the monitor implementation itself is correct. Similarly, we assume that the data fed into the monitor is trusted (i.e., correct input traces are used). These assumptions are discussed in more detail in Section 7.1.1.

1.2 Thesis Contributions

This thesis makes the following contributions:

- **I have identified a suitable runtime verification architecture for monitoring safety-critical embedded systems**. Safety critical systems have unique constraints which can subtly affect the applicability of runtime verification techniques. The possibility of black-box components adds even more restrictions to system monitoring. This thesis describes a semi-formal external bus monitor architecture which can be used directly as a broadcast bus monitor (and is also amenable to other configurations). This architecture is suitable for system testing as well as runtime fault detection on a live system (albeit with additional considerations) even with black-box components.
- **I provide a monitoring framework based on a formally proven monitoring algorithm and an informal system mapping interface**. I present a full end-to-end framework including our formal view of a system trace, semi-formal techniques to map real systems onto our formal view, a system specification logic, monitoring algorithms, specification design patterns, and a safety case example pattern.
- **I demonstrate the feasibility and show performance characteristics of the given**

monitor framework on several diverse systems. In this thesis the monitor is applied to different systems and specifications, including artificial traces showing performance characteristics as well as traces from real systems under test.

The rest of this thesis is organized as follows. Chapter 2 presents background and relevant related work. The monitoring framework architecture is described in Chapter 3, including our usability assisting patterns. The formal definitions and proofs of our monitoring algorithm are presented in Chapter 4. Chapter 5 describes our monitor implementations. In Chapter 6, the implemented monitors are evaluated against several different systems. We discuss design, trade-off issues and future work in Chapter 7. Chapter 8 presents our conclusions.

Chapter 2

Background and Related Work

2.1 Overview

This chapter describes the relevant background related to runtime monitoring of safety-critical embedded systems. This work relies on a broad set of existing research areas, ranging from traditional fault-tolerance and system safety to the much more theoretical formal methods in runtime verification.

2.2 Safety-Critical Systems

This thesis focuses on safety-critical embedded systems. Embedded systems are systems which include a computer but are not used for general purpose computing. Safety critical systems are systems whose failure can result in loss of life, significant property damage, or damage to the environment [14]. For our purposes, safety-critical systems are a representative example of the more general notion of mission-critical systems, which can have other critical properties relating to correct functionality and lack of failures that compro-

mise the system's ability to correctly perform its primary mission. Such systems generally have failure rate requirements ranging from 10^{-5} to 10^{-9} failures per hour or other suitable time period [15], with reliability encompassing the notion that the system is continuously operational and that it is operating with no functional defects during that time.

Traditional safety critical domains such as the aerospace, medical, chemical processing, and nuclear industries have the benefits of history and strong safety cultures [16] to mitigate the worst risks of incorporating safety-critical software. These domains have been conservative, slowly moving to rely upon software systems due to the difficulty of being able to prove that software-based systems will meet their desired operational reliability requirements. These (and other) industries are integrating additional software control as the advantages in efficiency, decreased costs, faster time-to-market, and advanced features continue to overtake the perceived risks.

Newer non-traditionally safety critical software domains including automotive, home automation, and civil infrastructure don't always have the experience or the safety cultures to help them accurately evaluate the benefits and risks of computer-based controls. Better technologies, processes, and standards that improve or ease the use of software in safety-critical domains are imperative to protect these domains where cost and functionality concerns may put pressure on safety principles. Regardless of the domain, acceptable mission-critical systems are unlikely to be built without good system engineering processes.

2.2.1 Safety Standards

As software safety has become more critical for safe operation, new standards have emerged which attempt to ensure that adequately safe software is used. These standards are increasingly suggesting and requiring formal methods as well as other best-practice development

processes. Standards generally can either be means-prescriptive (prescribing how to satisfy safety objectives) or objective-prescriptive (defining objectives and leaving the methods up to designers) [17]. Different standards may be anywhere from completely means-prescriptive (e.g., IEC 61508) to primarily objectives prescriptive (e.g., DO-178B/C).

There are a few major standards across the common safety-critical embedded system domains. IEC 61508 [18] is a generic standard concerned with improving the development of safety-related electrical/electronic/programmable electronic systems which are safety critical. The standard only deals with functional safety [19]. In aerospace, DO-178B/C [20] is an international standard which relates to civil aircraft. This standard is concerned solely with software. ISO 26262 [21] is an automotive standard for mass production of passenger cars loosely based on IEC 61508. Bowen provides a survey of some older safety standards and their relationship to formal methods in [22].

Safety Integrity Levels Many safety standards have adopted a Safety Integrity Level (SIL) approach to managing risk. SILs are classification levels indicating safety requirements in safety-critical systems [23]. Various standards across industries, such as IEC 61508 and ISO 26262, prescribe using SILs to classify the level of required safety integrity for systems and components. The standards define multiple SIL levels (e.g., SIL0-SIL4) which are used to allocate functional safety requirements to a system/component. Some standards also assign quantitative targets to the SILs such as a probability of failured on demand.

2.2.2 Hazard Analysis

A *hazard* is a system state or set of conditions together with other environmental conditions of the system that can lead inevitably to an accident [24]. Risk is the combination of the likelihood of a hazard leading to an accident and the hazard's consequences. The goal of safety engineering is to reduce risk, either by reducing the likelihood that a hazard will occur or by reducing its consequences. Safety must be designed into a system, so the primary concern of system safety is the management of hazards: identifying, evaluating, managing and controlling them throughout the design process [24].

Hazard analysis provides a structured method for reasoning about system hazards throughout the development lifecycle. Hazard analysis is the foundation of safety engineering – a system's safety case and safety requirements are identified through hazard analysis. We aim to improve system safety by utilizing runtime monitors, but a monitor that checks an incorrect or incomplete specification is useless. The specifications we wish to monitor are derived from safety requirements which are generated by hazard analysis.

Different hazard analysis techniques identify different sets of system hazards, so often many different analyzes are used across the development cycle. We review some common hazard analysis techniques here.

In the early stages of development hazard identification is often termed Preliminary Hazard Analysis (PHA). This is the initial effort to identify safety-critical areas of the system and identify hazards [25]. Identifying potential hazards, roughly ranking or evaluating the hazards severity, and identifying the required hazard mitigation or management techniques are all a part of PHA. The output of PHA is used in the rest of system development including developing system requirements, specifications, and test planning. The hazards and management techniques identified in this analysis strongly influence the system archi-

tecture and safety specification which determine how monitoring can be performed.

Hazards and Operability Analysis (HAZOP) is a qualitative technique for identifying hazards based on deviations from expected operation or state. HAZOP uses guide words to prompt analysts to consider the potential hazards caused by deviations of expected system state. Although HAZOP was originally developed for the chemical industry, it has also been applied to software systems [26].

Failure Modes and Effects Analysis (FMEA) is a forward search technique used to list potential faults in a system and analyze the probabilities of failures. The variant Failure Modes, Effects, and Criticality Analysis (FMECA) includes additional analysis of the criticality of identified failure modes. Software Failure Modes and Effects Analysis (SFMEA) is an extension of the FMEA technique to software, where the analysis is performed to identify cause/effect relationships in which data or software behavior can result in failure modes [27].

Fault Tree Analysis (FTA) [28] is a backward search technique for identifying the fault or failure in a system that is the root cause of a given hazard. Hazards that have been identified by other analysis techniques can be used within FTA to further analyze event combinations that can lead to the identified hazards. Software Fault Tree Analysis (SFTA) is a technique based on FTA used to analyze the safety of a software design [29]. Similar to traditional hardware FTA, SFTAs analyze software based hazards and can identify software properties or values that must be further analyzed and guaranteed to prevent the known hazards.

2.2.3 Embedded networks

Safety-critical distributed systems depend on their bus architectures to be reliable, especially as systems move towards more digital control such as X-by-wire control. Rushby compares four common aerospace and automotive bus architectures in [30].

Network buses are primarily one of two types, *time-triggered* or *event-triggered*. On time-triggered networks the activities (e.g., messages being sent) are driven by the passage of time (e.g., every 50ms send a message). On event-trigged networks, activities are driven by system events (e.g., when a value changes send a message).

Networks designed for use in safety critical systems are commonly time-triggered networks. Some networks are event-triggered but require extra mechanisms to help control network demand. This is because safety-critical systems need to safely handle network contention and bandwith allocation, even in the face of system faults. Time-triggered networks use a static pre-allocation of network bandwith which resolves contention at design time. Event triggered networks for embedded systems use a variety of techniques including priorities and token based methods to perform deterministic contention resolution.

Although time-triggered networks provide many useful properties for safety-critical operation, they are less flexible than event-triggered networks which can handle nonstationary workloads and use bandwith more dynamically. Because the flexibility to dynamically choose what messages to be sending is desirable in cost-sensitive systems, Flexray [31], a protocol designed by the auto industry, includes both a time-triggered and an event triggered portion.

2.2.3.1 Controller Area Network

Controller Area Network (CAN) is a widely used automotive network developed by Bosch in the 1980s. In this thesis we focus primarily on monitoring CAN because it is a common automotive bus which typically conveys a lot of the state we wish to observe. CAN is an event-based broadcast network with data rates up to 1Mb/s (although usually used at 125-500kbps). Messages on CAN are broadcast with an identifier which is used to denote both the message and the intended recipients. The message identifiers are also used as the message priorities for access control.

Although CAN is an event-based bus it is often used in a somewhat periodic, time triggered way so the network usage can be statically analyzed. Because of this our monitoring scheme is based on a time-triggered network sampling model, so it can monitor time-triggered networks as well.

2.3 Requirements and Specifications

In general, system development starts with defining system requirements. In traditional safety-critical design processes such as the Vee model [32], requirements elicitation or capture is performed to create a functional requirements document which describes the functional requirements of the system. Hazard and risk analysis may also be performed to create the system safety requirements which define what constitutes safe and unsafe system operation. From these requirements documents a system specification can be created and used as the basis for the system design.

In a review of the role of software in aerospace accidents, Leveson notes that almost all the identified software related accidents could be traced back to flaws in requirements

specification rather than coding errors [33]. That is, the software performed as the designers intended but the designed behavior was not safe from a system viewpoint [24]. This highlights the importants of identifying and validating the right set of system requirements. Just as building a system with incorrect requirements can lead to accidents, monitoring a system with an incorrect specification will lead to technically correct but practically useless monitoring results.

The way in which requirements are documented also plays an important role in their subsequent use [34]. A variety of formal, semi-formal, and informal languages have been suggested for specifying requirements documentation [35].

Informal specification languages are attractive because they can be easier to read, use, and understand, especially by non-experts. This includes natural language requirements in plain text as well as structured informal specification languages such as UML [36]. The ambiguities inherent in non-formal languages can lead to incorrect requirements which are a common cause of system failures [37].

Formal specifications have many benefits including being unambigious and checkable for syntactic correctness [38]. Formal specifications also enable other formal methods including verification and code generation and they can support testing in many ways, including guiding test selection and oracle generation [13].

Because expressing system specifications in formal languages such as temporal logic can be complex and error-prone, specification patterns have been proposed as a technique to help designers translate informal requirements to formal specification languages [12, 39, 40]. We utilize this idea ourselves, presenting our own specification templates based on these formats to help translate requirements into our specification logic.

2.3.1 Safety Cases

Safety cases, also known more generally as assurance cases, are a common tool that are a part of many different safety standards. Safety cases are "a documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment" [41]. A safety case is comprised of three main elements [1]:

- **Requirements** The safety goal(s) to be acheived.
- **Evidence** The available evidence about this goal.
- **Argument** The structured argument which describes the relationship between the requirements and evidence.

Safety cases are generally used to justify a certain level of confidence that the system will meet its safety (or other) goals with acceptable performance. Commonly, the level of confidence in a safety case is qualitative and somewhat informal. Bloomfield et. al. explore uncertainty in dependability cases including how to express quantitative confidence in a safety case in [42]. They explore confidence from a SIL perspective, judging claimed SIL levels based on beliefs about the system's probability of failure on demand, arguing that showing a higher than necessary SIL level can provide increased confidence in the claim about the (lower) desired SIL level.

Another framework for justifying confidence in the truth of a safety case claim is to use eliminative induction [43], a way to quantify the the confidence in a conclusion based on how many reasons for doubting the claim have been eliminated. Under this framework, rather than attempting to increase confidence in a claim by adding more evidence that supports the claim, the possible defeaters (sources of doubt about the claim) are identified and

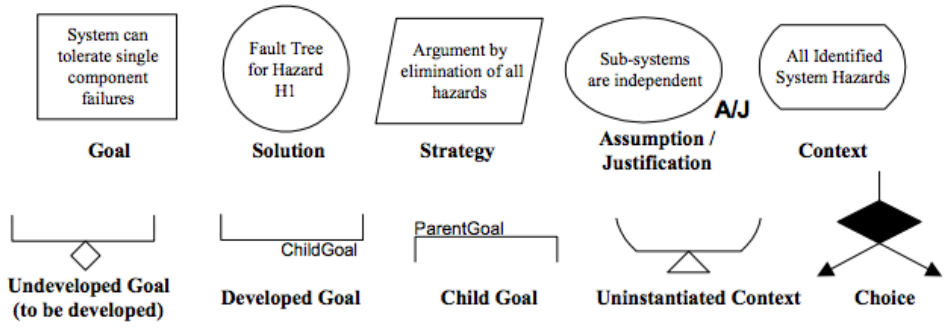


Figure 2.1: Principal elements of goal structuring notation, from [1]

then evidence is used to eliminate the defeaters. The confidence of a given argument can be presented as the Baconian probability $E|D$, with the number of identified defeaters D and eliminated defeaters E [44]. Although these Baconian probabilities cannot be directly compared to each other or reduced in any useful way (e.g., $1|2 \neq 2|4$), they do provide a quantative measure which can represent increasing confidence in an argument (i.e., eliminating a defeater increases confidence). So while not immediately intuitive, this does provide a clear, explicit way to discuss the confidence of an argument.

The Goal Structuring Notation (GSN) is a graphical notation for writing structured arguments and has been widely adopted for writing safety cases [1, 45]. GSN explicitly represents the individual elements (e.g., goals, evidence, context, etc.) and relationships between elements in a graphical form primarily using the elements shown in Figure 2.1.

Kelly introduces *safety case patterns* as a method to improve the problems with informal reuse of safety cases [46]. Safety case patterns include a GSN pattern and an accompanying pattern description. We present our own safety case pattern as a starting point for arguing the safety of an external monitor in Section 3.6.2. GSN has also been extended for modular safety cases to help with the management of more complex systems and safety cases [1].

2.4 System Verification

As systems become more complex, ensuring that the system was built matching the design becomes more and more difficult. Verification is the process of ensuring that the system adheres to its design specification. This thesis focuses on runtime verification, which provides verification about the actual system rather than a model of the system. A precise model of the environment is often not available, whether it is unknown, changing, or too complex to model. However, some system behaviors, especially in embedded systems, are dependent on the system's environment. In these situations, runtime verification can provide more accurate assurances than model checking or theorem proving where a simplified environment model must be used due to scalability issues. Examples include automobiles (e.g., different surfaces, lighting conditions, weather, etc.) and exploration vehicles (e.g., space, undersea, and underground drones/probes) which may encounter unanticipated environments. Incorrect internal environments (e.g., race conditions in multitasking OSes, corrupted RAM, etc.) are also often abstracted away in system models but can easily lead to system faults.

2.4.1 Runtime Verification

Runtime verification can be performed *online* (i.e., at runtime on a live system) or *offline* from recorded log files. We use the general term RV (or monitoring) for both online and offline monitoring.

In RV, a correctness property, usually specified in a formal logic is translated into a monitor which is used to check the target system. Linear Temporal Logic (LTL) [47] is perhaps the most common formal logic for RV. LTL is designed to be checked over infinite

traces, but runtime verification inherently deals with finite traces. This mismatch has been handled in many ways, from defining finite trace semantics for LTL to creating three-valued [48] or even four-valued [49] variants which can represent inconclusive values which occur when looking at finite prefixes of infinite traces.

LTL does not include time bounds, requiring the explicit definition of a clock variable to specify real-time properties. Metric temporal logic (MTL) [50] is a temporal logic similar to LTL with quantitative time bounds for specifying real-time properties. These logics are often propositional, though monitoring of first-order logics is becoming more common [51]. First-order logics are often used when monitoring logs for safety and privacy policies [52, 53, 54]. We utilize a propositional logic since they are more simple to check and embedded systems are generally fully defined at design time, so first-order quantifiers are generally unnecessary.

One interesting aspect of a logic choice is that some specified properties may not be monitorable. *Safety properties*, informally, are those which state that "something bad does not happen" while *liveness properties* state that "a good thing eventually happens" [55]. These notions were formalized based on finite prefixes of infinite traces in [56]. Not all specifiable properties can be monitored, but all safety properties can be. While liveness properties can be interesting for software systems (e.g., no deadlocks, guaranteed response, etc.), we generally want a bounded response to events when monitoring real-time system properties. Bounded liveness properties are essentially safety properties. We restrict our monitor to a bounded MTL which ensures that we only define safety properties in our specifications which guarantees that any monitor specifications we write can actually be monitored. This is a reasonable restriction for monitoring since there is no point in being allowed to create unmonitorable or uninteresting specifications. Checking that a response

eventually (in the infinite future) happens doesn't provide anything useful for RV since we cannot act on properties that take an infinite amount of time to check. One potential use case of more expressive logics is that the same logic and specification may be used for both model checking (where liveness properties are checkable, for example) and runtime verification of the same system.

2.4.1.1 Monitoring Safety-Critical Embedded Systems

Goodloe and Pike present a thorough survey of monitoring distributed real-time systems in [57]. Notably, they present a set of monitor architecture constraints and propose three abstract monitor architectures in the context of monitoring these types of systems. In [58] Pike et. al update these constraints with the acronym "FaCTS": Functionality, Certifiability, Timing, and SWaP (size, weight and power). The Functionality constraint demands that a monitor cannot change the system under observation's (SUO's) behavior unless the target has violated the system specification. The Timing constraint similarly says that the monitor can not interfere with the non-faulty SUO's timing (e.g., task period/deadlines). The Certifiability constraint is a softer constraint, arguing that a monitor should not make re-certification of SUO onerous. This is important because certification can be a major portion of design cost for these systems and nominally simple changes/additions to the SUO can require a broad and costly recertification. Lastly, safety critical systems are often extremely cost sensitive with tight tolerances for additional physical size, weight or required power. Any monitor we wish to add to an existing system must fit within these existing tolerances.

One of Goodloe and Pike's proposed distributed real-time system monitor architectures is the bus-monitor architecture. The bus monitor architecture has the monitor recieve net-

work messages over an existing system bus just like any other system component. The monitor can be configured in a silent or receive only mode to ensure it does not peturb the system. This is a simple architecture which requires few (essentially no) changes to the target system architecture. We utilize this architecture for our monitoring framework. The other proposed architectures require either additional buses or distributed monitors, both which add complexity and costs we wish to avoid when integrating a monitor.

2.4.2 Monitors

There are many existing runtime monitoring frameworks and monitoring algorithms with different primary uses. Monitoring frameworks provide not just the specification language and checking algorithm, but also the connection between the monitor and target system. In this section we highlight some existing monitoring algorithms and frameworks, especially existing algorithms and frameworks with similar approaches to ours (including some of which our framework is based on).

Watterson and Heffernan give an overview of runtime verification tools in [59]. Peters and Parnas present a requirements derived monitor in [60]. They utilize a four variable requirements model which separates the controlled and monitorable state in the specification to help create monitors from the system requirements. EAGLE [61] is a highly expressive logic for monitoring systems upon which other logics including LTL can be built. Because EAGLE is computationally expensive, RULER [62], a similar but simpler logic has also been implemented.

Temporal Rover [63] is a monitor which instruments the target source code with monitor checks. Specification assertions are written as comments in the target source code which is compiled into an instrumented executable which contains an inline monitor.

The NASA PathExplorer project has led to both a set of dynamic programming-based monitoring algorithms as well as some formula-rewriting based algorithms [64] for past-time LTL. These dynamic programming algorithms require checking the trace in reverse (from the end to the beginning) which makes them somewhat unsuitable for online monitoring [65]. The formula rewriting algorithms utilize the Maude term rewriting engine to efficiently monitor specifications through formula rewriting [66].

Thati and Roşu [67] describe an dynamic programming algorithm for monitoring MTL which is based on resolving the past and deriving the future. They perform formula rewriting which resolves past-time formulas into equivalent formulas without unguarded past-time operators and derive new future-time formulas which separate the current state from future state. They store formulas in a canonical form which allows expanding formulas to not grow larger than exponential in the size of the original formula and allows for updating formulas in exponential time. Like many similar algorithms, they store and calculate the necessary histories recursively to evaluate formulas. While they have a tight encoding of their canonical formulas, their monitoring algorithm still requires more state to be stored than some other algorithms (because formulas grow in size as they are rewritten), including the one presented in this thesis. Our monitoring algorithm is also based on formula-rewriting, although we use formula reduction only rather than a full set of rewriting rules. Our algorithm can also be thought of as a dynamic programming algorithm, building up history state from the smallest subformula up to the specification policy.

Basin et. al. describe a set of MTL monitoring algorithms, specifically focusing on how the time domain affects monitoring [68]. They point out that while point-based semantics seem fitting for real-time systems which are often viewed as a set of events, the point-based semantics can be unintuitive compared to interval semantics. Our monitoring algorithm

works in a very similar way to their point-based monitoring algorithm. They use an iterative recursive algorithm which calculates truth values over the target formula structures utilizing history structures. Our **reduce** procedure works similarly to their **step** procedure, except they only check past-time MTL so **step** is always guaranteed to return an answer whereas our **reduce** must handle inconclusive formulas as well.

There are some monitoring algorithms designed to handle incomplete trace information. Bauer et. al. present a policy logic for monitoring transaction logs with partial observability (i.e., not all parameters are observable) [69]. Basin et. al. also present a monitoring algorithm for incomplete logs due to logging failures or disagreeing logs in [70]. Other algorithms also handle log incompleteness [52, 53]. Our monitoring framework does not deal with incomplete logs, except for missing future-time trace entries which we expect to eventually obtain. We require that the entire state is available for all observed log steps.

Copilot is a Haskell-based embedded domain specific language for generating runtime monitors for real-time distributed systems [71]. Copilot specifications can be used to generate constant-time and constant-space C code which include their own scheduler and can be run alongside the program to be monitored. Unlike many of the other discussed monitors, Copilot is designed with distributed safety-critical embedded systems and their constraints in mind. Still, Copilot requires code source code access to instrument the target system (and is designed to run on-chip). This is not usable for black-box systems and common-mode faults between the monitor and target system may also be an issue.

The Monitoring and Checking (MaC) framework [72] is a generalized monitoring architecture which instruments the target program to send the targeted state to the monitor. MaC uses a two part specification which separates the implementation specific details from the requirements specification. The primitive event definition language (PEDL) is used to

specify the low level specification which defines the instrumentation and how the system state is transformed into monitor events. The meta event definition language (MEDL) defines the actual safety rules that get checked. Kim et. al. describe a Java implementation Java-MaC in [73]. Our semi-formal interface is similar to MaC's filters which are used to map the system to the checker's formal model. However, MaC's filters are implemented on the instrumented target system (which requires source instrumentation) whereas our mapping is performed on the monitor.

Monitor Oriented Programming (MOP) is a generalized framework for incorporating monitors into programs. BusMOP [74] is an external hardware runtime monitor designed to be used in verifying COTS components for real-time embedded systems. BusMOP is one of the few existing monitors which targets systems with COTS components (and thus cannot use any instrumentation). The monitor is an automatically generated FPGA monitor that can sniff a system's network (they use the PCI-E bus) to verify system properties. This is an external bus monitor architecture similar to our monitoring framework. BusMOP only supports past-time LTL and extended regular expressions so it cannot perform aggressive checking of future-based properties. BusMOP system mappings are defined directly in VHDL (which is compiled into the monitor) while the safety properties are written in a formal logic. Instead of having each monitor be generated based on its mapping, our monitoring algorithm is software based, so the mapping can be written in system level code (or, eventually in a simple domain specific language).

Reinbacher et. al. present an embedded past-time MTL monitor in [75]. They use a non-invasive FPGA monitor which is generated from the monitor specification. Their architecture is similar to ours, wiretapping the target system interface and passing it through an evaluation unit which creates atomic propositions out of the system state (similar to our

semi-formal interface). The actual implementation they describe does however presume system memory access to obtain system state (rather than using state from the target network). The generated atomic propositions are fed into the runtime verification unit which checks the desired ptMTL properties. This monitor is limited to past-time MTL which means it cannot check fully aggressive future properties unlike our monitor which allows both past and future bounded MTL.

Heffernan et. al. present a monitor for automotive systems using ISO 26262 as a guide to identify the monitored properties in [76]. They monitor past-time LTL formulas (using explicit time bounds when necessary) obtaining system state from target system buses (CAN in their example). They use "filters" as a system interface, allowing them to generate atomic propositions which get fed to the "event recognizer" (i.e., the monitor portion). Our semi-formal interface is equivalent to these filters. Their monitor is an on-chip SoC monitor based on previous work which used an informal hardware logic instead of past-time LTL [77]. The motivation and goals behind that work are very similar to ours, but they use on-chip SoC monitors with instrumentation to obtain internal system state. This is an important distinction since on-chip monitors aren't suitable for black-box systems. There is also the risk of common mode failures when the monitor is resident on the same chip as the target system which we try to avoid.

Our monitoring algorithm is inspired by the algorithms *reduce* [52] and *prècis* [53], adjusted for our aggressive monitoring and propositional logic use case. The algorithm *reduce* is an offline, iterative monitoring algorithm for auditing privacy and security properties (e.g., HIPAA [78] or GLBA [79] requirements) over incomplete logs. It checks a first order logic with restricted quantifiers [52] using an iterative, formula rewriting-based algorithm. Their audit log is a partial structure which maps every ground predicate to ei-

ther true, false, or unknown. They require the entire audit log to be stored and available for monitoring instead of summarizing the history in a structure. Storing the entire system log (i.e., the trace) is not a feasible approach for an embedded monitor because the traces continuously grow and thus can become very large. They also use explicit (quantified) time values rather than temporal logic to handle time-based constraints.

The structure of our monitoring algorithm is based on Garg's *reduce* algorithm. We also use an iterative, formula-rewriting based algorithm (with the primary procedure also named **reduce**), although our algorithm can be used for both online or offline monitoring. Our algorithm works in a similar way, recursively reducing subformula and returning residual formulas for "incomplete" or unreducable traces. We target a bounded propositional metric temporal logic, so we do not need to deal with substitution of quantified variables. We only need propositional logic because embedded systems are typically fully specified at design time (i.e., we know all the possible network nodes and messages). Both algorithms can return residual (i.e., incompletely reduced) formulas, but the reasons for incompleteness of these two algorithms are different. Garg et. al.'s *reduce* can return residual formulas due to unknown predicate substitutions or incomplete logs. Our **reduce** returns residual formulas when the truth value of a temporal formula is currently inconclusive (i.e., depends on future values). Our algorithm only handles incompleteness caused by needing to see future state. We do not consider incomplete traces due to missing information. We use a bounded propositional metric temporal logic rather than a first order logic, so we do not need structures to aid in substitution of quanitified variables. We do use structures to store any relevant state history to avoid storing the entire trace.

The online, iterative monitoring algorithm *prècis* generalizes Garg et. al's *reduce*. The *prècis* algorithm tries to summarize the log history as structures and falls back on *reduce*

style brute force checking when a summary structure cannot be built. Many existing monitoring algorithms are special cases of *prècis*. When it is possible for *prècis* to build structures for all subformulas it performs a typical runtime monitoring algorithm (i.e., checking the stored structure state following the semantics) whereas when it is not possible to build any structures *prècis* works similarly to *reduce*.

prècis performs online, iterative checking of metric first order temporal logic properties. Our overall algorithm **agmon** is based on *prècis*, with a similarly structured algorithm – updating all history structures then checking the desired formulas. Our algorithm **agmon** performs aggressive checking of future-time formulas (attempting to reduce them as soon as possible) while *prècis* delays the checking of future-time formulas until they are guaranteed to be reducable. We do use this delaying tactic in our conservative checking algorithm. We use a simpler propositional logic instead of *prècis*'s metric first-order temporal logic. Instead of storing summary structures for predicate substitutions we keep subformula history structures.

2.4.3 Formal Methods

Formal methods are mathematical based languages, techniques, and tools for specifying and verifying systems [80]. Hardware, software, and combined systems can all benefit from formal analysis. Though in the past formal methods were inadequate for use with real systems [80, 81], incorporating modern techniques can now be of more substantial benefit to designers.

The idea that formal methods could be an important tool in creating critical systems is not new [22, 82]. Safety-critical systems need some argument of correctness since they have the potential to cause costly damage and injuries but it is known that fully testing

systems with high reliability requirements is infeasible [6]. Since we cannot guarantee the behavior of these systems based on testing alone, the traditional approach has been to do it through careful design and analysis. One way to look at the concept of runtime verification is that it extends this analysis to run-time monitoring rather than ending the analysis phase when the product ships.

2.4.3.1 Model Checking

Model checking is a technique that relies on building a finite model of a system and checking that a formally specified desired property holds in that model [80]. If the model does not satisfy the given specification, then a counterexample is provided by the checker, which provides a detailed execution trace that demonstrates why the system model did not satisfy the specification.

RV has roots in model checking, but there are some key differences. First, model checking is used to check *all executions* of a model for the specified property, whereas runtime verification only checks the execution traces that are seen. This makes RV more suitable for black-box systems because model checking requires a precise model (an imprecise model leads to imprecise answers).

The primary hurdle for model checking is the state space explosion problem. Proving a property is true requires evaluating every possible state of the system. Although pruning techniques and identifying state equivalence classes can reduce the number of distinct states that must be evaluated, the number of states that needs to be checked grows quickly as the model gets more complicated. This is an especially big problem when discussing safety-critical embedded systems as they can have complex behaviors and environments that themselves require descriptive models.

Model checking and runtime verification can be used together in a complementary fashion. Models for model checking require a set of simplifying assumptions. Runtime verification techniques can be used to check these assumptions, providing assurance that the system is operating within its model checked proof [7]. ModelPlex [8] formally verifies properties of a system model and synthesizes monitors which validate the target system executions to that model.

2.4.4 Test Oracles

Test oracles are procedures that identify whether a given test has succeeded or failed. In traditional testing, human users, sometimes aided by automatic tools, act as test oracles. Automated oracles can provide more accurate (no mistakes) checking of test results at a faster rate than manual checking if they can be designed to accurately predict correct system behavior in response to test stimuli. The oracle problem is how to create such an automated predictor of system responses, including addressing situations in which such a predictor is impractical [83].

Partial oracles, which are oracles that can sometimes – but not always – correctly decide whether a test has succeeded or failed, can be easier to identify. Runtime monitors can be used as partial test oracles [9], evaluating test logs against a specification which defines the system behavior that identifies a passed or failed test.

In this way, RV can help improve testing in two primary ways. First, automated analysis of test results allows more automated checking which can help improve test coverage. Second, monitors can easily detect violations which may be hard to detect in other ways, including small transient violations which may not affect the system's observed behavior.

2.4.5 Enforcement Techniques

Safety kernels [84] are a technique used to ensure some specified properties of a system (safety properties, in this case). Safety kernels are based on security kernels, which enforce access control policies in information systems. Some of the primary benefits of safety kernels are also goals of our monitoring, namely ensured enforcement of rules regardless of the system implementation and simplicity/verifiability of the kernel (and monitor). Though similar in goals, safety kernels are an enforcement scheme (actively blocking "bad" commands) which is intrusive to the target system, as opposed to monitoring which is reactive (see "bad" state and then react). It is often true that checking is easier than doing (e.g., proofs), and thus we focus on monitoring. Connecting a monitor between system commands and the actuators (or having a monitor which can trigger recoveries fast enough to effectively block outputs) would essentially make a monitor-based safety kernel.

Security automata and runtime enforcement monitors are monitors which act similarly to security/safety kernels, enforcing system properties by ensuring the system trace satisfies a given specification. Different enforcement mechanisms exist, from shutting the system down when the specification is violated to the monitor editing the system requests to ensure the system matches the satisfaction. These monitors are discussed in more detail in [85]. A process algebra and synthesis algorithm to build controllers which mimic these security automata is presented in [87]. JavaMOP, a MOP based monitor for Java programs, has been used to monitor and enforce security policies in Java programs [86].

Chapter 3

Monitoring Architecture

There are many proposed architectures for use in runtime monitoring. Fundamental questions such as where the monitor executes (e.g., external hardware or on-system), what the monitor watches (e.g., memory values, executed instructions, etc.) and how the monitor obtains input (e.g., system instrumentation, external sensors) are dependent on both the properties of the system being monitored and the desired effects of monitoring (i.e., observation or enforcement/control).

Existing runtime monitoring techniques tend to clash with the constraints imposed by safety-critical embedded systems. Most current proposed monitors rely on automatic generation of instrumentation code or generation of the monitor itself (e.g., [58, 65]). This is unusable in black box or external supplier scenarios due to the lack of source code access and has a greater chance of affecting the non-faulty system behavior, especially timing in real-time systems. Instead, this thesis proposes a passive external monitor which only checks system properties that are observable by watching a broadcast network.

This chapter describes our runtime monitoring architecture which is focused primar-

ily on ground vehicles including automobiles and other industrial vehicles. Specifically, our framework is an external passive bus-monitor, primarily targeting vehicular CAN networks, that is capable of checking bus-observable system properties. This type of monitor treats all system components as black-boxes, although more specific system information can be useful to help create more accurate specifications. This is important for commercial systems where some system components will inevitably be supplied by third parties who may or may not provide detailed information about their components. Although we specifically target ground vehicles and CAN in this work, the flexible interface and system model allow other similar types of systems to be monitored with this approach. Systems without broadcast buses may be monitored by exposing the desired system state to the monitor (either through instrumentation or intelligent monitor placement such as network gateways/routers).

3.1 Motivation

The choice of a passive external bus monitor arose from an emphasis on the following ideas/constraints:

1. System components may be black-box components, with no available information except interface specifications (e.g., network message dictionaries).
2. Bolt-on monitoring is preferred for easier development and deployment.
3. Using the monitor should not necessarily require expert specialists (e.g., formal methods, safety, etc.).
4. Overall costs (e.g., extra hardware, power, network, implementation, integration, etc.) should be kept minimal.

The possibility of having black-box components in the system forces us to perform monitoring without source code annotation/instrumentation. This precludes the use of many existing monitoring frameworks and limits the properties that can be checked to externally observable state. Externally observable state is the set of system state that can be obtained on external interfaces such as values exposed in messages on the monitored broadcast bus. Though additional knowledge about the system can be useful when creating the monitor specification, a black-box interface level of knowledge of a system is essentially the minimum necessary to be able to perform useful monitoring and is thus a good baseline for our framework.

A *bolt-on* monitor is a monitor which can be connected to an existing system without requiring significant modification to the target system. This allows the monitor to be connected to systems that were not designed specifically to be monitored (such as existing automobiles). The notion of a bolt-on monitor leads towards reasonable monitor isolation and system independence, which is helpful for functionality assurance and system certification. Although we focus on external bolt-on monitors in this work, more integrated monitors are possible (e.g., on-chip monitor which has it's own processing core). This is discussed further in Section 3.4.

It is common knowledge that formal methods can be intimidating and thus are rarely used in industry [88], although this is improving [89]. This thesis does not explore usability directly, but with this in mind, certain design choices have been made which should ease the use of the monitor. For example, providing both future and past time logic allows the specification to use the most intuitive tense in any given case. The semi-formal interface between the monitor and system provides flexibility which helps reduce the complexity of the monitor and ensures that necessary but complex properties can be specified in our

framework. We provide a set of specification templates (see Section 3.6.1) to help non-specialists create formal specification rules from natural language specifications and an example safety case pattern (see Section 3.6.2) which can be used as a starting point for a real system monitor's safety case.

We have also emphasized a reasonable cost overhead to performing monitoring as a part of our design motivation. The underlying desire is that the addition of a monitor should add a minimal total cost overhead to the system (in terms of SWaP, design time, certification, etc.) so that it is practical for industry to use. This overhead can be balanced in different ways. This thesis focuses on isolation and certification ease at the expense of some SWaP costs but discusses other possible tradeoffs as well.

Because runtime monitors are essentially fault detectors, we look towards two primary applications: testing and run-time recovery. In system development and testing, monitors can be used as test oracles or debugging tools by providing a framework to analyze test results. In deployment, monitors could be used as a fault detector to trigger recovery actions such as degraded performance or safety shutdowns.

3.2 Monitor Architecture

An outline of our monitor architecture is shown in Figure 3.1. The monitor is connected to the system on its system broadcast bus. This bus is connected to the semi-formal interface which observes the bus and generates atomic propositions for the monitor based on the observed bus state, filling the monitor's stored state snapshot. The trace that is formally monitored is a series of these snapshots. The monitor algorithm takes the target specification φ and the trace step generated by the semi-formal mapping σ_i and outputs whether the

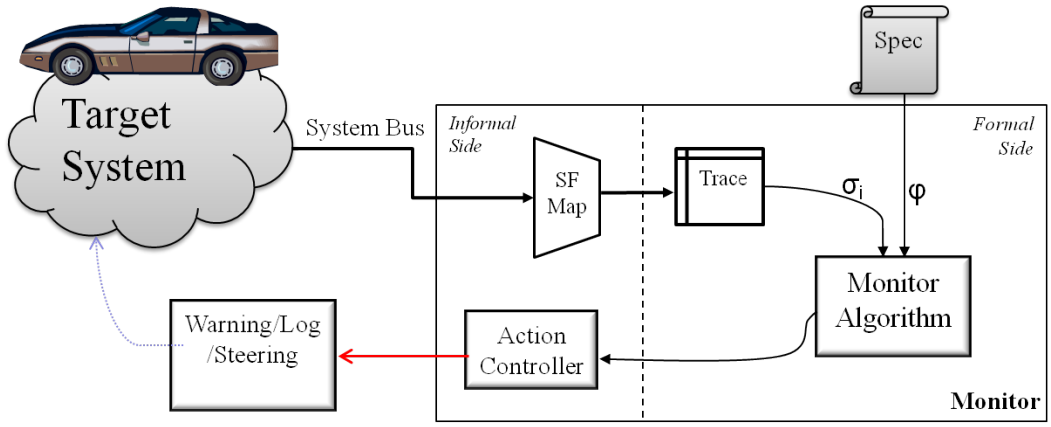


Figure 3.1: External monitor architecture outline

current trace satisfies or violates the given specification. This output is sent to an action controller, which chooses the desired action based on the monitor results. Possible actions include logging violations and activating warnings for passive monitors or triggering a recovery action such as a safety shutdown for more active monitors.

This architecture separates the system independent formal aspects of the monitor from the system dependent components including the semi-formal interface and system configurations. This allows us to utilize a core formal algorithm and framework with any system where an interface map can be used to create a state snapshot. Separating the system dependent and system-independent aspects of the monitor also lets the high level system requirements be somewhat abstracted away from the implementation. This means that changes to the target system may only require changes to the interface configuration and not the high level system specification. This is a similar situation to the two-level specifications used in the MaC framework [73].

3.2.1 Use Cases

The ability for runtime monitors to be semi-formal fault detectors leads to some interesting practical uses. This section discusses our two primary use cases.

3.2.1.1 Test Oracles

Testing is an important part of the system development process. Just as it is difficult to ensure a "finished" system is actually correct, deciding whether a complex system is behaving correctly during a test can be difficult. Some test failures, especially during functional behavior testing, are obvious and easy to see (e.g., an autonomous vehicle driving off the road). But subtle system failures are just as hard to identify during testing as in a finished system. This notion leads us to a simple idea: if monitors are a useful way to detect system faults at runtime, then they are also a useful way to detect system faults during development. Runtime monitors can be used as test oracles since they can check for the desired properties of a test trace [9].

Other aspects of our monitoring framework also lend themselves to being a useful test oracle. External bolt-on monitors can easily integrate with the target system since they do not (by design) require any changes to the target system for integration. Not only is this simpler than integrating inline or other annotation-based monitors, but it also means that changes to the system do not directly affect the monitor. This is important during development as the system may be rapidly changing which could introduce bugs into the monitor if it isn't isolated correctly.

An external monitor can be seen as a diagnostic tool rather than a part of the system that must be kept in sync to continue testing. The monitor being isolated from the target system also allows us to perform essentially any desired testing strategy without adding

integration work. An external monitor can check a system under hardware fault injection or robustness testing just as easily as during fault-less functional integration testing.

Beyond system independence, adding or updating specification rules is simple in our monitor thanks to the semi-formal interface. Being able to quickly update the monitor specification allows the monitor to be used as an exploratory diagnostic tool. The system behaviors can be explored by adjusting the monitor rules to quickly check for interesting system state or behaviors. This is especially useful for exploring the system's actual behavior which can be necessary to tune safety-margins or false positives in the monitor specification.

Because our monitor is designed to target black box systems, we can monitor at different levels of system abstraction depending on the current level of system development. Monitors can be used to check either individual components or a fully integrated system as long as the necessary observability is available. This allows monitors to be used throughout the entire development process regardless of which components are available. Individual components can be tested to ensure they follow their specifications, and as integration begins the partially integrated systems can be monitored as a whole, checking that component interactions don't cause system-level specification violations.

External monitoring only checks for the properties that are specified, not the underlying behaviors that cause them. From the perspective of the monitor all specification violations, whether due to software bugs, hardware faults, or a mistaken requirement, are equivalent. From a system verification standpoint, this is an important distinction from other verification techniques that focus on a formal model or even the system software. Even formally proven "correct" systems can fail due to problems that occur beyond the scope of the model (e.g., hardware data corruption). By checking the actual system state we can minimize the

risk of broken assumptions reducing the validity of our verification.

3.2.1.2 Runtime Fault Recovery

Runtime fault detection is an important aspect of fault-tolerant systems [90]. Most fault-tolerant system techniques rely on robust detection of faults. Fail-stop/silent components [91], which enable many fault-tolerant techniques, rely on a detection mechanism which can ensure that faulty messages do not propogate. Using fast failure detectors can even speed up fault-tolerant algorithms such as atomic broadcast and concensus [92].

As an abstract idea, monitors seem like a perfect fit to provide fault detection for systems or components, especially if used to trigger component recovery or shutdown. Isolated monitors even benefit from their independence from the system, since faults in a component won't also affect the monitor which is supposed to detect them, which is an assurance that inline monitors cannot provide.

Simple isolated monitors seem like a natural extension of simpler existing methods for protecting systems at runtime such as bus guardians [93]. Unfortunately, system recovery in a general sense is a hard problem. Systems or components which have simple generalized safety-shutdowns or systems for which a backup/fail-safe component is available naturally provide simple fault-recovery actions (e.g., shutdown or switch to backup). More complex systems run into the steering problem, where the necessary recovery action in a given situation may not be clear or commandable given a particular residual non-faulty portion of the system. Complex system recovery is discussed more in Section 7.3.1

While recovery in general is beyond the scope of this thesis, we can clearly see that monitors have the potential to perform complex fault detection, even of emergent system properties. It is possible that the existence of strong fault detectors can lead towards systems

incorporating simple recovery mechanisms.

3.3 Test Oracle Example

This section contains work previously discussed in [9]. In [9] we utilized a prototype monitor as a test oracle for a dSPACE hardware-in-the-loop (HIL) simulator at a commericial automotive research lab. The monitor was an informal prototype which did not utilize the monitoring algorithm and formal specifications described in this thesis. This earlier work provided experience about some difficult aspects of using monitors as test oracles for realistic systems and informed some of our future design decisions.

The simulated vehicle we tested was a prototype development platform for semi-autonomous driving features including full speed-range adaptive cruise control (FSRACC), automated lane keeping and emegency collision avoidance. Because feature development was still in progress we were only able to test the FSRACC features. We performed Ballista-style robustness testing [94] on the simulator to exercise the system and attempt to trigger faults that the monitor could identify. During and after this testing we learned that the simulator features, including the FSRACC, were not hardened for robustness and therefore our findings do not directly reflect upon the quality of production-grade features. However, the FSRACC did provide a prototype-quality realistic automotive feature for our testing purposes.

The simulator used MATLAB Simulink models to generate the code for individual electronic control units. CarSim [95] was used to provide the simulated vehicle and environment which the vehicle models on the HIL operated within. The simulator used the dSpace ControlDesk interface software to manage loading models and running tests (in-

cluding calibration, logging/measurements, and diagnostic access). ControlDesk includes a library (rtplib) allowing real-time scripting access to the models running on the HIL. We used this library to create some robustness testing scripts, and additionally used ControlDesk's control panel functionality to control the manual injection of test values into some individual signals. All logging was performed using ControlDesk's trace capture functionality.

To facilitate the black box interception and injection of vehicle network messages for robustness testing purposes, we added some instrumentation to the vehicle feature model as shown in Figure 3.2. These modifications were solely for input interception/injection to elicit system failures, and did not provide instrumentation for runtime monitoring. Moreover, they did not involve modification of the FSRACC code itself. Each input signal to the FSRACC module was routed through a test-controlling multiplexor which enabled the injection of test values (provided as an injection value) through an injection enable signal. This allowed us to have each input signal individually passed-through or overwritten by the chosen injection signal. These additional signals (the injection and enable signals) were accessible through ControlDesk's layouts and our rtplib-based Python scripts in a similar manner to the existing model signals. Because this part of the system was a simulation running in a fast HIL computer, the modifications did not affect system timing.

3.3.1 Robustness Testing

The primary goal of robustness testing the simulator was not necessarily to test the simulated vehicle's robustness directly, but to increase the chance of faults to better exercise the monitor. We tested the simulated vehicle mostly while in FSRACC following (with manual steering). During the testing we injected type-correct values into the target signals, which

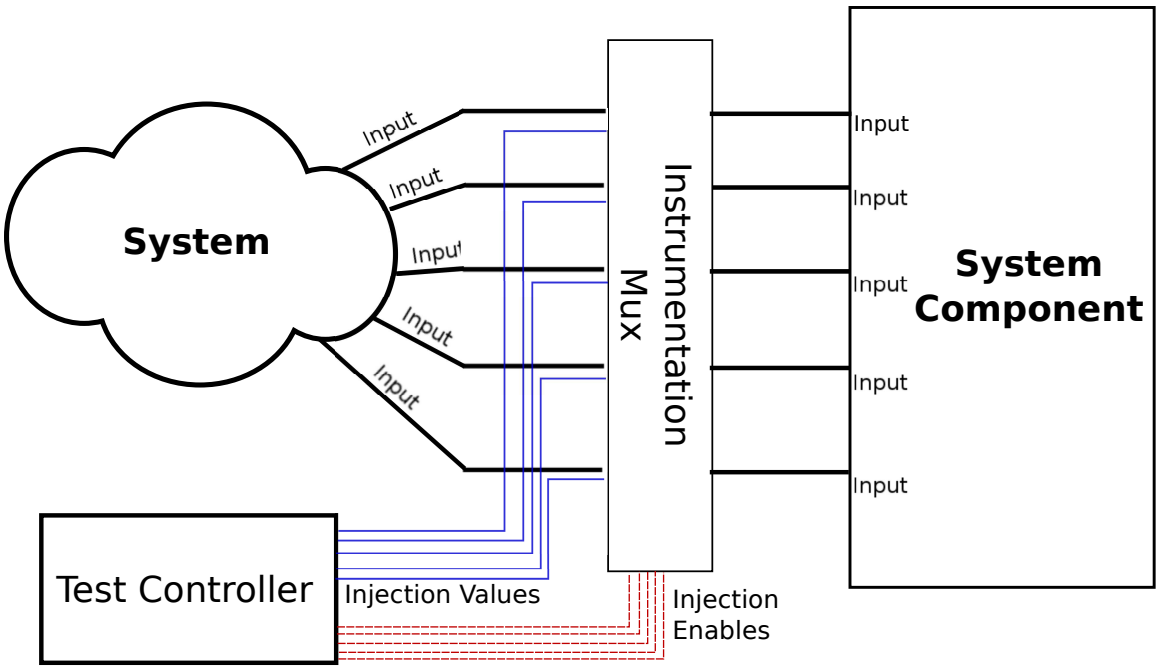


Figure 3.2: HIL feature instrumentation diagram

were either floating point numbers, booleans, or enumerations (positive integers).

We performed three different classes of robustness testing on the HIL: random bit flips (one, two, and four bits), random value injections, and exceptional value injections. For each testing type we injected a particular number of faults each for $20s$ (to allow time for the fault to manifest into a specification violation). Bits to flip were randomly chosen for each individual bit flip fault. For random value injection we injected values from $[-2000, 2000]$ for floats, $[0, 1]$ for booleans, and $[0, maxint]$ for enums. The float range was chosen such that it would go beyond the usual non-faulty values of the target messages while keeping the range small enough that at least some values chosen would land in the value's normal range. For Ballista style testing [94] we performed exceptional value injection which targeted float-typed messages with values chosen from the set {NaN, ∞, $-\infty$, 0.0, -0.0, 1.0, -1.0, π, $\frac{\pi}{2}$ $\frac{\pi}{4}$, 2π, e, $\frac{e}{2}$, $\frac{e}{4}$, $\sqrt{2}$, $\frac{\sqrt{2}}{2}$, $ln(2)$, $\frac{ln(2)}{2}$, 4294967296.000001, 4294967295.9999995,

I/O	Name	Type
Input	`Velocity`	float
Input	`AccelPedPos`	float
Input	`BrakePedPres`	float
Input	`ACCSetSpeed`	float
Input	`ThrotPos`	float
Input	`VehicleAhead`	boolean
Input	`TargetRange`	float
Input	`TargetRelVel`	float
Input	`SelHeadway`	float
Output	`ACCEnabled`	boolean
Output	`BrakeRequested`	boolean
Output	`TorqueRequested`	boolean
Output	`RequestedTorque`	float
Output	`RequestedDecel`	float
Output	`ServiceACC`	boolean

Figure 3.3: FSRACC module IO signals

4.9406564584124654e-324, -4.9406564584124654e-324}. Random injection values were used for exceptional-input injection when targeting non-float data types due to the strong value checking enforced on the HIL testbed.

3.3.2 Target Feature

The inputs and outputs of interest on the FSRACC module are listed in Figure 3.3. The module has other inputs and outputs that were disregarded for testing because they were uninteresting for this type of testing. For example, they had no observable effect, immediately cancelled cruise control, were interface indicators, or otherwise did not affect vehicle safety.

The `Velocity` input message is the forward speed of the vehicle. `AccelPedPos` gives the position of the accelerator pedal as a percentage (0 being fully released, 100 fully

depressed). The pressure applied to the brake pedal is given in `BrakePedPres` and the position of the throttle as a percentage (i.e., how open the throttle is) is `ThrotPos`. The commanded cruising speed is sent in the `ACCSetSpeed` message. The `VehicleAhead` message tells the ACC module whether a vehicle is detected ahead of it in the lane. `TargetRange` and `TargetRelVel` are the distance between the vehicle and the vehicle ahead of it (if one exists) and the relative velocity between those two vehicles respectively. The selected headway distance to the preceding car is an enum `SelHeadway`.

The output `ACCEnable` is whether the ACC thinks it is supposed to be in control of the vehicle or not (i.e., engine and brake controllers should ignore these output values if ACC isn't enabled). The `BrakeRequested` output is true when the ACC feature is requesting a deceleration. If the `BrakeRequested` output is true, then the `RequestedDecel` is a requested deceleration in m/s^2 for the brake controller to attempt to provide. If instead the message `TorqueRequested` is true then the `RequestedTorque` output is the additional amount of torque the engine controller should attempt to provide. The `ServiceACC` message is an error message used to enable an interface indicator to alert the driver that the feature has detected an error.

3.3.3 Safety Specification

To evaluate the use of these techniques we created partial behavioral specifications that were motivated by ensuring system safety. We used six safety rules that checked a mix of system robustness and functionality.

Since the feature under test is a third party provided code module designed mainly as a placeholder function to support early system integration, there was no available specification. Instead we created a set of specification rules based on "expert" elicited common

sense (i.e., properties a knowledgeable engineer would expect to hold based on automotive domain experience) through discussions with the system's engineers and looking over existing system metrics and other related documentation. While we would have preferred to have rules directly derived from system documentation, this is not always possible in industry (as in this example). In cases like this the usefulness of the monitoring results depend heavily on the experts and the quality of the rules they choose. Though expert derived rules may not provide as clear a notion of monitoring coverage, they can be made with the expert's direct needs in mind. It is important to note that partial coverage is still better than no coverage. For example, while the rules we checked in this section are in no way complete, they would be high priority (e.g., likely to lead to vehicle collisions) for a production quality feature.

The six rules we checked against the robustness testing traces are shown in Table 3.1. Because these rules were picked without any knowledge of the internal control algorithms or design parameters of the system, some of them may be too strict (as turned out to be the case). It seems likely that this sort of approach would be common when applying runtime monitoring to real-world systems, which often have incomplete specifications and opaque internal operation. Thus, the approach we took was to adopt these (potentially strict) rules and then relax them when false positives and uninteresting violations were found. We think this is a reasonable approach to employing runtime monitors in practice and also helps to build insight into the the system's operation as well. This is using the monitor as a diagnostic tool.

The issue of whether the data required to implement the monitor would be observable was simple since we were able to create our rules with knowledge of this restriction, writing rules based on system state available on the CAN bus. Observability would be a more

Table 3.1: Monitor specification for HIL simulator

Rule #	Informal Rule *Motivation*
0	If the `ServiceACC` signal is true, then `ACCEnabled` must be false. *A simple consistency check to ensure that the feature does not continue to attempt to control the vehicle when it knows something is wrong.*
1	If the actual vehicle headway time is below 1.0s, then it must be recovered to greater than 1.0s within 5s elapsed time. *This rule is derived from an existing headway metric for another similar system.*
2	If `TargetRange` is less than half the desired headway, then `RequestedTorque` should not be increasing. *Check for feature trying to increase speed when it is already too close to the target vehicle.*
3	If `Velocity` is greater than `ACCSetSpeed` and `RequestedTorque` is less than 0, `RequestedTorque` must still be less than 0 in the next timestep. *Check for vehicle attempting to increase speed when already above the set speed, avoiding tripping on control oscillations by only checking after there are no active requests.*
4	If `Velocity` is greater than `ACCSetSpeed` then `RequestedTorque` must not be increasing at some point within 400ms. *Similar to #3: if vehicle velocity is increasing while above set speed, should start slowing down (or at least hold speed) within 400ms.*
5	If `BrakeRequested` is true then `RequestedDecel` must be less than or equal to 0. *Checks if the value of a requested deceleration is in fact a deceleration (negative).*
6	If `VehicleAhead` is true and `TargetRange` is less than 1, then `TorqueRequest` must be false or `RequestedTorque` must be less than 0. *Checks for near collisions – assuming a feature should not be requesting a increase in speed when the target vehicle is extremely close.*

difficult issue when deriving rules from system requirements which may include requirements on properties that are not externally observable. We discuss this further in Section 3.4.1.

Coverage of the safety rules was not intended to be complete. Rather, the idea is to express a set of safety rules that are useful and see if runtime monitoring detects rule violations for a black-box system which has not been augmented with additional monitoring information.

3.3.3.1 System Interface

The semi-formal interface mapping required to create the propositions in this situation was straightforward. We needed to convert boolean values to truth propositions, compare floating point integers against thresholds, and check whether values were increasing between samples. Although the direct mapping required between the system and the monitor specification was simple, we encountered other system mapping issues including state synchronization across multiple message periods and discrete value jumps.

Mutiple Message Periods The first issue was handling multiple message periods. In the vehicle we tested there were two relevant message periods, with some messages being updated four times slower than the others. At first we simply assumed that these slower values stayed constant between updates. But, dealing with values across multiple timesteps required more care when trying to obtain it's change over time, because a slowly sampled value that is in fact increasing would appear to be unchanging for several cycles while the faster samples were being checked.

As an example, to see if the FSRACC feature was requesting increasing torque, we

would calculate the difference of the previous and current `RequestedTorque` value. However, if the held value is used in a monitor that updates four times between every `RequestedTorque` update, the torque would appear to be constant for three samples out of four due to the repetition of the most recent sampled value being held. Additionally, jitter would sometimes cause slower-period messages to be delayed, resulting in five faster frequency message updates occuring between the slower message updates. Once recognized, it was relatively simple to work around these problems in an ad hoc manner by checking whether these values increased over four timesteps instead of one. The observation remains that runtime monitoring that involves data sampled at different periods can be tricky, and a runtime monitoring architecture should have a uniformly applied mechanism to deal with that issue.

Discrete Value Jumps Another network value issue that we came across was ensuring that rules could handle message transitions from non-active to active. Some messages in a system, such as `TargetRange` can perform large discrete jumps when they are activated even though they represent continuous physical properties. As an example, `TargetRange` is 0 when there is no target being tracked, but once a target is found this value will immediately jump to the actual range. This was noticed for rules that checked if the ACC would command control when the change in `TargetRange` did not agree with the sign of `TargetRelVel`. While these values should always agree in a non-fault condition, there is one normal situation where they may not: when a vehicle comes into sensor view the relative velocity may be correctly reported as negative, but the first change in range seen is necessarily positive (change from zero to the actual positive range). Delaying the check of such a rule until after the activation (allowing the "change" variable to

initialize before testing) avoids this problem.

Other message or rule types may also have initialization issues, such as rules that rely on an integrator or running average of a value. A general observation is that run-time monitors should have a uniform way of "warming up" monitors for data that changes state abruptly, especially when changing from invalid to valid, to avoid false alarms.

3.3.4 Testing Results and Lessons Learned

For each of the either target signals we ran three different types of robustness tests – Ballista style injections, bit flips, and random value injections. All three test types found similar violations which we attribute to the lack of robustness of the target system (any value mangling could cause the type of bad input values that caused the violations). The monitor identified violations of all the specification rules except Rule #0.

Some of the violations turned out to be overly strict specification rules. One example of this is most of the violations of Rule #5 which was often caused by control system overshoot causing a single cycle positive acceleration when the brakes were released. Similar single-cycle violations which may be legitimate violations were also identified showing the usefulness of an automated test checker for finding transient specification violations. Without thorough documentation we cannot be sure whether the system should contain short violations of this type or not.

The process of performing robustness testing with a monitor-based test oracle led us to identify and discuss three major research challenges. The first, *intent estimation* is the challenge of approximating or representing desired system intent based on the observable system properties. Intent estimation is discussed in Section 3.4.1.1.

The other two challenges identified were identifying the necessary language features

required for efficiently specifying a desired system property and how to map the target system onto the monitor's system model. In this thesis both these challenges are handled by the semi-formal interface. We use a bounded propositional temporal logic as a specification language which keeps the monitoring complexity down, but there are many system properties that are not directly expressible or cumbersome to express in this logic. Instead of choosing a more expressive language (which would increase the difficulty of monitoring), we have supplemented the logic by allowing a configurable interface to fill the propositions that are used in the logic. This provides flexibility to build the propositions, which protects the monitor from being unable to specify a new property that does not fit within the specification language. The system mapping is handled through the semi-formal interface in the same way. Again, the semi-formal interface provides the ability to map many different target systems into our monitor model.

3.4 External Bus monitoring

Although external monitoring follows naturally from targeting black-box systems, the isolation from the system provided by an external monitor has other benefits as well. Isolation between a monitor and its target SUO is a simple way to ensure that the monitor fits within the known constraints of safety-critical system monitoring. Isolation in general reduces the amount of possibly dangerous interactions that can occur between the monitor and the system. Specifically, using an external, isolated monitor simplifies the argument that a monitor fits the FaCTS constraints discussed in Section 2.4.1.1, especially the timing and functionality constraints. Notably, isolation provides a straightforward argument that the addition of the monitor does not make the system less safe and thus any benefits the monitor pro-

vides are strictly gained benefits. This isolation will be utilized heavily in our safety case pattern instantiation in Section 3.6.2.

Arguing that an external passive monitor will not affect the functionality or timing of the SUO is straightforward, since isolation by definition precludes any interactions. Only shared resources or potentially weakly isolated components need to be considered as potential problems. External monitors also may be independently certifiable, allowing a modular certification [96] approach. Because the monitor is isolated (e.g., by ensuring it never sends a message), there is little coupling or interactive complexity with the target system, making independent certification straightforward. Having changes in either the system or monitor be independent of each other with regards to certification provides flexibility in the development cycle. For example, with a certified working monitor, last minute changes to the system functionality do not affect the certified correctness of the monitor. This is especially useful for research or prototype systems where changes may occur later in the development process than is usual for production systems.

Size, Weight, and Power. The downside of using an external monitor is that it requires extra resources (in general, more hardware). External monitors do somewhat contradict the goal of minimizing the size, weight, and power contributions of the monitor. There is a fundamental tradeoff between spending additional resources to obtain better isolation or using more integrated resources to save costs. As with redundancy for fault-tolerance, if the system needs strongly guaranteed isolation for assurance purposes, then there is no avoiding the extra SWaP costs. The necessary level of isolation and thus extra SWaP costs are system and situation dependent, but with hardware costs and size/power requirements continuously decreasing the costs of isolation are constantly improving.

There may be situations where making room for extra hardware resources is impossible (e.g., space probes with extreme weight constraints), but for systems where a strong guarantee of isolation is valuable it cannot be obtained in any other way. For systems where full isolation is not worth the additional SWaP costs, this same framework can be applied in a consolidated architecture on a single Electronic Control Unit (ECU) or network gateway with existing system functionality. The monitor will still be performing external monitoring of the system, just from within an existing component. This can reduce the SWaP overhead and provides the same monitoring output but may require more effort to show monitor correctness due to possible interference on or from the shared system. Our monitor design is based on a fully external monitor so it can be fully bolt-on and to simplify the safety case. This isolation/consolidation tradeoff is discussed more in Section 7.1.2.

3.4.1 Observability

External monitoring of a SUO is limited by the external observability of the system. The *external observability* of a system is the set of system state that can be obtained from outside the system. For example, internal software variables of a component are not externally observable, while values sent out in network messages are.

Obviously, a monitor can only check specifications over properties that it can observe. Passive external monitors are restricted in that they only have access to system properties that are already externally observable. This is in stark contrast to less passive and on-chip (internal) monitoring approaches where it is possible to instrument a system or perform inline monitoring directly on internal system state. Although being limited to observable state is a seemingly tight restriction, broadcast buses tend to have useful observable state such as component status messages and feature requests.

The external observability of a system is an extremely system-dependent property. While a certain amount of information is likely to be available for any given system domain due to common designs or standards (e.g., ground vehicles are likely to at least have the common J1939 messages [97]), there is no guarantee that any given system will have any certain observable properties. Some other solution needs to be found if an unobservable property is required to check a desired specification. In some cases it may be possible to calculate a proxy value from observable values. Imagine a sensor system which detects obstructions ahead of a vehicle and provides a distance to the target object but no relative or actual velocity of the target. In this case a target velocity can be derived from consecutive samples of the distance to the target and the host vehicle's velocity. More intrusively, the system could be modified to expose the desired property, either by instrumenting the system components themselves or by adding additional sensors connected to the monitor. When possible, architecting externally visible state into the system is a general solution to the observability problem. This does require identifying the necessary state that needs to be exposed but allows monitoring many different properties. The MaC framework's instrumentation works in this way, instrumenting the target program to expose the monitored values. If no feasible instrumentation solutions exist, then the problematic requirements may be deemed externally unmonitorable for the existing architecture.

In this thesis we work within the constraints of our target system's observability without making any changes to the system. While some requirements are not monitorable given this restriction, we can often find a way to check useful or related properties. This is less of an issue if the monitor design is a part of the system design since necessary properties can be purposefully exposed.

3.4.1.1 Intent Estimation

A subtle problem that can arise due to observability issues is the *intent estimation* problem. This is the question of how to represent a system or feature's intent to perform some high level action based on an observable set of lower level properties [98]. This problem has been explored in many domains including defense aerospace [99], unmanned undersea vehicles [100] and automobile driver intent [101].

System observability bleeds into the problem of intent estimation directly. System specifications, especially informal specifications, often utilize the intent of a system without providing a directly observable proxy or behavioral definition. For example, during monitoring of a prototype adaptive cruise control feature under robustness testing (see Section 3.3) we had a desired informal specification rule: "If the vehicle's velocity is greater than the cruise control set speed, then cruise control should not request an increase in velocity". Since the cruise control component only output requested torque to the engine controller, we had no way to directly look for a request for increased velocity. We instead used an increase in requested torque as an estimation for the ACC feature intending to accelerate the vehicle. This was a reasonably conservative and causal estimation (increasing engine torque generally increases vehicle speed) but it is not perfect as the necessary torque to even sustain a given vehicle speed depends on a host of factors such as road conditions and grade. Besides road conditions, instantaneous torque requests do not tell the whole story about vehicle state due to system inertia (duration and amplitude of the torque requests are important as well).

In this type of case, we can imagine the desire to implement a dynamics model in the monitor to attempt to create a more accurate model of the ACC feature attempting to accelerate the vehicle. This is possible but goes against our motivation from runtime

verification of performing simple checking to reduce the reliance on abstract models. We still do allow utilizing moderately complex system models in the semi-formal interface, but integrating complex models increases the risk that the monitoring results are undermined by an incorrect model.

An important factor in choosing a useful estimation is understanding the abstraction being used and how that affects the false positive and false negative rate of the monitor output. For many cases, estimations that limit false negatives are ideal since they at least preserve the detection ability of the monitor.

Intent estimation can be a problem while monitoring white-box systems as well. Even though having access to a feature's internals gives a more inclusive view of a system's process, intent can be an emergent property that does not appear in any given system variable. Similar estimation solutions (finding acceptable estimations/proxy values) can be applied to the white-box system case, albeit with more options of existing state to choose from and usually a better understanding of what intent means.

3.4.2 Sampling Based Monitoring

Our monitor framework is a sampling-based time-triggered monitor. The monitor samples the system state at a constant frequency and checks the specification against this series of sampled state snapshots. This allows the monitor to be used to monitor both event-triggered and time-triggered system architectures in a straightforward manner with predictable overhead and no issues with event bursts [102].

Systems which are not inherentely time triggered can be monitored in this model by using observed system events to fill a system state model which can be frozen and copied when the monitor needs a snapshot. For example, when monitoring a CAN bus the mon-

itor's system state view is updated by every relevant incoming message. At each monitor checking step (i.e., periodically) a snapshot of the current system state can be taken and used as the system's state at that step. This allows the live system state view to continue to be updated while the monitor is checking a given step.

Time-triggered architectures naturally fit within this framework, but event-triggered system architectures require a bit more care to ensure that the monitor is checking what users expect it to be checking. The monitor's checking period (or sampling frequency) must be fast enough to ensure that system behaviors which need to be distinguished from each other can be. System events (changes in state) which occur between monitor checks cannot be ordered or distinguished. This means that the sampling period must be twice as fast as the fastest monitored event to avoid aliasing which could lead to inaccurate monitoring answers. It should be noted that this is only with regards to values that affect the monitored specification and need to be distinguished in time. Even if the system has state which updates at 10ms, if all the specification-related values only update at 50ms, we can safely check at 25ms, not 5ms.

Because we monitor the target system's observable state, we avoid synchronization issues in general by basing the monitor specifications on observable system state rather than internal system state. For example, we may monitor that a given vehicle state obtained from a status message (e.g., a button press) requires some response message within a bounded time. Note that this is a requirement on the bus, the messages themselves, not the internal system state. The desired timing of these state updates and requests on the bus is known and can be encoded in the specification. Time-triggered system buses such as Flexray and Time-Triggered Protocol (TTP) provide their own global time synchronization, so when monitoring on these types of buses we are given more timing guarantees. Even though

CAN is an event triggered bus, by sampling the bus observed state fast enough we can ensure that we obtain a true view of the state of the system (as far as the bus is concerned). This highlights that the safety margins, not only on property values but also on temporal rule durations, need to be chosen to ensure that the monitor is checking stable system state.

3.5 Semi-formal Monitoring

Using formal methods has some clear benefits such as removing ambiguity, enabling early defect identification and allowing automated checking of properties. Even so, industrial adoption of formal methods, especially in software and system engineering has been slow (with hardware design being an exception) due to well known challenges such as a lack of tools and difficulty integrating with real systems [103]. With regards to formal specifications, one possible cause is that completely describing a real system in most formal languages can be difficult or imposible. This is a part of Knight et al's formal specification evaluation criteria *coverage*, which notes that a formal specification must permit description of all of an application or be designed to operate with any other notations that will be required [104]. This is a key concept for any practical and general runtime verification framework for real-world systems. Different systems will have varying specification needs, and if these needs are not easily met history shows that system designers are likely to just use an informal specification instead, which restricts the use of formal verification techniques.

To ensure specification flexibility, we have implemented a semi-formal specification which combines a formal specification with a semi-formal *system mapping* which translates the actual system values into the monitor's more abstract execution trace. This semi-formal

interface allows considerable leeway in designing the translation between the system and monitor's model. This leeway is both powerful and potentially dangerous. By allowing certain transformations in the interface, we are able to remove complexity from the monitor itself while ensuring that we can monitor a diverse set of needs. This approach needs to be used carefully because it shifts part of the specification away from being formally defined, creating more opportunity for design mistakes and eroding some of the benefits that lead us to formal runtime verification in the first place.

The two part specification feeds into two parts of the monitor, as shown in Figure 3.4. The high-level formal specification contains the formal logic formulas that are directly checked by the formal monitoring algorithm. The lower-level semi-formal mapping is used to define the system to monitor interface which generates the formal trace propositions that are checked by the monitoring algorithm. From an framework perspective, the semi-formal interface has minimal design constraints. As long as the interface provides a system trace of the properties present in the formal specification, we do not restrict how this trace is created.

Because the correctness of the monitor ultimately relies on the correctness of the interface transformations, limiting the interface to a simple, verifiable set of transformations is key to providing guarantees about the monitor's output.

The interface has two primary purposes:

1. To convert actual system values (e.g., network message data) into a propositional execution trace.
2. To provide additional semantic power to create useful propositions (e.g., state machines).

The interface translates observable system state into a trace of propositions which can

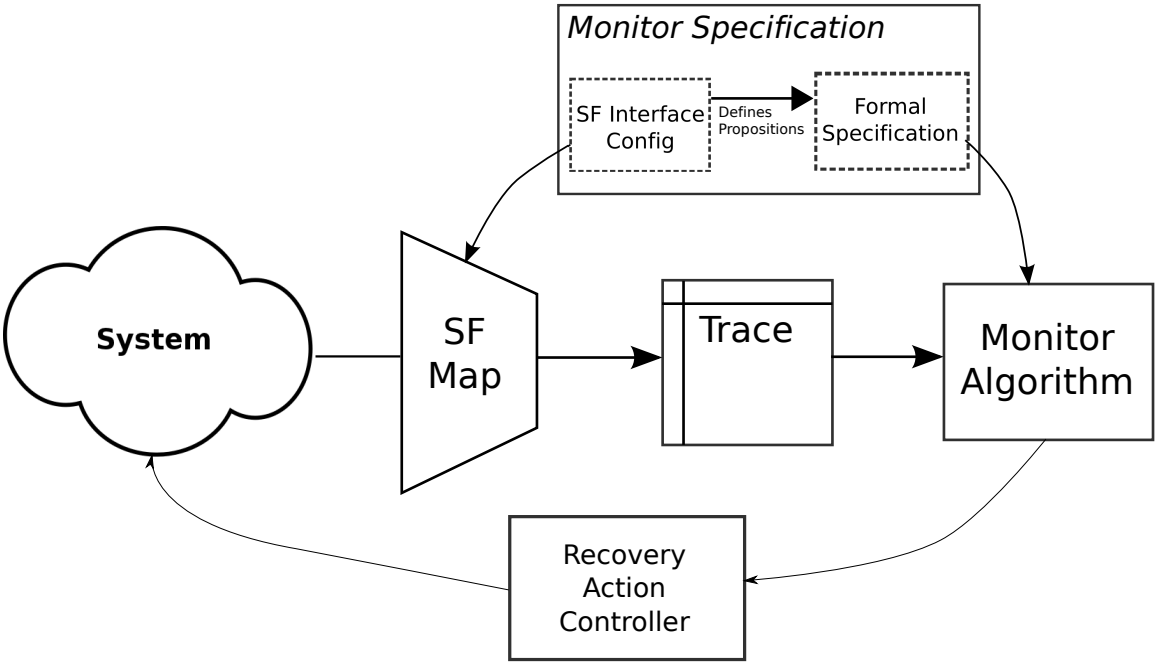


Figure 3.4: Monitor architecture showing multi-part specification

be checked against a specification by our monitoring algorithm. The interface provides the ability to extract the relevant data out of network messages in a flexible, network agnostic way. It can be used to extract the desired portion of a message, change units or typecast as necessary and then store the created proposition value correctly for use with the monitor. This ensures that the monitor is not restricted to networks or message types that it already knows how to transform. Any necessary data transformations can also be done in the interface. Besides simple transformations (e.g., storing boolean values into propositions, arithmatic comparisons, etc.), we can also do more complex compututations and conversions in the interface such as building state machines whose state can be used to fill propositions.

3.5.1 Semi-Formal Interface Design

The semi-formal interface is used provide the monitoring algorithm with the state trace. There are no inherent limitations on the semi-formal interface besides that it must create a viable trace for the monitor. There is, however, a practical limitation – the more complex the interface becomes the less confidence we should have that it works correctly. Given this, we'd like to restrict the interface as much as possible while still providing the necessary flexibility.

From our experience with system requirements and monitoring, the important transformations that are common for these types of systems are simple arithmetic, boolean connectives, comparisons to recent values, and simple state machines. Because of this, we restrict our own semi-formal mapping syntax to a simplified subset of the C programming language. We allow all the basic algorithmic, bitwise, and comparison operators and statically allocated variables of primitive types (i.e., integers, floating point). We do allow simple loops, for example over a static array holding a set of values, for calculating averages and other simple values. We do not currently have a formal definition of the semi-formal syntax, instead we trust that users can decide how restricted they wish to and realistically can be.

Our monitor implementation's semi-formal mapping configuration is integrated into the monitor's code directly by a specification compiler. This is essentially an internal domain specific language (DSL) [105]. Using a DSL to specify the semi-formal mappings allows us to restrict the specification of the interface to keep verification (of the interface) reasonable while still providing the ease of use and power of a programming language. Other prototype monitors we designed have used more traditional interface DSLs where the specifications have been used to insert and generate the code used by the monitor for the interface. More

discussion about using domain specific languages for the semi-formal interface is included in Section 7.3.2.

The motivation behind the restricted interface syntax is to provide enough power to define the system propositions necessary to monitor any reasonable desired property while incentivizing the monitor designers to work within the formal specification as much as possible. Given a desired set of system properties, it is easy to imagine the pathological case where the monitor is configured to check the proposition `SystemOK` and the interface is built to compute this value directly. While this technically would work, it is obviously an abuse of the interface and a bad (non?) use of the monitor framework. Doing this essentially creates a completely informal monitor inside the interface, wasting the proven correctness of the formal monitoring algorithm.

A more interesting case is when we have a system message which includes a value and a validity bit (signifying whether the message value is valid). We can imagine that most rules utilizing this value will want to check both some property of the value and whether it is valid at the same time (e.g., `valuePositive` and `valueValid`). This could be done on the formal side in the logic by using $\texttt{valuePositive} \wedge \texttt{valueValid}$ or by combining them in interface into one proposition `valuePositiveValid` which is calculated from both incoming values.

In this case it is less clear which method is superior, but we can suggest a few heuristics to help decide. Perhaps the most straightforward way to choose is to attempt to mirror the design documentation and requirements in the monitor specification. If the system requirements mention the validity either explicitly (as a message value) or implicitly (by mentioning a "valid positive value") then the monitor specification should also directly mention validity and use the validity message value directly. If the requirements do not

mention validity, then we can treat validity as an implementation detail and hide it away in the interface's proposition transformation. This follows the idea that the high-level requirements are abstracted away from the implementation. So if the validity bit is an artifact of the implementation then it is intuitive to deal with it in the interface, but if the validity bit is a part of the higher level system design then it should also be a part of the monitor specification.

Other factors can also affect the decision, such as monitoring complexity (moving transformations to the interface can reduce the amount of work the monitoring algorithm has to do) or the desire to maximize the amount of monitoring that is formally performed. These decisions should be kept consistent across specification rules to avoid confusion or misunderstanding.

In an ideal situation, the formal specification would contain all the relevant safety specification parts, leaving only the actual translation of system values to the semi-formal interface. More complex translations than simple threshold comparisons or copying booleans may require some semi-formal transformation if the desired properties cannot be expressed in propositional logic. Moving parts of the transformation to the interface side is useful when necessary, but as the interface configuration complexity increases it becomes more important to validate and verify the interface itself.

3.6 Usability Concerns

One of the biggest hurdles to the adoption of formal techniques is the perceived difficulty by non-experts of using them. In an attempt to diffuse these perceptions and provide a guide to simple integration, we provide two design artifacts: specification patterns and a

safety case pattern.

3.6.1 Specification Patterns

The difficulty, both perceived and real, of applying formal methods to real systems includes issues with using formal specification languages. Expressing system properties in formal languages such as temporal logic can be difficult for formal method experts, let alone non-expert system designers. Formal specification patterns have been proposed as a tool to support users in creating formal specifications with pre-specified, generic patterns [12].

Specification pattern based approaches have been proposed in many different domains [39, 106]. We have created a list of patterns common to autonomous ground systems. The patterns are presented as a catalogue which provides a description, intent, usage information and the specification rule in our bounded MTL for each pattern. We will refer to our bounded logic as BMTL. This logic is described in Section 4.1.2. Users of the patterns can create their system requirements using traditional requirements analysis methods (FTA, FMEA, etc.) and then use the specification pattern catalogue to translate those requirements into a formal specification.

One example of a specification pattern is shown in Table 3.2 (the rest of the pattern catalogue is provided in Appendix A).

Each specification pattern has a unique number and descriptive name denoting the pattern class. The intent entry provides a description of the high level intent of the pattern. The example entry shows an example instantiation of the pattern, showing how the pattern should be used in an applied setting. Next, the pattern provides the example in BMTL formula form and then provides the generic pattern BMTL and ASCII BMTL formulas. The variables entry describes each component of the formula in more detail. The description

entry gives a more thorough description of the formula and its actual meaning. Limitations are also noted, explaining potential misconceptions, pitfalls, etc., of the specific pattern. Other similar patterns are also listed, to help find the desired pattern if the current one does not exactly match the desired specification.

The example in Table 3.2 is a straightforward requirement that could exist in a passenger vehicle. The informal example requirement is "If the unlock doors button is depressed then the driver side door must be unlocked within 500ms". Given this requirement, the appropriate specification pattern would be identified in the catalogue which in this case is our example pattern 1.a Bounded Response. Once this pattern was chosen the user would match the informal specification to the variables to fill in the formula. In this case, the triggering event T is the unlock doors button being depressed, the triggered state E is that the driver side door is unlocked, and the maximum time for the doors to open within h is 500ms. Filling the identified values into the formula, we would obtain the example formula `UnlockDoorsPressed -> <0,500> DriverDoorUnlocked`.

We do not claim that our pattern catalogue is currently complete, nor do we make any claim about the coverage of our patterns. It is clear that even if a complete set of specification patterns for a given domain could be created, it would still require a matching semi-formal mapping which is similarly an expert creation. Nonetheless, the use of patterns in this way should help usability by letting non-experts create formal specifications by applying the patterns.

3.6.2 Safety Case Templates

A primary aspect of the motivation and usefulness of an external bolt-on monitor is the ease of composability both from a physical perspective (i.e., just connect the monitor to the

Table 3.2: Specification pattern example

Name	1.a Bounded Response	
Intent	To describe a relationship between two states where there must be an occurance of the second within a bounded amount of time of an occurance of the first	
Example	If the unlock doors button is depressed then the driver side door must be unlocked within 500ms	
Ex Formula	`UnlockDoorsPressed -> <0,500> DriverDoorUnlocked`	
Formula	BMTL ASCII	$T \rightarrow \Diamond_{[l,h]} E$ `T -> <l,h> (E)`
Variables	T E l h	Triggering event/state Triggered event/state Minimum time between occurance of T and occurance of E Maximum time between occurance of T and occurance of E
Description	This template is used for the common basic pattern where some state requires that another state be occurring in some bounded amount of time. As an invariant, note that any time t the guard condition T is true, then E must be true at some point in the future interval $[t + l, t + h]$	
Known Uses	This pattern can be used any time an event requires a change in state, such as user input (button/pedal presses, etc.) which cause a system mode change (turning off a feature, beginning some transition) or requires a bounded response. Care should be taken that one of the more specific rules is not actually desired	
Limitations	Temporal	Note that T and E only need to be true for a single time step. If a specific duration is required then a more specific rule should be used
See Also	1.b Bounded Response with duration	

bus) and a verification (of correctness/certifiability) perspective. But is this composition of systems actually simple? Can we actually argue for independent certification of the monitor and target system? The degree of system independence and the ease of monitor integration is system dependent, but we can at least outline the safety-case argument which shows that the monitor and system is safely (and relatively easily) composable.

In this section we present a safety case pattern which outlines the safety case for safe composability of the monitor and a target system. Our safety case pattern will follow the format of [46], which includes a GSN pattern and an accompanying text description. Our pattern is based on the existing *Hazard Avoidance Pattern* with two main classes of important hazards, bad monitor actions and bad monitor interactions, directly emphasized. The example instantiation of this pattern for our embedded ARM monitor is helpful to direct the types of hazards that should be explored. Besides acting as a starting point for a monitor's safety case, the pattern and example also highlight the important properties of a monitor implementation that affect its independence from a target system.

The underlying motivation here is to provide a safety case which argues that adding an external isolated monitor such as our design to an existing system is safe. The pattern attempts this by arguing that the monitor does not add any additional hazards to the system. This case hinges on the monitor's isolation from the target system, which comes down to two primary properties. First, connecting the monitor to the system should not add any unmitigated hazards. External monitor integration hazards can occur due to shared resources such as power or the connected network interface. As long as the hazards of any sharing are known and addressed, the integration itself will be safe. Second, if the monitor can perform any external actions (e.g., trigger a recovery) then these actions and any possible side-effects must not be hazardous. This covers both directly monitor-caused

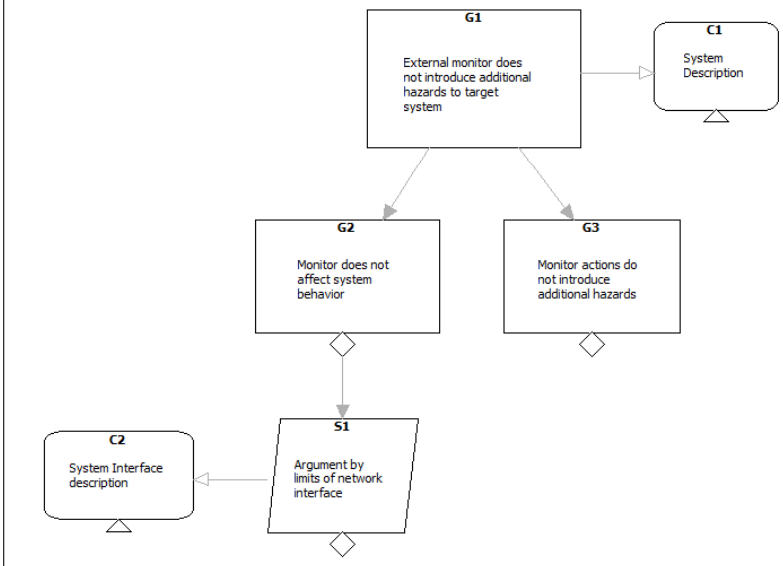
actions (e.g., monitor commands an unsafe behavior) and potential interactions between monitor actions and the real system (e.g., monitor disengages a necessary system feature, monitor messages flood network bus, etc.).

Note that this pattern does not incorporate or require that the monitor's software is fault free (for which we could incorporate the GSN fault free software pattern). By using passive monitors or monitors which only can actuate safe recoveries, which are always safe, even if not necessary, the correctness of the monitor is a reliability concern but does not affect the safety of the target system. That is, we wish to show that no matter what the monitor does, it cannot affect the safety of the system. In this way, the correctness of the monitor (besides isolation-affecting configuration concerns) does not affect system safety. Since there are integration hazards that can arise regardless of the correctness of the monitor, we frame the argument that integration is safe on inherent limitations of the integration (which may depend on the monitor configuration being correct).

Our safety case pattern is shown in Table 3.3 and the example instantiation is shown in Figure 3.5 for better readability.

Intent	The intent of this pattern is to provide a framework for arguing that the hazards of connecting a bolt-on monitor to a target system have been mitigated.
Also Known As	
Motivation	The motivation for this pattern is to communicate the key claims that need to be put forth to demonstrate that an external monitor does not add unmanaged hazards to a target system.

Structure


G1
External monitor does not introduce additional hazards to target system
C1
System Description
G2
Monitor does not affect system behavior
G3
Monitor actions do not introduce additional hazards
S1
Argument by limits of network interface
C2
System Interface description

Participants	**G1** This goal sets out the overall objective of the pattern – to be able to claim that introducing an external monitor does not add hazards to the system. **G2** Defines the goal that the integration of the monitor does not introduce hazards by affecting the system behavior. **G3** Defines the goal that the monitor itself does not perform actions which introduce hazards to the system. **S1** The argument approach defined by this strategy is to show that the monitor cannot introduce integration hazards due to inherent limits of the network interface. **C1** This context should be instantiated to refer to the system design documentation. Specifically this should be used to create the context in which the new hazards could occur. **C2** This context describes the system network, most importantly providing information about possible limitations or actions that the monitor can do to affect the system over the network.

Collaborations	The system and interface descriptions set the overall context. The **Monitor Actions** strongly affects the entire case. A completely passive monitor has no actions and so G3 becomes trivial. G2 depends only on physical issues such as power and network load.
Applicability	This pattern is applicable in contexts where an external monitor is being connected to a target system through a network interface. This pattern is mainly for use with passive monitors or monitors which utilize specific recovery subsystems rather than trying to alter system behavior. This pattern also assumes that connecting the monitor to the interface causes a negligible affect on the target system (e.g., network has spare load capacity).
Consequences	After instantiating this pattern, a number of unresolved goals will remain: G2: Monitor does not affect system behavior G3: Monitor actions do not introduce additional hazards See *Participants* for a description of the forms of support argument expected for these goals

Implementation	Implementation of this pattern first requires instantiating the contexts C1 and C2. The possible hazards that goals G2 and G3 show as mitigated are defined by the context of C1 and C2. Both goals G2 and G3 may need more subgoal instantiations as hazards are identified.
Examples	See Figure 3.5
Known Uses	Example CAN ARM monitor safety case, see Figure 3.5
Related Patterns	Hazard Avoidance Pattern [46]

Table 3.3: Safety case pattern

Being based on the safety case pattern, the partially instantiated safety case in Figure 3.5 has the same top-level goal which breaks down into two strategies: all integration hazards are addressed (S1) and all monitor action hazards are addressed (S2). Both these strategies depend on the identification of these classes of hazards, which is represented by the dependence on the contexts C1 and C2.

This partially instantiated case is based on a fully-passive version of our CAN monitor which silently listens on the CAN network rather than sending a warning message as we do in Section 6.3. On the integration side, we see three goals (G2,G3,G4) which correspond to three identified integration hazards. These are that the network may not have physical capacity for another node (the monitor), that shared power between the monitor and system could cause a fault, and that the monitor could cause the network schedule to fail. We partially expand G2, leaving all three of these goals requiring further work to complete evaluation. One subgoal G6 is expanded under goal G2, which is that the monitor does not send network messages. A node on the network that doesn't send a message cannot

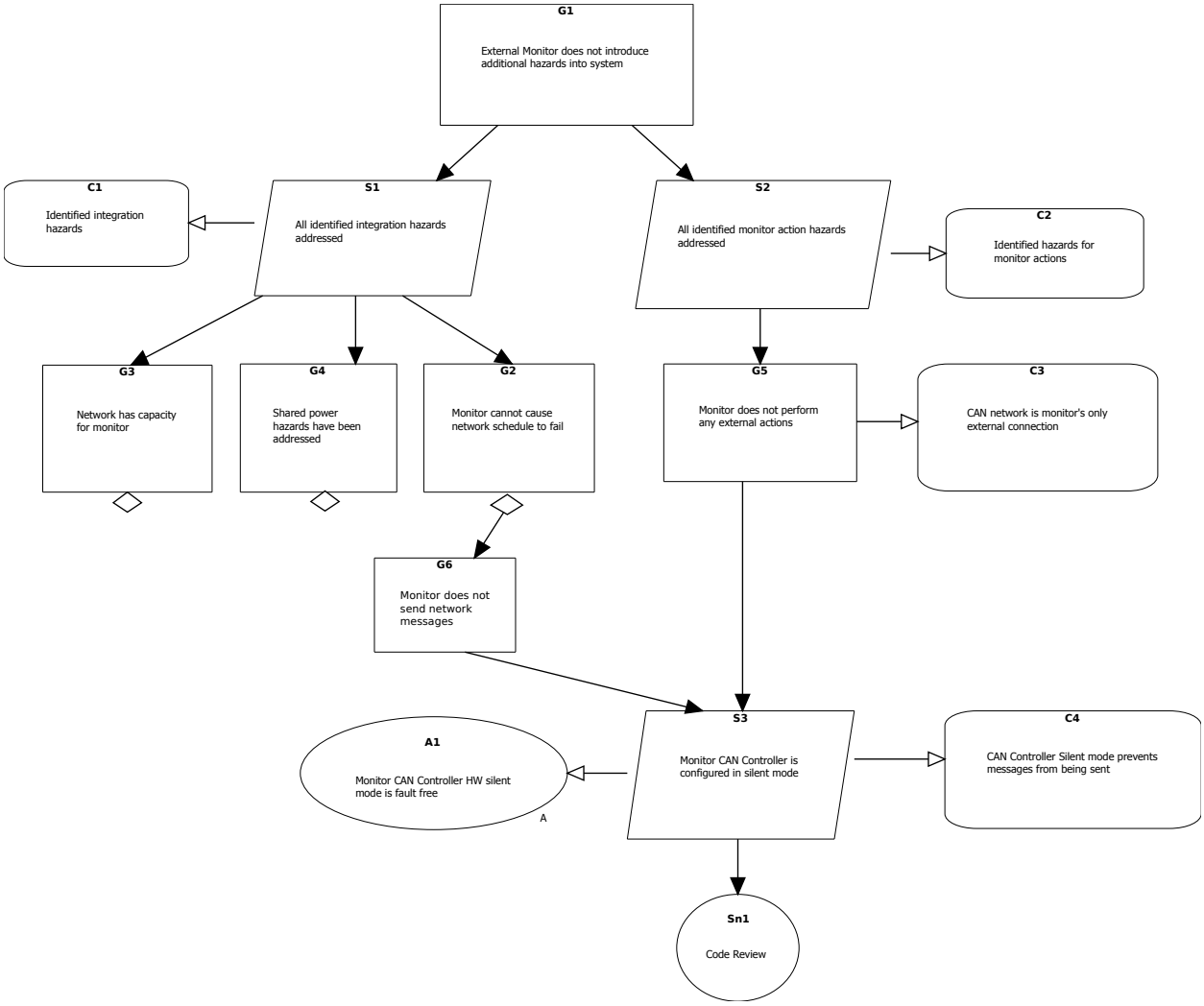


Figure 3.5: Partially instantiated safety case pattern

affect the network schedule directly by use (although it could cause load problems or other physical faults which could affect the schedule). The strategy used to show that the monitor cannot send messages on the network is to show that the monitor's CAN controller is configured in silent mode, in which the controller receives but can not send messages (from context C4). Given the assumption (A1) that the CAN controller is correct we can use a code review (Sn1) of the monitor to show this is true.

This same strategy is utilized by the other high level branch, which argues that all identified monitor action hazards have been addressed. This can be shown by verifying goal G5 which states that the monitor does not perform any external actions. Given the context that the monitor is not connected externally to anything but the CAN network (C3), this can also be shown by S3.

Chapter 4

Formal Monitoring

We present the runtime verification algorithm **agmon** which takes a trace and specification and reports whether the trace satisfies or violates the specification. **agmon** is an iterative, sampling-based algorithm so it can be used at runtime to continually check the specification against the current trace.

4.1 Preliminaries

The specification logic is built upon a set of atomic propositions which represent system properties (e.g., a proposition $speedLT40mph$ stating that the vehicle speed is less than 40mph). These propositions are created from the observable system state by the semi-formal interface. Let AP be the set of atomic propositions. A *state* $s : AP \to \{\top, \bot\}$ is a mapping from AP to the set of truth values. A *trace* $\sigma = s_0 s_1 s_2 \ldots s_n$ is a finite sequence of states. We use σ_i to denote the state s_i at position i in σ. A time series $\tau = t_0 t_1 t_2 \ldots t_n$ is a finite series of timestamps $t \in \mathbb{T}$. The pair σ, τ together represent a timed trace where each timestamp τ_i is the time associated with the occurance of state s_i.

4.1.1 Time Model

Temporal logics have many different characteristic parameters which are discussed and compared in [107]. Time models for real-time logics are based on two sets of parameters. First is whether they are continuous or discrete, that is, whether there are infinitely many or a finite number of underlying time points in a given interval. A time model is also either interval-based or point-based. Interval-based models treat traces as a series of intervals of continuous system state whereas point-based models treat traces as a sequence of system states (or transitions). Dense, interval-based time domains are thought of as more intuitive/natural [68] but they have more computationally difficult decision problems.

We utilize a discrete pointwise semantics with the time domain $\mathbb{T} \equiv \mathbb{N}$ for simplicity (though other discrete domains could be used instead). In general, pointwise semantics are best used when modeling the system as a series of events [108], but there is a risk that point-based semantics can be unintuitive in contrast to interval-based semantics [68]. By using our point-based semantics within a periodic sampling architecture we can avoid these potential intuition mismatches while still benefiting from a conceptually simpler pointwise algorithm. This requires forcing all temporal bounds to be multiples of the monitoring period so that all checks are performed on the exact sample points and not between samples. This can be easily ensured by using step-bounds in temporal operators instead of absolute time bounds.

Our trace model is based on a time-triggered, sampling-based monitor strategy where the system state is periodically sampled and then checked. Since the monitor state can only change at these sample points, we can treat the state changes as instantaneous which allows us to completely specify our interval traces with single points as discussed in [109]. This timed trace model has a clean mapping to two common actual system trace types:

state snapshots and update message logs. State snapshots are a series of (usually periodic) timestamped system states often coming from instrumented or simulated systems. Value update messages are aperiodic system property "updates" such as a network log where each message carries an update to a single or small collection of property values. System traces that are a series of state snapshots naturally fit into our execution trace. In this case the execution trace is the series of timestamped state snapshots. A series of update messages actually fits the point-based semantics more directly. We can treat each incoming message as a transition which updates the system state. If the system is being sampled at a rate slower than the incoming messages, then multiple update messages can be functionally combined into a single state transition.

4.1.2 Specifications

A *specification* is a set of system invariants written in our bounded MTL variant BMTL which is a past- and future-time MTL with only finite intervals. We refer to the invariants as the specification *rules* or *policies*. The specification rules are checked at every monitored step, so each rule has an implicit unbounded *always* over it (similar to safety rules in [110]). The syntax of BMTL is shown below:

$$\psi ::= p \mid \neg\psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \mathcal{U}_{[l,h]} \psi_2 \mid \psi_1 \mathcal{S}_{[l,h]} \psi_2$$

Policy formulas are denoted by ψ. Policy formulas can include two bounded temporal operators: the past-time operator *since* ($\mathcal{S}$) and the future-time operator *until* ($\mathcal{U}$). Both temporal operators include a time bound interval $[l, h]$ where $l, h \in \mathbb{T}$ and $0 \leq l \leq h$. These intervals bound the time in which the triggering subformula ψ_2 is evaluated. An

until formula $\alpha \mathcal{U}_{[l,h]} \beta$ states that α must be true from the current step i until (but not including) the time step k where $\tau_k \in [\tau_i + l, \tau_i + h]$ and β is true. *Since* formulas are similar but in the past: $\alpha \mathcal{S}_{[l,h]} \beta$ states that α must have been true since (but not including) timestep k where $\tau_k \in [\tau_i - h, \tau_i - l]$ and β was true (where i is the current time step).

Specification policies are interpreted over a timed system trace σ, τ. We write $\sigma, \tau, j \vDash \varphi$ to represent that the policy φ is satisfied by the timed trace (σ, τ) at trace position j. We use a common definition of $\vDash$ defined inductively as follows:

$$
\begin{array}{ll}
\sigma, \tau, j \vDash \top & \\
\sigma, \tau, j \vDash p & \text{if and only if } \sigma_j(p) = \top \\
\sigma, \tau, j \vDash \neg\psi & \text{if and only if } \sigma, \tau, j, \nvDash \psi \\
\sigma, \tau, j \vDash \psi_1 \vee \psi_2 & \text{if and only if } \sigma, \tau, j \vDash \psi_1 \text{ or } \sigma, \tau, j \vDash \psi_2 \\
\sigma, \tau, j \vDash \psi_1 \mathcal{U}_{[l,h]} \psi_2 & \text{if and only if } \exists k : (l \leq \tau_k - \tau_j \leq h), \sigma, \tau, k \vDash \psi_2 \\
 & \qquad \text{and } \forall k' : (\tau_j \leq \tau_{k'} < \tau_k), \sigma, \tau, k' \vDash \psi_1 \\
\sigma, \tau, j \vDash \psi_1 \mathcal{S}_{[l,h]} \psi_2 & \text{if and only if } \exists k : (l \leq \tau_j - \tau_k \leq h), \sigma, \tau, k \vDash \psi_2 \\
 & \qquad \text{and } \forall k' : (\tau_k < \tau_{k'} \leq \tau_j), \sigma, \tau, k' \vDash \psi_1
\end{array}
$$

We use the usual notational conveniences for the other common logic operators: *eventually* ($\Diamond_{[l,h]}\psi \equiv \top \mathcal{U}_{[l,h]}\psi$), *always* ($\Box_{[l,h]}\psi \equiv \neg\Diamond_{[l,h]}\neg\psi$), *past eventually* ($\blacklozenge_{[l,h]}\psi \equiv \top \mathcal{S}_{[l,h]}\psi$), *past always* ($\blacksquare_{[l,h]} \equiv \neg\blacklozenge_{[l,h]}\neg\psi$) and the boolean connectives *and* and *implies* ($\wedge, \rightarrow$).

4.2 Practical Issues

An important aspect of the presented monitoring algorithm is the actual use of the algorithm to perform runtime verification of safety-critical systems. We discuss three primary aspects:

(1) *Correctness*, (2) *Promptness*, and (3) Trace Indepedence.

Obviously, we want the monitoring algorithm to correctly monitor the target system. We want to identify all violations and not any false positives. To do this we need to prove that the monitor correctly implements the logic semantics, which we do in Section 4.4. One important thing to note is that since we use a two-valued logic, we must ensure that aggressive checking of incomplete traces does not return premature monitoring decisions on inconclusive traces which could lead to false positives or worse, undetected violations. Because we use a two-valued logic there are traces which are neither good nor bad prefixes (i.e., the continuation of the trace could lead to either truth value [111]) but do have a truth value based on our semantics [112]. In other words, formulas in our logic always have a true or false interpretation even if that interpretation may change given more (future) information. For example, given the formula *It will rain today or tomorrow* ($\Diamond_{[0,1]} rain$), if it is currently not raining we cannot conclusively answer for all possible futures. According to the semantics of our logic, this formula is false (currently) if we have not seen rain. If it does rain tomorrow, then the premature false interpretation will have been wrong. One way to solve this dilemna is to use a three-valued logic which also includes a value representing *unknown* [113]. Instead of using a three-valued semantics, we have designed our monitoring algorithm to distinguish between conclusive and inconclusive traces. The algorithm delays checking formulas for which the current trace is inconclusive. Essentially, rather than report a potentially incorrect answer, the algorithm puts the question aside until it can conclusively answer. Our formal definitions of monitor correctness take this into account – we require that all answers the monitor provides are correct but we do not require the monitor to provide immediate answers for inconclusive traces. We do require immediate answers for conclusive traces, which is accounted for by the algorithm's promptness

property.

To account for inconclusive traces, we utilize the notion of *promptness* which states that the monitor will have checked a formulas as soon as the trace is guaranteed to be conclusive for that formula. Every formula has a static delay (provided by $\Delta^w(\phi)$ which defines how long after the current time t the monitor must wait before the trace is guaranteed to be conclusive for ϕ at t. Because we use a bounded logic, this delay is guaranteed to be finite and we thus know how long we must wait to answer any given formula. The aggressive monitoring algorithm does attempt to check formulas early (i.e., before the trace is guaranteed to be conclusive) but if the formula can't be conclusively answered the monitor delays answering.

An important property for live, resource constrained monitoring is that the monitor be relatively independent of the system trace length. We do not want to store the entire system trace since it is constantly growing which would make storing the trace for long running systems difficult or impossible. Our monitoring algorithm is an iterative algorithm that stores the necessary history state (for checking temporal formulas) in structures but does not store the entire trace. This algorithm is essentially trace length independent, performing the same work at every iteration regardless of where in the trace it is currently checking. The algorithm as presented below does keep a constantly growing history structure and the entire timestamp sequence, but this can be easily pruned to a bounded amount of history dependent on the formula's temporal duration. We only need to keep enough history to ensure that we can conclusively check the specification at every monitor step. The amount of history required to check any given step is defined by the formula's storage delay ($\Delta^S(\phi)$) so we can remove entries older than this delay. Instead of keeping the entire timestamp sequence τ we can instead store individual τ_k's for each stored history step k in the history

structures which bounds the number of stored timestamps to the number of history entries. These stored timestamps would then be pruned along with the history structure entries.

4.3 Monitoring Algorithms

The proposed monitoring algorithm is an aggressive, iterative algorithm that performs formula reduction based on formula rewriting utilizing history structures to save any necessary trace state. The formulas are eventually reduced to truth values which show whether a given formula is satisfied or violated the given trace. The algorithm is aggressive in that it attempts to check future-time temporal formulas as soon as possible rather than waiting until an answer is guaranteed to be available. This aggressiveness is achieved by performing short-circuiting of boolean operators and checking if temporal formulas can conclusively be answered at all steps. This is in contrast to many existing runtime monitoring algorithms which either are restricted to past time logics which can be checked with only the current and past state (i.e., they are never inconclusive) [65, 114] or those that only check properties after their delay which guarantees they are conclusive [53, 110, 115]. The algorithm is iterative in that the history structure must be updated at each timestep in the trace in order. This allows us to only store the necessary state history (in history structures) rather than keep the entire system trace. As noted, these history structures can be pared based on the policy delay to bound their maximum size.

4.3.1 Definitions

Residues A residue $r_\phi^j = \langle j, \psi \rangle_\phi$ is a tagged pair where $j \in \mathbb{N}$ is a position in the trace and ψ, ϕ are well-formed formulas. The monitoring algorithm is built on these residual for-

mulas which are used to represent the currently unreducible fragments (due to incomplete information) of a given parent formula. The residual formula ψ is a reduced equivalent form of the formula ϕ at time j. A residue is considered *correct* if ψ and ϕ have equivalent truth values at j. Residues are used to hold the history information of a formula at a specific time for checking temporal formula. The $\mathbf{reduce}$ function performs formula rewriting by reducing a given residue based on the current state and history information.

Formula Delays There are two important delay functions that are used for this monitor. These delays are similar to the sampling bound operator $\#(\phi)$ from [115] where traces generated by a system simulation were monitored. The sampling bound $\#(\phi)$ describes how much farther the simulation should be run to generate a trace long enough to guarantee that ϕ can be conclusively checked. This bound is the same as the amount of wait time needed to obtain the necessary state from a running system (rather than a simulation).

Our first delay is the wait delay $\Delta^w(\phi)$, which defines an upper bound on the maximum time the monitor needs to wait before ϕ can be evaluated. The second delay function $\Delta^S(\phi)$ defines the storage requirements (i.e., how far back in time history needs to be kept) for the history structure S_ϕ. This is similar to the wait delay of the formula ϕ but combines the future and past delays additively since we must store the past history to evaluate past-time formulas. Residues r^j where $\tau_i - \tau_j > \Delta^S(\varphi)$ in any history structure S^i_ψ where $\psi \in \mathbf{tempsub}(\varphi)$ can be pruned away to save storage space as they are no longer needed by the algorithm.

$$\Delta^w(\phi) = \begin{cases} 0 & \text{iff } \psi \equiv \top \\ 0 & \text{iff } \psi \equiv \bot \\ 0 & \text{iff } \psi \equiv p \\ \Delta^w(\psi) & \text{iff } \phi \equiv \neg\psi \\ max(\Delta^w(\alpha), \Delta^w(\beta)) & \text{iff } \phi \equiv \alpha \vee \beta \\ h + max(\Delta^w(\alpha), \Delta^w(\beta)) & \text{iff } \phi \equiv \alpha\, \mathcal{U}_{[l,h]}\, \beta \\ max(\Delta^w(\alpha), \Delta^w(\beta)) - l & \text{iff } \phi \equiv \alpha\, \mathcal{S}_{[l,h]}\, \beta \end{cases}$$

$$\Delta^S(\phi) = \begin{cases} 0 & \text{iff } \psi \equiv \top \\ 0 & \text{iff } \psi \equiv \bot \\ 0 & \text{iff } \psi \equiv p \\ \Delta(\psi) & \text{iff } \phi \equiv \neg\psi \\ max(\Delta(\alpha), \Delta(\beta)) & \text{iff } \phi \equiv \alpha \vee \beta \\ h + max(\Delta(\alpha), \Delta(\beta)) & \text{iff } \phi \equiv \alpha\, \mathcal{U}_{[l,h]}\, \beta \\ h + max(\Delta(\alpha), \Delta(\beta)) & \text{iff } \phi \equiv \alpha\, \mathcal{S}_{[l,h]}\, \beta \end{cases}$$

Simplify The $\mathbf{simplify}(\phi)$ function is used to rewrite a given formula in a syntactically reduced form. For the restricted logic, simplify only rewrites two types of formulas. It removes negations from formulas when the truth value is known and reduces disjunctive formulas based on any known truth values. Otherwise $\mathbf{simplify}(\phi)$ returns the formula unchanged.

$$\mathbf{simplify}(\phi) = \begin{cases} \top & \text{if } \phi \equiv \neg\bot \\ \bot & \text{if } \phi \equiv \neg\top \\ \top & \text{if } \phi \equiv \alpha \vee \top \text{ or } \phi \equiv \top \vee \beta \\ \alpha & \text{if } \phi \equiv \alpha \vee \bot \\ \beta & \text{if } \phi \equiv \bot \vee \beta \\ \phi & \text{otherwise} \end{cases}$$

Subformulas of Temporal Formulas The only trace history that is necessary for monitoring in this algorithm is the history of the direct child formulas of temporal subformula. That is, for $\alpha \mathcal{U}_{[l,h]} \beta$ we need to save the history of α and β (and if either of those are also a temporal formula then we need their subformula's history as well). The function $\mathbf{tempSub}(\psi)$ returns a list of all the subformula of ψ that need their history to be saved to monitor ψ.

$$\mathbf{tempSub}(\psi) = \begin{cases} \emptyset & \text{if } \psi \equiv p \\ \{\alpha\} \cup \{\beta\} \cup \mathbf{tempSub}(\alpha) & \\ \quad \cup \mathbf{tempSub}(\beta) & \text{if } \psi \equiv \alpha \mathcal{U}_{[l,h]} \beta \text{ or } \psi \equiv \alpha \mathcal{S}_{[l,h]} \beta \\ \mathbf{tempSub}(\alpha) \cup \mathbf{tempSub}(\beta) & \text{if } \psi \equiv \alpha \vee \beta \\ \mathbf{tempSub}(\alpha) & \text{if } \psi \equiv \neg\alpha \end{cases}$$

Formula Length The length $|\phi|$ of a formula ϕ is the number of subformulas in a formula and is defined as follows:

$$|\phi| = \begin{cases} 1 & \text{if } \phi \equiv p \\ 1 + |\phi'| & \text{if } \phi \equiv \neg\phi' \\ 1 + |\alpha| + |\beta| & \text{if } \phi \equiv \alpha \vee \beta \\ 1 + |\alpha| + |\beta| & \text{if } \phi \equiv \alpha\, \mathcal{U}_{[l,h]}\, \beta \\ 1 + |\alpha| + |\beta| & \text{if } \phi \equiv \alpha\, \mathcal{S}_{[l,h]}\, \beta \end{cases}$$

Free Propositions A free proposition in a formula is a proposition which is not a subformula of a temporal operator. The procedue $\mathbf{freep}(\phi)$ provides a list of free propositions in ϕ. This will be used to define *reducable* residues which is necessary to ensure that residues are initialized with their original state (since that state will not be saved).

$$\mathbf{freep}(\phi) = \begin{cases} p & \text{if } \phi \equiv p \\ \mathbf{freep}(\alpha) & \text{if } \phi \equiv \neg\alpha \\ \mathbf{freep}(\alpha) \cup \mathbf{freep}(\beta) & \text{if } \phi \equiv \alpha \vee \beta \\ \emptyset & \text{if } \phi \equiv \top \text{ or } \phi \equiv \bot \\ & \quad \text{or } \phi \equiv \alpha\mathcal{U}_{[l,h]}\beta \\ & \quad \text{or } \phi \equiv \alpha\mathcal{S}_{[l,h]}\beta \end{cases}$$

History Structures A history structure $S^i_\phi = \{r^0_\phi, r^1_\phi, \ldots r^i_\phi\}$ is a list of residues which is used to store the history of a formula. We use $\mathbb{S}^i_\phi$ to represent the set of history structures for all temporal subformula of ϕ, i.e., $\mathbb{S}^i_\phi = \bigcup_{\psi \in \mathbf{tempSub}(\phi)} S^i_\psi$. So $\mathbb{S}^i_\phi$ holds all the history structures necessary to check the formula ϕ.

History structures summarize the relevant portions of the past elements of a trace that are (possibly) required to evaluate the truth value of the property in question. More precisely, the history structure S^i_ϕ for a given formula φ stores the truth values of the formula ϕ at prior positions (including the current position) of the trace. This history is used as the

values of ϕ when checking any parent temporal formulas of ϕ.

History structures need to be updated when new state (i.e., trace entries) are encountered. This requires both reducing all the structure's residues with the new state, which updates them based on the new state, as well as adding a residue (i.e., a history entry) for the current encountered state. The $\mathbf{incrS}(S_\psi^{i-1}, \mathbb{S}_\psi^i, \sigma_i, \tau, i, \psi)$ procedure performs this update for structure S_ψ^{i-1}, reducing all of the structure's residues as well as adding a reduced residue for the current state.

The increment function takes the following parameters: The history structure from the previous step we wish to increment S_ϕ^{i-1}, the set of necessary subformula history structures for solving ψ $\mathbb{S}_\psi^i$ (which may be empty), the current state σ_i, the timestamp sequence τ, the current step i, and the formula ψ for which the history structure is currently being updated. The function returns the updated history structure S_ψ^i.

$$\mathbf{incrS}(S_\psi^{i-1}, \mathbb{S}_\psi^i, \sigma_i, \tau, i, \psi) = \left(\bigcup_{r \in S_\psi^{i-1}} \mathbf{reduce}(\sigma_i, \tau, \mathbb{S}_\psi^i, r) \right) \cup \mathbf{reduce}(\sigma_i, \tau, \mathbb{S}_\psi^i, \langle i, \psi \rangle)$$

Monitor Algorithm The high level aggressive monitoring algorithm is shown in Figure 4.1. First, the history structure set $\mathbb{S}_\varphi = \{S_{\phi_1}, S_{\phi_2}, \ldots, S_{\phi_n}\}$ is built by identifying the required history structures S_{ϕ_i} needed to check the policy φ using $\mathbf{tempSub}(\varphi)$. The history of these subformula and the current state at any given step are the only history required by the algorithm. Once the structure $\mathbb{S}_\varphi$ is built, the monitoring loop begins.

First, In each step, all the identified history structures are incremented with the current step of the trace. This is done in increasing formula size since larger formula can depend on the history of smaller formula which may be their subformula (i.e., this way when

1: For all recognized formulas $\psi \in \mathbf{tempSub}(\varphi)$: $S_\psi^{-1} \leftarrow \emptyset$
2: $S_\varphi^{-1} \leftarrow \emptyset$
3: $i \leftarrow 0$
4: **loop**
5: Obtain next trace step (σ_i, τ_i) and extend τ with τ_i
6: **for** every $\psi \in \mathbf{tempSub}(\varphi)$ in increasing size **do**
7: $S_\psi^i \leftarrow \mathbf{incrS}(S_\psi^{i-1}, \mathbb{S}_\psi^i, \sigma_i, \tau, i, \psi)$
8: $S_\varphi^i \leftarrow \mathbf{incrS}(S_\varphi^{i-1}, \mathbb{S}_\varphi^i, \sigma_i, \tau, i, \varphi)$
9: **for** all $\langle j, \bot \rangle \in S_\varphi^i$ **do**
10: `Report violation on` σ `at position` j
11: $i \leftarrow i + 1$

Figure 4.1: Aggressive Monitoring Algorithm

incrementing ψ every structure in $\mathbb{S}_\psi$ has already been updated since they are all structures of smaller formula). Each structure is updated using $\mathbf{incrS}(S_\psi^{i-1}, \mathbb{S}_\psi^i, \sigma_i, \tau, i, \psi)$ which reduces all residues in the structure and adds a reduced residue for the current trace step. Then, the same procedure is performed for the top level policy that is being monitored – the policy's structure is updated with $\mathbf{incrS}(S_\varphi^{i-1}, \mathbb{S}_\varphi^i, \sigma, \tau, i, \varphi)$. Once updated, this structure is checked for policy violations (i.e., false residues) before the algorithm continues to the next trace step. Any false residue in this structure represents a timestep when the formula is false, so false residues in the policy's structure show violations of the policy.

It is important to note that due to the recursive nature of the monitor algorithm, the top-level policy is treated exactly as any temporal subformula would be (which follows from the implicit *always* on all policies) except that violations of the top level policy are reported. We separate incrementing the subformulas from the specification policies in the algorithm description for clarity only. We could instead define $\mathbf{tempsub}(\varphi)$ to include φ which would include the policy (and some other extraneous formula structures) in the loop.

Reduce The $\mathbf{reduce}$ function is the backbone of the monitor algorithm. It takes a residue and the current trace state and returns the residue in a reduced form.

The $\mathbf{reduce}$ function takes the following parameters: the current state σ_i, the timestamp sequence τ, the current step i, the set of history structures $\mathbb{S}^i_\phi$, and a residue $\langle j, \psi \rangle_\phi$. $\mathbf{Reduce}$ returns a reduced residue $\langle j, \psi' \rangle_\phi$. Note that for $\phi = \mathbf{reduce}(\alpha \mathcal{U}_{[l,h]} \beta)$ (and also *since*) we know that $\mathbb{S}^i_\phi$ at least contains S^i_α and S^i_β (and possibly other structures if there are temporal subformulaof α or β).

Residues which have truth values are already fully reduced, so $\mathbf{reduce}$ returns these residues unchanged:

$$\mathbf{reduce}(\sigma_i, \tau, i, \mathbb{S}^i_\top, \langle j, \top \rangle) = \langle j, \top \rangle$$
$$\mathbf{reduce}(\sigma_i, \tau, i, \mathbb{S}^i_\bot, \langle j, \bot \rangle) = \langle j, \bot \rangle$$

For residues whose formula is a proposition, $\mathbf{reduce}$ fills in this proposition with its current truth value:

$$\mathbf{reduce}(\sigma_i, \tau, i, \mathbb{S}^i_p, \langle j, p \rangle) = \langle j, \sigma_i(p) \rangle$$

For residues whose formula is a negated subformula $\neg\psi$, $\mathbf{reduce}$ is called on the subformula ψ and the negation is added back onto the resulting reduced subformula. This newly reduced and negated subformula is passed through $\mathbf{simplify}$ to evaluate the negation and then returned:

$$\mathbf{reduce}(\sigma_i, \tau, i, \mathbb{S}^i_{\neg\psi}, \langle j, \neg\psi\rangle) = \begin{cases} \text{Let } \alpha \leftarrow \mathbf{reduce}(\sigma_i, \tau, i, \mathbb{S}^i_{\psi}, \langle j, \psi\rangle) \\ \quad \alpha' \leftarrow \mathbf{simplify}(\neg\alpha) \\ \quad \mathbf{return}\ \langle j, \alpha'\rangle \end{cases}$$

For residues whose formula is a disjunction of subformula $\alpha \vee \beta$, **reduce** is called on both subformula α and β. The resulting reduced subformula are recombined into a disjunction $\alpha' \vee \beta'$ which is passed through **simplify** to evaluate the disjunction and then returned:

$$\mathbf{reduce}(\sigma_i, \tau, i, \mathbb{S}^i_{\alpha\vee\beta}, \langle j, \alpha \vee \beta\rangle) = \begin{cases} \text{Let } \alpha' \leftarrow \mathbf{reduce}(\sigma_i, \tau, i, \mathbb{S}^i_{\alpha}, \langle j, \alpha\rangle) \\ \quad \beta' \leftarrow \mathbf{reduce}(\sigma_i, \tau, i, \mathbb{S}^i_{\beta}, \langle j, \beta\rangle) \\ \quad \phi' \leftarrow \mathbf{simplify}(\alpha' \vee \beta') \\ \quad \mathbf{return}\ \langle j, \phi'\rangle \end{cases}$$

For residues whose formula is an *until* formula $\alpha\mathcal{U}_{[l,h]}\beta$, the history structures S^i_α and S^i_β are used to reduce the formula. If the formula can be evaluated conclusively then the truth value is returned, otherwise the residue is returned unchanged. We utilize five marker variables to evaluate the formula over the current trace:

- a_a is the earliest in-bounds ($[\tau_j, \tau_j + h]$) step at which α is false. This represents the latest time step at which β can be true for $\alpha\mathcal{U}_{[l,h]}\beta$ to be true.
- a_u is the latest in-bounds ($[\tau_j, \tau_j + h]$) step at which α has been true since the residue time step. This represents the latest time step that the *until* formula would be conclusively true if β was true at that step.

- b_a is the earliest in-bounds ($[\tau_j + l, \tau_j + h]$) step at which β is not conclusively false. This is used to check whether the formula is still satisfiable or not.
- b_t is the earliest in-bounds ($[\tau_j + l, \tau_j + h]$) step at which β is conclusively true.
- b_n is true if the current step is later than the wait delay $\Delta^w(\psi)$ of the residue and all residues in bounds are false (that is, β is conclusively false at all steps in bounds).

Using these marker variables, we can evaluate the semantics of *until*. If $a_u \geq b_t$ (and b_t exists) then we know that α is true from the residue step j until the step b_t where β is true, which satisfies the semantics of *until* (so we return $\top$). Otherwise, if $a_a < b_a$ then α is not true until the first possible step that β could be true (since b_a is the earliest non-false step). $\alpha \mathcal{U}_{[l,h]} \beta$ cannot be true in this case, so we return $\bot$. Similarly, if b_n is true, then there is no β in bounds and the formula is false. If none of these cases exist then the trace is inconclusive, so the residue is returned unchanged.

$$
\mathbf{reduce}(\sigma_i, \tau, i, \mathbb{S}^i_{\alpha \mathcal{U}_{[l,h]}\, \beta}, \langle j, \alpha \mathcal{U}_{[l,h]}\, \beta \rangle) = \begin{cases}
\text{let } a_a \leftarrow min(\{k | \tau_j \leq \tau_k \leq \tau_j + h \wedge \langle k, \bot \rangle \in S^i_\alpha\}, i) \\
a_u \leftarrow max(\{k | \tau_k \in [\tau_j, \tau_j + h] \\
\qquad \wedge \forall k' \in [j, k-1].(\langle k', \alpha' \rangle \in S^i_\alpha \wedge \alpha' \equiv \top\}, i) \\
b_a \leftarrow min(\{k | \tau_j + l \leq \tau_k \leq \tau_j + h \wedge \langle k, \beta' \rangle \in S^i_\beta \wedge \beta' \neq \bot\}) \\
b_t \leftarrow min(\{k | \tau_j + l \leq \tau_k \leq \tau_j + h \wedge \langle k, \top \rangle \in S^i_\beta\}) \\
b_n \leftarrow \top \text{ if } (\tau_i - \tau_j \geq \Delta^w(\psi)) \\
\qquad \wedge \forall k.(\tau_j + l \leq \tau_k \leq \tau_j + h).\langle k, \bot \rangle \in S^i_\beta \\
\text{if } b_t \neq \emptyset \wedge a_u \geq b_t \\
\quad \mathbf{return} \langle j, \top \rangle \\
\text{else if } (b_a \neq \emptyset \wedge a_a < b_a) \text{ or } b_n = \top \\
\quad \mathbf{return} \langle j, \bot \rangle \\
\text{else} \\
\quad \mathbf{return} \langle j, \alpha \mathcal{U}_{[l,h]}\, \beta \rangle
\end{cases}
$$

For residues whose formula is an *since* formula $\alpha \mathcal{S}_{[l,h]} \beta$, the history structures S^i_α and S^i_β are used to reduce the formula. If the formula can be evaluated conclusively then the truth value is returned, otherwise the residue is returned unchanged. We utilize five marker variables to evaluate the formula over the current trace:

- a_a is the latest in-bounds ($[\tau_j - h, \tau_j]$) step at which α is false. This represents the earliest time step at which β can be true for $\alpha \mathcal{S}_{[l,h]} \beta$ to be true.
- a_s is the earliest in-bounds ($[\tau_j - h, \tau_j]$) step at which α has been true since then (up until the residue step j). This represents the earliest time step that the *since* formula would be conclusively true if β was true at that step.
- b_a is the latest in-bounds ($[\tau_j - h, \tau_j - l]$) step at which β is not conclusively false. This is used to check whether the formula is still satisfiable or not.
- b_t is the earliest in-bounds ($[\tau_j - l, \tau_j - h]$) step at which β is conclusively true.
- b_n is true if the current step is later than the wait delay $\Delta^w(\psi)$ of the residue and all residues in bounds are false (that is, β is conclusively false at all steps in bounds).

$$\mathbf{reduce}(\sigma_i, \tau, i, \mathbb{S}^i_{\alpha\,\mathcal{S}_{[l,h]}\,\beta}, \langle j, \alpha\,\mathcal{S}_{[l,h]}\,\beta\rangle) = \begin{cases} \text{let } a_a \leftarrow max(\{k|\tau_j - h \leq \tau_k \leq \tau_j \wedge \langle k, \bot\rangle \in S^i_\alpha\}, i) \\ a_s \leftarrow min(\{k|\tau_k \in [\tau_j - h, \tau_j] \\ \qquad \wedge \forall k' \in [k+1, j].(\langle k, \alpha'\rangle \in S^i_\alpha \wedge \alpha' \equiv \top\}, i) \\ b_a \leftarrow max(\{k|\tau_j - h \leq \tau_k \leq \tau_j - l \wedge \langle k, \beta'\rangle \in S^i_\beta \wedge \beta' \neq \bot\}) \\ b_t \leftarrow max(\{k|\tau_j - h \leq \tau_k \leq \tau_j - l \wedge \langle k, \top\rangle \in S^i_\beta\}) \\ b_n \leftarrow \top \text{ if } (\tau_i - \tau_j \geq \Delta^w(\psi)) \\ \qquad \wedge \forall k.(\tau_j - h \leq \tau_k \leq \tau_j - l).\langle k, \bot\rangle \in S^i_\beta \\ \text{if } b_t \neq \emptyset \wedge a_s \leq b_t \\ \quad \mathbf{return}\langle j, \top\rangle \\ \text{else if } (b_a \neq \emptyset \wedge a_a > b_a) \text{ or } b_n = \top \\ \quad \mathbf{return}\langle j, \bot\rangle \\ \text{else} \\ \quad \mathbf{return}\langle j, \alpha\,\mathcal{S}_{[l,h]}\,\beta\rangle \end{cases}$$

4.4 Correctness of the algorithms

We will show the correctness of our top-level algorithm **agmon** through mutual induction. We start with some definitions.

4.4.1 Definitions

A *correct* residue is one whose residual formula has an equivalent truth value with the parent formula at the residue's step. In other words, for a residue $\langle j, \phi\rangle_\psi$, the truth of ϕ must be equivalent to the truth of ψ at step j.

Definition 1 (Residue correctness)**.** *The residue $r^j_\psi = \langle j, \phi\rangle_\psi$ is correct if for all traces σ, timestamp series τ, timestep j, and formulas ψ, ϕ that $\sigma, \tau, j \vDash \psi$ iff $\sigma, \tau, j \vDash \phi$.*

A *prompt* residue is one which is fully reduced to a truth value if the formula is guar-

anteed to be reducable. That is, if at least the parent formula's delay has passed since the residue's time step, ($\tau_j + \Delta^w(\phi) \leq \tau_i$) then the residue value must be a truth value.

Definition 2 (Residue promptness)**.** *The residue $r = \langle j, \phi \rangle_\psi$ is prompt at i iff for all traces σ, timestamp series τ, timesteps j, i where $j \leq i$, and formulas ψ, ϕ that $\tau_i - \tau_j \geq \Delta(\psi)$ implies $\phi \in \{\top, \bot\}$*

A *reducible* residue is one which has already been initialized by its starting state (replacing all free propositions with truth values) or which is a residue of the current state (which can be initialized). If a residue is not reducible at i then there is state unavailable at step i which is necessary to reduce the residue.

Definition 3 (Reducible residues)**.** *A residue $\langle j, \psi \rangle_\phi$ is* reducible *at i if $j = i$ or there are no free propositions in ψ (i.e., $\mathbf{freep}(\psi) = \emptyset$).*

A history structure is correct if all the residues it contains are correct.

Definition 4 (Correctness of history structures)**.** *A history structure S^i_ψ is correct iff for all timesteps i, formulas ψ, traces σ, and timestamp series τ: $\forall k \in [0, i], r^k \in S^i_\psi$ and r^k is correct.*

A history structure is prompt if all the residues is contains are prompt.

Definition 5 (Promptness of history structures)**.** *A history structure S^i_ψ is prompt if for all timesteps i, formulas ψ, traes σ, and timestamp series τ: $\forall r^k \in S^i_\psi$, r^k is prompt at i.*

A history structure is reducible if all the residues is contains are reducible.

Definition 6 (Reducibility of history structures)**.** *A history structure S^i_ψ is reducible if for all timesteps i, formulas ψ, traes σ, and timestamp series τ: $\forall r^k \in S^i_\psi$, r^k is reducible at i.*

With these properties defined, we can prove the correctness of the top-level algorithm $\mathbf{agmon}$. There are two separate properties we want for $\mathbf{agmon}$ to be correct. First, we want it to provide *correct* answers, i.e., it should only say a policy is violated if it is conclu-

sively violated and only report a policy is satisfied if it is conclusively satisfied. This means that if **agmon** reports a violation of policy ϕ at step j, then $\sigma, \tau, j \not\models \phi$. We also want to guarantee that we will get answers when they are available, so we want *Prompt* answers: **agmon** should report violations/satisfaction for any timesteps older than the policy delay. Thus, if $\sigma, \tau, j \not\models \phi$ then **agmon** must report the violation by time $\tau_j + \Delta^w(\phi)$.

Theorem 1 (Correctness and Promptness of **agmon**)**.** *For all $i \in \mathbb{N}$, all formula φ, all time stamp sequences τ and all traces σ it is the case that (1) if $\langle j, \bot \rangle \in S^i_\varphi$ then $\sigma, \tau, j \not\models \varphi$ and if $\langle j, \top \rangle \in S^i_\varphi$ then $\sigma, \tau, j \models \varphi$ (Correctness) and (2) if $\tau_i - \tau_j \geq \Delta^w(\varphi)$ then if $\sigma, \tau, j \not\models \varphi$ then $\langle j, \bot \rangle \in S^i_\varphi$ and if $\sigma, \tau, j \models \varphi$ then $\langle j, \top \rangle \in S^i_\varphi$ (Promptness)* .

Proof. By mutual induction over i and φ.

To show the correctness and promptness of **agmon** we just need S^i_φ to be correct and prompt.

We can show that each history structure S^i_ψ (where $\psi \in \{\varphi, \mathbf{tempsub}(\varphi)\}$) is correct, prompt, and reducible at every step by showing $\mathbb{S}^i_\phi$ and S^{i-1}_ϕ to be correct, prompt, and reducible at every step.

First, we show that $\mathbb{S}^i_\phi$ is correct, prompt, and reducible by construction due to the order the history structures are incremented. For all $\phi \in \mathbf{tempSub}(\varphi)$ in increasing order, $S^i_\phi \leftarrow \mathbf{incrS}(S^{i-1}_\phi, \mathbb{S}^i_\phi, \sigma, \tau, i)$. Because for all $S^i_\psi \in \mathbb{S}^i_\phi$, $|\psi| < |\phi|$ (by Lemma 2) and the iteration over ψ is done in increasing size, we know that $\mathbb{S}^i_\phi$ will be correct, prompt, and reducible for each subformula because all the history structures in $\mathbb{S}^i_\phi$ are smaller than ϕ and thus were already incremented (and correct/prompt/reducible due to **incrS** by Lemma 3).

Next, we show that S^{i-1}_ϕ is correct, prompt, and reducible at every step. At step $i = 0$, for all $\phi \in \mathbf{tempSub}(\varphi)$, S^{-1}_ϕ is trivially correct, prompt, and reducible be-

cause S^{-1} is empty. Together this means that S^i_ϕ is reducible, prompt and correct for all $\phi \in \mathbf{tempSub}(\varphi)$ at step $i = 0$. With all S^i_ϕ prompt, correct, and reducible for $\phi \in \mathbf{tempSub}(\varphi)$, $\mathbb{S}^i_\varphi$ is also prompt, correct, and reducible. Along with S^{-1}_φ being trivially reducible, prompt, and correct and $\mathbf{incrS}()$ being correct, S^0_φ is also prompt, correct, and reducible.

This proof step follows for $i > 0$. S^{i-1}_ψ is correct and prompt from the previous step. Since we increment the structures in order and $\mathbf{incrS}()$ is correct, $\mathbb{S}^i_\psi$ is also prompt and correct for every $\phi \in \mathbf{tempSub}(\varphi)$. This means that $\mathbb{S}^i_\varphi$ is correct and prompt and from this it follows that S^i_φ is correct and prompt.

4.4.2 Proof of agmon Correctness

In this section we present some assisting lemmas and definitions used to show the correctness of **agmon**.

First, we note that $\mathbf{simplify}(\psi)$ preserves the correctness of ψ. In other words, simplify does not change the truth evaluation of ψ at any step. This is clearly true since only simple syntactic and semantic reductions are used.

Lemma 1 (Correctness of $\mathbf{simplify}(\psi)$)**.** *For all positions i, formulas ψ, traces σ, and time sequences τ it is the case that if $\psi' \leftarrow \mathbf{simplify}(\psi)$ then $\sigma, \tau, i \vDash \psi \leftrightarrow \sigma, \tau, i \vDash \psi'$*

Proof. Trivial by the semantics of $\mathbf{simplify}$.

We also note that subformula are always smaller than their parent formula, which is clear since the size of a formula includes the size of all of its subformula.

Lemma 2 (Subformula sizes)**.** *For all formulas ϕ, ψ, if $\phi \in \mathbf{tempSub}(\psi)$ then $|\psi| > |\phi|$.*

Proof. If $\phi \in \mathbf{tempSub}(\psi)$ then ϕ is a subformula of ψ. Subformula are inherently smaller than their parent formulas since the size of the parent $|\psi|$ is at least the size of ϕ (it is contained in ψ) plus at least the one node which makes it a parent formula, thus $|\psi| > |\phi|$.

The $\mathbf{incrS}$ procedure is used to update history structures given a new input state. Here we show that $\mathbf{incrS}$ creates correct, prompt and reducable structures if its input is correct, prompt, and reducable. This relies on the fact that $\mathbf{incrS}$ only adds a residue for the current state and reduces this and all the existing residues in the structure. Since $\mathbf{reduce}$ preserves correctness, promptness, and reducability, $\mathbf{incrS}$ does as well.

Lemma 3 (Correctness of $\mathbf{incrS}$)**.** *For all traces σ, timestamp series τ, timesteps i, formulas ψ, correct, prompt, and reducible structures S_ψ^{i-1}, and correct, prompt, and reducible structure sets $\mathbb{S}_\psi^i$, if $S_\psi^i \leftarrow \mathbf{incrS}((S_\psi^{i-1}, \mathbb{S}_\psi^i, \sigma, \tau, i)$ then S_ψ^i is correct, prompt, and reducible.*

Proof. The correctness/promptness/reducibility of $\mathbf{incrS}$ follows from the correctness/promptness/reducibility of $\mathbf{reduce}$. For S_ψ^i to be correct, prompt, and reducible it must have a residue entry for all $k \in [0, i]$ and every entry must be correct, prompt, and reducible. From the definition of $\mathbf{incrS}$ we can see that all entries in $[0, i]$ exist in S_ψ^i because S_ψ^{i-1} is correct (and thus has a residue entry for all $k \in [0, i-1]$) and the $\langle i, \psi \rangle$ entry is added to this. All the newly reduced residues from S^{i-1} are correct/prompt/reducible because $\mathbf{reduce}$ preserves those properties and they were already correct/prompt/reducible (since S_ψ^{i-1} is correct/prompt/reducible). The new reduced residue $\langle i, \psi \rangle_\psi$ is also correct/prompt/reducible due to the correctness of reduce following that $\mathbb{S}_\psi^i$ is correct and

prompt and $i = j$ (so the residue is reducible).

All of **agmon** relies on **reduce** preserving correctness, promptness, and reducibility. We show this in three parts. First, we show that **reduce** preserves correctness. This means that all transforming reductions that reduce performs must preserve the residue's eventual truth value. Second, we show that **reduce** creates prompt residues. If enough time has elapsed since the residue's timestep, **reduce** is guaranteed to fully reduce the residual formula to a truth value. Lastly, we show that **reduce** preserves reducibility. Once a residue has had all of it's free propositions removed it is always reducible, so all that is necessary is that when reduce is called on a residue $\langle j, \phi \rangle_\psi$ with $i = j$ that it removes all free propositions.

Lemma 4 (Correctness of $\mathbf{reduce}(\sigma, \tau, i, \mathbb{S}^i_\psi, \langle j, \psi' \rangle_\psi)$)**.** *For all timesteps $i, j \leq i$, formula ψ, traces σ, timestamp series τ, formulas ϕ, where $\langle j, \phi \rangle_\psi$ is reducible at i and correct, reducible, and prompt history structure sets $\mathbb{S}^i_\psi$, if $\langle j', \phi' \rangle_\psi \leftarrow \mathbf{reduce}(\sigma, \tau, i, \mathbb{S}^i_\psi, \langle j, \phi \rangle_\psi)$ then $\langle j, \phi' \rangle_\psi$ is correct.*

Proof by induction over ϕ.

Case $\phi \equiv p$

There are two cases: $\sigma, \tau, j \vDash p \rightarrow \sigma, \tau, j \vDash \phi$ and $\sigma, \tau, j \vDash \phi \rightarrow \sigma, \tau, j \vDash p$.

Subcase 1 If $\sigma_j(p) = \bot$ then this is trivially true, so we look at the case $\sigma_j(p) = \top$. Since the residue is reducible and there is a free proposition (p) we know that $i = j$, and from the definition of reduce we can see that ϕ will reduce to $\sigma_i(p) = \top$. So, $\sigma, \tau, j \vDash \top \rightarrow \sigma, \tau, j \vDash \top$.

Subcase 2 Since the residue is reducible and there is a free proposition p, we know that $i = j$. There are two possible subcases here, either $\sigma_j(p) = \bot$ or $\sigma_j(p) = \top$. If $\sigma_j(p) = \bot$ then the implication is trivially true. If $\sigma_j(p) = \top$ then we have $\sigma, \tau, j \vDash \top \rightarrow \sigma, \tau, j \vDash p$ which is true from the semantics of p (since $\sigma_j(p) = \top$).

Case $\phi \equiv \neg\psi_1$

There are two cases: $\sigma, \tau, j \vDash \neg\psi_1 \rightarrow \sigma, \tau, j \vDash \phi$ and $\sigma, \tau, j \vDash \phi \rightarrow \sigma, \tau, j \vDash \neg\psi_1$.

Subcase 1 From the semantics of *not* we know that if $\sigma, \tau, j \vDash \neg\psi_1$ then $\sigma, \tau, j \nvDash \psi_1$. Since reduce is correct we can see from the definition of reduce that $\psi_1' \leftarrow \mathbf{reduce}(\langle j, \psi_1 \rangle)$ will be $\langle j, \bot \rangle$ and then $\mathbf{simplify}(\neg\psi_1')$ will return $\top$.

Subcase 2 If $\sigma, \tau, j \vDash \phi$ then from the definition of reduce we see that $\phi \leftarrow \mathbf{simplify}(\neg\phi')$. Since $\mathbf{simplify}$ is correct, this means that $\sigma, \tau, j \nvDash \phi'$. Since $\mathbf{reduce}$ is correct, from line one we see that $\sigma, \tau, j \vDash \phi' \leftrightarrow \sigma, \tau, j \vDash \psi_1$. So $\sigma, \tau, j \nvDash \phi' \rightarrow \sigma, \tau, j \nvDash \psi_1$ and thus $\sigma, \tau, j \vDash \neg\psi_1$.

Case $\phi \equiv \psi_1 \vee \psi_2$

There are two cases: $\sigma, \tau, j \vDash \psi_1 \vee \psi_2 \rightarrow \sigma, \tau, j \vDash \phi$ and $\sigma, \tau, j \vDash \phi \rightarrow \sigma, \tau, j \vDash \psi_1 \vee \psi_2$.

Subcase 1 From the semantics of *or* we know that $\sigma, \tau, j \vDash \psi_1$ or $\sigma, \tau, j \vDash \psi_2$. Because $\mathbf{reduce}$ is correct we know that either α' or β' will be $\langle j, \top \rangle$. Since simplify conserves correctness, this means ϕ' will be $\langle j, \top \rangle$.

Subcase 2 In this case we know that $\phi' = \langle j, \top \rangle$ and backtracking through simplify we see that either α' or β' must be $\top$. Since reduce is correct, this means that either $\sigma, \tau, j \vDash \alpha$ or $\sigma, \tau, j \vDash \beta$ which by the semantics of *or* gives us $\sigma, \tau, j \vDash \psi_1 \vee \psi_2$

Case $\phi \equiv \alpha \mathcal{U}_{[l,h]} \beta$

There are two cases, 1) $\sigma, \tau, j \vDash \phi' \rightarrow \sigma, \tau, j \vDash \alpha \mathcal{U}_{[l,h]} \beta$ and 2) $\sigma, \tau, j \vDash \alpha \mathcal{U}_{[l,h]} \beta \rightarrow \sigma, \tau, j \vDash \phi'$.

Subcase [$\sigma, \tau, j \vDash \alpha \mathcal{U}_{[l,h]} \beta \rightarrow \sigma, \tau, j \vDash \phi$**]** By the semantics of *until*, we know that $\exists k.(l \leq \tau_k - \tau_j \leq h).(\sigma, \tau, k \vDash \beta \wedge \forall k' \in [j, k-1].(\sigma, \tau, k' \vDash \alpha))$. Since S^i_β is correct we have $\langle k, \top \rangle \in S^i_\beta$ and since S^i_α is correct $\forall k' \in [j, k-1].\langle k', \top \rangle \in S^i_\alpha$.

Given these values in the history structs, we can see from the definition of reduce that $b_t \leq k$ and $a_u \geq k$ (if there is a k then $\forall k' \in [j, k-1].\langle k', \top \rangle \in S^i_\alpha$ and $a_u \geq k$ otherwise there is no such k and $a_u = i \geq k$ by the completeness of S^i_β). Combining these, we find that $a_u \geq b_t$ and so $\phi' = \top$.

Subcase [$\sigma, \tau, j \vDash \phi' \rightarrow \sigma, \tau, j \vDash \alpha \mathcal{U}_{[l,h]} \beta$**]** There are three possible subcases here: $\phi' = \top$, $\phi' = \bot$, and $\phi' = \alpha \mathcal{U}_{[l,h]} \beta$.

Sub-Subcase $\phi = \top$ If $\phi = \top$ then we know that $a_u \geq b_t$ and $b_t \neq \emptyset$. For this to be the case there must exist a k such that $\tau_k \in [\tau_j + l, \tau_j + h] \wedge \langle k, \top \rangle \in S^i_\beta$ (i.e., $b_t = k$). Since S^i_β is correct, we know that $\sigma, \tau, k \vDash \beta$. We also know from the definition of a_u and $a_u \geq b_t$ that $\forall k' \in [j, b_t - 1]$ that $\langle k', \top \rangle \in S^i_\alpha$ which means that for all these k' that $\sigma, \tau, k' \vDash \alpha$. Given this, we see from the semantics of *until* that $\sigma, \tau, j \vDash \alpha \mathcal{U}_{[l,h]} \beta$

Sub-Subcase $\phi' = \bot$ This case is trivially true, since $\sigma, \tau, j \nvDash \bot$.

Sub-Subcase $\phi' = \alpha \mathcal{U}_{[l,h]} \beta$ This case is trivial since ϕ' is returned.

Case $\phi \equiv \alpha \mathcal{S}_{[l,h]} \beta$ There are two cases, 1) $\sigma, \tau, j \vDash \phi' \rightarrow \sigma, \tau, j \vDash \phi$ and 2) $\sigma, \tau, j \vDash \phi \rightarrow \sigma, \tau, j \vDash \psi'$.

Subcase [$\sigma, \tau, j \vDash \alpha \mathcal{S}_{[l,h]} \beta \rightarrow \sigma, \tau, j \vDash \phi'$**]** By the semantics of *since*, we know that $\exists k.(l \leq \tau_j - \tau_k \leq h).(\sigma, \tau, k \vDash \beta \wedge \forall k' \in [k+1, j].(\sigma, \tau, k' \vDash \alpha))$. Since S^i_β is correct we

know $\langle k, \top \rangle \in S^i_\beta$ and from S^i_α being correct we have $\forall k' \in [k+1, j].\langle k', \top \rangle \in S^i_\alpha$.

Given these values in the history structs, we can see from the definition of reduce that $b_t \geq k$ and $a_s \leq k$ (from the definition of a_s and b_t). Combining these, we find that $a_s \leq b_t$ and so $\phi' = \top$.

Subcase [$\sigma, \tau, j \vDash \phi' \rightarrow \sigma, \tau, j \vDash \alpha \mathcal{S}_{[l,h]} \beta$**]** There are three possible subcases here: $\phi = \top$, $\phi = \bot$, and $\phi = \psi'$.

Sub-Subcase $\phi = \top$ If $\phi = \top$ then we know that $a_s \leq b_t$ and $b_t \neq \emptyset$. For this to be the case there must exist a k such that $\tau_k \in [\tau_j - h, \tau_j - l] \wedge \langle k, \top \rangle \in S^i_\beta$ (i.e., $b_t = k$). Since S^i_β is correct, $\sigma, \tau, k \vDash \beta$. We also know from a_s that that $\forall k' \in [k+1, j].(\langle k', \top \rangle \in S^i_\alpha)$ (i.e., $a_s \leq k$). Since S^i_α is correct and $a_s \leq k = b_t$ this tells us that $\forall k' \in [k+1, j].\sigma, \tau, k' \vDash \alpha$. Given this, we see from the semantics of *since* that $\sigma, \tau, j \vDash \alpha \mathcal{S}_{[l,h]} \beta$.

Sub-Subcase $\phi = \bot$ This case is trivial, since $\sigma, \tau, j \nvDash \bot$

Sub-Subcase $\phi = \psi'$ This case is trivial, since $\sigma, \tau, j \vDash \psi' \rightarrow \sigma, \tau, j \vDash \psi'$

As discussed, $\mathbf{reduce}$ also preserves residue promptness. For non-temporal formulas this is trivial since they are always prompt. The definitions of $\mathbf{reduce}$ for *until* and *since* are designed such that given the conclusive trace history (which is guaranteed after the formula delay) they will always return a truth value.

Lemma 5 (Promptness of $\mathbf{reduce}(\sigma, \tau, i, \mathbb{S}^i_\psi, \langle j, \psi' \rangle_\psi)$)**.** *For all timesteps $i, j \leq i$, formulas ψ, traces σ, timestamp series τ, formulas ϕ, and correct, reducible, and prompt history structures sets $\mathbb{S}^i_\psi$ if $\langle j', \phi' \rangle_\psi \leftarrow \mathbf{reduce}(\sigma, \tau, i, \mathbb{S}^i_\psi, \langle j, \phi \rangle_\psi)$ then $\langle j, \phi' \rangle_\psi$ is prompt at i.*

Proof by induction over ϕ.

All cases but $\phi \equiv \alpha \mathcal{U}_{[l,h]} \beta$ and $\phi \equiv \alpha \mathcal{S}_{[l,h]} \beta$ follow simply from reduce being prompt. We show these more difficult cases:

Subcase $\phi \equiv \alpha \mathcal{U}_{[l,h]} \beta$ For $\langle j', \phi' \rangle$ to be prompt at i we must see that $(\tau_i - \tau_j \geq \Delta^w(\psi) \rightarrow \phi \in \{\top, \bot\}$. First, we note that in this case both S^i_α and S^i_β are prompt at k such that $\tau_k \in [\tau_j, \tau_j + h]$. We show this for α and note that β follows. First, $\Delta^w(\phi) - h = max(\Delta^w(\alpha), \Delta^w(\beta))$ from the defintion of Δ^w. So $\Delta^w(\alpha) \leq \Delta^w(\phi) - h$. Since $\tau_i - \tau_j \geq \Delta^w(\phi)$ we have $\Delta^w(\alpha) \leq (\tau_i - \tau_j) - h$. Note that $\tau_k \leq \tau_j + h$, and substituting above we obtain $\Delta^w(\alpha) \leq \tau_i - \tau_k$. So if S^i_α is prompt at j, it is also prompt at these k's.

There are two possible cases, either there is no true β residue or there is a true β residue.

Subcase no-β In this case, $\forall k.(\tau_j + l \leq \tau_k \leq \tau_j + h).\langle k, \bot \rangle \in S^i_\beta$. If this is true and $\tau_i - \tau_j \geq \Delta^w(\psi)$ then we see from the definition of reduce that $b_n = \top$ and so $\phi' = \bot$.

Subcase β **exists** In this case, $\exists k.(\tau_j + l \leq \tau_k \leq \tau_j + h)$ such that k is the minimum step where $\langle k, \psi' \rangle \in S^i_\beta$ and $\psi' \not\equiv \bot$. Since S^i_β is prompt, ψ' must be $\top$. From the definition of $\mathbf{reduce}$ we see that $k = b_t = b_a$.

Within this case, there are two sub-cases: either α is true until β ($\alpha \mathcal{U} \beta$) or it is not ($\neg(\alpha \mathcal{U} \beta)$). In either case we know that S^i_α is correct and prompt at all k with $\tau_{k'} \in [\tau_j, \tau_k]$ (as shown above). So all residues $\langle k', \psi \rangle \in S^i_\alpha$ exist and all residual formulas $\psi \in \{\top, \bot\}$.

Sub-Subcase not α **until** β In this case, if $\exists k' \in [j, k-1]$ where $\langle k', \bot \rangle \in S^i_\alpha$ then from the definition of $\mathbf{reduce}$ we can see that $a_a < k = b_a$ and so $\phi' = \bot$

Sub-Subcase α **until** β Otherwise, $\forall k' \in [j, k-1].\langle k, \top \rangle \in S^i_\alpha$ (due to promptness of S^i_α). From the definition of $\mathbf{reduce}$ we can see that $a_u \geq k = b_t$ so $a_u \geq b_t$ and $\phi' = \top$.

Subcase $\phi \equiv \alpha \mathcal{S}_{[l,h]} \beta$ For $\langle j', \phi' \rangle$ to be prompt at i we must see that $(\tau_i - \tau_j \geq \Delta(\psi) \rightarrow \phi \in \{\top, \bot\}$. First, we note that in this case both S^i_α and S^i_β are prompt at k such that $\tau_k \in [\tau_j - h, \tau_j]$. We show this for α and note that β follows. First, $\Delta^w(\phi) + l = max(\Delta^w(\alpha), \Delta^w(\beta))$ from the defintion of Δ^w. So $\Delta^w(\alpha) \leq \Delta^w(\phi) + l$. Since $\tau_i - \tau_j \geq \Delta^w(\phi)$ we have $\Delta^w(\alpha) \leq (\tau_i - \tau_j) + l$. We can rearrange this to $-\Delta^w(\alpha) \geq \tau_j - l - \tau_i$. Note that $\tau_k \leq \tau_j - l$, and substituting above we obtain $-\Delta^w(\alpha) \geq \tau_k - \tau_i$. Flipping the sign again we see $\Delta^w(\alpha) \leq \tau_i - \tau_k$. So if S^i_α is prompt at j, it is also prompt at these k's.

There are two possible cases, either there is no true β residue or there is a true β residue:

Subcase no-β In this case, $\forall k.(\tau_j - h \leq \tau_k \leq \tau_j - l).\langle k, \bot \rangle \in S^i_\beta$. If this is true and $\tau_i - \tau_j \geq \Delta(\psi)$ then we see from the definition of reduce that $b_n = \top$ and so $\phi' = \bot$.

Subcase β **exists** In this case, $\exists k.(\tau_j - h \leq \tau_k \leq \tau_j - l)$ such that k is the maximum step where $\langle k, \psi' \rangle \in S^i_\beta$ and $\psi' \neq \bot$. Since S^i_β is prompt, ψ' must be $\top$. From the definition of $\mathbf{reduce}$ we see that $k = b_a = b_t$.

Within this case, there are two sub-cases: either α is true until β ($\alpha \mathcal{S} \beta$) or it is not ($\neg(\alpha \mathcal{S} \beta)$). In either case we know that S^i_α is correct and prompt at all k with $\tau_{k'} \in [\tau_k, \tau_j - l]$ (as shown above). So all residues $\langle k', \psi \rangle \in S^i_\alpha$ exist and all residual formulas $\psi \in \{\top, \bot\}$.

Subcase not α **since** β In this case, if $\exists k' \in [k+1, j].\langle k, \bot \rangle \in S^i_\alpha$ then from the definition of $\mathbf{reduce}$ we can see that $a_a \geq k' > k = b_a$ and so $a_a > b_a$ and $\phi' = \bot$

Subcase α **since** β Otherwise, $\forall k' \in [k+1, j].\langle k, \top \rangle \in S^i_\alpha$ (due to promptness of S^i_α). From the definition of $\mathbf{reduce}$ we can see that $a_s \leq k = b_t$ so $a_s \leq b_t$ and $\phi' = \top$.

Showing that $\mathbf{reduce}$ preserves reducibility is straightforward. A residue is reducible at all steps once all of its free propositions have been removed and $\mathbf{reduce}$ always removes any free propositions in the formula by evaluating them at the current state. Evaluating the propositions at the current step is the correct action to take since the residue must be reducible and thus if it has any free propositions, $j = i$. So, if there are any free propositions in a residue, $\mathbf{reduce}$ is guaranteed to remove them all so the resulting reduced residue will be reducible at all steps.

Lemma 6 (Reducibility of $\mathbf{reduce}(\sigma, \tau, i, \mathbb{S}^i_\psi, \langle j, \psi' \rangle_\psi)$)**.** *For all timesteps $i, j \leq i$, formula ψ, traces σ, timestamp series τ, formulas ϕ, and correct, reducible, and prompt history structure sets $\mathbb{S}^i_\psi$ if $\langle j', \phi' \rangle_\psi \leftarrow \mathbf{reduce}(\sigma, \tau, i, \mathbb{S}^i_\psi, \langle j, \phi \rangle_\psi)$ then $\langle j, \phi' \rangle_\psi$ is reducible at all steps.*

Proof.

There is only one important case. If $\phi \equiv p$ then from the definition of $\mathbf{reduce}$ we see that $\phi' = \sigma_i(p)$. This means that any free propositions are replaced with their truth value, so the returned residue will be reducible at all steps. For any other ϕ, $\mathbf{reduce}$ preserves the reducibility of the residue since only free propositions affect reducibility.

Chapter 5

Monitor Implementation

This chapter describes our implementation of the monitoring framework. We have implemented the basic monitoring algorithm as an ANSI C codebase which can be utilized in different scenarios. We have created both a PC-based offline monitor and an ARM based real-time CAN monitor implementation utilizing our framework monitor codebase.

5.1 Implementation Overview

The primary monitor codebase provides an implementation of the monitoring algorithm in portable ANSI C. It contains the necessary data structures and an implementation of the subcomponents of the **agmon** monitoring algorithm.

The code is designed to be used by a driver program which builds the system trace and implements **agmon** (or another monitoring algorithm based on **agmon**'s history structures). The driver program interfaces with the system, acting as the semi-formal interface, and fills the monitor data structures. It also implements the actual monitoring algorithm by incrementing the structures at the appropriate time.

The monitor specification is translated into configuration code which is compiled into the subsequent monitor program directly. This is an easy way to implement the monitor with the limitations described below in Section 5.1.1. This was done instead of a run-time configuration to simplify the implementation. A run-time configurable monitor could still be made in the exact same way as our monitor by adding logic to build the formula and data structures at runtime from the configuration instead of at compile time.

We currently have two implemented driver files, a PC-based offline monitor which reads comma sepearated value (CSV) formatted traces, and an embedded ARM driver which we run on an ARM-based development board to monitor CAN networks in real-time.

5.1.1 Embedded Limitations

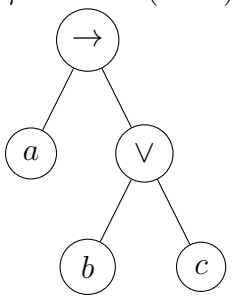
As has been noted, software designed for safety-critical embedded systems has more restrictive design constraints than typical system software. Two important limiting constraints which strongly affect the implementation of **agmon** are avoiding recursion and no dynamic memory allocation [116]. We focus on these limitations because they directly impact the implementation so it was imperative to ensure they could be handled in a practical way.

Static Memory Allocation Most safety-critical coding guidelines discourage or prohibit the use of dynamic memory allocations after initialization [117]. This is primarily to avoid memory leaks but also avoids unpredictable execution time within memory allocators and garbage collectors. Using only statically allocated memory also provides a straightforward upper bound on memory use and guides the system towards a more constant-time and constant-space design which helps in calculating worst case execution time.

While static memory allocation is safer and easier to analyze, dynamic allocation is useful. **agmon** makes use of dynamic sets and lists which add and remove residues constantly. Since our specification logic is bounded, it is possible to statically allocate storage space for the maximum number of elements in the history structure lists (and elsewhere when a temporary data structure is useful) in place of dynamic lists.

One issue from static allocation that permeates the entire design is how to handle representing formulas with limited memory. The straightforward approach to implementing **agmon** stores residue formulas as their abstract syntax trees (AST), which makes formula traversal and rewriting easy. Storing an AST for every residue in the monitor wil take up a lot of space for long formulas or formulas with long temporal durations. Statically allocating enough storage space to hold a full AST for every potential live residue in the system could require a large amount of storage space. Our solution is to use a single global formula tree and have the formulas in residues be indices into a table pointing to this global tree. This reduces the monitor's storage requirements dependence on formula durations. By using a global formula we only require an extra index for every timestep duration rather than an entire AST which allows us to monitor bigger formulas with long durations without running into space limitations. For a specification formula which has a one second duration and a 20ms monitoring period, the global formula only requires storing one global tree (the formula AST plus any reducable subtrees) rather than storing 50 individual formula ASTs. Even accounting for a large global tree (i.e., a formula with many reduction subformulas) and storage for the 50 residue indices, using a global tree can provide a large (5-50x) reduction in storage requirements.

One issue with global formula trees in our algorithm is that they must include every subformula of the specification that can occur in the monitor due to rewriting. When residues



Formula	Node ID	Global Formula
a	0	0
b	1	1
c	2	2
$b \vee c$	3	$2 \vee 3$
$a \to (b \vee c)$	4	$0 \to 3$
$a \to b$	5	$0 \to 1$
$a \to c$	6	$0 \to 2$

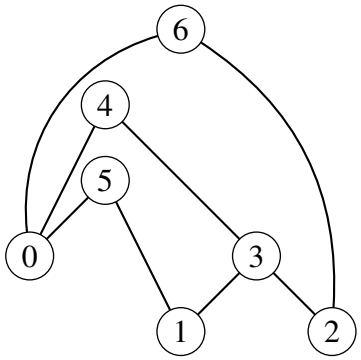


Figure 5.1: Static global formula representation

contain an actual AST it is easy to dynamically rewrite the formulas (i.e., just update the AST in place), but since we use pointers to a static set of trees we must ensure that every possible residual formula (including subformula in the history structures) has an entry in the global formula tree. For example, the formula $\psi = (a \vee b) \to (c \wedge d)$ contains possible reduced subformula such as $b \to d$ which do not directly exist as a subformula of ψ. All of these subformula which are reachable by reduce need to be present in the global formula tree which is generated by the specification compiler described in Section 5.1.2. Figure 5.1 shows a specification formula AST and the global table and global formula tree that would be generated from it. The global formula includes the extra subformulas ($a \to b$ and $a \to c$) required by reduce that are discussed below.

Avoiding Recursion Recursion is also generally prohibited in safety-critical design guidelines because it can be difficult to guarantee a maximum stack depth when using recursion. Obviously, **agmon** utilizes recursion heavily, so we need a way to implement the recursive aspects of the algorithm without actually recursively calling the **reduce** function.

Instead of recursively calling **reduce** over a residue, we utilize an iterative traversal implementation which walks the formula tree. The iterative **reduce** is based on a straight-

forward iterative depth-first traversal. Nodes cannot be marked as visited during traversal since traversal is done over the global formula tree. Instead, the three statically allocated stacks (`traversal`, `values`, and `direction`) are used to keep track of the traversal. The `traversal` stack is used as in a traditional iterative depth-first search to hold list of nodes to be visited. Every step of the iterative reduce loop, the next node to visit is popped off the `traversal` stack. The `direction` stack keeps track of the traversal directions (up/down and left/right) and is used to decide whether to continue traversal down the formula tree (when traversing down) or evaluate the current node (when traversing up). The `values` stack is used to hold the values of evaluated nodes, so when evaluating a node, the values of its evaluated children can be found next on the `values` stack. Figure 5.2 shows a call of the iterative reduce algorithm on a simple formula. At each step, the current node location of the traversal is highlighted, while the table shows each stack's state at the beginning of that iteration.

5.1.2 System Specifications

The implementation takes system specifications as a list of ascii-formatted BMTL formulas. This list is given to the configuration compiler which generates a C file which is compiled into the monitor. This file provides the functions to build and increment the history structures for the given specification formulas. This method was chosen as a straightforward and simple way to configure the monitor with a static allocation. It is also possible to provide a memory space which could then be filled by the monitor itself dynamically at startup allowing the monitor to be reconfigured with a new specification without recompiling or reflashing. This setup would be preferable in a commercial design to gain the benefits of a quickly configurable monitor (although recompiling/flashing is not slow with

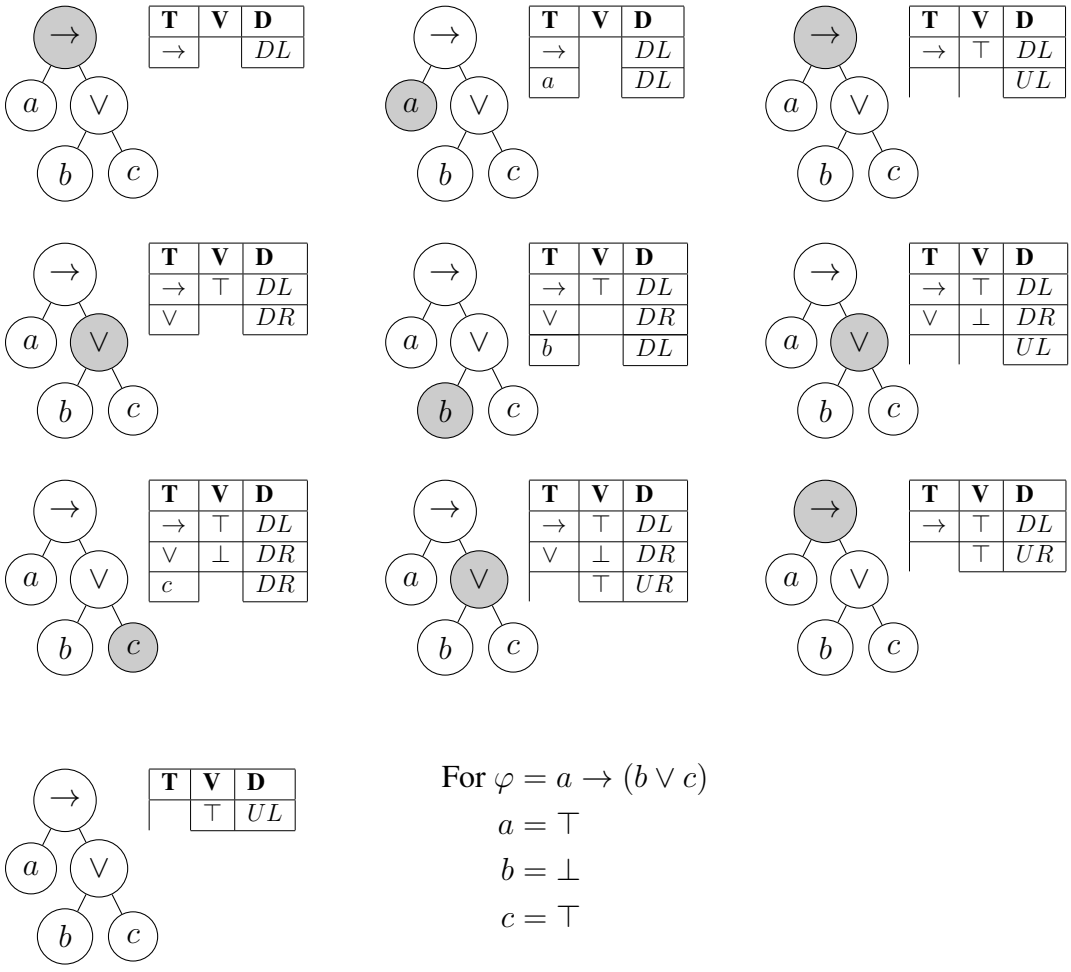


Figure 5.2: Example of iterative reduce execution

the current code). The semi-formal interface is defined separately as a part of the monitor driver and is triggered by incoming messages.

The specification compiler generates code for the following tasks:

- Simplification tables for constant-time lookups to simplify
- Static allocation of all the necessary data structures
- Initializing the history structures
- Building the global formula table and tree
- Defining configuration values (e.g., number of specification rules, proposition names, etc.)

5.1.3 Optimizations

There are many possible optimizations and useful trade-offs that can be implemented on top of the base **agmon** algorithm. Two high-value optimizations that are extremely useful for efficient monitoring are interval history structures and directly implementing the extended logic operators. Due to the high value of these optimizations, they are used in our implementation. The implementation of these optimizations is discussed in this section and they are evaluated in Section 6.1.1.

5.1.3.1 Intervals

The basic approach to implementing **agmon** requires iterating over the history structure lists to find the desired entries when checking temporal formula. This requires performing a search over every entry in the set, which can be a problem for formulas with long durations (i.e., long lists) or nested temporal formulas where the number of checks increases

$$S^i_\phi = \begin{cases} \langle 0, \top\rangle, \langle 1, \top\rangle, \langle 2, \top\rangle, \langle 3, \bot\rangle, \\ \langle 4, \bot\rangle, \langle 5, \phi'\rangle, \langle 6, \phi\rangle, \langle 7, \top\rangle \end{cases} \equiv \begin{cases} T : \{[0,2],[7,7]\} \\ F : \{[3,4]\} \\ R : \{\langle 5, \phi'\rangle, \langle 6, \phi\rangle\} \end{cases}$$

Figure 5.3: Comparison of history structure list and interval representations

multiplicatively with each nesting.

Instead of storing individual residues for each reduced timestamp, we can combine reduced residues into intervals of truth values. Rather than only containing a list of residues, each interval history structure contains a `true` interval list (T), a `false` interval list (F), and a list of active residues (R). When a residue reduces to true or false, its timestamp t is added into the appropriate interval list by either extending an existing interval in the list or adding the empty interval $[t, t]$. Instead of iterating over an entire list of residues, checking temporal formula using the structures requires searching the shorter list of intervals. Figure 5.3 compares the two history structure representations. Even in this small case, there are less entries to search over in the interval notation than the list notation.

Although in the worst case this optimization provides no benefits (same number of items, potentially more absolute space, and not increasing speed), realistic traces rarely are worst case with respect to the interval notation. In fact, realistic traces tend to fit nicely within this optimization by rarely having unreducable holes and because truth values tend to cluster in real systems (due to physical/control inertia and modes/activation guards).

5.1.3.2 Additional Logic Operators

The specification logic we use defines a moderately minimal logic which utilizes notational equivalences for many common logical operators. While this keeps proofs of correctness

small it can reduce the efficiency of the monitor.

First, there are considerable space savings to be had by reducing the size of the specification formulas since a small change in formula size can be a large change in monitor configuration size (due to the need to explicitly define every reachable subformula). For example, a very straightforward formula $(a \vee b \vee c \wedge d)$ translates into $(a \vee b \vee \neg(\neg c \vee \neg d))$ which requires 62 formulas in the global formula tree. By directly implementing the *and* operators we can reduce this to 18 formulas – a considerable space savings. The savings increases as more equivalence-defined operators are utilized, with $a \wedge b \wedge (c \rightarrow d)$ requiring 402 formulas versus 18 when fully implemented.

Besides reducing the amount of space used, reducing the size of a formula can shorten the execution time of **reduce** by reducing the number of nodes it needs to visit. Every aliased *and* node adds up to six extra iteration steps (three nodes both descending and ascending) more than a direct implementation due to the three additional *not* nodes in the translation ($a \wedge b \equiv \neg(\neg a \vee \neg b)$). Even more importantly, the directly implemented temporal operators can potentially save checking an entire list. For example, to check $\Diamond_{[l,h]} b$ we only need to search over the history structure of b, but if we check this in the restricted logic as $\top \; \mathcal{U}_{[l,h]} \; b$ we have to check the history list of $\top$ as well (although using intervals makes checking the $\top$ list fast).

Implementing six extra common operators (*and*, *implies*, *always*, *eventually*, *past always*, *past eventually*) is simple and straightforward. Showing the correctness is straightforward as well (though we omit that here) since the nontemporal operators are straightforward and the temporal operators are sub-operators of *until/since* (their implementations exist within the *until/since* implementations).

5.1.3.3 Semi-Aggressive

The least amount of work that can be done to completely check a trace is to check each trace entry once. The way to ensure this occurs is to check each step once after it is guaranteed to be checkable. This is the basic conservative check (based on *prècis*), which evaluates each formula after waiting long enough to guarantee that it can be evaluated (i.e., at least Δ^w time). Aggressive checks perform more work, doing extra checks hoping that the formulas can be evaluated early. The benefit of aggressive checking is that we may detect specification violations earlier than conservative approaches, providing more time to attempt a recovery action. This extra recovery time may help avoid or reduce the damage caused by a violation (e.g., engaging emergency stop earlier means a vehicle may avoid a collision or collide with less speed).

There are situations, such as when fully aggressively checking a formula would take too much computation time to guarantee correctness but checking a portion of the existing residues aggressively is necessary or useful. A straightforward implementation of a semi-aggressive monitoring approach is a hybrid conservative/aggressive approach which performs conservative checking and uses any spare time to check remaining residues aggressively. This approach, which we use and discuss in Section 5.2.2.3, utilizes a conservative checking approach to provide a promptness guarantee while still trying to benefit as much as possible from aggressive checking. This is not the only possible hybrid approach, but any method which provides some assurance of correctness or completeness will likely be similar since all steps must eventually be checked.

All semi-aggressive approaches require deciding the order in which the history should be checked (and thus, which residues may not get checked aggressively). This decision is system dependent and expert knowledge can increase the efficiency of this type of monitor-

ing. Simple factors such as deciding between checking oldest or newest residues first are obvious tuning mechanisms, but there may be situations where more dynamic strategies (e.g., check the oldest residue and if it doesn't reduce start checking from newest) may perform better if certain likely system behaviors/inertia are known. More specific approaches such as only aggressively checking certain specification rules or only aggressively checking when the system is in a certain mode may also be useful.

Choosing a monitoring approach boils down to understanding the target system and trying to attain the best monitoring results with the given monitoring resources. This is anchored on the level of correctness guarantees that can be made by any given approach. To provide guarantees about checks beyond the simple conservative promptness, more timing analysis is necessary to understand exactly how many residue checks can be performed in the worst case. Given a certain number of checks, an aggressive strategy which is guaranteed to execute the highest priority checks can be used.

5.2 Implementations

We currently have two monitor implementations built on the monitoring code framework, a PC-based CSV log monitor and an embedded ARM CAN monitor. This section describes these two implementations.

5.2.1 PC-based Monitor

The PC-based monitor is an offline monitor designed to check specifications over CSV formatted log files. Most log formats can be converted into CSV's in a straightforward manner, and the CSV logs are reasonably easy for a person to manually check for false

positives/negatives and even try to diagnose identified violations. A monitor targeting CSV logs is versatile since text-based tools can easily manipulate the logs. The sampled state view of a system also maps perfectly onto CSV logs, where each line entry can be a new timestamped state with each column representing a specific system property.

The PC-based monitor contains the necessary framework calls to set up monitoring, the interfacing code to read CSV logs and fill the framework's state buffer, and the code to actually perform the monitoring.

The monitor is designed with separate conservative and aggressive checks. They are configured such that either can be enabled or disabled, allowing this monitor to be used to investigate performance with the different algorithms. When both are enabled the conservative checks run first for all policies before the aggressive checking loop is run.

When executed, the monitor initializes itself (building the formulas, initializing data structures, etc.) and then enters a loop which performs the following for every entry in the log file:

1. Updates the system state buffer from the log entry (i.e., semi-formal interface)
2. Increments the existing history structures with the new state
3. Adds the specification policies to their history structures
4. If enabled, performs a conservative check of each policy (i.e., checks the oldest residue)
5. If enabled, performs aggressive checking of each policy

The separated aggressive/conservative checks simplify performance analysis for the different implementations. The monitor also has configuration flags for other optimizations, allowing us to enable or disable the full logic implementation and intervals (although the

full logic using lists is not implemented and thus disallowed). The specification compiler takes these configuration flags and enables features using `#defines`.

5.2.2 Embedded ARM Monitor

Runtime monitoring of real-time embedded systems is only useful if it can be performed fast and cheap enough to be practical. We have implemented an **agmon** based monitor on a STMicro STM32F4-Discovery development board and designed it to enable real-time monitoring of CAN networks (although other interfaces are certainly possible).

5.2.2.1 Monitor Hardware

The STM32F4 microcontroller is a 32-bit ARM Cortex-M4 based microcontroller targeted at DSP and other high-performance applications. This chip has a reasonable amount of external connectivity including 6x USARTS, I2C, CAN, Ethernet, SDIO, and multiple ADCs. More importantly, this chip comes in large memory options, with the version we use containing 1MB of flash and 192KB SRAM. Although this chip is likely more powerful (and costly) than would be used to implement a mass-produced commercial monitor, the extra memory provides a useful amount of leeway during implementation which allows us to experiment with the limits of scalability of the approach.

The STM32F4-Discovery development board is a simple demonstration board for the STM32F4 microcontroller. The board includes an ST-LINK/V2 debug tool, LEDS, a push-button, a few other accessories (MEMS, accelerometer, microphone, etc.) and extension headers for microcontroller I/O including the CAN controller. The microcontroller has a CAN controller but does not include a CAN transceiver, so we utilize an external TI CAN transceiver on a powered breadboard to interface with the target CAN networks.

5.2.2.2 Simulator

The μVision IDE from Keil (ARM's development tools branch) is a standard IDE for ARM based chips which includes a microcontroller simulator. Unfortunately, the simulator does not fully implement all the peripherals of our STM32F4 chip, so some peripherals, including the Nested Vector Interrupt Controller which handles processor interrupts, are simulated using the default peripheral simulation driver. This is primarily an issue because we would like to simulate interrupts for both the CAN controller and some timers. It is possible to manually trigger interrupts through a dialog but we cannot automatically simulate timer interrupts periodically. This leads us to do most evaluation with the PC implementation or onboard instead of in simulation, which unfortunately provides less relevant information (from the PC version) or is much less flexible (onboard). Still, simple debugging, especially of non-timing related aspects can be performed in simulation rather than on the microcontroller.

5.2.2.3 Hybrid Conservative/Aggressive Implementation

Although we can calculate worst-case execution time for checking any given specification we must perform a new analysis every time any specification rule is changed at all. Some specifications may not be worst-case schedulable under aggressive monitoring (which can be much more computationally expensive than conservative monitoring). This means that for those specifications, we cannot get the early detection benefits that come with aggressive checking without giving up some guarantees of monitor correctness. Specifications which may in practice always be checkable can have worst-case execution times which are too slow to guarantee monitor correctness. In many instances the system trace will not exhibit worst-case monitoring behavior and an aggressive monitoring scheme may be able to fully

check a property. Other specifications may have execution times such that we can guarantee that they can be fully aggressively checked. Even if we know that in practice that the aggressive check would always finish, the monitor cannot be trusted with high-criticality if it isn't guaranteed to finish executing on time.

In order to safely mix guaranteed specification checking with the benefits of aggressive checking we have implemented a hybrid approach combining a conservative wait-to-check algorithm (based on *prècis* [53]) with our aggressive algorithm. Under our periodic sampling design, the conservative check is guaranteed to only need to check a single residue for each specification policy in every step (plus updating structures or saving history once per step). This conservative check can be done quickly at each period, leaving any extra time until the next check for aggressive checking. This provides a guarantee that at least the specification is checked within a known delay (i.e., a promptness guarantee) and allows the monitor to aggressively check as much as possible. As long as we know that the worst case execution time for message handing, incrementing the structures, and a single residue check is short enough to finish within a monitor period then we are guaranteed at least a (delayed) correct and prompt output.

Any early detection, even if it is not guaranteed, is still a benefit and can be important for system safety if it provides additional time to perform a recovery. The cases where the aggressive check is not guaranteed to finish are where a semi-aggressive strategy or an intelligent order for aggressively checking can really provide an increased benefit (such as increasing the likelyhood of aggressively catching failures).

The embedded monitor is designed as a multi-tasking system with four steps which are separated into three tasks: updating the trace (i.e., the semi-formal interface), updating the structures, the conservative check, and the aggressive check. This system is implemented

Task	**Priority**	**Location**
Trace Update	Highest	CAN receive ISR
Structure update	High	Timer 2 ISR
Conservative Check	High	Timer 2 ISR
Aggressive Check	Low	Main Loop

Table 5.1: ARM hybrid monitor task allocation

as an interrupt service routine (ISR) based multitasking system. This could also be implemented through multi-tasking without ISRs, but the ARM implementation is a bare-metal C implementation so using the CAN receive ISR for interface handling and timer ISRs to control the periodic monitor check was simpler than building a task scheduler. The trace update task is performed inside the CAN receive ISR. The structure update and conservative check are performed by a timer ISR, and the aggressive check is performed in the monitor's main loop. This task assignment is shown in Table 5.1

This is a straightforward allocation of work which fits well with the desired monitoring scheme. The trace update task keeps a copy of the system state updated constantly. Whenever a new incoming message is received, this interrupt is called and the task performs the necessary semi-formal mapping to update the current state. The monitor's checking frequency is controlled by the Timer 2 interrupt. The monitor is currently configured to run at a 25ms period, so every 25ms the Timer 2 interrupt fires, running the structure update and conservative check function. This function samples a copy of the current system state. It then increments the history structures based on this new state and performs a conservative check for each policy.

Once the monitor is configured and the main monitor thread enters its checking loop, it continuously performs an aggressive check over the specification policy structures. The aggressive and conservative checks share the same policy lists and history structures so

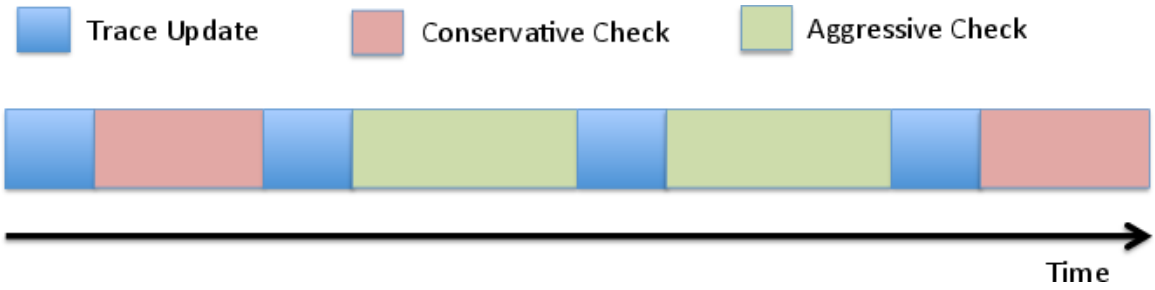


Figure 5.4: Hybrid monitor ideal task schedule

that they do not duplicate work. This requires some straightforward locking to ensure the aggressive check does not check inconsistent data. If the aggressive checker detects that it has been interrupted by the conservative check, it throws away the in-progress check and starts over at the new current (i.e., the next) state. If the aggressive check finishes checking the specification before the checking period is over, it enters a busy-wait until the next period occurs.

In an ideally scheduled monitor, the execution would occur as depicted in Figure 5.4, with the aggressive monitor finishing all checks within each period. Figure 5.5 shows the execution of the embedded monitor instrumented to output the currently executing task to an oscilloscope. With this specification, the specification used in Section 6.3 plus another 200 residue *eventually* rule, there was still a large portion of idle time – 23ms of the 25ms loop was spent idle. This shows the aggressive checking finished reasonably quickly and the monitor could handle much longer duration or more complex formulas before the execution time was bad enough to not guarantee it will complete the aggressive check.

In the next section we evaluate the monitor performance and scalability.

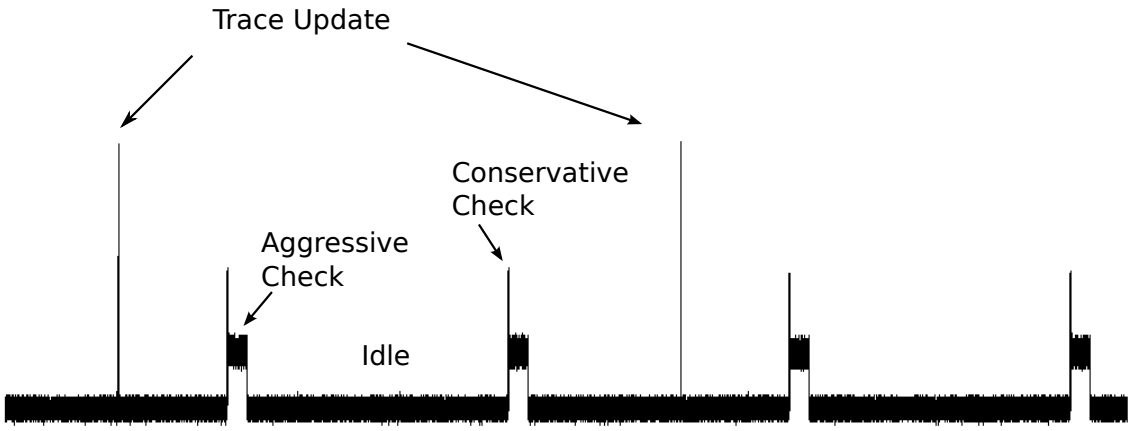


Figure 5.5: Oscilloscope capture of embedded monitor task execution

Chapter 6

Monitor Evaluation

In this chapter the monitor implementation is evaluated in use checking logs from real systems under test. We run the PC-based monitor against artificially generated logs to analyze basic performance characteristics as well as use it to check logs obtained from a test drive of a semi-autonomous research vehicle. The embedded monitor is used to check a replay of CAN logs from testing of a separate autonomous vehicle component.

6.1 Analysis

Looking at both the algorithm and our implementation of **agmon**, there are three primary parameters that affect the overall space usage and execution time: the number of policy formulas, the number of required history structures and the total formula delay of these structures. The actual values that occur in the trace and the monitor's checking frequency also have a strong effect.

We can divide the **agmon** evaluation loop into four phases: trace capture, updating the structure, the conservative check, and the aggressive check as shown in Figure 6.1.

1: Initialization (all the vars, interface, build structure, build formula).
2: **for** every step **do**
3: **Trace Capture**: update system state model
4: **History Update**: increment history structures and policy lists
5: **Conservative Check**: for each policy, check oldest residue
6: *With available time*: **Aggressive Check**: for each policy, check entire residue list

Figure 6.1: **agmon** Evaluation loop

Trace capture is the phase where the monitor takes in the trace data and fills the required propositions. Depending on the complexity of the mapping this can be negligibily fast (e.g., copying boolean bytes out of the trace/network into variables) or more complex (e.g., executing state machines, performing running averages/integrations of floating point values). We recommend that the mapping used be relatively simple, so this phase is a small component of monitoring, or at least an quick one in absolute terms. The practical limitation here is that the mapping computations need to be computable in real-time to keep up with the incoming system values coming from the network as well as leave enough time between mappings to allow the monitoring algorithm to complete.

The history structure update phase is where the history structures are updated. This step includes adding the current step to each structure and reducing each unresolved residue in every structure. In the worst case this leaves us with $\sum_{S_\phi \in \mathbb{S}_\psi} \Delta^S(\phi)$ reductions (to check formula ϕ). If some formulas can be reduced before their delay, this can be reduced in the best case up to the delay times (i.e., $\sum_{S_\phi \in \mathbb{S}_\psi} 1$).

The conservative check is a single step of reduce for every policy rule, so this is only affected linearly with the number of policies. The aggressive check requires a reduce for every unresolved residue for every rule, so similar to the history structure we have $\sum_{S_\phi \in \mathbb{S}_\psi} (1 + \Delta^S(\phi))$ reductions in the worst case and no reductions in the best case (since the conservative check already handles them).

From this we can see that the worst scaling issues will come from increasing temporal formula durations (big Δ^S's) and then from the number of history structures required (more $S_\phi \in \mathbb{S}$) and the total number of policies. The worst case of adding an additional policy rule is that the rule is completely independent of the existing specification which adds all the required work to check the new rule. This is the same as running a separate monitor with the rule. In some cases rules may share subcomponents such as history structures. Adding rules which share history structures require less extra work to check than independent rules.

6.1.1 Artificial Traces

The first thing we look at is the implementation's scalability against different parameters (such as number of formulas, temporal formula durations, formula size, etc.) to compare optimizations and to get a sense for the realistic limits. We perform these tests with our PC-based offline monitor checking artificially generated traces to keep the analysis simple. In these analyses we use the notion of *best* and *worst* case traces to see how the algorithm performs given different types of traces. Different formulas have different best/worst case traces, so the trace values used will be noted with each formula. We also note that we do not care much about absolute time values here, except as a benchmark for potential performance. We can compare relative times to understand scaling and basic monitor behavior, but since these runs are performed on a PC using non-real time clocks and sharing resources with other processes there are caching issues, jitter, and other delays which would not affect a bare-metal implementation such as our ARM monitor. All values presented in this section are averages of five runs unless otherwise noted.

Figure 6.2 shows the computation time per monitor step of a few simple formulas. We can see that as expected, runtime per step is constant for a given formula as trace length

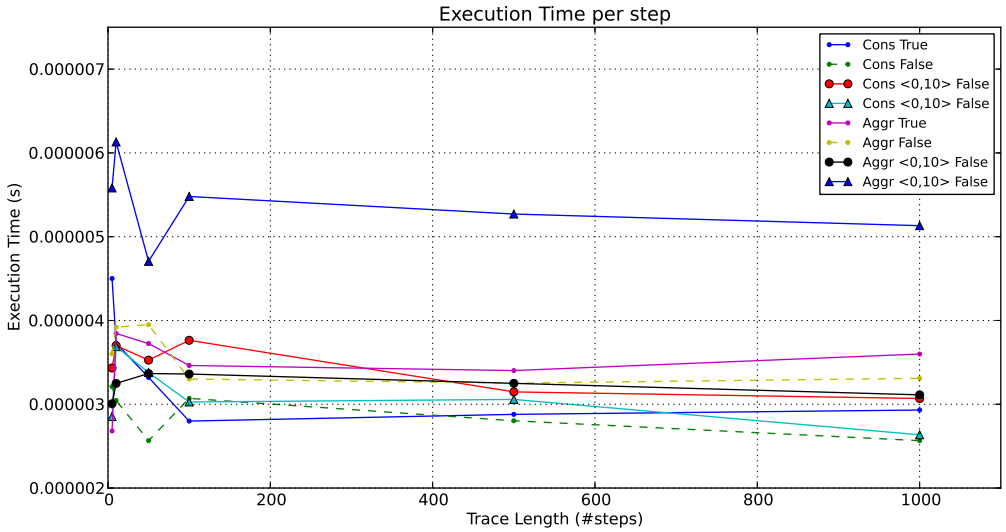


Figure 6.2: Execution time per step for simple formula

increases. The straight propositional policies (true and false) and the best-case eventually policies (i.e., always true traces which always reduce) all have similar checking times while the aggressive worst-case eventually (all false values) has a slightly higher runtime per step because it has to check more live residues at every step.

The number of policies checked also has linear execution scaling. Each additional policy is in the worst case independently checked, so it's as if a separate monitor was run for each policy. This is shown in Figure 6.3 which shows the execution time for an increasing number of policy formulas. The figure shows the execution time per monitor step for two different types of specifications, one filled with unique propositions and another with unique eventually formulas. Each of these policies is checked with both all true valuations and all false valuations. As expected, the execution time increases linearly with each new policy rule and the eventually rules take longer per step than the plain proposition rules.

Increasing the duration of a temporal formula (e.g., $\Diamond_{[0,X]}p$ for increasing X) causes

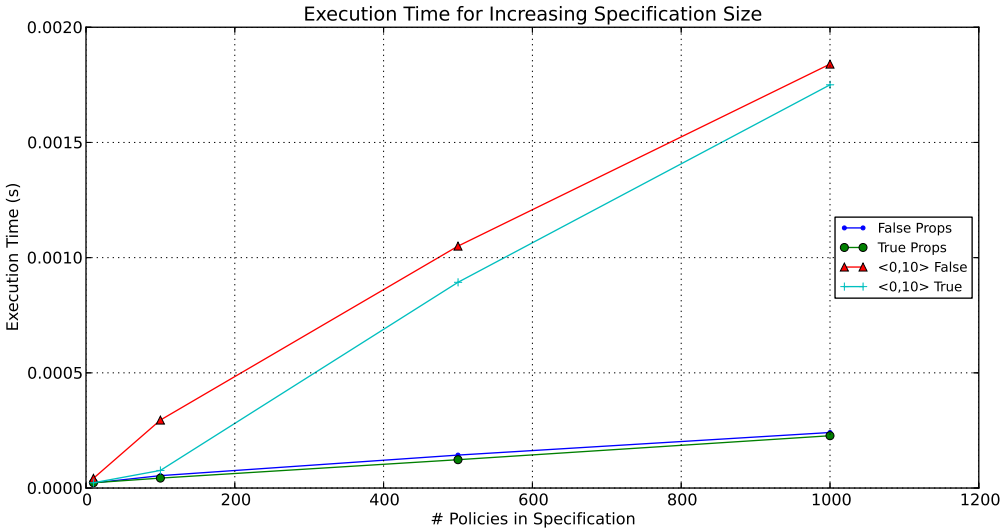


Figure 6.3: Execution time per step of specifications with increasing number of policies

a linear increase in execution time. Figure 6.4 shows this increase with both best (true) and worst (false) case traces. The conservative algorithms and best case aggressive (true proposition values) checking stay relatively constant time. Conservative is naturally constant since only one check is performed in every step, regardless of the policy. In this best case situation, the aggressive algorithm also only performs one check every step because the formula can be reduced at every step so there are no left over unreduced residues to check later. Worst case aggressive execution time (false proposition values) increases in the expected manner, since each additional step in the duration adds an additional residue which will be live and needs to be checked in every step. Nested temporal formulas can add a multiplicative scaling, as shown in Figure 6.5, although we can see that utilizing intervals can keep the scaling modest.

Intervals vs. Lists As discussed in Section 5.1.3, straightfoward algorithm optimizations including intervals and implementing a more full logic semantics can provide strong prac-

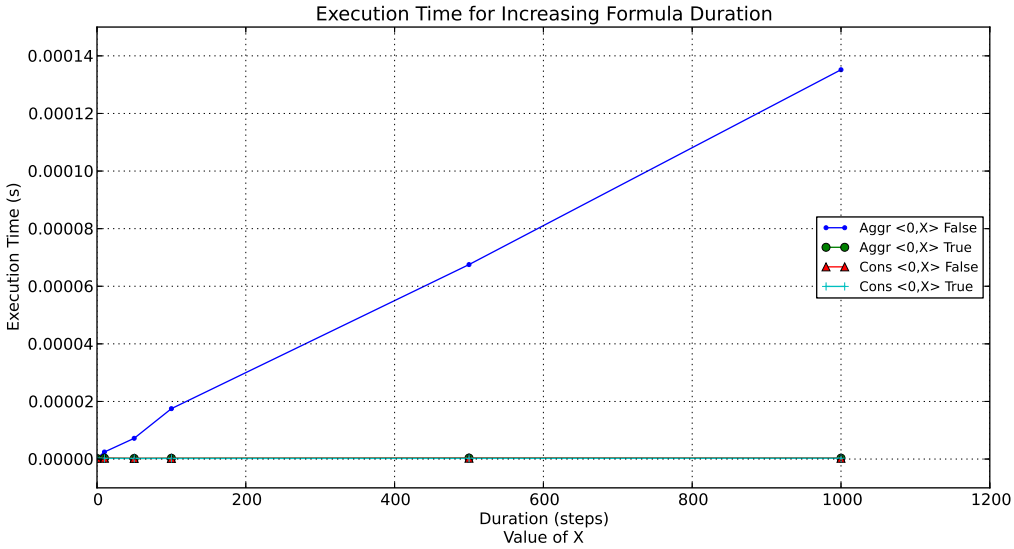


Figure 6.4: Execution time for formulas with increasing temporal durations

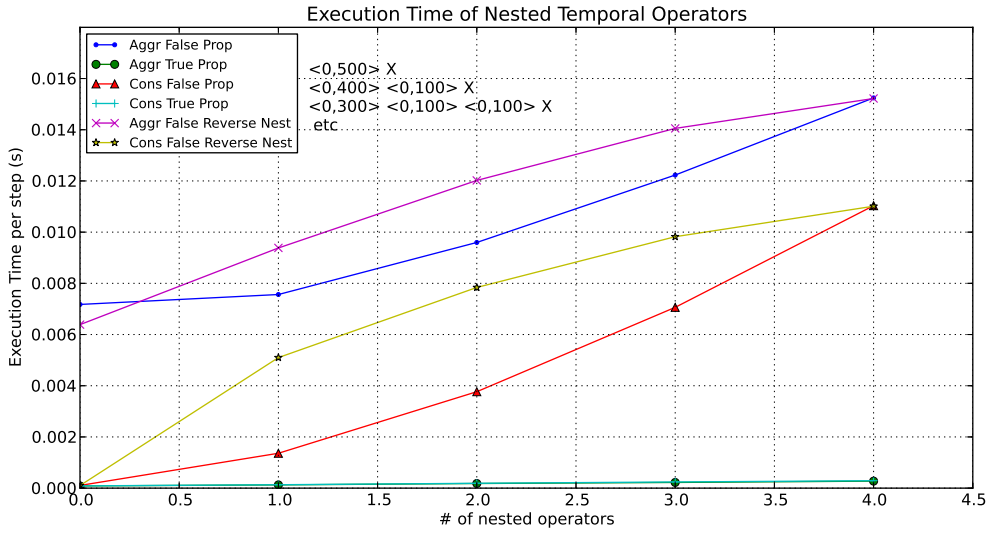


Figure 6.5: Execution time for nested temporal formulas

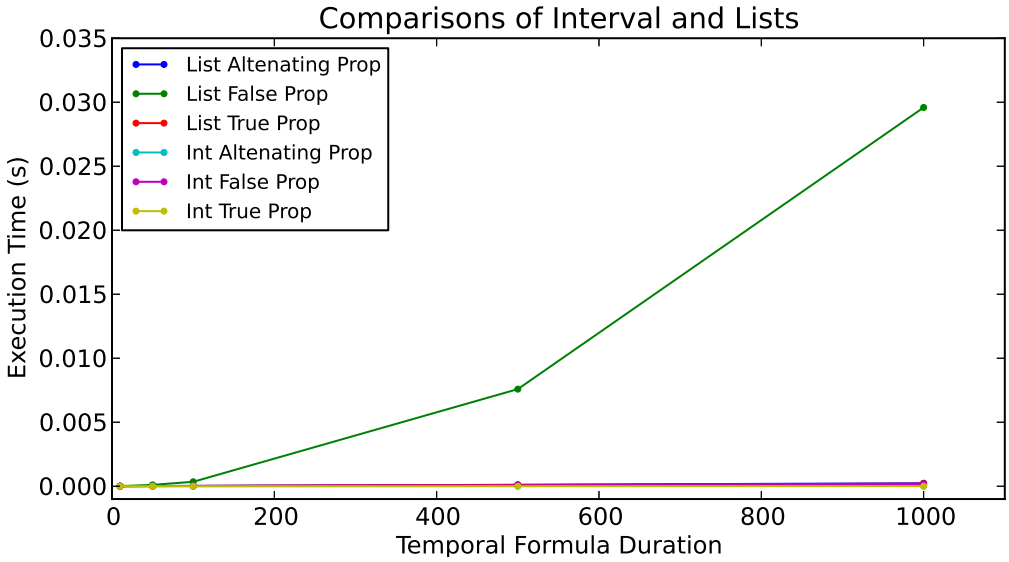


Figure 6.6: Comparison of intervals and lists

tical benefits. Figure 6.6 compares the execution time of monitoring with residue intervals rather than basic residue lists for formulas with different durations. The worst case aggressive behavior for lists here is a false value which requires iterating over every step in the duration. Using lists with a worst-case temporal trace (eventually with all false values) has an execution time with gets much worse as the duration increases. Both using intervals or a having a nicer (i.e., not worst case) trace keep the execution time down. The interval implementation tends to have a smaller iteration since any adjacent residues will combine, reducing the total number of iterations. The worst case behavior for intervals can be worse than lists if the trace does not permit the combining of any steps (i.e., no two adjacent timesteps have the same value), but this is unlikely. It requires a trace which matches the specific worst-case of the specification, and even in some of these cases intervals can still outperform lists.

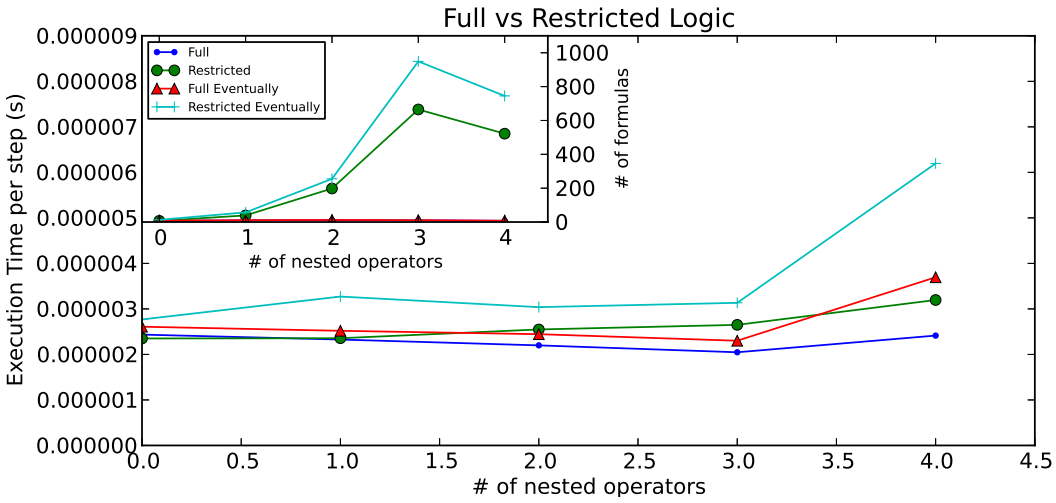


Figure 6.7: Comparison of full and restricted logic

Extended Logic The other major optimization that we've implemented is defining more of the logic semantics directly. As discussed in Section 5.1.3.2 the size of the specification formulas can be reduced by directly implementing the extra commonly used operators. The memory savings alone are important for embedded implementations which have limited available memory, but smaller formulas can also save execution time. Not only are smaller formulas easier to check since they requires fewer iterations to traverse, but the directly implemented temporal operators (eventually, always, etc.) are easier to check than their until- or since-based definitions. Figure 6.7 shows the execution time and number of formulas with respect to the number of unrestricted subformulas in the specification. This uses the formula $a \vee b \vee c \vee d \vee e$ where each added extended operator transforms one of the or's ($\vee$) into and and ($\wedge$). For example, two nested operators uses the formula $a \vee b \vee c \wedge d \wedge e$ Execution time is not heavily affected for this simple formula and trace set, but the number of required formulas is controlled much better using the full logic (three nested operators requiring 11 full-logic formulas versus 950 restricted logic formulas)

The benefits gained by directly implementing the extended logic operators obviously depend on how pervasive the new operators are in the specification. On one side, a specification which does not use the extended operators will gain no benefit (and in fact pay some cost from the larger code size). On the other hand, certain specifications and traces can create real differences. For example, checking the rule $\Diamond_{[0,50]}\Box_{[0,49]}value$ with a trace that has runs of values for 50 timesteps at a time takes half the time with the full logic implemented (5s vs 3s).

6.2 Offline Vehicle Logs

In this section we describe the use of our offline PC-based monitor as a tool to check the logged results of a vehicle test for safety and other interesting property violations. We used this monitor to check test logs from a research vehicle under development at a commercial automotive research lab.

The tests we analyze are from an approximately 16 hour drive with the primary purpose of testing the autonomous lane centering (ALC) feature. The test logs were provided in a proprietary format and were converted to CSV logs for use with the monitor.

The vehicle is a standard modern automobile which also contains new autonomy features, some under development at the research facility and some COTS components from third-party suppliers. We primarily focus on high level safety properties for a few obviously safety-critical features including adaptive cruise control and automated lane keeping.

6.2.1 Rule Elicitation

Specification rule elicitation was somewhat informal. Because the vehicle was in a rapid state of development and some features were supplied by third-parties, there was no formally defined specification of the vehicle or features. Also, no system safety requirements documentation was available to us. Since we could not obtain any documents which clearly led us to a useful monitoring specification, we first had to identify our own system specification. We derived our specification rules based on an analysis of existing design metrics, existing system design documents, discussions with system engineers, and observable system state available in the test logs.

The full logs have over 35,000 individual values (fields). We identified 244 potentially interesting values and exported logs containing these values to reduce the total amount of data being investigated (and help get a better understanding of what values were available).

Based on previous monitoring experiences and the testing's focus on the lane keeping feature, we attempted to identify a set of rules which would trigger on lane keeping issues. Because we had minimal documentation the tests logs were used extensively to understand the basic system dynamics and architecture. This was a prime example of a black box testing situation. We have no system implementation information whatsoever, even which messages each feature actually utilizes was not entirely clear from the network's data dictionary.

Our two primary monitoring targets were the adaptive cruise control and lane keeping features. With this in mind we looked for observable state which could be used to monitor safety rules about ACC and ALC behavior such as feature state and appropriate sensor messages. We identified cruise control messages which provided ACC control state, ACC acceleration requests, and the ACC selected speed among others. We also identified lane

Table 6.1: Offline log monitoring specification

Rule #	Informal Rule BMTL
0	If the brake pedal has been pressed, then within 200ms cruise control should be disengaged for 100ms $BrkPedalPressed \rightarrow \Diamond_{[0,200]}\Box_{[0,100]}\neg CrsControlEngaged$
1	The vehicle should not be within 1m of either lane edge for 1s while the automated lane centering feature is active $\neg\Box_{[0,1000]}(ALCEngaged \wedge ((rdrDistToLeftLT1 \wedge rdrDistToLeftVal) \vee (rdrDistToRightLT1 \wedge rdrDistToRightVal))$
2	The vehicle should not be within 1m of either lane edge for 1s while the automated lane centering feature is active $\neg\Box_{[0,1000]}(ALCEngaged \wedge (visDistToLeftLT1 \vee visDistToRightLT1)$

centering outputs and multiple sensors which provided lane information. With the knowledge of what system properties were observable, we created the three specification policies shown in Table 6.1.

Rule #0 comes from the simple idea that the driver should be able to override the cruise control system by depressing the brake pedal. We gave the system a 200ms response time between the pedal press being acknowledged on the bus and the cruise control feature acknowledging that it has disengaged. We also require that the cruise control stay disengaged for a 100ms duration. This helps to ensure that control is actually relinquished. Without this, a single 10ms step where the cruise control message stated it was disengaged (even if it re-engaged afterwords) would be enough to satisfy the rule. This is a prime example of a problematic ambiguity in informal specifications; we have to make a decision about the actual meaning of "ACC is disengaged".

Rules #1 and #2 are the same rule covering two different input sensors. There are multiple lane position inputs available in the system although we don't know which are actually used by the features. By comparing these sensor values with other information

Table 6.2: Offline log monitoring propositions

Proposition Name	System Variables	Mapping
$BrkPedalPressed$	Brake Pedal Travel message	Direct Boolean
$CrsControlEngaged$	ACC Activated message	Direct Boolean
$ALCEngaged$	Lane Centering Feature state message	Value Comparison
$rdrDistToLeftLT1$	Radar Left Lane Position message	Value Comparison
$rdrDistToLeftVal$	Radar Left Lane Position validity message	Direct Boolean
$rdrDistToRightLT1$	Radar Right Lane Position message	Value Comparison
$rdrDistToRightVal$	Radar Right Lane Position Validity message	Direct Boolean
$visDistToLeftLT1$	Vision Left Lane Position message	Value Comparison
$visDistToRightLT1$	Vision Right Lane Position message	Value Comparison

in the traces (including captured video) we chose two different but generally compatible sensor messages, one radar and one vision based, to use in our monitor specification. This mostly was exploratory in nature. It's easier to monitor both sensors and examine the results before finding a good way to choose one or merge them. Monitoring both sensors also helps look at system "ground truth", as we'd expect both sensors to agree reasonably with the actual state. If they do not mostly agree then we can assume something is wrong.

Given these policies and the observable system data, we need to create our semi-formal mapping. Table 6.2 shows the propositions we monitor and the basic mapping they use. Note that with these rules we only used direct boolean mappings (i.e., taking a single bit from the network message and using that as the proposition value) and arithmetic comparisons (i.e., comparing an integer or floating point value to a threshold). None of the more complex powers of the semi-formal interface (e.g., saved values, averages, etc.) were necessary.

6.2.2 Monitoring Results

We used the PC-based monitor to check the previously mentioned rules over our test traces. The 16 hours of traces were split into 240 files, each approximately four minutes long (an artifact from the original proprietary logs which are large even at 4 minute durations).

Running the monitor with the specification given above took approximately 4 minutes (under 3 minutes of compute time according to the unix `time` program) on a 2.5Ghz Intel Core i5 laptop with 8GB of RAM. Of the 238 log files tested, 65 contained specification violations.

There we no violations of Rule #0. To see if the brake was used to cancel cruise control in the logs, we tested the instanteous response version of Rule #0 $BrkPedalPressed \rightarrow \neg CrsControlEngaged$. There were violations of this rule in the logs. Seeing these expected violations reinforces our understanding of the system dynamics and shows that the brake was applied while cruise control was engaged in the logs. This is a simple example of using the monitor to explore the system for debugging purposes. It's useful to be able to quickly check whether a certain scenario occured in the logs. This also helps check our understanding of the system and specification.

We did find violations of both Rule #1 and #2 which may be actual problems and thus would be useful to bring to the attention of the system designers. While violations tended to occur in both rules relatively simultaneously, there were logs where one of the rules was violated and the other was not. We chose these vision and radar sensor values because they seemed accurate based on manual inspection and they generally contained matching values. The vision sensor lane position value has more significant figures than the radar sensor message. So while the two sensor values agree under rounding, there are situations in the data where this rounding affects whether a trace satisfies or violates the monitoring

specification. We could tune our specification to try to avoid these types of situations, but we do need to be aware of the resolution of our data when writing specifications We can see that the choice of which sensor value to monitor could affect our results.

6.2.3 Exploratory Example

Creating a monitoring specification for a system with limited documentation such as the scenario above requires experience and intuition. A benefit of our monitoring framework is that it can be used to easily explore a black box system such as this, even if we are limited to testing logs. Exploring a system by trying different policies and adjusting thresholds or formula durations based on the monitoring results is a straightforward method to help understand the target system when other information is unavailable. In general, this type of exploration can be done by identifying a desired property and trying a tightly bound (i.e., possibly overspecified) policy that expresses the property. It is easier to start with an overly-strict policy because any false positives provide information which can then be used to tune the policy. The location (e.g., the timestep) of a potentially false-positive violation can help identify the portions of a log which should be further investigated to improve understanding of the system.

We saw one way how this can be done in the previous section when we checked that braking did not instantaneously cause cruise control to be disabled. We explore this type of usage more thoroughly with a separate monitoring property. In this case, we want to specify a policy which ensures that the ACC feature does not command acceleration when it is too close to a target vehicle.

We first need to pick the general requirement we want to explore. Headway is a measurement of the time for a following vehicle to cover the distance between it and a lead

vehicle [118]. In this case, we can use headway as our notion of "too close" to the target. A commonly used minimum safe headway value is 2s [119], so that will be our starting point. Deciding how to represent the ACC feature commanding acceleration is more difficult. This is a part of the intent estimation problem discussed in Section 3.4.1.1. Before we make a decision on how we will represent this, it's best to inspect the system and see what state is available to work with.

Armed with this general idea for a rule, we can inspect the system documentation and test logs to identify the observable state which we will use to monitor this property. For this system, we find an obstacle tracking feature which provides us with (among other values) targets, distances, and relative velocities. It turns out that our exported log values do not include the vehicle speed. This provides an opportunity to demonstrate using the semi-formal interface to utilize what observable state is available. Instead of checking the headway directly with our desired headway time, we will use the ACC set speed and our desired headway time to calculate our desired following distance which we can compare to the actual following distance. We also have ACC messages which contain activation state, engine torque requests, target speeds, etc. As we have done previously, we can start by using an increase in ACC torque request as a metric for ACC acceleration.

Because the obstacle tracker provides separate messages for each of its tracking channels, we have two options for monitoring this data. We could create a rule for each channel, monitoring them independently. We could also use the semi-formal interface to choose the in-path obstacle and fill the propositions with the appropriate channel's values so we only need to monitor a single policy. These two options may not have the same meaning depending on the implementation. One difference that needs to be accounted for is if there is a lane change by the host vehicle or target vehicles such that the host is always behind a tar-

Table 6.3: Propositions for headway specification

Proposition Name	**System Variables**	**Mapping**
$VehInPath$	Obstacle X In Path message	Boolean Choice
$HeadwayLT2s$	Obstacle X Distance ACC Driver Selected Speed	Compare distance to speed calculated headway
$ACCAccel$	ACC Torque Request	History Differential

get vehicle. The version with independent rules for each channel will not see a continuous headway in this case whereas the merged rule may. If this is seen as a continuous headway a violation could be detected that is caused by following two separate vehicles too closely for too long.

Generally we would suggest using separate rules to keep the semi-formal interface as simple as possible, but here we will use the merged version to demonstrate some of the semi-formal interface's power. Our policy rule is thus $(VehInPath \wedge HeadwayLT2s) \rightarrow \Diamond_{[0,200]}\Box_{[0,100]}\neg AccAccel$. Informally, if there is a vehicle in our path and the headway is less than two seconds, then in the next 200ms there must exist the start of a 100ms duration where the ACC torque request is not increasing. The timing durations will likely need adjustment. We need them to allow for the component's response time and have some margin for small transients. We have just picked "reasonable" values until we have more information.

The policy rule is straightforward here, but the semi-formal mapping is more involved. In this case the mapping requires multiple system state variables to be used in the creation of individual propositions. Table 6.3 describes the propositions and the inputs we will create them with.

Checking this rule over the test logs, we find 21 logs which contain violations. Inspecting a few of these logs, we find some where the system seems to be behaving in a

reasonable, but not specification satisfying, way. In these cases when the distance gets smaller towards the rule's threshold the torque requests are also decreasing until no more torque is being requested. Since it looks like the system is behaving reasonably, we can try lengthening the duration of the eventually subforumla, giving the ACC more time to respond to the small headway.

Increasing the eventually duration to 500ms and then 1s removed many but not all of the violations. Assuming this specification rule is reasonable (i.e., that the headway should always recover in some set amount of time), we have either found a real problem or one of our approximations is wrong. We could continue trying to alter the specified rule, perhaps changing our headway approximation or choosing a new method to decide the ACC is accelerating besides any increase in torque. At this point, however, we would be best served by finding more information about the system and whether this rule is reasonable or that we are specifying a behavior that is not required of the system.

6.3 Embedded Monitor

To evaluate the embedded monitor we need to check an actual CAN bus. Since we did not have access to a real system with CAN and a data dictionary to monitor, we instead performed real-time replay of timestamped CAN logs captured during testing of a real system onto a bench CAN bus which the monitor was connected to, as shown in Figure 6.8.

The system under test that provided the logs is an autonomous heavy truck which is being designed for use in vehicle platoons. The system has multiple internal buses, some CAN and some Ethernet, connecting different system components. The logs we monitor are from

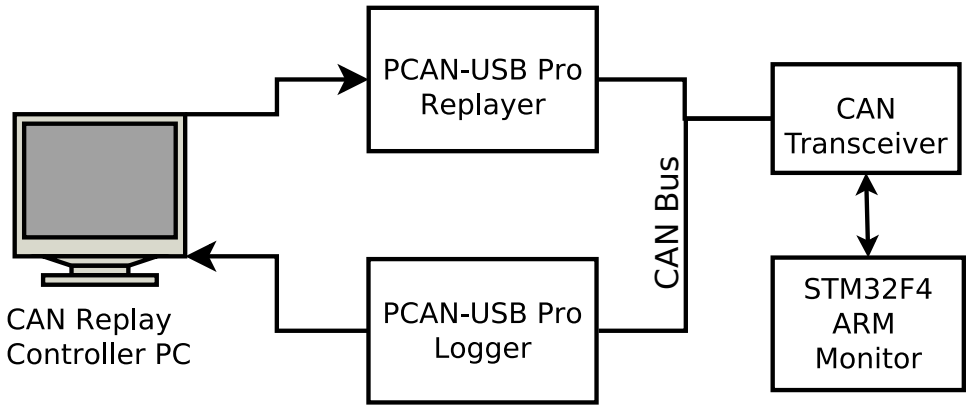


Figure 6.8: CAN replay setup

robustness testing of the interface controller, so we focus on its CAN bus which contains communication between the interface controller and the primary vehicle controller. The logs contain both normal operation as well as some operation under injected robustness testing. During robustness testing, the testing framework can hijack targeted network messages on the bus to inject testing values.

A PC was connected to a PCAN-USB Pro [120] device which provides a USB interface to two CAN connections. One CAN channel was used as the log replayer, while the other was used as a bus logger for analysis purposes. We performed log replay with a PC-based script which would take a test log and replay it on the CAN bus based on the log's timestamps. A separate script used the second CAN connection to log the CAN network traffic. The replay timing is based on a busy-wait using the log timestamps. We compared the replayed log timings to the original test logs to ensure this replay was accurate. The percent error in message timestamps relative to the start of the message in our longest logs had an average of less than 0.01% error. The absolute error was generally sub-millisecond, which is accurate enough for our 25ms monitor with its greater than 50ms minimum time step resolution.

Rule #	Informal Rule BMTL
0	A feature heartbeat shall be received every 500ms $HeartbeatOn \rightarrow \Diamond_{[0,500ms]} HeartBeat$
1	The interface component heartbeat counter is correct $HeartbeatOn \rightarrow HeartbeatCounterOk$
2	The vehicle shall not transition from manual mode to autonomous mode $\neg((\blacksquare_{[1,1]} IntManualState) \wedge IntAutoStat)$
3	The vehicle controller shall not command a transition from manual mode to autonomous mode $\neg((\blacksquare_{[1,1]} VehManualModeCmd) \wedge VehAutoModeCmd)$
4	The vehicle shall not transition from system off mode to autonomous mode $\neg((\blacksquare_{[1,1]} IntSDState) \wedge IntAutoStat)$
5	The vehicle controller shall not command a transition from system off mode to autonomous mode $\neg((\blacksquare_{[1,1]} VehSDModeCmd) \wedge VehAutoModeCmd)$

Table 6.4: CAN replay monitoring specification

6.3.1 Rule Elicitation

For this system we did have requirements documentation which could more directly lead to a monitoring specification. Since we wanted to monitor the interface bus, we identified requirements which we could monitor or partially monitor based on the state available on this bus. The specifications we used on the embedded monitor are shown in Table 6.4 and the propositions used in these rules are described in Table 6.5.

The rules were derived from the vehicle safety requirements documentation. Limited to the observable state on the interface bus, we used the user interface LEDs as proxies for the actual system state. This is an approximation we would feel is reasonable in most systems, and in this case there were also safety requirements which state that the output LEDs should be correct. This provides us more assurance that the approximation is reasonable.

Rule #0 is a heartbeat detection which ensures that the interface component is still run-

ning (essentially a watchdog message). Rule #1 is a second component of this check. The system's heartbeat message contains a single heartbeat status bit which we checked directly in Rule #0, but the message also has a rolling counter field. We used the semi-formal interface to create a proposition that represents whether the counter is incrementing correctly (i.e., one value at a time). To block false-positive violations during initialization, we blocked these rules from being checked until after the first heartbeat message was received by creating a guard proposition $HeartbeatOn$ with the semi-formal interface. Initialization issues are discussed in more detail in Section 6.4.0.4.

We also watched the system state for illegal state transitions. Although the actual mode decisions and commands are on a different bus, we can still monitor the system state through the user interface LEDs which show the mode state. We created rules for two of the illegal transitions, from manual mode to autonomous driving and from system off to autonomous driving. We independently checked both the vehicle controller's LED command messages and the interface's LED status message for these transitions.

The proposition mappings here were straightforward. The LED command and status messages were single bit fields and we checked the heartbeat's single bit status message. We checked the rolling heartbeat counter for consistency (i.e., that it counted up and wrapped correctly) by comparing it against the previously seen value in the semi-formal interface. We also created the guard $HeartbeatOn$ which is false until the first heartbeat message is seen, and true from then on. Using a guard proposition is our primary method to implement unbounded since/until type rules within our bounded logic. If we had unbounded operators we could use the formula $(\Diamond_{[0,500]} Heartbeat)\ \mathcal{S}\ (Heartbeat)$ instead of a mode-based guard proposition, but without unbounded operators using guard propositions to enable or disable a formula is a reasonable replacement.

Table 6.5: CAN replay propositions

Proposition Name	**System Variables**	**Mapping**
$HeartBeat$	Feature Status Message Heartbeat field	Fresh direct boolean
$HeartbeatOn$	Interface HB message	System Mode
$HeartbeatCounterOk$	Interface HB message	Comparison with Past Value
$VehManualModeCmd$	Vehicle command message	Direct Boolean
$VehAutoModeCmd$	Vehicle command message	Direct Boolean
$VehicleSDModeCmd$	Vehicle command message	Direct Boolean
$IntManualStat$	Interface status message	Direct Boolean
$IntAutoStat$	Interface status message	Direct Boolean
$IntSDStat$	Interface status message	Direct Boolean

6.3.2 Monitoring results

We monitored the CAN log replays on our test CAN network with the specification discussed above. To capture violations for analysis we configured the monitor to send a CAN message denoting the violated policy when violations were detected. These violation messages were rate limited to one message per second to allow the violation message to have a high CAN priority without letting it take over the network.

The monitor found heartbeat violations in the logs captured during robustness testing. Three different types of heartbeat violations were identified after inspecting the monitor results. The first is a late heartbeat message. In one of the robustness testing logs the heartbeat message was not sent on time, which is clearly a heartbeat violation. Figure 6.9 shows the heartbeat counter values and the inter-arrival time of the heartbeat messages over time for this violation. We can see here that the heartbeat counter did in fact increment in a valid way, just too slowly.

The second violation is on-time heartbeat status message but the heartbeat status field is 0. We do not know from the available documentation whether a bad status in an on-time

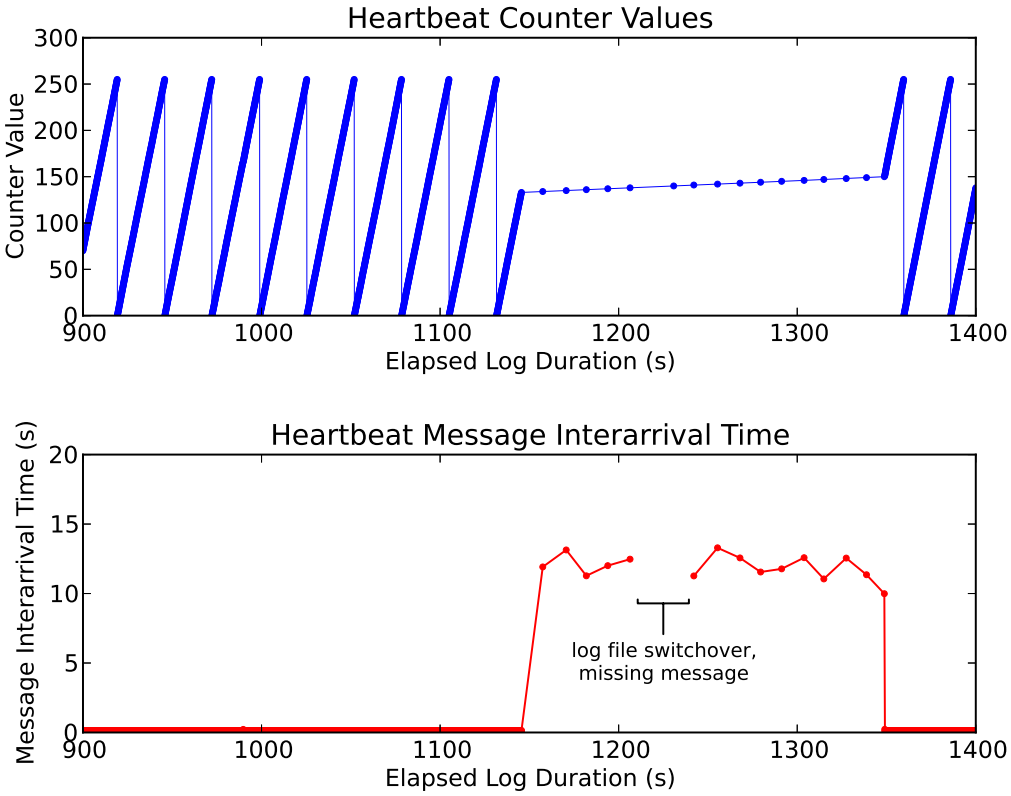


Figure 6.9: Heartbeat counter values over time

message with a good counter is valid or not. So without more information we cannot tell whether these violations are false positives or not. This is worthy of further investigation.

The last type of violation is a bad counter. We have defined a good counter as one which increments by one every message up to its maximum (255 in this case) before wrapping back to zero. Every consecutive heartbeat status message must have an incremented heartbeat counter or a violation will be triggered. Figure 6.10 shows the counter value history for one of the traces with a heartbeat violation caused by a bad counter value. Further inspection of this violation showed that the bad counter values were sent by the testing framework rather than the actual system. In this case, the network traffic the monitor is seeing is not real system state but actually it is messages being injected by the testing framework. This is not a real violation (since the violating state is not the actual system state), and so we consider this a false positive violation.

Different counter restrictions could also be used, such as allowing unchanging as well as incrementing values or requiring an increment to occur within a time threshold rather than every message. Once again, without documentation to point us towards the right restriction, all we can do is try a restriction and after seeing the monitoring results decide whether we believe our restriction is accurate or causes too many false positives.

There were also violations of the transition rules, but these, similar to the heartbeat counter violation, also turned out to be false positives triggered by message injections by the robustness testing harness. Since the monitor checks network state, if we perform testing that directly affects the values seen on the network (such as injection/interception of network messages) we may detect violations which are created by the testing framework rather than the system. This is a common issue when using monitors as a test oracle with some sort of fault or behavior injection – the monitor needs to know which state is from

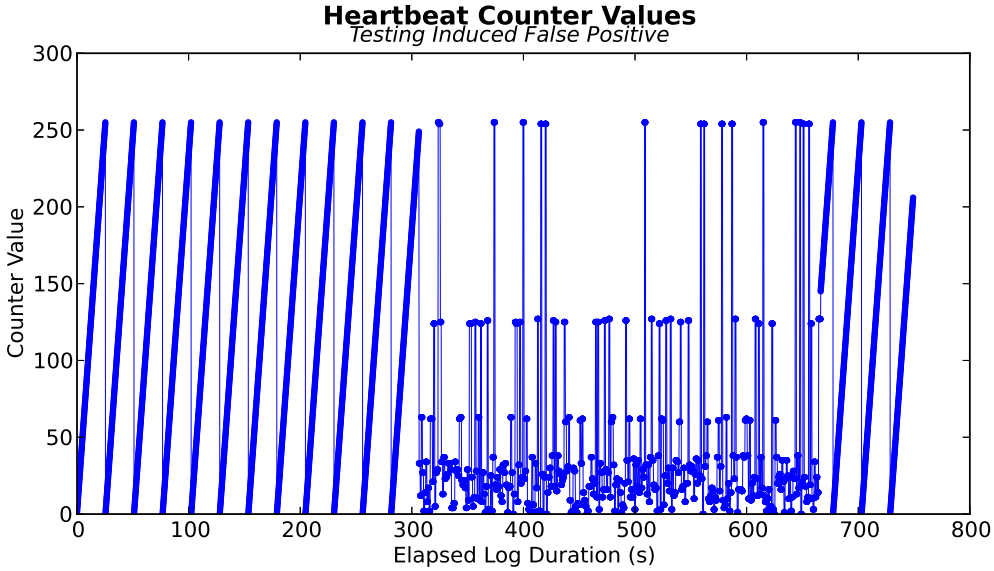


Figure 6.10: Bad heartbeat counter values

the test and which is from the system. Information about the test configurations can be used to filter out these types of false positives which arise from test-controlled state. This type of filtering can be automated if the test information can be input to the monitor, either directly on the network (e.g., adding a message value to injected messages) or through a side-channel (i.e., building a testing-aware monitor).

Comparing the violation messages from the monitor with the actual network state we can see the monitor's detection speed. The deteection time for the monitor should approximately be the monitor's period plus the time to perform the monitoring and the time to send the detection message once the violation is detectable. This is approximately two monitoring periods (given that the time to send a high priority message is negligable). This bears out in the replay testing, where the violation messages come approximately 555ms after the last good heartbeat message, which is a 55ms response time and close to double the monitor's 25ms period.

6.4 Lessons Learned

In this section we discuss some lessons learned from the evaluation of the monitor implementation. In particular, we look at four important lessons:

1. The optimizations and trace values affect monitor execution time.
2. The monitor specification quality depends on the quality of system requirements and documentation from which it is derived.
3. Detection accuracy, especially handling false positives, depends on an accurate monitor specification.
4. The monitor specification must take state initialization and system modes into consideration.

6.4.0.1 Optimizations and Traces

Checking the artificial traces has shown that the algorithm scales about as expected. Increased formula durations and additional temporal subformula (which means saving more history) can cause increases in monitor execution time per step. The two major optimizations we implemented, intervals and the extended logic, help keep the monitor execution time down in the face of these parameters. Interval structures keep the execution time lower as temporal durations increase and the extended logic keeps the required monitor storage (i.e., number and size of history structures) down.

Another important thing to note is how the specific trace values affects the execution time. The difference between best and worst case traces can be very large. Because formula duration and number of checks are the biggest contributors to execution time, bad traces which leave many residues (requiring a lot of checks) cause much worse performance than

good traces which may only require one single residue check per step even for complex formula. It is possible, and may be desired, to write specification rules designed specifically to avoid worst-case behavior based on the known usual state of the system. Modifying rules with this in mind may help make worst-case traces either extremely unlikely or even less bad.

6.4.0.2 System Requirements

It is clear that having accurate and complete requirements to derive the monitoring specification is key. The usefulness of monitoring results stems from having a specification that provides a useful envelope of system behavior. If the system requirements can be used to generate the formal monitoring specification directly then many of the specification elicitation and exploration issues we saw do not exist. Good requirements need a full understanding of the scope of observable system state, including the available sensors, sensor accuracy, and contain explicit state requirements rather than implicit intent. Less complete requirements can obviously still be used, as we showed by performing monitoring essentially without written requirements. Building a monitor specification that isn't directly generated from the system requirements requires experience, time spent exploring the available system information (both documentation and monitor-based exploration), and even some luck that necessary and useful system properties are observable. This is especially apparent when identifying the right system state and proxy value definitions. Accurately building the system trace so it best represents the ground truth of the system depends heavily on a complete understanding of the system itself.

6.4.0.3 Detection Accuracy

When building these higher-level monitoring specifications we tend to avoid many important false-negatives because the specification rules explicitly define the bad state we wish to identify. False-positives, on the other hand, are harder to deal with because the systems can violate these rules in small, interesting ways. Each identified false-positive violation has to be removed individually by adjusting the specification, which can be time consuming unless common specification misunderstandings are identified. This specification tuning is a difficult part of specification building which depends heavily on how exact the system requirements describe the actual system behaviors. The more thoroughly the requirements describe the actual system behavior, the easier it is to specifically include only true violations into the monitor specification. For testing purposes, false-positive detections may not be a major problem, but they may be costly on a deployed monitor which can trigger recovery actions. In these cases, avoiding these types of false positives is paramount.

In the embedded monitor example, we had requirements documentation and straightforward rules which allowed us to create a monitor specification. But even in this situation we were unsure whether an on time but zero-valued heartbeat status counted as a missing heartbeat. This is a situation where a full requirements specification (or more likely, access to knowledgeable designers) which explains what constitutes a valid and invalid system behavior would be necessary.

6.4.0.4 Initialization

An issue that arises out of our use of invariants as specification policies is that we must ensure that our specifications are designed to actually be satisfied by the system at all times, including initialization. This means that in practice, most policies must be guarded by some

state or action. Many system properties do not hold during system initialization, and even for properties that do hold the network-based view of the system state might not yet show a fully correct system state. This points towards a careful choosing of value initializations in the monitor to ensure that the monitor doesn't trigger specification violations before it sees real system state.

We can see this in action with our $HeartbeatOn$ guard from the embedded monitor example. Although we'd expect a component heartbeat to be the type of property that is actually invariant across system execution, it's possible that some system components can be started before the heartbeat component. This can cause a missing heartbeat failure to be detected even though the component which is supposed to send the heartbeat hasn't started executing yet. Depending on the situation, different guard approaches may be used, including waiting for the first target component message or not starting the monitor until a specific system state is reached.

For the embedded monitor example, we did not start the monitoring until the first CAN message was received. This still does not protect against the early missing heartbeat for some logs where the heartbeat component wasn't immediately started. We added the more explicit guard to protect against this. We also see somewhat hidden guards in the offline vehicle logs example. Both lane keeping rules (#1 and #2) are guarded by the $ALCEngaged$ proposition to ensure the feature is enabled when we check its behavior.

Chapter 7

Discussion

Real systems and operating environments can vary widely, making generalizations about embedded systems risky. The caveat *system-dependent* is common when discussing possible trade-offs and design decisions. Because of this, we generally attempt to make reasonable choices and provide information relating the possible tradeoffs inherent in any decision. This chapter presents more discussion on some of our design decisions and assumptions, explaining trade-offs and the justifications for our choices.

7.1 Design Issues

Our design and implementation choices were heavily motivated by the autonomous ground vehicle use case. This led to some design choices which may not be applicable to all other systems. These assumptions and design decisions are discussed here.

7.1.1 Monitor Correctness

We have made the helpful assumption in this thesis that the monitor implementation and input data is correct.

7.1.1.1 Monitor Correctness

We utilize a proven monitoring algorithm which ensures that we get prompt and correct answers, but this does not protect the implementation, specification, or system mapping from being incorrect. Instead, we assume that the monitor implementation is correct, or more importantly, could feasibly be made correctly if desired. Given this, the trick becomes creating the specification and mapping that correctly checks the desired properties.

As a research prototype, our monitor implementation inevitably has some bugs. But due to its relative simplicity, we can envision the monitor implementation being built correctly through formal techniques and strong software design process. Comparatively, the monitor implementation is much simpler than the systems it is designed to monitor. This means that regardless of the level of design rigor, we can expect that the monitor can be made near-perfect in a much less costly way than the target system (e.g., an automobile). At worst, it will be easier to make the monitor perfect than the system perfect for any nontrivial system.

The transfer of criticality possible by using a monitor as a fault detector which can trigger recoveries can lead to a reduction of the size of critical system components. In the end case, if system safety relies entirely on a working monitor (including a perfect recovery mechanism), then the target system itself is no longer safety-critical at all (except the recovery mechanism) and can be made to any desired level of quality. In this case, instead of safety, the correctness of the target system only affects system availability and the usefulness of the output. Reducing the amount of system that is critical means more effort can be

spent ensuring that the smaller critical components are actually correct, which should improve safety for the entire system. This does rely on a trusted recovery mechanism, which is discussed more in 7.3.1.

7.1.1.2 Input Data

The output of any analysis method, including a runtime monitor, can only be as correct as the input fed into it. This is commonly referenced as "garbage-in, garbage-out". In our case, this means that monitoring results are only as accurate, or real, as the network bus's system state model is correct. An inaccurate network model can make a monitor essentially worthless. We will call the monitor's network-based view of system the *network model*.

There are two main ways the network model can affect the correctness of our monitor output. The first is simply that the network model does not match the actual system, which can occur due to system faults or design errors. If the system state that we are monitoring (the network model) is not equivalent to the actual system state, then obviously our monitor results are also not equivalent. The other, slightly more insidious problem is that a misunderstanding of the network model affects the overall meaning of the monitoring specification. This is technically a fault of the specification, but we will discuss it here since it can cause a similar outcome.

The primary potential causes of incorrect network state are network faults, system faults, and design errors. Network faults such as dropped or mangled messages can cause system state to be lost or delayed (which can be just as bad as lost in real-time systems). Some of these faults may be detected directly by the monitor (e.g., dropped/lost heartbeat messages can be detected), but others can cause inconsistent monitor state which can make the monitor results innacurate. A faulty system or component on the network which does

not fail silently may also cause an incorrect value to be present on the bus. Lastly, design errors can also cause a bad network state. Problems including mismatched units or incorrectly specified/implemented messages can leave the network model in a inaccurate state. One solution to help monitoring in the face of potentially faulty network state is utilizing monitor-dedicated sensors. Extra sensors can be used to generate system state used to perform sanity checks. For example, rotary encoders can be added to a wheel to obtain low-resolution wheel speed which can be cross-checked with the network model state.

All of these issues are expected in distributed systems and there are plenty of known protection mechanisms. The level of assurance needed is system dependent, but it is clear that the level of fault-tolerance in the network is a major component to the trustworthiness of a network-based monitor. A useful notion about these systems is that they generally require some amount of fault-tolerance for functional reasons regardless of the monitor. So we know that the network is at least accurate enough for the system to work as designed. Obviously, this idea is of little help in the face of systems which aren't designed to the desired level of fault tolerance that we want to obtain from the monitor, but it is a start. For testing, assuming the network is reliable may be reasonable due to the small exposure time, but on a deployed system with the monitor actually affecting system safety it may not be. Whether the target network is correct enough for our needs or not, fault-tolerance for distributed systems is a well researched area and plenty of techniques exist for different fault models (although other design constraints may cause problems) [121, 122].

The mismatch between the actual meaning of the network model and the meaning assumed by the specification can be subtle yet can lead to monitoring problems. These are errors in translating the requirements into a specification. They arise when the true meaning of an observable network state is misunderstood. Examples of this mismatch include

treating control requests as actions or using the wrong version of a message value (e.g., if many different system nodes send messages denoting the same system-wide property). Our offline test vehicle logs include multiple ACC engaged message values being sent from different system components (human-machine interface controllers, ACC controllers, engine controllers, etc.). Choosing to monitor the wrong message value could lead to undetected violations if the monitored value does not represent the system state that the specification writer expected. These errors can be difficult to notice since the validation steps used to check them may include the same misunderstandings.

Avoiding these errors requires careful process with strong validation. This ultimately inherits from the difficulties of mapping the system to the monitor model. Care must be taken to ensure that this mapping and the specification agree.

7.1.2 Monitor Consolidation

Fully isolating a system monitor provides many benefits, including protection from common faults and allowing independent certification. True full isolation is rarely used, except in the most critical of systems. Often some resources, whether power or even the physical enclosure, is shared between system components to reduce costs.

A separate hardware monitor has a strong use case as a testing oracle, since it can be temporarily connected to an existing system without requiring much design effort for integration.

In some systems the costs associated with extra physical hardware can be debilitating. Commercial automobiles are moving towards more feature consolidation onto fewer control units to save costs [123]. Our monitor framework can easily be put into a network gateway or consolidated ECU as an extra task, as demonstrated by the PC-based monitor

version. Doing so can reduce the validity of independence arguments for correctness, but modern real-time operating system process isolation might work well enough in practice. Again, just as with the argument for network value correctness, if the system designers trust combining multiple system processes on a single chip, it is likely also reasonable to trust combining the monitor with similar tasks. Utilizing an extra core for the monitor could isolate an integrated monitor, mitigating the monitor's effect on system timing as well as the system's effects on the monitor. This type of integration does remove the ability to partition criticality onto just the monitor, but performing monitoring, even if not as independent as ideal, still can provide the benefits of fault detection.

7.1.3 Semi-Formal Interface

The semi-formal interface provides flexibility and power to the monitor, especially in the face of real systems which do not always map nicely onto rigid formal frameworks. The usefulness of being able to define essentially any desired property that can be derived from the system state is obvious, but we want the monitor specification to be as formal as possible. Since we don't know where to draw the line that separates what should be formally defined and what must be informal to incorporate the real system, we need the semi-formal interface to be powerful and flexible enough to allow us to move this line ourselves as needed. If a desired property cannot be fit into the formal specification language, it should be creatable in the semi-formal interface. We see a similar setup in other monitoring frameworks which utilize "filters" or similarly named components which generate atomic propositions to be monitored from the system state. Performing verification on a real system requires some informal to formal interface that bridges the real system with the verification trace.

The monitor checks the specification it is given, no more, no less. Creating the right specification is the real challenge. While the semi-formal interface provides a way to avoid the formal aspects of monitoring, even so far as allowing the creation of a completely informal monitor, it also provides a way to ensure that regardless of how difficult a desired specification is to write formally we can mold the available system state enough to gain some amount of formally guaranteed checking. This leaves the balance between informal and formal aspects of the specification in the hand of the specification writers, who should have the knowledge available to choose the right balance.

7.1.3.1 Choice of semi-formal semantics

We have provided a suggested semi-formal semantics which limits the semi-formal transformations to simple, common computational elements which provide most of the necessary power to extract useful properties from the system state while promoting the use of the formal specification logic rather than the semi-formal interface.

Our restrictions were chosen based on needs that have arisen in our experience implementing this type of monitoring. We can easily imagine wanting or needing more flexibility or even more restrictions (e.g., no memory storage, no functions, etc.) which could ease verification of the interface. A monitor for any system needs the ability to map system state of different types to monitor propositions, such as copying booleans to a proposition or comparing an integer to a threshold. There are also more complex transformations such as comparing a running average of a value to a threshold or comparing system state to a simple dynamics model that may be necessary in other situations.

Ultimately, it is clear that at some level the real system needs to be mapped to a formal model if we want to perform formal checks of the system. Exactly what should occur in

this mapping is still an open question, but we have shown that a semi-formal monitoring framework which combines an informal mapping with formal checking is viable and useful.

7.2 Time Model

The monitor presented in this thesis is a time-triggered monitor, yet the monitoring algorithm **agmon** is actually a pointwise asynchronous monitoring algorithm. This is somewhat a quirk of development, but it ends up providing a stronger overall use case for our monitoring framework.

The time-triggered monitor checks a trace which is a list of periodic snapshots. We treat the traces as snapshots of events that represent proposition updates. In this way we can take a pointwise event semantics (i.e., trace of instantaneous update events) and view the system trace as a series of state intervals between these events. This lets us treat the trace as a time-triggered interval state trace (i.e., a list of intervals for each state), which is more intuitive for discussing system state rather than a series of events. This interval model fits well with time-triggered, state broadcast systems where the current system state values are periodically broadcast to the network. These value update messages can be used as the events which update their respective propositions. For example, if we receive a vehicle velocity message stating the velocity is now 10m/s, the constant state model treats the velocity as 10m/s until the next velocity message is seen. This model is useful for physical system state since the real system state must always have some value and it handles actual instantaneous events as well. To use instantaneous events and keep the interval semantics, they can be treated as short duration constant states.

When used completely synchronously as shown in this thesis the monitoring speci-

fications are checked in the intuitive way. The monitoring can also be done completely asynchronously, where incoming instantaneous events are checked against each other but the trace doesn't carry state. If we want to check instantaneous events and constant state together at the same time, the safest way is to use the synchronous system but give asynchronous events negligible durations. Another more direct way is to build a new trace model and use a dual algorithm based on **agmon** where asynchronous messages trigger an asynchronous check. The semi-formal interface can be used to make asynchronous propositions not carry state while carrying synchronous values. The heartbeat proposition in the ARM example worked in a similar way, where a received heartbeat message set the heartbeat proposition for just the current state (it was not carried over to the next sample).

7.3 Future Work

7.3.1 System Recovery

Once you have a monitor that is capable of correctly identifying when a system is violating its safety specification, the next obvious step is to enable it to perform recoveries by *steering* the system into a safe state.

A simple and straightforward steering strategy for violating systems is to engage an emergency stop/shutdown (which many systems already have as part of their safety design). For some violation scenarios and systems this is a perfectly fine response, but many systems either cannot just be shut down (e.g., aerial vehicles) or prefer a more controlled shutdown to avoid system damage (e.g., trains, chemical/industrial plants, etc.).

There are many possibilities for doing more elaborate shutdowns such as graduated warnings, graduated shutdowns (e.g., turning off individual features/subcomponents that

are faulty) or multiple step controlled shutdowns. Some systems may even want to attempt to ride-through [124] safety violations if it may be safely possible to improve system reliability. Because what constitutes a desired shutdown is very system dependent in both how and when the shutdown is performed, steering is an interesting and hard problem.

One of the hardest aspects of performing complex recoveries is that they are initiated only when the system is faulty, and so the recovery controller needs to be robust to different fault scenarios. It's possible that the fault which necessitates recovery also blocks the preferred recovery tactic. A fault diagnosis and system partitioning which guaranteed that certain recovery actions are still available given the known set of specification violations would allow choosing the right recovery actions.

7.3.2 Semi-Formal DSL

The semi-formal interface presented in this thesis is relatively ad-hoc. We recommomend a set of restrictions but allow any desired transformations. Defining the specification language of the semi-formal interface in a more formal and standard way would both encourage correct usage and could lead to useful verification techniques on the mapping itself.

An internal domain specific language in the monitor's implementation language (in our case C) or one which could be compiled to the target implementation language would be useful. The features necessary in the interface specification language are common across systems. We would need to define the set of basic operators, a set of simple variable types, a simple syntax for performing calculations over sets of variables, and the semantics for simple state machines.

Before a "final" DSL can be defined, many design decisions would need to be better understood. Two important decisions are the basic language types and allowed complex-

ity. A common set of basic data types which is adequate for specifying the interface for any target system and also has a clean mapping into the monitor implementation language would need to be identified. It is important that the values from the system can be cleanly represented in the interface specification, and perhaps more importantly, that the interface implementation be easily generated from the DSL. If the DSL contains types that are hard to implement in the target monitor, bugs and misunderstandings are more likely.

Understanding the correct computational constraints would also be important. What operators are allowed? How does the language allow iteration, branching (e.g., if `mode1` then set `val2`, otherwise set `val3`), and other dynamic computation constructs while still restricting the interfaces complexity? If the DSL is not simplified in a way that improves verification of the interface, then there is no reason to use the DSL instead of manually designing the interface. The tradeoff between necessary complexity and verifiability is fundamental, and finding the right combination that fits a general set of target systems is nontrivial.

Chapter 8

Conclusion

As safety-critical systems become more and more complex, including the use of software, the importance of strong system verification has increased dramatically. These systems have unique constraints which affect the use of existing runtime verification techniques, especially the use of more COTS, black-box components.

8.1 Thesis Contributions

To address the difficulties in monitoring safety-critical embedded systems, this thesis makes the following contributions:

8.1.1 Identifying suitable runtime verification architecture

I have identified a suitable runtime verification architecture for monitoring safety-critical embedded systems.

Based on the motivation to create a bolt-on monitor for ground vehicle architectures,

I have presented a monitoring architecture which uses an external bus monitor connected to the target system using a semi-formal interface. The semi-formal interface provides flexibility to handle unique system-dependent properties.

The primary motivations for my monitor architecture are presented in Section 3.1. The monitor architecture was chosen specifically for distributed systems with broadcast buses and black-box components which is a common architecture in modern ground vehicles. Although the architecture was chosen to directly fit these systems, our system model is generic enough that different target system architectures can be accommodated through instrumentation. Chapter 3 details the identified monitoring architecture, discussing design decisions and how the constraints of safety-critical systems can be appropriately handled.

Passive external monitors have many benefits that line up well against the constraints imposed by safety-critical embedded systems. Isolating the monitor from the target system helps ensure that system functionality is not compromised by the inclusion of the monitor.

We have shown that the presented architecture is suitable for performing runtime monitoring of safety-critical systems by applying our monitor to example systems discussed in Chapter 6.

8.1.2 Monitoring Framework

I provide a monitoring framework based on a formally proven monitoring algorithm and an informal system interface

I have presented an end-to-end framework which fits the identified runtime verification architecture into the more broad needs of actual monitoring integration. This includes not only the monitoring algorithms and specification logic but guidelines for the semi-formal interface, and design patterns for specifications and a monitor safety case.

Chapter 4 presents the formal definitions that make up our monitoring framework. A formal, bounded metric temporal specification logic is presented in Section 4.1.2. Our aggressive runtime monitoring algorithm is presented in Section 4.3. It performs an iterative check of a given specification against a trace step by step, so it can be used at runtime to check a trace of sampled system state. Section 4.4 contains correctness proofs of this algorithm, showing that the algorithm gives both correct and prompt answers.

To improve usability two different types of design patterns were presented. I have created a set of specification patterns which can be used to translate informal system requirements into our bounded temporal logic. I also provide a safety case pattern and example instantiation of the pattern as a starting point to creating a safety case which argues that incorporating a monitor based on this framework into an existing system is safe. These patterns are discussed in Section 3.6 and Appendix B.

An implementation of our monitor framework is detailed in Chapter 5. The implementation contains two different monitors (a PC-based log monitor and an embedded CAN monitor) based on a portable monitor code library. The implementation fulfills some standard safety-critical embedded constraints including avoiding recursion and no dynamic memory.

8.1.3 Feasibility of real-time monitoring

I demonstrate the feasibility and show performance characteristics of the monitoring framework on multiple diverse systems

Runtime monitoring is a well researched field, but few monitors for safety-critical embedded systems that can be used in real-time exist. Chapter 6 contains an evaluation of the monitor implementation, detailing performance characteristics of the monitor framework

and showing that real-time monitoring of a CAN bus is feasible.

An offline log monitor was used to show performance characteristics of our monitoring framework with artificial traces in Section 6.2. The offline monitor was also used to check test result logs from a commercial research lab's autonomous research vehicle. We also evaluated an ARM based network monitor with CAN logs replayed onto a live CAN network, showing that real-time runtime monitoring of a CAN network with realistic specifications is feasible using commonly available embedded microcontrollers.

Appendix A

Acronyms

ACC Adaptive Cruise Control
ALC Autonomous Lane Centering
AST Abstract Syntax Tree
BMTL Bounded Metric Temporal Logic
CAN Controller Area Network
COTS Commercial off the Shelf
CSV Comma Separated Values
DSL Domain Specific Language
ECU Electronic Control Unit
FMEA Failures Modes and Effects Analysis
FMECA Failures Modes, Effects, and Criticality Analysis
FaCTS Functionality, Certifiability, Timing, Size, Weight, and Power
FSRACC Full Speed Range Adaptive Cruise Control
FTA Fault Tree Analysis
GSN Goal Structuring Notation
HAZOP Hazards and Operability Analysis
HIL Hardware-in-the-Loop
ISR Interrupt Service Routine
LTL Linear Temporal Logic
MaC Monitoring and Checking
MTL Metric Temporal Logic

MEDL Meta Event Definition Language

MOP Monitor Oriented Programming

PEDL Primitive Event Definition Language

RV Runtime Verification

SFTA Software Fault Tree Analysis]

SIL Safety Integrity Level

SFMEA Software Failure Modes and Effects Analysis

SUO System Under Observation

SWaP Size, Weight, and Power

TTP Time-Triggered Protocol

PHA Preliminary Hazard Analysis

UML Unified Markup Language

Appendix B

Specification Patterns

Table B.1: Pattern 1.a Bounded Response

Name	1.a Bounded Response	
Intent	To describe a relationship between two states where there must be an occurance of the second within a bounded amount of time of an occurance of the first	
Example	If the unlock doors button is depressed then the driver side door must be unlocked within 500ms	
Ex Formula	`UnlockDoorsPressed -> <0,500> DriverDoorUnlocked`	
Formula	BMTL ASCII	$T \rightarrow \Diamond_{[l,h]} E$ `T -> <l,h> (E)`
Variables	T E l h	Triggering event/state Triggered event/state Minimum time between occurance of T and occurance of E Maximum time between occurance of T and occurance of E
Description	This template is used for the common basic pattern where some state requires that another state be occurring in some bounded amount of time. As an invariant, note that any time t the guard condition T is true, then E must be true at some point in the future interval $[t + l, t + h]$	
Known Uses	This pattern can be used any time an event requires a change in state, such as user input (button/pedal presses, etc.) which cause a system mode change (turning off a feature, beginning some transition) or requires a bounded response. Care should be taken that one of the more specific rules is not actually desired	
Limitations	Temporal	Note that T and E only need to be true for a single time step. If a specific duration is required then a more specific rule should be used
See Also	1.b Bounded Response with Duration	

Table B.2: Pattern 1.b Bounded Response with Duration

Name	1.b Bounded Response with Duration	
Intent	To describe a relationship between two states where there must be an occurance of the second state for a specified duration within a bounded amount of time of an occurance of the first state	
Example	If the brake pedal is depressed then within 500ms cruise control should be disabled and stay disabled for at least 200ms	
Ex Formula	`BrakePressed -> <0,500> [0,200] ACCDisabled`	
Formula	BMTL	$T \rightarrow \Diamond_{[l_1,h_1]}\Box_{[l_2,h_2]}E$
	ASCII	`T -> <l1,h1> [l2,h2] (E)`
Variables	T	Triggering event/state
	E	Triggered event/state
	l_1	Minimum time between occurance of T and occurance of E
	h_1	Maximum time between occurance of T and occurance of E
	l_2	Time between start of E and start of required E duration
	h_2	Time between start of E and end of required E duration
Description	This template is used for the common basic pattern where some state requires that another state occur for a duration in some bounded amount of time.	
Known Uses	This pattern can be used any time an event requires a response state which must occur for a specified duration. This rule is often the correct choice instead of the bounded response, because a little duration ensures that the response or state change actually holds (i.e., it isn't a short, transient state).	
Limitations	Temporal	Note that with this rule, the duration of E only must start within the first eventually bounds. It does not need to be finished within those bounds.
See Also	1.a Bounded Response	

Table B.3: Pattern 1.c Bounded Response with Cancel

Name	1.c Bounded Response with Cancel	
Intent	To describe a relationship between two states where the occurance of a triggering state requires the occurrence of a second state within a bounded time unless a cancel event occurs.	
Example	If automated lane centering is enabled the vehicle should return to the lane center within 5s unless ALC is canceled.	
Ex Formula	`ALC -> <0,5s> (VehicleCentered || ALCCancel)`	
Formula	BMTL	$T \rightarrow \Diamond_{[l,h]}(E \vee C)$
	ASCII	`T -> <l,h> (E || C)`
Variables	T	Triggering Event/State
	E	Triggered Event/State
	C	Cancel Event/State
	l	Minimum time between occurance of T and occurance of E
	h	Maximum time between occurance of T and occurance of E
Description	This template is used for the pattern where some state requires that either another state occur in some bounded amount of time or an event cancel occur.	
Known Uses	This pattern can be used any time an event requires another event or state in a bounded amount of time but some other state change can cancel the necessary response. This pattern adds a cancel to the bounded response pattern, which is important for policies which are generally required but can be canceled by another event. Canceling events include explicit cancels (e.g., turning off the feature that requires a request) as well as events or state changes which make a rule no long apply. For example, a rule that requires a specific headway be regained within a certain amount of time may be canceled by a new vehicle cutting infront of it.	
Limitations	Temporal	This rule requires the cancel occur within the same bounds as the target. Many cancel events should use a ¡0,h¿ bound instead to ensure that any cancel that occurs before the time limit is caught. This rule would be $T \rightarrow ((\Diamond_{[l,h]}E) \vee \Diamond_{[0,h]}C)$
See Also	1.a Bounded Response	
	1.d Bounded Response with Duration and Cancel	

Table B.4: Pattern 1.d Bounded Response with Duration and Cancel

Name	1.c Bounded Response with Duration and Cancel	
Intent	To describe a relationship between two states where there must be an occurance of the second state for a specified duration or a response cancel within a bounded amount of time of an occurance of the first state	
Example	If ACC is enabled and the target headway is less than 2s, then a 2s headway should be regained for a duration of 1s within 5s	
Ex Formula	`(ACCEnabled && HeadwayLT2s) -> <0,5s> ([0,1s] ~HeadwayLT2s) || CutIn`	
Formula	BMTL	$T \rightarrow \Diamond_{[l_1,h_1]}(\Box_{[l_2,h_2]}E) \vee C)$
	ASCII	`T -> <l1,h1> ([l2,h2] (E) || C)`
Variables	T	Triggering Event/State
	E	Triggered Event/State
	C	Cancel Event/State
	l_1	Minimum time between occurance of T and start of duration of E
	h_1	Maximum time between occurance of T and start of duration of E
	l_2	Minimum delay between end of the outer bound and start of duration of E
	h_2	Maximum delay between end of the outer bound and end of duration of E
Description	This template is used for the pattern where some state requires that either another state occur for a duration in some bounded amount of time or an event cancel occur.	
Known Uses	This pattern should be used if an event requires a response state with a specified duration but the response can be canceled by another event. This pattern adds a cancel to the bounded response with duration, which is important for policies which usually occur but can be canceled by another event. Canceling events include explicit cancels (e.g., turning off the feature that requires a request) as well as events or state changes which make a rule no longer apply. For example, a rule that requires a specific headway be regained within a certain amount of time may be canceled by a new vehicle cutting infront of it.	
Limitations	Temporal	This rule requires the cancel occur within the same bounds as the target. Many cancel events should use a ¡0,h¿ bound instead, as in pattern 1e
See Also	1.b Bounded Response with Duration 1.c Bounded Response with Cancel	

Table B.5: Pattern 2.a Conflicting State

Name	2.a Conflicting State	
Intent	To describe a property where two states are conflicting and thus cannot both be true at the same time.	
Example	The ACC feature should not request both braking and acceleration at the same time	
Ex Formula	$\sim$(`ACCBrakeRequest && ACCAccelRequest`)	
Formula	BMTL ASCII	$\neg(A \wedge B)$ $\sim$(`A && B`)
Variables	A B	First Event/State Second Event/State
Description	This pattern is used for situations where two states or events are conflicting and cannot both be true at the same time.	
Known Uses	This pattern is used when there are conflicting system state properties or events which should not occur together. This includes directly conflicting requests (e.g., brake and accelerate) as well as illegal actions due to modes (e.g., cruise control enabled while in park).	
Limitations	Temporal	Note that this rule only covers instantaneous state. Many seemingly conflicting states are actually temporal responses, e.g., brake and cruise control can happen together when the brake is first pressed since the cruise control can't respond before it's seen the incoming state.
See Also	2.b Conflicting State Duration	

Table B.6: Pattern 2.b Conflicting State with Duration

Name	2.b Conflicting State with Duration	
Intent	To describe a property where two states are conflicting and cannot both be true for some specified duration.	
Example	Cruise control cannot be enabled for 300ms while the brake is also being depressed	
Ex Formula	`~ [0,300] (ACCBrakeRequest && ACCAccelRequest)`	
Formula	BMTL ASCII	$\neg\Box_{[l,h]}(A \wedge B)$ `~[l,h](A && B)`
Variables	A B l h	First Event/State Second Event/State Start of Time Bound for State Duration End of Time Bound for State Duration
Description	This template is used for situations where two states or events are conflicting and cannot both be true at the same time for a specified duration.	
Known Uses	This pattern is used any time there is state cannot be in conflict for some specified duration. This includes directly conflicting requests (e.g., brake and accelerate) as well as illegal actions due to modes (e.g., cruise control enabled while in park). Small durations may be used to create functionally instantaneous conflicting state rules which are more robust to small amounts of jitter.	
Limitations	Temporal	This pattern uses future time, which makes it useful for composing with other guards or triggering events, but slightly less useful for direct invariant rules. The past-time version often makes more sense for pure invariant rules.
See Also	2.a Conflicting State Duration 2.c Past-Time Conlficting State with Duration	

Table B.7: Pattern 2.c Past-Time Conflicting State with Duration

Name	2.c Past-Time Conflicting State with Duration	
Intent	To describe a property where two states are conflicting and cannot both be true for some specified duration.	
Example	Cruise control cannot have been enabled for 300ms while the brake is also being depressed	
Ex Formula	`~ [[0,300]] (ACCBrakeRequest && ACCAccelRequest)`	
Formula	BMTL ASCII	$\neg\blacksquare_{[l,h]}(A \wedge B)$ `~[[l,h]](A && B)`
Variables	A B l h	First Event/State Second Event/State Start of Time Bound for State Duration End of Time Bound for State Duration
Description	This template is used for situations where two states or events are conflicting and cannot both be true at the same time for a specified duration in the past.	
Known Uses	This pattern is used any time there is state that cannot be in conflict for some specified duration. This includes directly conflicting requests (e.g., brake and accelerate) as well as illegal actions due to modes (e.g., cruise control enabled while in park). Small durations may be used to create near-instantaneous conflicting rules which can handle small amounts of jitter.	
Limitations	Temporal	This pattern uses past-time, which makes it useful direct invariants because there is no waiting for the event to occur, which reduces the amount of residue storage slightly. This pattern will check past state when composed with other patterns, which is useful if we want to check for conflicting state before a given trigger
See Also	2.a Conflicting State 2.b Conlficting State Duration	

Table B.8: Pattern 3.a No Instantaneous Transition

Name	3.a No Instantaneous Transition	
Intent	To describe an illegal instantaneous transition.	
Example	The cruise control mode cannot immediately transition from Off to Engaged.	
Ex Formula	$\sim$ `(CruiseEngaged && [[p,p]] CruiseOff)`	
Formula	BMTL ASCII	$\neg(A \wedge \blacksquare_{[p,p]} B)$ $\sim$ `( A && [[p,p]] B)`
Variables	A B p	First Event/State Second Event/State Monitor Period
Description	This template is used for situations where a transition between states is illegal.	
Known Uses	This pattern is used any time there are two states that the system cannot immediately transition between. The pattern must use the monitor period p as the time bound for instantaneous transitions to be disallowed. Other patterns can be used to limit transitions within a duration.	
Limitations	Temporal	This pattern only checks for instantaneous transitions. If transitions are disallowed for a certain duration a different pattern is necessary.
See Also	3.b No Transition within Duration	

Table B.9: Pattern 3.b No Transition within Duration

Name	3.a No Transition within Duration	
Intent	To describe a transition which is illegal for a set duration.	
Example	The cruise control cannot be engaged for 250ms after the brake pedal has been pressed.	
Ex Formula	`~ (CruiseEngaged && <<0,250>> BrakePressed)`	
Formula	BMTL	$\neg(A \wedge \blacklozenge_{[l,h]} B)$
	ASCII	`~ ( A && <<l,h>> B)`
Variables	A	First Event/State
	B	Second Event/State
	l	Lower Bound of Limited Duration
	h	Upper Bound of Limited Duration
Description	This template is used for situations where a transition between states is illegal for some specified duration.	
Known Uses	This pattern is used any time there are two states that the system cannot transition between for some specified duration. If there is an event or state which precludes some other event or state for a known amount of time, this pattern should be used.	
Limitations		
See Also	3.a No Instantaneous Transitions	

Table B.10: Pattern 4.a Always

Name	4.a Always	
Intent	To describe a proposition which should always be true.	
Example	Vehicle speed should always be below 100mph	
Ex Formula	`SpeedLT100`	
Formula	BMTL	A
	ASCII	`A`
Variables	A	System State/Proposition
Description	This template is used for basic invariant properties.	
Known Uses	This is the most basic pattern for describing simple invariant propositions	
Limitations		
See Also	4.b Guarded Always (Implies)	

Table B.11: Pattern 4.b Guarded Always (Implies)

Name	4.b Guarded Always (Implies)	
Intent	To describe a proposition which should always be true if another specific proposition is true.	
Example	If cruise control is active, the vehicle speed should always be above 35mph	
Ex Formula	`CruiseActive -> SpeedGT35`	
Formula	BMTL	$A \rightarrow B$
	ASCII	`A -> B`
Variables	A	Guard State/Proposition
	B	System State/Proposition
Description	This template is used for basic guarded invariant properties.	
Known Uses	This is a basic pattern for describing simple properties that are guarded or activated by another simple property. The second property B will not be checked unless A is already true. This pattern is used often to check for initialization or system mode before checking a property.	
Limitations		
See Also	4.b Guarded Always (Implies)	

Table B.12: Pattern 5.a Periodic State

Name	5.a Periodic State	
Intent	To describe a proposition which should occur periodically.	
Example	A component alive message should be received every 400ms	
Ex Formula	`<0,400> Alive`	
Formula	BMTL ASCII	$\Diamond_{[l,h]} E$ `<l,h> E`
Variables	E l h	Periodic Event/State Delay before start of periodic occurence Maximum delay before occurance of event
Description	This template is used for periodic events such as heartbearts or required commands	
Known Uses	This pattern is used to specify properties that are periodically required.	
Limitations	Temporal	This pattern uses just an eventually time bound, so it only requires a single occurance within each period, not a specific frequency of events. More specific time bounds can be used to specify event frequency.
See Also	5.b Periodic State with Duration	

Table B.13: Pattern 5.b Periodic State with Duration

Name	5.b Periodic State with Duration	
Intent	To describe a proposition which should occur periodically with a specified duration.	
Example	The flow valve should be closed for at least 1s every 10s	
Ex Formula	`<0,10s> [0,1s] ValveClosed`	
Formula	BMTL	$\Diamond_{[l_1,h_1]}\Box_{[l_2,h_2]}E$
	ASCII	`<l1,h1> [l2,h2] E`
Variables	E	Periodic Event/State
	l_1	Delay before start of periodic occurence
	h_1	Maximum delay before occurance of event
	l_2	Minimum delay before start of event duration
	h_2	Maximum delay before start of event duration
Description	This template is used for periodic events such as heartbearts or required commands that must be active for a specific duration	
Known Uses	This pattern is used to specify periodic properties that much hold for some specified duration.	
Limitations	Temporal	This pattern only requires the event duration occur within the eventually bounds, it does not create a tight event frequency which can be specified by using stricter eventually bounds.
See Also	5.a Periodic State	

Bibliography

[1] T. Kelly, "A systematic approach to safety case management," *SAE 04AE-19,*, 2003. (document), 2.3.1, 2.1, 2.3.1

[2] G. Leen and D. Heffernan, "Expanding automotive electronic systems," *Computer*, vol. 35, no. 1, pp. 88–93, 2002. 1

[3] P. Koopman, "Embedded system security," *Computer*, vol. 37, no. 7, pp. 95–97, July 2004. 1

[4] D. Jackson and M. Rinard, "Software analysis: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 133–145. [Online]. Available: http://doi.acm.org/10.1145/336512.336545 1

[5] M. Leucker and C. Schallhart, "A brief account of runtime verification," *Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293 – 303, 2009, the 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07). [Online]. Available: http://www.sciencedirect.com/science/article/B6W8D-4TK7X4V-2/2/0442d2c20315ab050bb913af605cc126 1

[6] R. Butler and G. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *Software Engineering, IEEE Transactions on*, vol. 19, no. 1, pp. 3 –12, jan 1993. 1, 2.4.3

[7] A. Bayazit and S. Malik, "Complementary use of runtime validation and model checking," in *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*, nov. 2005, pp. 1052 – 1059. 1, 2.4.3.1

[8] S. Mitsch and A. Platzer, "Modelplex: Verified runtime validation of verified cyber-physical system models," in *Runtime Verification*, ser. Lecture Notes in Computer Science, B. Bonakdarpour and S. Smolka, Eds. Springer International Publishing, 2014, vol. 8734, pp. 199–214. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11164-3_17 1, 2.4.3.1

[9] A. Kane, T. Fuhrman, and P. Koopman, "Monitor based oracles for cyber-physical system testing: Practical experience report," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, June 2014, pp. 148–155. 1, 2.4.4, 3.2.1.1, 3.3

[10] P. Koopman, "Challenges in representing cps safety," in *Workshop on Developing*

Dependable and Secure Automotive Cyber-Physical Systems from Components, March 2011. [Online]. Available: http://users.ece.cmu.edu/~koopman/pubs/koopman11_cps_safety.pdf 1

[11] M. Perhinschi, M. Napolitano, G. Campa, B. Seanor, J. Burken, and R. Larson, "Design of safety monitor schemes for a fault tolerant flight control system," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 42, no. 2, pp. 562 – 571, 2006. 1

[12] F. Bitsch, "Safety patterns - the key to formal specification of safety requirements," in *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security*, ser. SAFECOMP '01. London, UK, UK: Springer-Verlag, 2001, pp. 176–189. [Online]. Available: http://dl.acm.org/citation.cfm?id=647399.724860 1.1, 2.3, 3.6.1

[13] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, pp. 9:1–9:76, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1459352.1459354 1.1, 2.3

[14] J. Knight, "Safety critical systems: challenges and directions," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, 2002, pp. 547 – 550. 2.2

[15] N. G. Leveson, "Software safety: why, what, and how," *ACM Comput. Surv.*, vol. 18, pp. 125–163, June 1986. [Online]. Available: http://doi.acm.org/10.1145/7474.7528 2.2

[16] I. N. S. A. Group, "Safety culture," International Atomic Energy Agency, Safety Report SAFETY SERIES No. 75-INSAG-4, 1991. [Online]. Available: http://www-pub.iaea.org/MTCD/publications/PDF/Pub882_web.pdf 2.2

[17] P. Baufreton, J. Blanquart, J. Boulanger, H. Delseny, J. Derrien, J. Gassino, G. Ladier, E. Ledinot, M. Leeman, P. Quéré *et al.*, "Multi-domain comparison of safety standards," in *Proceedings of the 5th International Conference on Embedded Real Time Software and Systems (ERTS2), Toulouse, France*, 2010. 2.2.1

[18] I. S. 65A, "Functional safety of electrical/electronic/programmable electronic safety-related systems," The International Electrotechnical Commission, 3, rue de Varembé, Case postale 131, CH-1211 Genève 20, Switzerland, Tech. Rep. IEC 61508, 1998. 2.2.1

[19] R. Panesar-Walawege, M. Sabetzadeh, L. Briand, and T. Coq, "Characterizing the chain of evidence for software safety cases: A conceptual model based on the iec 61508 standard," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, april 2010, pp. 335 –344. 2.2.1

[20] RTCA, *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Commission for Aeronautics (RTCA) Std., 1992. 2.2.1

[21] ISO, "ISO/DIS 26262 - Road vehicles – Functional safety," Geneva, Switzerland, Tech. Rep., November 2011. 2.2.1

[22] J. Bowen, "Formal methods in safety-critical standards," in *Software Engineering Standards Symposium, 1993. Proceedings., 1993*, aug-3 # sep 1993, pp. 168 –177. 2.2.1, 2.4.3

[23] Y. Papadopoulos, M. Walker, M.-O. Reiser, M. Weber, D. Chen, M. Törngren, D. Servat, A. Abele, F. Stappert, H. Lonn, L. Berntsson, R. Johansson, F. Tagliabo, S. Torchiaro, and A. Sandberg, "Automatic allocation of safety integrity levels," in *Proceedings of the 1st Workshop on Critical Automotive Applications: Robustness & Safety*, ser. CARS '10. New York, NY, USA: ACM, 2010, pp. 7–10. [Online]. Available: http://doi.acm.org/10.1145/1772643.1772646 2.2.1

[24] N. G. Leveson, *Safeware - system safety and computers: a guide to preventing accidents and losses caused by technology*. Addison-Wesley, 1995. 2.2.2, 2.3

[25] H. E. Roland and B. Moriarty, *Preliminary Hazard Analysis*. John Wiley & Sons, Inc., 2009, pp. 206–212. [Online]. Available: http://dx.doi.org/10.1002/9780470172438.ch23 2.2.2

[26] J. McDermid, M. Nicholson, D. Pumfrey, and P. Fenelon, "Experience with the application of hazop to computer-based systems," in *Computer Assurance, 1995. COMPASS '95. Systems Integrity, Software Safety and Process Security. Proceedings of the Tenth Annual Conference on*, Jun 1995, pp. 37–48. 2.2.2

[27] R. Lutz and R. Woodhouse, "Contributions of sfmea to requirements analysis," in *Requirements Engineering, 1996., Proceedings of the Second International Conference on*, Apr 1996, pp. 44–51. 2.2.2

[28] J.-M. Flaus and J.-M. Flaus, *Fault Tree Analysis*. John Wiley & Sons, Inc., 2013, pp. 229–251. [Online]. Available: http://dx.doi.org/10.1002/9781118790021.ch12 2.2.2

[29] N. Leveson and P. Harvey, "Analyzing software safety," *Software Engineering, IEEE Transactions on*, vol. SE-9, no. 5, pp. 569–579, Sept 1983. 2.2.2

[30] J. Rushby, "A comparison of bus architectures for safety-critical embedded systems." Springer-Verlag, 2001, pp. 306–323. 2.2.3

[31] D. A. G. Bmw Ag, *FlexRay Communications System Protocol Specification Version 2.1 Revision A*, F. Consortium, Ed., Dec. 2005. 2.2.3

[32] K. Forsberg and H. Mooz, "The relationship of system engineering to the project cycle," *INCOSE International Symposium*, vol. 1, no. 1, pp. 57–65, 1991. [Online]. Available: http://dx.doi.org/10.1002/j.2334-5837.1991.tb01484.x 2.3

[33] N. G. Leveson, "A systems-theoretic approach to safety in software-intensive systems," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 66–86, Jan. 2004. [Online]. Available: http://dx.doi.org/10.1109/TDSC.2004.1 2.3

[34] B. Nuseibeh and S. Easterbrook, "Requirements engineering: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 35–46. [Online]. Available: http://doi.acm.org/10.1145/336512.336523 2.3

[35] A. v. Lamsweerde, "Formal specification: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 147–159. [Online]. Available: http://doi.acm.org/10.1145/336512.336546 2.3

[36] OMG, *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*, Object Management Group Std., Rev. 2.4.1, August 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1 2.3

[37] D. Berry and E. Kamsties, "Ambiguity in requirements specification," in *Perspectives on Software Requirements*, ser. The Springer International Series in Engineering and Computer Science, J. do Prado Leite and J. Doorn, Eds. Springer US, 2004, vol. 753, pp. 7–44. [Online]. Available: http://dx.doi.org/10.1007/978-1-4615-0465-8_2 2.3

[38] M. Heimdahl and C. Heitmeyer, "Formal methods for developing high assurance computer systems: working group report," in *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on*, 1998, pp. 60–64. 2.3

[39] 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*, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 411–420. [Online]. Available: http://doi.acm.org/10.1145/302405.302672 2.3, 3.6.1

[40] S. Konrad and B. H. C. Cheng, "Real-time specification patterns," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, May 2005, pp. 372–381. 2.3

[41] P. Bishop and R. Bloomfield, "A methodology for safety case development," in *SAFETY-CRITICAL SYSTEMS SYMPOSIUM, BIRMINGHAM, UK, FEB 1998*. Springer-Verlag, ISBN 3-540-76189-6, 1998. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.9838 2.3.1

[42] R. Bloomfield, B. Littlewood, and D. Wright, "Confidence: Its role in dependability cases for risk assessment," in *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, June 2007, pp. 338–346. 2.3.1

[43] J. Goodenough, C. Weinstock, and A. Klein, "Toward a theory of assurance case confidence," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. (CMU/SEI-2012-TR-002), 2012. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=28067 2.3.1

[44] C. B. Weinstock, J. B. Goodenough, and A. Z. Klein, "Measuring assurance case confidence using baconian probabilities," in *Proceedings of the 1st International Workshop on Assurance Cases for Software-Intensive Systems*, ser. ASSURE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 7–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2662398.2662401 2.3.1

[45] T. P. Kelly, "Arguing safety – a systematic approach to managing safety cases," Ph.D. dissertation, University of York, 1998. [Online]. Available: http://www-users.cs.york.ac.uk/tpk/tpkthesis.pdf 2.3.1

[46] T. Kelly and J. McDermid, "Safety case construction and reuse using patterns," in *Safe Comp 97*, P. Daniel, Ed. Springer London, 1997, pp. 55–69. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-0997-6_5 2.3.1, 3.6.2, 3.3

[47] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annual Symposium on*, 311977-nov.2 1977, pp. 46 –57. 2.4.1

[48] A. Bauer, M. Leucker, and C. Schallhart, "Monitoring of real-time properties," in *In Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), volume 4337 of LNCS*. Springer, 2006, pp. 260–272. 2.4.1

[49] ——, "Comparing ltl semantics for runtime verification," *J. Log. and Comput.*, vol. 20, no. 3, pp. 651–674, Jun. 2010. [Online]. Available: http://dx.doi.org/10.1093/logcom/exn075 2.4.1

[50] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Syst.*, vol. 2, pp. 255–299, October 1990. [Online]. Available: http://dx.doi.org/10.1007/BF01995674 2.4.1

[51] A. Bauer, J.-C. Kster, and G. Vegliach, "From propositional to first-order monitoring," in *Runtime Verification*, ser. Lecture Notes in Computer Science, A. Legay and S. Bensalem, Eds. Springer Berlin Heidelberg, 2013, vol. 8174, pp. 59–75. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40787-1_4 2.4.1

[52] D. Garg, L. Jia, and A. Datta, "Policy auditing over incomplete logs: Theory, implementation and applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 151–162. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046726 2.4.1, 2.4.2

[53] O. Chowdhury, L. Jia, D. Garg, and A. Datta, "Temporal mode-checking for runtime monitoring of privacy policies," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds. Springer International Publishing, 2014, vol. 8559, pp. 131–149. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-08867-9_9 2.4.1, 2.4.2, 4.3, 5.2.2.3

[54] D. Basin, M. Harvan, F. Klaedtke, and E. Zalinescu, "Monitoring usage-control policies in distributed systems," in *Temporal Representation and Reasoning (TIME), 2011 Eighteenth International Symposium on*, Sept 2011, pp. 88–95. 2.4.1

[55] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Softw. Eng.*, vol. 3, pp. 125–143, March 1977. [Online]. Available: http://portal.acm.org/citation.cfm?id=1313313.1313439 2.4.1

[56] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed Computing*, vol. 2, pp. 117–126, 1987, 10.1007/BF01782772. [Online]. Available: http://dx.doi.org/10.1007/BF01782772 2.4.1

[57] A. Goodloe and L. Pike, "Monitoring distributed real-time systems: a survey and future directions," no. NASA/CR-2010-216724, July 2010 [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.159.4769 2.4.1.1

[58] L. Pike, S. Niller, and N. Wegmann, "Runtime verification for ultra-critical systems," in *Proceedings of the 2nd Intl. Conference on Runtime Verification*, ser. LNCS. Springer, September 2011 2.4.1.1, 3

[59] C. Watterson and D. Heffernan, "Runtime verification and monitoring of embedded systems,"

Software, IET, vol. 1, no. 5, pp. 172 –179, 2007. 2.4.2

[60] D. K. Peters and D. L. Parnas, "Requirements-based monitors for real-time systems," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 146–158, February 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=506201.506204 2.4.2

[61] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Eagle monitors by collecting facts and generating obligations," Tech. Rep., 2003. 2.4.2

[62] H. Barringer, D. Rydeheard, and K. Havelund, "Rule systems for run-time monitoring: From eagleto ruler," in *Runtime Verification*, ser. Lecture Notes in Computer Science, O. Sokolsky and S. Tasiran, Eds. Springer Berlin / Heidelberg, 2007, vol. 4839, pp. 111–125, 10.1007/978-3-540-77395-5_10. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-77395-5_10 2.4.2

[63] D. Drusinsky, "The temporal rover and the atg rover," in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. London, UK: Springer-Verlag, 2000, pp. 323–330. [Online]. Available: http://portal.acm.org/citation.cfm?id=645880.672089 2.4.2

[64] K. Havelund and G. Rosu, "Efficient monitoring of safety properties," *Int. J. Softw. Tools Technol. Transf.*, vol. 6, no. 2, pp. 158–173, Aug. 2004. [Online]. Available: http://dx.doi.org/10.1007/s10009-003-0117-6 2.4.2

[65] ——, "Synthesizing monitors for safety properties," in *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '02. London, UK, UK: Springer-Verlag, 2002, pp. 342–356. [Online]. Available: http://dl.acm.org/citation.cfm?id=646486.694486 2.4.2, 3, 4.3

[66] G. Rosu and K. Havelund, "Rewriting-based techniques for runtime verification," *Automated Software Engineering*, vol. 12, no. 2, pp. 151–197, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10515-005-6205-y 2.4.2

[67] P. Thati and G. Roşu, "Monitoring algorithms for metric temporal logic specifications," *Electron. Notes Theor. Comput. Sci.*, vol. 113, pp. 145–162, Jan. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2004.01.029 2.4.2

[68] D. Basin, F. Klaedtke, and E. Zalinescu, "Algorithms for monitoring real-time properties," in *Runtime Verification*, ser. Lecture Notes in Computer Science, S. Khurshid and K. Sen, Eds. Springer Berlin Heidelberg, 2012, vol. 7186, pp. 260–275. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29860-8_20 2.4.2, 4.1.1

[69] A. Bauer, R. Gor, and A. Tiu, "A first-order policy language for history-based transaction monitoring," in *Theoretical Aspects of Computing - ICTAC 2009*, ser. Lecture Notes in Computer Science, M. Leucker and C. Morgan, Eds. Springer Berlin Heidelberg, 2009, vol. 5684, pp. 96–111. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03466-4_6 2.4.2

[70] D. Basin, F. Klaedtke, S. Marinovic, and E. Zlinescu, "Monitoring compliance policies over incomplete and disagreeing logs," in *Runtime Verification*, ser. Lecture Notes in Computer

Science, S. Qadeer and S. Tasiran, Eds. Springer Berlin Heidelberg, 2013, vol. 7687, pp. 151–167. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-35632-2_17 2.4.2

[71] L. Pike, "Schrödinger's CRCs (fast abstract)," in *40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010)*, June 2010, participants' proceedings. Paper available at http://www.cs.indiana.edu/~lepike/pub_pages/dsn.html. 2.4.2

[72] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "Runtime assurance based on formal specifications," in *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999. 2.4.2

[73] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, "Java-mac: A run-time assurance approach for java programs," *Form. Methods Syst. Des.*, vol. 24, no. 2, pp. 129–155, Mar. 2004. [Online]. Available: http://dx.doi.org/10.1023/B:FORM.0000017719.43755.7c 2.4.2, 3.2

[74] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu, "Hardware Runtime Monitoring for Dependable COTS-Based Real-Time Embedded Systems," *2008 Real-Time Systems Symposium*, pp. 481–491, Nov. 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4700460 2.4.2

[75] T. Reinbacher, M. Fgger, and J. Brauer, "Runtime verification of embedded real-time systems," *Formal Methods in System Design*, pp. 1–37, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10703-013-0199-z 2.4.2

[76] D. Heffernan, C. MacNamee, and P. Fogarty, "Runtime verification monitoring for automotive embedded systems using the iso 26262 functional safety standard as a guide for the definition of the monitored properties," *Software, IET*, vol. 8, no. 5, pp. 193–203, October 2014. 2.4.2

[77] D. Heffernan, S. Shaheen, and C. Watterson, "Monitoring embedded software timing properties with an soc-resident monitor," *Software, IET*, vol. 3, no. 2, pp. 140 –153, april 2009. 2.4.2

[78] U. Congress, "Health insurance portability and accountability act of 1996, privacy rule. 45 cfr 164," August 2002. [Online]. Available: http://www.access.gpo.gov/nara/cfr/waisidx_07/45cfr164_07.html 2.4.2

[79] ——, "Gramm-leach-bliley act, financial privacy rule. 15 usc §6801-§6809," November 1999. [Online]. Available: http://www.law.cornell.edu/uscode/usc_sup_01_15_10_94_20_I.html 2.4.2

[80] E. M. Clarke and J. M. Wing, "Formal methods: state of the art and future directions," *ACM Comput. Surv.*, vol. 28, pp. 626–643, December 1996. [Online]. Available: http://doi.acm.org/10.1145/242223.242257 2.4.3, 2.4.3.1

[81] J. Bowen and V. Stavridou, "The industrial take-up of formal methods in safety-critical and other areas: A perspective," in *FME '93: Industrial-Strength Formal Methods*, ser. Lecture Notes in Computer Science, J. Woodcock and P. Larsen, Eds. Springer Berlin / Heidelberg, 1993, vol. 670, pp. 183–195, 10.1007/BFb0024646. [Online]. Available:

http://dx.doi.org/10.1007/BFb0024646 2.4.3

[82] J. S. Ostroff, "Formal methods for the specification and design of real-time safety critical systems," *Journal of Systems and Software*, vol. 18, no. 1, pp. 33 – 60, 1992. [Online]. Available: http://www.sciencedirect.com/science/article/B6V0N-48TD2S0-9S/2/06b26071a0f11b04e57a132def29e23b 2.4.3

[83] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982. [Online]. Available: http://comjnl.oxfordjournals.org/content/25/4/465.abstract 2.4.4

[84] K. Wika and J. Knight, "On the enforcement of software safety policies," in *Computer Assurance, 1995. COMPASS '95. 'Systems Integrity, Software Safety and Process Security'. Proceedings of the Tenth Annual Conference on*, jun 1995, pp. 83 –93. 2.4.5

[85] Y. Falcone, L. Mounier, J.-C. Fernandez, and J.-L. Richier, "Runtime enforcement monitors: composition, synthesis, and enforcement abilities," *Formal Methods in System Design*, vol. 38, no. 3, pp. 223–262, 2011. [Online]. Available: http://dx.doi.org/10.1007/s10703-011-0114-4 2.4.5

[86] S. Hussein, P. Meredith, and G. Roşlu, "Security-policy monitoring and enforcement with javamop," in *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, ser. PLAS '12. New York, NY, USA: ACM, 2012, pp. 3:1–3:11. [Online]. Available: http://doi.acm.org/10.1145/2336717.2336720 2.4.5

[87] F. Martinell and I. Matteucci, "Through modeling to synthesis of security automata," *Electronic Notes in Theoretical Computer Science*, vol. 179, no. 0, pp. 31 – 46, 2007, proceedings of the Second International Workshop on Security and Trust Management (STM 2006). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066107003003 2.4.5

[88] J. Bowen and M. Hinchey, "Seven more myths of formal methods: Dispelling industrial prejudices," in *FME '94: Industrial Benefit of Formal Methods*, ser. Lecture Notes in Computer Science, M. Naftalin, T. Denvir, and M. Bertran, Eds. Springer Berlin Heidelberg, 1994, vol. 873, pp. 105–117. [Online]. Available: http://dx.doi.org/10.1007/3-540-58555-9_91 3.1

[89] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, no. 4, pp. 19:1–19:36, Oct. 2009. [Online]. Available: http://doi.acm.org/10.1145/1592434.1592436 3.1

[90] M. Reynal, "A short introduction to failure detectors for asynchronous distributed systems," *SIGACT News*, vol. 36, no. 1, pp. 53–70, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1052796.1052806 3.2.1.2

[91] R. Schlichting, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *Transactions on Computer Systems*, vol. 3, no. August, pp. 222–238, 1983. [Online]. Available: http://portal.acm.org/citation.cfm?id=357371 3.2.1.2

[92] M. Aguilera, G. L. Lann, and S. Toueg, "On the impact of fast failure detectors on real-time

fault-tolerant systems," *Distributed Computing*, pp. 354–369, 2002. [Online]. Available: http://www.springerlink.com/index/E03YF4ETBNLE9728.pdf 3.2.1.2

[93] C. Temple, "Avoiding the babbling-idiot failure in a time-triggered communication system," in *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, June 1998, pp. 218–227. 3.2.1.2

[94] P. Koopman, K. Devale, and J. Devale, *Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project*. John Wiley & Sons, Inc., 2008, pp. 201–226. [Online]. Available: http://dx.doi.org/10.1002/9780470370506.ch11 3.3, 3.3.1

[95] M. S. Corporation, "Carsim overview," November 2013. 3.3

[96] J. Rushby, "Modular certification," NASA Contractor Report NASA/CR-2002-212130, Dec 2002. [Online]. Available: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20030001129.pdf 3.4

[97] *Recommended Practice for a Serial Control & Communications Vehicle Network*, SAE Std. SAE J1939, 2004. 3.4.1

[98] P. H. Foo, G.-W. Ng, K. H. Ng, and R. Yang, "Application of intent inference for surveillance and conformance monitoring to aid human cognition," in *Information Fusion, 2007 10th International Conference on*, 2007, pp. 1–8. 3.4.1.1

[99] K. Lee and J. Lunas, "Hybrid model for intent estimation," in *Information Fusion, 2003. Proceedings of the Sixth International Conference of*, vol. 2, 2003, pp. 1215–1222. 3.4.1.1

[100] E. H. L. Fong, "Maritime intent estimation and the detection of unknown obstacles," Master's thesis, MIT, 2004. [Online]. Available: http://hdl.handle.net/1721.1/30279 3.4.1.1

[101] S. Lefevre, J. Ibanez-Guzman, and C. Laugier, "Context-based estimation of driver intent at road intersections," in *Computational Intelligence in Vehicles and Transportation Systems (CIVTS), 2011 IEEE Symposium on*, 2011, pp. 67–72. 3.4.1.1

[102] B. Bonakdarpour, S. Navabpour, and S. Fischmeister, "Sampling-based runtime verification," in *Proceedings of the 17th international conference on Formal methods*, ser. FM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 88–102. [Online]. Available: http://dl.acm.org/citation.cfm?id=2021296.2021308 3.4.2

[103] J. Knight, "Challenges in the utilization of formal methods," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, ser. Lecture Notes in Computer Science, A. Ravn and H. Rischel, Eds. Springer Berlin Heidelberg, 1998, vol. 1486, pp. 1–17. [Online]. Available: http://dx.doi.org/10.1007/BFb0055331 3.5

[104] J. Knight, C. DeJong, M. Gibble, and L. Nakano, "Why are formal methods not used more widely?" in *Fourth NASA Formal Methods Workshop*. Citeseer, 1997. 3.5

[105] H. Barringer and K. Havelund, "Internal versus external dsls for trace analysis," in *Runtime Verification*, ser. Lecture Notes in Computer Science, S. Khurshid and K. Sen, Eds. Springer Berlin Heidelberg, 2012, vol. 7186, pp. 1–3. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29860-8_1 3.5.1

[106] D. Bianculli, C. Ghezzi, C. Pautasso, and P. Senti, "Specification patterns from research to industry: A case study in service-based applications," in *Software Engineering (ICSE), 2012 34th International Conference on*, june 2012, pp. 968 –976. 3.6.1

[107] P. Bellini, R. Mattolini, and P. Nesi, "Temporal logics for real-time system specification," *ACM Comput. Surv.*, vol. 32, no. 1, pp. 12–42, Mar. 2000. [Online]. Available: http://doi.acm.org/10.1145/349194.349197 4.1.1

[108] J. Ouaknine and J. Worrell, "Some recent results in metric temporal logic," in *Proceedings of the 6th international conference on Formal Modeling and Analysis of Timed Systems*, ser. FORMATS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 1–13. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85778-5_1 4.1.1

[109] R. Alur and T. Henzinger, "Logics and models of real time: A survey," in *Real-Time: Theory in Practice*, ser. Lecture Notes in Computer Science, J. de Bakker, C. Huizing, W. de Roever, and G. Rozenberg, Eds. Springer Berlin / Heidelberg, 1992, vol. 600, pp. 74–106, 10.1007/BFb0031988. [Online]. Available: http://dx.doi.org/10.1007/BFb0031988 4.1.1

[110] D. Basin, F. Klaedtke, S. Mller, and B. Pfitzmann, "Runtime monitoring of metric first-order temporal properties," in *Proceedings of the 28th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 08)*, vol. Dagstuhl Seminar Proceedings, 2008, pp. 49–60. 4.1.2, 4.3

[111] O. Kupferman and M. Vardi, "Model checking of safety properties," *Formal Methods in System Design*, vol. 19, no. 3, pp. 291–314, 2001. [Online]. Available: http://dx.doi.org/10.1023/A%3A1011254632723 4.2

[112] A. Bauer, M. Leucker, and C. Schallhart, "The good, the bad, and the ugly, but how ugly is ugly?" in *Proceedings of the 7th international conference on Runtime verification*, ser. RV'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 126–138. [Online]. Available: http://portal.acm.org/citation.cfm?id=1785141.1785155 4.2

[113] ——, "Runtime verification for ltl and tltl," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 14:1–14:64, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/2000799.2000800 4.2

[114] T. Reinbacher, J. Brauer, M. Horauer, A. Steininger, and S. Kowalewski, "Past time ltl runtime verification for microcontroller binary code," in *Formal Methods for Industrial Critical Systems*, ser. Lecture Notes in Computer Science, G. Salan and B. Schtz, Eds. Springer Berlin Heidelberg, 2011, vol. 6959, pp. 37–51. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24431-5_5 4.3

[115] P. Zuliani, A. Platzer, and E. Clarke, "Bayesian statistical model checking with application to stateflow/simulink verification," *Formal Methods in System Design*, vol. 43, no. 2, pp. 338–367, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10703-013-0195-3 4.3, 4.3.1

[116] G. Holzmann, "The power of 10: rules for developing safety-critical code," *Computer*,

vol. 39, no. 6, pp. 95–99, June 2006. 5.1.1

[117] MIRA Ltd, *MISRA-C:2004 Guidelines for the use of the C language in Critical Systems*, MIRA Std., Oct. 2004. [Online]. Available: www.misra.org.uk 5.1.1

[118] L. Evans, *Traffic safety and the driver.* Van Nostrand Reinhold Co, 1991. 6.2.3

[119] N. S. Council, "Defensive driving course," 1992, (Course Guide). 6.2.3

[120] "Pcan-usb pro: Peak system," http://www.peak-system.com/PCAN-USB-Pro.200.0.html?&L=1. 6.3

[121] D. A. Rennels, "Fault-tolerant computing concepts and examples," *IEEE Trans. Comput.*, vol. 33, pp. 1116–1129, December 1984. [Online]. Available: http://portal.acm.org/citation.cfm?id=1311963.1312787 7.1.1.2

[122] V. Nelson, "Fault-tolerant computing: fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, July 1990. 7.1.1.2

[123] N. Navet, A. Monot, B. Bavoux, and F. Simonot-Lion, "Multi-source and multicore automotive ecus - os protection mechanisms and scheduling," in *Industrial Electronics (ISIE), 2010 IEEE International Symposium on*, July 2010, pp. 3734–3741. 7.1.2

[124] A. Kane and P. Koopman, "Mode Based Ride-Through for Embedded System Runtime Monitoring," *Dependable Systems & Networks, 2013.*, 2013, (in review). 7.3.1