Lightweight Formalizations of System Safety Using Runtime Monitoring of Safety Proof Assumptions

Submitted in partial fulfillment of the requirements for

the degree of

Master of Science

in

Electrical and Computer Engineering

Casidhe Hutchison

B.S., Electrical and Computer Engineering, Lafayette College
B.S., Mathematics, Lafayette College

Carnegie Mellon University
Pittsburgh, PA

Aug 2016

Acknowledgements

First and foremost, I'd like to thank Milda Zizyte, for being there for me through thick and thin, and being the best grad school buddy and friend one could hope for. I'd also like to thank my parents, Gary and Johannah Hutchison, for ingraining in me a love for both technology and creative pursuits, and for being an unwavering source of support.

I would also like to thank Phil Koopman, my advisor, for giving me support and experience, and Mike Wagner for providing real world context and opportunities for my research.

I'd like to thank my fellow graduate students, both in my research group and out. Thanks especially to my fellow advisees Aaron Kane, John Filleau, Malcolm Taylor, and Justin Ray and again to Milda Zizyte (who can not receive enough gratitude). Thanks also to Ben Niewenhuis, who provided endless feedback on innumerable drafts of this manuscript.

Lastly I'd like to 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 (GM-CMU VIT-CRL) and Connected and Autonomous Driving Collaborative Research Lab(GM-CMU CAD-CRL). It was also supported by SYSU-CMU joint Institute of Enginering.

Abstract

As embedded systems become more entrenched in safety critical applications (such as autonomy systems), formal verification of safety properties becomes critically important. However, two of the chief roadblocks to formal verification of embedded systems are: first, it can be difficult to ensure the accuracy of a model of an embedded system, and second, design of that model itself can be prohibitively difficult.

To this end, we introduce PinkySwear, a system to ease development of formal safety arguments by automating these analysis steps using a system description written using a simple specification language of common patterns for behavioral requirements. PinkySwear both automatically generates a model for formal verification of system adherence to safety properties, and checks the accuracy of that model using runtime monitoring expressions on captured debug traces or live system behavior. PinkySwear is designed to be usable by non-experts and requires only a high-level understanding of formal methods.

In this paper, we outline the development of PinkySwear, the first tool to be designed explicitly for use in safety critical embedded applications by non-experts, and include a case study evaluation to provide empirical evidence of its soundness and suitability for this purpose. The case study demonstrates that the specification language is sufficiently expressive for this example system, and includes fault injection experimentation to evaluate PinkySwear as a development tool for detecting faults and as a failsafe monitoring solution.

Contents

Contents iii

List of Figures iv

List of Tables v

1 Introduction 1

2 Background 3

2.1 Comparison to Related Work 4

3 PinkySwear Framework 6

3.1 High Level View of PinkySwear 6

3.2 System Specification 10

3.3 Tool Chain 16

4 Case Study 19

4.1 Target System 20

4.2 Are Behavior Templates Sufficient? 21

4.3 Is PinkySwear an Effective Development Tool? 21

4.4 Is PinkySwear an Effective Failsafe? 23

5 Discussion 27

5.1 Monitor Scope 27

5.2 Synchronous Model of Execution 28

5.3 Small Case Study Size 28

6 Conclusions 29

6.1 Contributions 29

6.2 Threats to Validity of the Work . . . 30
6.3 Future Work . . . 31

A A More Complicated Safety Pattern **32**

B Test Case Artifacts **34**

Bibliography **37**

List of Figures

3.1 PinkySwear generates a formal model and monitor invariants to check the design and implementation for faults. . . . 8
3.2 The PinkySwear safety chain: If a trace from the implementation satisfies the design, and the design has been proved to allow only safe behavior, then the trace cannot violate safety rules. . . 9
3.3 One of the strongly typed interface definitions for the elevator Door Controller . . . 11
3.4 An example Critical Property for the distributed elevator test case . . . 12
3.5 An example behavioral specification for the elevator Door Controller used in the case study. . . 14
3.6 An example constraint for the elevator Dispatcher. . . . 15
3.7 Behavioral specifications from the Door Controller, and the SMV code they are used to generate. 17
3.8 Behavioral specifications from the Door Controller, and the monitor expressions they are used to generate. . . . 18

4.1 Case study elevator network architecture diagram[29] . . . 20
4.2 Using correctness checking results in earlier detection of fault activation. . . . 25

B.1 XML description of the interface and behavior of the Call button in the Test Case Elevator. . . . 35
B.2 The generated SMV model for the Call button, generated using the *Priority* conflict resolution . 36

B.3 The generated monitor expressions, and propositionalizer expressions, generated using the *Priority* conflict resolution . . . 36

List of Tables

4.1 Injected design faults and the PinkySwear analysis results. . . . 22
4.2 Injected implementation faults and the PinkySwear analysis results. . . . 23
4.3 Injected runtime faults and the PinkySwear failsafe detection results. . . . 24

Chapter 1: Introduction

Embedded systems in safety critical domains, such as automotive and autonomy systems, are becoming increasingly complex in order to provide more feature rich experiences. As these systems become more complicated, it is increasingly advisable to formally verify their adherence to safety properties to prevent loss of life. Formal verification uses a mathematical model of a system to try to prove safety properties about the system (e.g. an elevator never opens the doors while traveling between floors.) These verification results are an excellent way to examine complex system behavior that can be difficult to understand by manual inspection. However, the results are only accurate if the system model accurately describes behavior of the system[25].

There are many things that could cause the real system behavior to differ from the system model. These models necessarily contain simplifications to make analysis easier, and these simplifications cause the model behavior to be inexact. Additionally, implementations in the real world, even mature ones, often contain bugs which can cause the system to behave aberrantly. Lastly, even if the design and implementation are fault free, transient faults such as failure of a sensor or other electronic component can cause behaviors that are unanticipated by the model. Since the proof of system safety depends on the accuracy of the model, system behavior needs to be examined to see if it actually fits the model. One approach to doing this is to use runtime monitoring to examine system traces for compliance with model behavior[11].

Formal verification and runtime monitoring assume different fault models from each other, but both fault models are relevant to complex embedded systems. Formal verification assumes the faults are in the design of a system, by exploring the system model trying to find where these faults could cause a violation of formal safety properties. Runtime monitoring, on the other hand, has a fault model that includes transient faults and latent implementation errors such as coding errors, which it catches when these faults propagate to become failures in monitored properties.

As both of these fault models are of concern for safety critical systems, we developed PinkySwear, a complementary framework for verification and validation of behavioral requirements and interfaces, as a fault detection tool for system development. PinkySwear uses formal verification to analyze the system design, and runtime validation to analyze the implementation of the design. Using these two techniques together, PinkySwear aims to expand detection of faults during development without requiring extensive expertise in either method of analysis.

PinkySwear also aims to function as a safety monitor. It uses the verification of the design to assert that design compliance monitors are sufficient for safety monitoring. Because the design has been verified to not allow the system to violate the safety specification, then system behavior complying with the design guarantees that the system is not violating safety rules. If the system deviates from the expected behavior, then the system model used for verification no longer accurately describes the system. In this case, the verification of the design doesn't apply to the system anymore, and the system may become unsafe.

One of the roadblocks to this framework is finding a language for the system design that is structured enough to serve as a system model but is still easy to check against runtime traces. Additionally, research in the area of system descriptions has revealed that inconsistent requirements, as well as interface inconsistencies/misunderstandings, have a huge impact in safety related errors in distributed embedded systems, both in the design and the implementation of complex embedded systems[23]. Thus, it is critically important that the system description language be unambiguous to both designers and implementers to reduce the likelihood of errors, and be formal enough to support analysis.

To address these needs, PinkySwear uses a strict design language to define both the interfaces and the behaviors of the system components. This language is based on existing formal languages and distributed system specification techniques. Language components were selected to support three important properties: the interfaces must be type checkable, description of system architecture should be modular and compositional, and behaviors must be able to have non-determinism in timing and value.

Finally, to demonstrate the effectiveness of PinkySwear, we analyze a distributed elevator system. This case study provides empirical evidence of PinkySwear's effectiveness as a development tool and a failsafe system, as well as demonstrates the applicability of the tool to a reasonably complex system. The architecture and design of this elevator are used to provide examples of the specification language used in this research.

Chapter 2: Background

Formal methods techniques can be split broadly into two categories: **verification techniques** and **validation techniques**. Verification techniques, such as model checking and theorem proving, take formal descriptions of the system and use these descriptions to prove formal properties about the described system behavior[6]. On the other hand, validation techniques, such as runtime monitoring and code contracts, take a series of rules and check the system behavior against them[11]. In a sense, verification starts with system behaviors and tries to prove rules, while validation starts with rules and checks system behaviors to see that the rules are followed.

The verification technique we use is model checking[6]. Model checking uses a model of system behavior to create a large state space that encompasses all of the possible system states. The model checker then uses explorations of this statespace to check if desired system properties hold true. The model is a formal description of the behavior of the system (usually in a description language specific to the model checker), while the properties are typically specified in some formal logic. In comparison to other verification techniques, such as a proof engine, model checkers are slower but much more automated, requiring less user expertise[6][25].

PinkySwear uses model checking to provide a proof of design compliance with a particular set of safety rules, finding errors in the system design. However, this model checking doesn't detect faults in the implementation, such as coding errors or component failures, since it is based solely on the specification.

Validation techniques, on the other hand, are designed to detect faults in a running system. Unlike models, monitors make no assumptions about system behaviors, and instead observe the actual behavior of a system and check it for rule compliance. The validation technique we use is runtime monitoring. Runtime monitoring collects a log of the system behavior (called a system trace) and compares it against a set of monitoring rules. This comparison can be done on collected and saved logs, known as offline monitoring, or it can be done on traces as they are generated in real time, known as online monitoring[11].

PinkySwear uses runtime monitoring to ensure that system behavior conforms to design specification. Since we are using this monitor on safety critical systems, the monitor we use is a bus monitor, a type of external system monitor. External system monitors have an advantage over the more powerful and more integrated software monitors for safety critical applications, as they can be removed from the system without the system requiring any changes that might invalidate all the monitoring results[16]. Bus monitors work by reading values off the system data bus. These values are used as input to the monitoring expressions[26]. As these values are on the bus under normal operation, the system is not affected by the presence of the monitor.

It is important to note, however, that while runtime monitoring provides an analysis of observed system behavior and sees faults that may not be caught by user observation (either due to the speed of the fault, or the fault remaining in the internal behavior of the system), it is still a testing tool. No amount of passive monitoring can provide a guarantee of system safety; it can only provide empirical observations that critical properties were not violated in a particular test run. Since a large number of design errors are revealed in primarily off-nominal conditions[12], it's possible, or even likely, that flaws still exist in the design or implementation of systems that have been monitored for faults during normal acceptance and integration testing.

In order to provide greater debugging and safety monitoring capability, we use a carefully designed combination of both model checking and runtime monitoring to detect faults in system design and implementation.

2.1 Comparison to Related Work

There are a number of other tools that are based on the concepts of formal verification and system validation using system specification documents. But while these tools are based on similar ideas, PinkySwear is the first tool designed specifically for easy use by non-experts in the safety critical embedded systems domain.

The first paper to significantly explore the idea of generating monitors from a specification and develop a tool chain for it was the MaC group at UPenn[17][21]. The MaC (Monitoring and Checking) framework was designed with the intent to "bridge the gap between formal specification, which ensures the correctness of a design, and testing, which partially validates an implementation"[17]. The MaC framework is primarily a debugging tool, designed to supplement testing, and while it has some discussion of integration with formal verification, it is for the most part a monitoring solution.

While the MaC framework showed the effectiveness of monitoring systems for compliance with specification, the framework's target was primarily for software systems with instrumented code, requiring modifications to the source code and perhaps making it unsuitable for usage on safety-critical embedded applications. Additionally, this means the approach would not work for systems without available source code, such as many of the commercial off-the-shelf (COTS) components that are ubiquitous in today's embedded systems.

Later, in 2005, Bayazit's "Complementary Use of Runtime Validation and Model Checking"[2] explored combining monitoring and model checking as a development tool to ensure correct code. In Bayazit's paper, model checking was used to verify global properties of the system, while runtime validation was used to check local properties.

Bayazit et al. demonstrated the effectiveness of using a monitor to make model checking more tractable. However it was not done using a systematic approach, but rather models and monitors were written ad hoc, with no guarantees of logical consistency between the two.

Most recently, "ModelPlex: Verified Runtime Validation of Verified Cyber-Physical System Models"[25] used the KeYmaera proof engine to verify the safety of a Cyber-Physical System model, and generated monitors to check system traces for compliance with the model. In this way, it acted as a runtime safety monitor, by ensuring verification results from the safety proof applied to the actual system.

While checking the applicability of the safety proof is similar to the approach employed by PinkySwear, ModelPlex focuses on the development of a formal proof of system safety, and then generates monitors from this proof, rather than generating them from a system description. Thus, when a monitor detects a violation, it comes from a computationally reduced form of the system level dynamics equations, which can make it difficult to diagnose which component has failed. This makes ModelPlex more suited to safety monitoring than debugging.

Additionally, Modelplex requires a high degree of user expertise both to write the original model in Dynamic Logic (the KeYmaera mathematical language) and to guide the proof engine towards a formal proof of the property (for example, a user may be required to specify a differential invariant, or a monotonic function over a series).

PinkySwear brings these components together in a framework designed specifically for both debugging and monitoring safety critical embedded systems, with minimal required expertise in formal methods.

Chapter 3: PinkySwear Framework

The goal of PinkySwear is to detect a) faults in the design of embedded systems that could lead to violations of the safety critical properties, and b) faults in the implementation of embedded systems that could lead to violations of the design specification. PinkySwear detects these faults by generating a model to verify that the design does not violate any safety properties, and by generating monitors to check that the implementation does follow the design. PinkySwear uses these fault detection capabilities to provide safety assurances about implemented systems, giving PinkySwear utility both as a development tool and as a safety monitoring solution.

3.1 High Level View of PinkySwear

The PinkySwear framework starts with a specification document that defines the interfaces and behaviors of the components in a system. The specification document also contains safety rules for the system that don't affect system behavior, but are rules the system should follow when executing.

This specification document is based on a periodic model of execution for the controllers, with outputs that change once per period based on the inputs and internal state of the controller. Due to this model of execution, PinkySwear may have difficulty representing some event driven systems, particularly when they are highly aperiodic.

The specification document is used to generate a formal system model for safety analysis, as well as monitors to check that the system complies with this specification. The safety analysis checks a formal model of the system against the safety rules, exploring all the behaviors allowed in the specification document. The monitors examine actual system traces and try to find deviations from the behaviors described in the specification document.

PinkySwear is designed using these analysis techniques so that in may be used as a development tool and safety monitoring solution for even those embedded systems for which source code may not be available. However, because neither monitoring nor model checking directly analyze the code, PinkySwear cannot provide any guarantee of code correctness.

3.1.1 PinkySwear as a Development Tool

One of the goals of PinkySwear's design is use as a debugging tool to try to detect faults in a system during development. It is designed to catch both design faults and implementation faults, to be a development tool for both those designing a system at a high level, and those taking that design and trying to implement it. Below we outline how PinkySwear is designed to meet meet this goal.

Design faults may result in ill-defined corner cases and race conditions, which are difficult to detect in normal testing[3][1]. Because these obscure timing conditions occur with low probability, it's rare to see them in short testing time frames. Once deployed, however, production quantities and long product lifetime greatly increases the number of exposure hours, and thus increases the risk of these faults being exercised. Unlike testing, however, model checking exhaustively explores all possible paths. To ensure that even rare timing conditions are examined for possible violations of the safety properties, PinkySwear generates models that encapsulate system timing.

Even if there are no faults in the design of the system, the system might still experience unsafe behavior if the implementation does not follow the design. PinkySwear is also designed to detect faults in the implementation that cause violation of the design specification, such as control flow errors, misunderstandings of the design, or overaggressive optimization (which may cause a component to violate the design specification).

It is difficult to manually observe when a fault in an individual component of a complex system may cause that component to violate behavioral specifications. Violations are especially difficult to see in cases where the majority of the time the variation is masked in greater system behavior, and so does not lead to a user observable failure. However, these usually unobservable violations can cause problems in rare conditions. For example, a 100 ms excessive delay on an actuator response may seem innocuous, and would likely go unnoticed by a user, but it might cause catastrophic problems if it violates a timeout or interlocking mechanism. To detect faults in individual component behavior, PinkySwear generates runtime correctness monitors, which check the outputs of the system components and detect if they violate the system specification.

PinkySwear uses the system specification document to a) generate models that try to detect faults in the design that could lead to the violation of critical properties, and b) generate monitors to check for implementation faults (Figure 3.1).

The design choice to generate formal models and runtime monitors from requirements for verification and validation was based on the prior success of using automatically generated models/monitoring rules for system analysis. Generation of formal models from requirements has been shown to be very effective

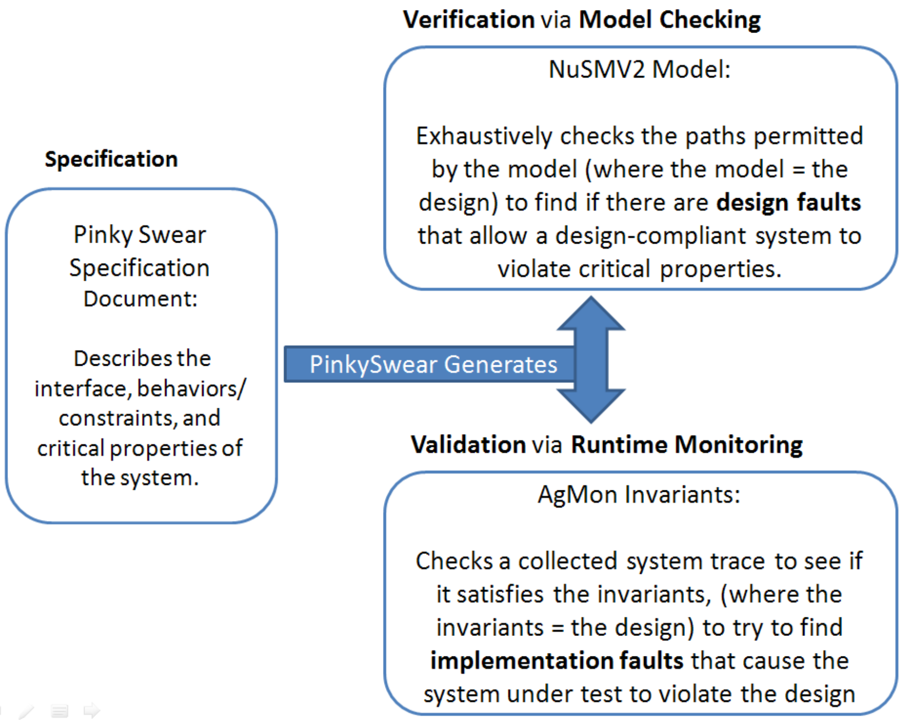

Figure 3.1: PinkySwear generates a formal model and monitor invariants to check the design and implementation for faults.

at detecting design flaws[12][15], and generated runtime monitors have been demonstrated to be similarly effective at detecting deviations from specified behavior in running systems[26][16].

PinkySwear for Safety Monitoring

Another goal of PinkySwear is to provide utility as a failsafe monitoring solution on production systems, such as chemical plants or autonomous vehicles. It provides an end-to-end safety argument, which is used to make monitoring more effective, as well as provide lower detection latency and more versatile checking. This is done by using model checking to check for complex safety properties, and thereby reducing the monitoring task to checking the correctness of the implementation.

The key idea behind safety monitoring using PinkySwear is that monitors and models that are generated from the same underlying system description give a stronger "safety chain" argument for the safety of the system than the techniques used independently. The hypothesis is that if the design is fault free (i.e. it does not violate the critical properties) and the implementation is fault free (i.e. the implementation does not violate the design), then the implementation is safe. This safety chain hypothesis is outlined in Figure 3.2

This hypothesis starts with a semi-formal partial specification (Φ), which includes interface informa-

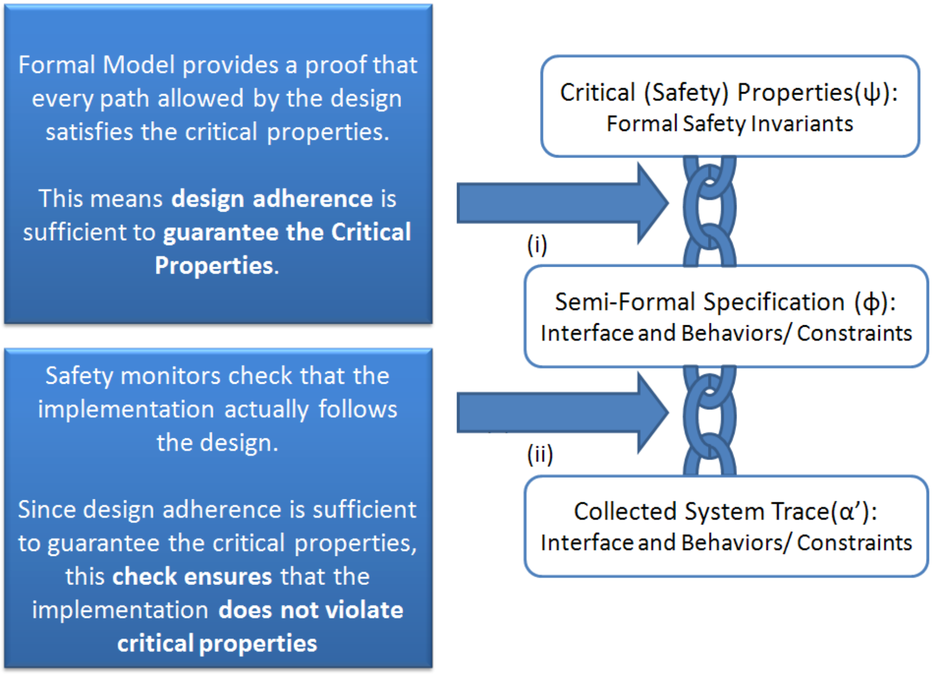

Figure 3.2: The PinkySwear safety chain: If a trace from the implementation satisfies the design, and the design has been proved to allow only safe behavior, then the trace cannot violate safety rules.

tion and behavioral requirements, as well as the safety properties to be checked. The specification is partial, meaning that it is composed of non-deterministic behavioral rules, which define a large set of possible execution paths for a given input, of which the actual system will necessarily take only one[28]. Because PinkySwear is checking safety properties, if we can show all paths are safe, then any individual path must be safe. This allows room for implementers to make optimizations of behavior without the need to overly complicate the specification with implementation details; as long as the optimization is within the paths defined by Φ, we know it is safe.

A model is generated to explore all possible behaviors permitted by the specification. Since the specification is partial (non-deterministic) there are multiple possible responses to the same input sequence, so to verify the design we need to check critical properties over all paths. The formal model is checked to show that for all paths α that are permitted by the specification Φ, none violate the critical property ψ (arrow i in Figure 3.2). If a path can be found that does violate ψ, then this means it is possible to build the system according to design and still have it be unsafe; therefore the system design is not strong enough to guarantee system safety, and must be refined. If no such path exists, then all the possible execution paths that satisfy the specification are safe.

Similarly, monitors are generated from the specification Φ that check to see if the implemented system conforms to the partial specification (arrow *ii* in Figure 3.2). That is, given a collected system trace α', is α' permitted by Φ? If so, then we know that α' satisfies the safety property ψ since our formal analysis has shown that *any* path that is permitted under Φ satisfies ψ. In the case that the trace α' is not permitted under Φ, then the component is not behaving correctly (due to some fault or defect) and therefore the system might be unsafe.

To summarize, PinkySwear uses formal methods to prove the design only allows safe behavior, and generates monitors to check system traces for design compliance. If we have a proven safe design ($\forall \alpha$ s.t. $\alpha \models \Phi . \alpha \models \psi$), given by the formal analysis, correctness of a system trace ($\hat{\alpha} \models \Phi$) implies safety of the trace($\hat{\alpha} \models \psi$), and runtime correctness monitoring is sufficient to show a given run is safe.

3.2 System Specification

Whether PinkySwear is being used as a development tool or a failsafe mechanism, the process starts with a system specification document. As a key goal of PinkySwear is to be usable to non-experts, this specification document needs to be easy to write, while still having the formality and expressiveness necessary to perform analysis. To help make specification documents easy to write, they are based on UML descriptions of system architecture, and use templates to make writing the formal language portions of the document simpler.

The specification document is comprised of a list of components. Each component contains a description of its interface, behavior, operational properties (such as control period), and any internal states, as well as instances of other components and connections between them. Components also contain a list of critical properties, or properties that the component must not violate for the system to be safe.

To illustrate the parts of a component description, we are going to use an example component. The component we are going to use is the Door Controller from the distributed elevator case study, which is detailed in Chapter 4. The Door Controller provides the logic that controls an individual door on the elevator, and decides to open or close the door to allow passengers on, based on the state of the elevator. It accepts input from the dispatcher, the elevator call buttons, and some of the smart sensors that send information about the physical state of the system across the network.

While excerpts from the component specification of the Door Controller are used here as an example, the full component specification for another controller from the test case system can be found in Appendix B, Figure B.1, which demonstrates how these pieces fit together.

3.2.1 Component Interface Definition

The interface definition of a component details the inputs, outputs, and strong typing information for each (see Figure 3.3), similar to other specification languages[22]. This gives us a list of variables (X), and the possible values that each variable may assume ($N_\chi | \chi \in X$). These outputs may also contain a default value. If a default value is not specified then the output can assume any value at start up, in which case it is assumed that the output during startup is not known, and all possible values are tested.

```
<output>
    <name>Door_Command</name>
    <type default="stop"><enum>{stop,open,close,nudge}</enum></type>
    <description>The drive controller output</description>
</output>
```

Figure 3.3: One of the strongly typed interface definitions for the elevator Door Controller

3.2.2 Critical Properties

Critical properties are restrictions on behaviors that the system must not violate in order to remain safe. These can be simple speed limits or complicated conditional/temporal statements. They are often emergent properties, meaning that they depend on the interaction of more than one component of the system or more than one behavioral requirement of a component.

These properties are usually created by the system designer examining the system at a high level and looking at potentially unsafe situations and how they could occur. For example, an unsafe situation for an automated electric vehicle might be "the motors are unpowered without the brakes enabled." This could lead to uncontrolled movement of the vehicle, so a safety rule might be power to the motors is never cut unless the brakes are engaged.

These critical properties need to be written in a formal language so that the model checker can verify that they hold. These properties can be written in any of the languages supported by the NuSMV model checker, including Linear Temporal Logic(LTL), Computation Tree Logic(CTL), and Process Specification Language(PSL). The examples presented in this paper are written in LTL[27]. LTL is a modal logic, with the normal boolean logical operators ($\neg, \wedge, \vee$) and additional temporal (modal) operators (Globally ($\Box$), Eventually ($\Diamond$), and Until ($\mathcal{U}$)).

While it is the goal of PinkySwear to be usable without deep knowledge of formal languages, some level of formality is necessary in order to provide verification of the system behavior. In order to ease the generation of critical properties, we recommend usage of the formal language pattern repositories, such as [10],[8], or [4] . This allows the system designer to specify the safety properties without needing a deep understanding of formal languages.

For example, in the elevator case study, one critical property we wish to express is that "the elevator never moves with the door open." To get the LTL expression for this safety requirement, we use the pattern repository in [10] to look up the correct pattern. The modal operator here (the timing aspect) is "never", so we are going to use the Absence pattern: $\Box(\neg p)$. The remaining exercise is determining the proposition p, which is a matter of expressing what "moving with doors open" is in terms of system variables. Since this is similar to creating guards for "if" or "case" statements, we can assume the user is able to do this. The formal language expression of this safety property is shown in Figure 3.4. For a more intricate example of using this pattern repository to generate critical properties refer to Appendix A.

$$\Box\neg(\neg(\text{Door\_Closed\_all}) \wedge \neg((\text{Drive\_Control} = \text{stop})\vee (\text{Drive\_Control} = \text{level\_up}) \vee (\text{Drive\_Control} = \text{level\_down})))$$

Figure 3.4: An example Critical Property for the distributed elevator test case

3.2.3 Behavioral Requirements Language

Another important piece of making PinkySwear easy to use for non-experts is that the language used to define the behavioral requirements needs to be intuitive, yet still formal enough to have strict unambiguous semantics. There are several languages designed specifically for behavioral requirements, including table based specification[15] and state-machine based languages [22][14]. However, these languages are either exhaustive, making them difficult to use to model complex algorithmic behavior, or do not provide the capacity for non-deterministic timing requirements.

To find a language that addresses our needs, we have used a subset of bounded Metric Temporal Logic (bMTL)[16], which is a variant of LTL as described above, where the modal operators are substituted with bounded time operators. For example, $\Box_{\{0,t\}}A$ indicates that A holds over the next t seconds.

To make usage of bMTL as a specification language easier, we use specific bMTL constructions of common intended system behavior patterns, which are set forth in [16]. These patterns include the **bounded response** pattern (pattern 1), which states that the system must respond to an input within a specified about of time, and the exclusion **constraint** pattern (pattern 4), which states that propositions can not be true at the same time (for example, a mutex can not allocate a resource to two processes at the same time).

3.2.4 Response Behavior Specifications

The model PinkySwear uses to define response behaviors is based on the specification pattern 1.a from [16] for bounded response: `If` *guard condition* `Then` *response* `within bounds` (t_1, t_2). These response behavior specifications describe the behavior of the system at the interface level, and do not cover implementation details.

Semantically, these response specifications are constructed from a single guard condition (ϕ), and a non-empty list of actions(ρ_i) where

$$\phi \rightarrow \bigwedge_i \rho_i.$$

The guard condition(ϕ) is a bMTL expression over propositions, restricted to logical operators and the $\blacksquare_{\{t,0\}}$ (always in the past, since time t) temporal operator:

$$\phi ::= \neg\phi | \phi \wedge \phi | \phi \vee \phi | \theta | \blacksquare_{\{t,0\}}(\theta),$$

where θ is a logical proposition. These expressions describe formulas of the form:

$$\theta ::= \chi < \sigma | \chi \leq \sigma | \chi > \sigma | \chi \geq \sigma | \chi = \sigma | \chi \neq \sigma,$$

where σ is either some value $\nu \in N_\chi$ or some variable $\hat{\chi} \in X$ such that $N_\chi = N_{\hat{\chi}}$, where $N_{\hat{\chi}}$ is the set of values χ can assume based on the interface specification.

The actions that occur when a guard is met are also bMTL expressions, constrained to the form:

$$\rho ::= \chi = \nu | \bigvee_j \chi = \nu_j | \Diamond_{\{t_1,t_2\}} \chi = \nu | \Diamond_{\{t_1,t_2\}} \left(\bigvee_j \chi = \nu_j \right),$$

where $\nu, \nu_j \in N_\chi$. The temporal operator $\Diamond_{\{t_1,t_2\}}$ is the "happens" operator, which is satisfied if and only if the subformula is satisfied at some point between the next t_1 and t_2 seconds. These expressions (ρ) describe assignments to variables, where the timing may be non-deterministic (in the range t_1 to t_2) and the value may be chosen non-deterministically from the set of ν_js.

An example behavioral specification for the Door Controller is shown in Figure 3.5. This behavior describes the response of the Door Controller to a "Door_Reversal" signal, which is triggered when a passenger obstructs the door. The behavior requires that the internal flag "Has_Reopened" be set immediately to record that this has happened (ρ_1), and that the door motor is set to reopen the door within the next 100 ms (ρ_2). The flexibility in the timing of this requirement lets the system implementer decide what the best timing is to discourage people obstructing the doors, without making them feel as if the doors are dangerous.

$$\overbrace{\text{Door\_Reversal} \land \neg\text{Has\_Reopened} \land (\text{Door\_Command} = \text{close})}^{\phi}$$
$$\rightarrow \underbrace{(\text{Has\_Reopened} = \text{true})}_{\rho_1} \land \underbrace{\overbrace{\Diamond_{0,100}}^{happens} (\text{Door\_Command} = \text{open})}_{\rho_2}$$

Figure 3.5: An example behavioral specification for the elevator Door Controller used in the case study.

Conflict Resolution

Since it is often possible for multiple behavioral specification guards in a component to be satisfied at once, it is necessary to establish some form of conflict resolution. There are several possibilities to address this issue systematically.

First, *Priority* gives precedence to behavioral specifications that are defined first. Thus, each guard can be read as "guard is satisfied and all previous guards are not." Alternatively, it can be thought of in program flow terms as a series of "elseif" statements. While this is the most straightforward method of conflict resolution, it can lead to the component doing unexpected things if the designer is not anticipating this precedence.

A second method is to *Satisfy Any* response requirements, where, if several guards are satisfied, then taking a behavioral path that satisfies any of the assignments can be considered valid. This method of resolution makes the most sense in the context of a service system, where satisfying any of several requests constitutes correct action. Again, this can lead to the component behaving unexpectedly, as requirements can seem to be ignored unexpectedly when other guards are satisfied, with no clear precedence or consistency.

A final method for resolving conflicts, is to strictly *Satisfy All* behaviors. That is to say, if two guards are satisfied, then both of their assignments must be satisfied. In cases where there is no value that satisfies both assignment expressions, there is a unresolvable conflict. For the design to still be valid in these cases, the guards must be mutually exclusive so that the conflict never arises in the system. This can happen logically, (e.g. "$A \rightarrow B = 1, \neg A \rightarrow B = 0$") or due to some higher level behavior (e.g. it is physically impossible to for both "Door_Opened" and "Door_Closed" to be true at the same time in the Elevator, though it is not symbolically exclusive).

To ensure unresolvable conflicts do not happen regardless of symbolic reduction, PinkySwear adds an additional safety property to the specification of the component that both guards are never satisfied at the same time. The model checker is then under the obligation to prove that this unresolvable conflict never occurs, and the runtime monitor must check that this is the case at runtime.

PinkySwear supports all three of these methods, and by default uses *Satisfies All* as it is the most strict.

3.2.5 Constraints

It is also useful to be able to specify the behavior of a component in terms of constraints, restricting the behavior of the component without explicitly stating it as a response to an input. These constraints are based on patterns 2.a and 4.a from [16], and take the form of logical propositions that must hold true in every state:

$$\textit{invariant}\ \texttt{is always True.}$$

Constraints are useful in allowing flexibility of design and the separation of design details from implementation details. For instance, rather than checking an optimized path planning algorithm for the elevator, our system design may instead specify that the elevator should be moving towards any valid floor. This way, if the algorithm is re-tuned, the design does not need to be revalidated to demonstrate the safety of picking floors in a new order.

Since the Door Controller does not have any constraints in its specification, the example in Figure 3.6, is an example from the elevator Dispatcher. The Dispatcher receives calls over the CAN bus and sets the target floor for the elevator correspondingly.

$$\text{Target\_Floor} = 1 \rightarrow (\text{At\_Floor} = 1 \vee \text{Car\_Call\_1} \vee \text{Hall\_Call\_F1\_U})$$
$$\text{Target\_Floor} = 2 \rightarrow (\text{At\_Floor} = 2 \vee \text{Car\_Call\_2} \vee \text{Hall\_Call\_F2\_D} \vee \text{Hall\_Call\_R2\_D})$$

Figure 3.6: An example constraint for the elevator Dispatcher.

Constraint Satisfaction Strictness

There is no conflict resolution for constraints, since they must each always be satisfied, however there are different ways to interpret the meaning of "always satisfied". For example, a constraint that a slow controller must mirror the input from a faster controller will not be satisfied while the slower controller is idle, but it may be sufficient for the controller to mirror the the input every time it ticks. To accommodate this, we allow for two levels of strictness for satisfying constraints.

First is *Always* satisfied, in which a constraint must always be satisfied regardless of the state of the controller. This is likely to fail for periodic systems as described above, but is ideal for event driven systems, where components do not have a control period delay.

The less strict option is *On Tick*, where the component must satisfy the constraint when it takes action, but if it is currently idle, then the constraint need not be satisfied. This makes a sense for periodic systems, where the component is only able to react to input values once per controller period.

3.3 Tool Chain

So far we have outlined the PinkySwear description language for distributed embedded systems, which provides the framework with well defined interfaces and formalized behavioral requirements. This gives an unambiguous description of how a system is laid out and how it behaves, which will allow us to generate the input to the analysis tools.

PinkySwear uses two open source tools for verification and validation: the NuSMV2 model checker[6] to validate the system design, and AgMon, the formal bus monitor developed in Chapter 3 of [16], to monitor the system design adherence.

3.3.1 NuSMV2

PinkySwear uses model checking for design verification (over proof engines) as it is heavily automated, requiring no user interaction past the initial model creation. While many proof engines are moving to greater levels of automation[25], they still require user intervention for many tasks. Model checkers are also more intuitive for the analysis of clocked systems than proof engines, since the timing of controllers is handled implicitly in the model[20][30].

The selection of the NuSMV2 model checking engine is mostly due to its excellent size reduction of models, making analysis of larger systems tractable[7], and its use in Bayazit's work showing its effectiveness[2].

To do the formal methods analysis, the system description is translated into a NuSMV model. This is done by finding all the modules needed for the unit under test by recursive descent through instantiated modules. The system's greatest common divisor period is calculated and used to provide timing for the periodic system components[20].

Next, each component is generated as follows. The component interface description is used to generate the module definition, and the behavioral requirements are translated from an ascii representation of bMTL to NuSMV code. Each behavior is parsed and each assignment and corresponding guard is added to a set for the assignment's output/state variable in the component. Then, for each output/state variable, a block case statement in NuSMV code is generated. Extra cases are generated as needed to satisfy the conflict resolution method and timing constraints of the specifications.

Figure 3.7 shows the translation of two behavioral specs into the portion of the NuSMV case statement for the Door_Command output. For a full example of a generated SMV model, refer to Appendix B, Figure B.2.

```
...
<spec type="behavior">Door_Open &amp; !(Has_Reopened) -&gt;
   &lt;1000ms,5000ms&gt;(Door_Command = close)</spec>
<spec type="behavior">Door_Reversal &amp; !(Has_Reopened) &amp;
   (Door_Command = close) -&gt; (Door_Command = open) &amp; (Has_Reopened)
   </spec>
...
```

(a) PinkySwear Specification Document

```
init(Door_Command):= stop;
next(Door_Command):= case
   !(DOORCTRL_IS_TICK): {Door_Command};
   s_1_1_bounds_timer.expired | s_1_1_bounds_timer_non_det_taken: {close};
   ((Door_Reversal & !(Has_Reopened)) & (Door_Command = close)): {open};
...
```

(b) Generated NuSMV Code

Figure 3.7: Behavioral specifications from the Door Controller, and the SMV code they are used to generate.

These generated models have non-deterministic timing and assignments that allow for a variable to take one of a set of values. NuSMV explores all paths that are permitted by the specification, meaning each valid timing and each possible value choice. This means that the safety proof applies to any system design that follows one of the paths allowed by the specification.

3.3.2 AgMon

Since PinkySwear is designed for safety critical systems, we used AgMon, a non-intrusive bus monitor designed with safety monitoring of embedded systems in mind[16]. This type of monitor is ideal for the monitoring PinkySwear performs, as the system description is based on component interfaces and behaviors at that interface. This monitoring approach is known as interface monitoring[28][26].

Interface monitoring is especially useful in systems involving smart sensors transmitting directly over the bus, and COTS components where it is most likely not possible to run a software monitor onboard the component. As more systems move towards integrating COTS components (or subcontracting the production of some system components), bus monitoring becomes an attractive option for monitoring systems for functional correctness[16].

Another reason for our selection of the AgMon is that it is a formal bounded-time Metric Temporal Logic monitor. As this is the language that PinkySwear uses to specify system behavior, translation from design to monitors is fairly straightforward. Translation to monitoring expressions simply substitutes propositions into behavioral requirements, and accounts for the latency of communication between system components, calculating the best and worst case latency based on controller periods.

Since bMTL works over boolean propositions, AgMon uses a propositionalizing layer that takes expressions over non-boolean variables and evaluates them. For example, the propositionalizing layer would evaluate the expression "Door_Command = stop" and provide the monitoring expressions with boolean proposition "Door_Command_EQ_stop". Since the grammar is restricted to deliberately allow non-boolean variables only in the θ grammar, PinkySwear uses each θ as an expression for the propositionalizer.

PinkySwear also generates monitors to validate the assumption that components do not change their output or state unless explicitly required to do so by the design specification. This means that from a monitoring perspective, if we observe an output change, then there should be a behavioral requirement that caused that action (i.e. whose guard condition has been satisfied).

```
...
<spec type="behavior">Door_Open &amp; !(Has_Reopened) -&gt;
    &lt;1000ms,5000ms&gt;(Door_Command = close)</spec>
...
```

(a) PinkySwear Specification Document

```
{...,
       Door_Command_EQ_close : "Door_Command = close",
       ...}

[...,
       "(Door_Opened -> (<0,530>(Door_Command_EQ_close)))"
       "(Door_Command_EQ_close -> ([[10,10]](Door_Command_EQ_close) ||
            <<1000,5000>>(Door_Opened)))",
       ...]
```

(b) Generated Prepositionalizers and AgMon Invariants

Figure 3.8: Behavioral specifications from the Door Controller, and the monitor expressions they are used to generate.

Figure 3.8 shows the design specification and the generated monitors. Once again, a full example set of generated monitors and propositions can be found in Appendix B, Figure B.3.

Chapter 4: Case Study

To evaluate PinkySwear we needed make sure that PinkySwear is capable of analyzing a test case embedded system. Since Pinkyswear is designed for highly distributed embedded systems with written requirements documents, our test case was a simulated elevator architecture that comes with a set of design documents that outline the system architecture, as well as component interfaces and behaviors[29].

This architecture has a corresponding simulator, and was designed as a tool for teaching safety-critical design, and thus includes both a failsafe system (emergency brake) and support in the simulator for the elevator to become unsafe and trigger said failsafe[19]. The specific implementation used in this case study was a student submission for the project, for which the author was a contributor. The elevator architecture and simulator have been used as a case study for similar types of analysis before[5][24].

We use this test case to assess three of PinkySwear's goals.

1. Can the we use PinkySwear's behavior templates to sufficiently describe real world systems?
2. Is PinkySwear effective as a development tool to detect design faults and implementation faults?
3. Is PinkySwear effective as a failsafe monitor, using correctness monitoring to guarantee the applicability of a safety proof?

To answer the first question, we need to find out if PinkySwear is sufficiently descriptive to write the specification for the test case system in the PinkySwear format. As this is a pre-existing design/architecture and a reasonable analogue of a real world system, if the design can be expressed in PinkySwear, then there is at least an indication that there are real world systems that can be represented.

To determine PinkySwear's effectiveness finding faults as a development tool, we performed a small fault injection campaign on the test case system. Two fault injection campaigns were run to test PinkySwear's effectiveness: one using design faults, and one using faults in the implementation.

Finally, to determine PinkySwear's utility as a failsafe monitor, we compared it against the simulator's built in safety failsafes. This comparison is based on the same faulty implementation from the above campaign, as simulation of the fault free elevator implementation did not trigger either failsafe.

4.1 Target System

The system used for the case study is a distributed elevator architecture. The elevator is a single car, with a left and a right door and call buttons in the car and hallways. The elevator has a drive motor with an acceleration profile that allows for high speed design (i.e. where the elevator design must slow for approach substantially before entering sensor range of the destination floor).

The architecture is a distributed model, shown in Figure 4.1, with components communicating over a CAN (Control Area Network) bus. The architecture includes controllers for the actuators (such as door motors) and smart sensors that report the physical state of the system (such as the door closed sensors). This architecture also includes call buttons, direction lanterns, and floor indicators, for the passengers to interact with the system.

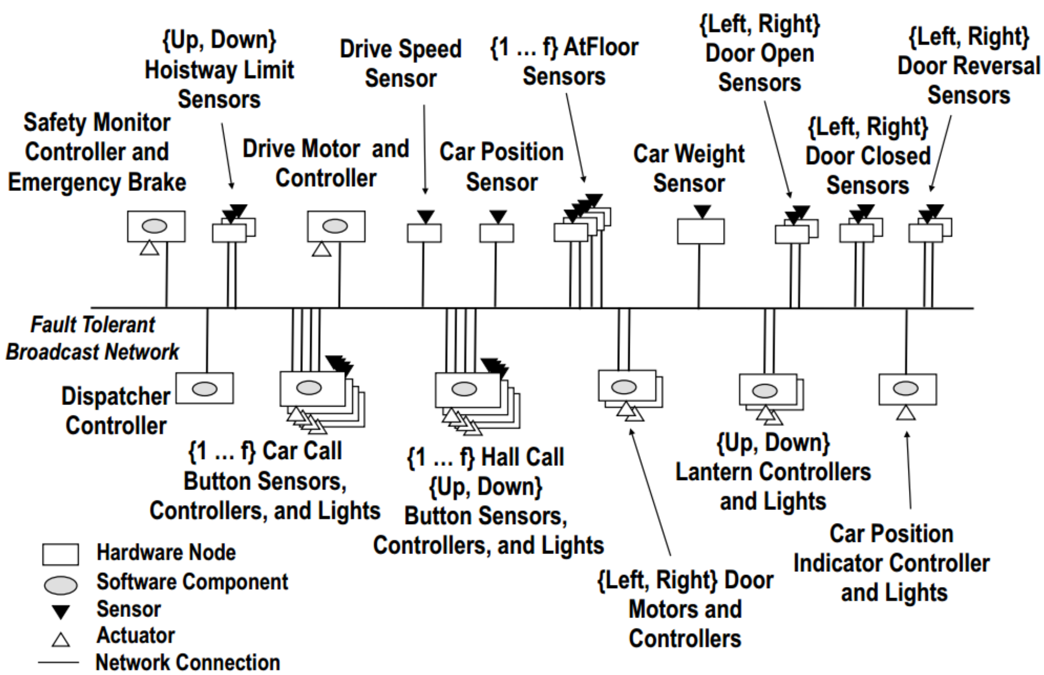

Figure 4.1: Case study elevator network architecture diagram[29]

The architecture has a corresponding 10,000-line discrete event simulator, with independent software components, kinematic profiles for actuators, and simulated passengers who exercise the use cases of the design (from pressing buttons and interacting with doors, to changing the weight of the car). The simulation includes packet level simulation of the CAN bus for communication between the controllers that accurately simulates network message delay.

The critical properties that we checked for this system were:

- The elevator never departs a floor with the doors open
- After startup, the elevator only goes to a floor after it has been called

The first property is a safety property, and derives from an analysis of physical hazards to riders in an elevator. This property is dependent upon the interaction of several of the distributed controllers in the elevator, and is more difficult to monitor, as well as more difficult to ensure through code review, than design requirements that pertain only to the behavior of a single controller.

The latter property is a liveness property, meaning that it makes eventual guarantees about the system that do not necessarily hold true in any specific bounded time. While often not as critical as safety properties, liveness is still a property of interest to system engineers; for example, ensuring that there are no deadlocks in their system.

The formalizations of these properties can be found in Section 3.2.2 and Appendix A.

We will use this system as a test case to demonstrate that the PinkySwear framework is effective both as an analysis tool and as a failsafe monitor.

4.2 Are Behavior Templates Sufficient?

The first important question is whether or not behavioral templates are sufficient to describe the elevator system.

To answer this, we first translated the design document for this system to a PinkySwear specification document using the templates given in Section 3.2.4, to make sure that the templates were adequate for the design. While the full text of this description document is too long to be included, an excerpt can be found in Appendix B.

This description also needed to encapsulate the behavior of the system accurately. To check this, we use the generated models and monitors, and compared them against the simulator. To compare the monitors against the simulation, we ran the simulator under a variety of workloads, and confirmed that there were no violations. For the model, we used the NuSMV simulation feature to manually compare the model behavior against the behavior of the elevator simulation, and confirmed that it did correctly describe the elevator behavior.

4.3 Is PinkySwear an Effective Development Tool?

To determine if PinkySwear can catch faults during system development, we performed two short fault injection campaigns. We were interested in two classes of fault: design faults and implementation faults.

Design Faults

To test if PinkySwear is capable of detecting design faults, we injected a few common design faults, as identified by [23]. Of these identified faults, process faults (2.B & 3) were out of scope, and syntax faults (1.A) are usually caught by a compiler. Coding errors (2.A) are addressed below in implementation faults. The remaining faults are outlined in Table 4.1 below, along with the results of PinkySwear analysis on an representative fault.

Fault	Results of PinkySwear analysis
Interface Fault(1.B): Attempt to access an interface that doesn't exist	The fault is caught when NuSMV attempts to read the generated model, returning a "Component.variable undefined" error message, identifying the nonexistent interface.
Functional Fault(1.C): Attempt to assign an invalid type (or out of range value) to a variable	The fault is caught when the assignment is made, triggering a "type system violation" or a "cannot assign value" error, displaying the variable and value of the faulty assignment.
Errors in Recognizing/Deploying Requirements (2.C): A clause missing in a boolean expression in a requirement	Fault causes the model checker to determine that the safety property does not hold. NuSMV provides a counterexample trace, which describes a system run in which the safety property is violated.

Table 4.1: Injected design faults and the PinkySwear analysis results.

For each of the faults in the table above, a representative fault was introduced into the design (e.g. "<spec>A -> bool_message = 3</spec>"). The design is then compiled to a model by PinkySwear, and handed over to NuSMV for analysis. NuSMV performs its analysis to detect faults in the model that could cause violations of the safety properties. Faults 1.B and 1.C are detected by NuSMV's consistency checks as it reads the model. Fault 2.C creates a functional model, but the model checker finds a trace in this model that causes a violation of the critical property.

To confirm that fault 2.C does in fact cause a safety violation, we ran the simulator with code that implemented this fault. The simulation did in fact detect safety violations when given the input pattern from the example trace found by NuSMV.

Implementation Faults

We also needed to demonstrate that PinkySwear catches implementation faults. To do this, we injected some common software faults as identified by [9].

As the elevator simulator is primarily used as a demonstration platform for teaching safety-critical design process, the code structure makes several of the faults presented unfeasible (e.g. All variable access is through function calls, which do not allow assignment, so "assign instead of compare" faults would not result in compilable code).

We injected several of the remaining faults, focusing on examples that were not immediately obvious as faulty to a user (e.g. door never closes, and the elevator doesn't move). These injected faults, and the PinkySwear monitoring results are outlined below in Table 4.2.

Fault	Total Activations	% Unmasked Activations	% Unmasked Caught by PinkySwear	% Unmasked Caught by Safety Monitors
Parens Fault: A mismatched parenthesis in a logical expression	2266	1.59%	100.00%	13.89%
Missing Statement Fault: A part of a multi-line boolean guard is missing	1750	2.74%	100.00%	6.25%
No Initialization Fault: A peripheral (one of the CAN mailboxes) is not initialized properly	6068	0.18%	100.00%	36.36%
Total	10084	0.94%	100.00%	12.63%

Table 4.2: Injected implementation faults and the PinkySwear analysis results.

As shown in the table above, most of the time these faults were masked, meaning that the faulty behavior matched the desired behavior for the design. Additionally, it is often the case that even the unmasked faulty behavior does not result in a safety violation. Fault behavior that doesn't result in safety violation is difficult to spot using safety monitoring or user observation. PinkySwear, on the other hand, catches these faults every time they exhibit unmasked behavior, about eight times more frequently than relying on failures caught by the simulator's built in safety monitor.

Additionally, because PinkySwear checks individual component for behavioral correctness, it identifies the component and behavioral requirement that was violated, which is not the case when directly monitoring the safety properties. For example, if the high-level safety property "the car never moves with the door open" is violated, it is difficult to tell which controller is at fault. On the other hand, if a behavioral requirement of the Door Controller is violated, then it is known that the fault is in the Door Controller. This can help developers find these faults faster by reducing the number of places that need to be checked.

4.4 Is PinkySwear an Effective Failsafe?

PinkySwear is also designed to be used as a failsafe monitor. Using the safety argument laid out in Section 3.1.1, PinkySwear proposes to effectively monitor safety by monitoring the design compliance of individual components.

When used as a failsafe monitor, PinkySwear relies on the successful verification of the design by model checking, thus design faults are outside the scope of this analysis. The faults that PinkySwear is designed to provide a failsafe against are implementation faults, which are described in the previous section (4.3), and transient faults.

Transient faults are temporary errors, such as a network error, bit flip due to cosmic radiation, or other short duration events. The transient error we used in this case study was internal state corruption, where the internal state of a controller changes erroneously from one mode to another (e.g. the Door Controller's internal state is corrupted from "OPEN" to "CLOSED"). In real systems, this type of fault could be caused by a number of root causes, such as a network error (providing the wrong information to a controller and making it behave incorrectly) or a bit flip corrupting the state directly.

	Unmasked Activations		Activations that Cause Safety Violations	
Fault	Total	% Caught	Total	% Caught
Parens Fault	36	100.00%	5	100.00%
Missing Statement Fault	48	100.00%	3	100.00%
No Initialization Fault	11	100.00%	4	100.00%
Transient Fault: a corruption of internal state due to some short duration event	33	90.90%	8	100.00%
Total	128	97.66%	20	100.00%

Table 4.3: Injected runtime faults and the PinkySwear failsafe detection results.

As show in Table 4.3, PinkySwear catches 97% of unmasked fault activations, and 100% of fault activations that result in safety violations. PinkySwear does not catch all unmasked fault activations, as in some instances a fault can cause behavior that is not masked (i.e., is different than the fault-free behavior of the system) but still complies with the specification. An example for the elevator might be a faulty algorithm that sometimes chooses an inefficient but still valid floor due to a corner case. These faults are not detected by PinkySwear's monitors. However, as they are still within the design specification, the safety proof guarantees that these fault activations will not cause safety violations.

For the fault activations that do cause safety violations, not only does PinkySwear detect them, but the PinkySwear approach results in faster detection time when compared to direct monitoring of safety properties. In cases where the fault causes a safety violation, the correctness monitors found violations (and therefore invalidations of the safety proof) on average ∼50 ms (5 times the period of the fastest controller) earlier in simulated operation time than violations reported by the safety monitor, as shown in

Figure 4.2.

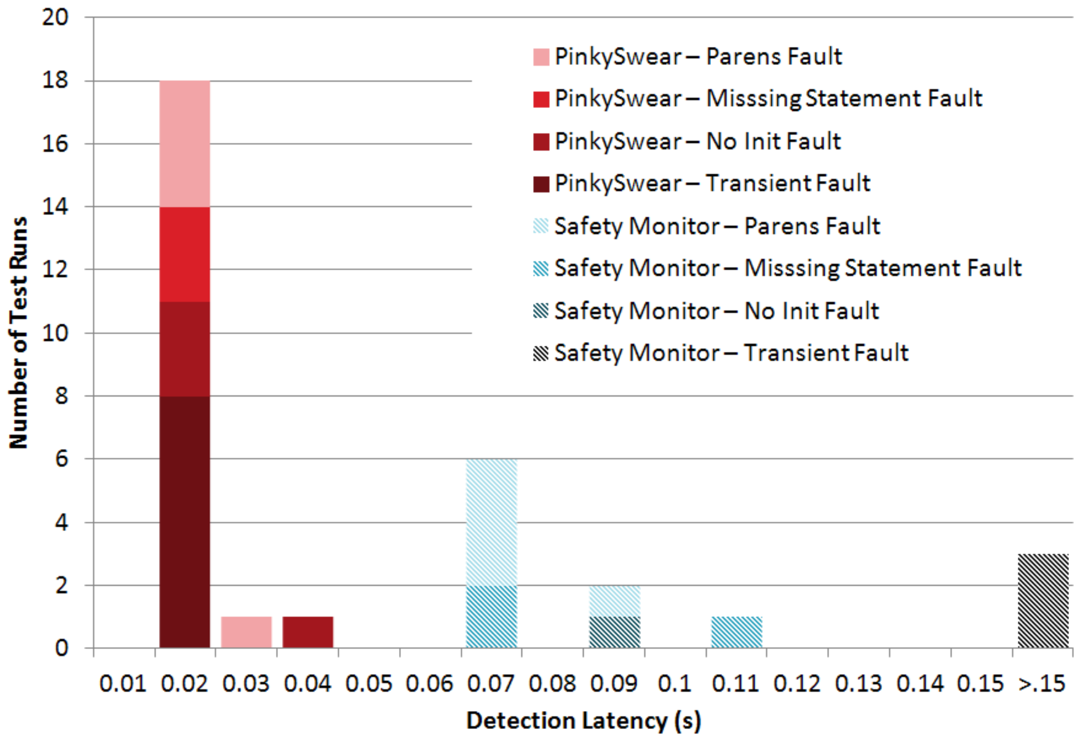

Figure 4.2: Using correctness checking results in earlier detection of fault activation.

This difference occurs due to latency from the controller behaviors to the safety critical output. This can come from physical inertia of a controlled plant, or long data paths in the system. For example, in the elevator, faulty behavior from the Door Controller takes time to move the doors enough for the system to become unsafe. By monitoring individual controller behaviors, PinkySwear can catch the activation of faults before they propagate to make the system unsafe.

4.4.1 Liveness Properties

Some system properties fall into the category of "infinite time behaviors" of the system. In the elevator, one such property might be the liveness property "while there are calls, the elevator keeps answering them." This example can be approximated using a timeout, since in most cases a bounded response time is desirable. On the other hand, there are other properties such as fairness (e.g "If calls for floor X happen infinitely often, then the elevator services floor X infinitely often") that are important to system designs and may translate less easily to timeout style monitoring, depending on the requirements of the system under analysis.

Violation of liveness properties is normally detected in the elevator simulator with a timeout set to a substantial duration (one minute) so as to ensure that the system is in fact in a dead/livelock situation, and not just responding to a situation that inherently requires a long response time. This long response time can occur during perfectly normal operation of service systems, where users may cause delays by their interactions with the system (e.g. A full load of passengers disembarking the elevator followed by a full load getting on). This means that the response time for detecting failures with a timeout must be much slower than controller periods, otherwise the risk of false positives increases.

On the other hand, PinkySwear generates liveness proofs using model checking, while the monitors check only the design compliance. Because this correctness monitoring involves properties that are all bounded time actions (in bMTL), monitoring can be used to determine if liveness properties of the system may be violated in the current trace. These correctness time bounds are usually on the order of controller periods, making it much lower latency than timeouts for detecting faults that cause liveness failures.

In the case study, there were five instances of an injected fault in the door controller causing the elevator to be stuck in a deadlock situation, where no progress was being made. In these cases, the fault was detected by the PinkySwear correctness monitors in approximately 20 ms, while the timeout took 60 of simulated operation time seconds to register that the system had deadlocked.

Chapter 5: Discussion

PinkySwear is designed to serve as a bug finding development tool and safety monitor for distributed embedded systems. This is a wide class of systems, and while design decisions were made to try and take advantage of commonalities, there are several threats to the validity of results obtained by the PinkySwear tool: the relatively limited scope of the monitor, the synchronous model of execution, and the small size of the case study.

5.1 Monitor Scope

The main threat to the validity of results obtained from PinkySwear is the limitations of monitoring in PinkySwear's interface based approach. PinkySwear uses a bus monitor to isolate the monitor from the system under test. For the most part, this design decision presents no issue, as the properties and component descriptions are defined at an interface level. However, not all interfaces are on the bus. For example, the physical components of the elevator, such as the door motor and the hoistway motor, do not provide outputs that are visible to the monitor.

In the case that the physical system model proves to be inaccurate, the system may end up in an unsafe situation despite the verification of the design and validation of the system run. For example, if the design of an adaptive cruise control mechanism can be verified to ensure that the distance to the car in front is never less than 20 feet, this property may be violated in a fault free run, by a car changing lanes in front of the test vehicle. In this case the fault was in the physical system model, and this safety violation could go undetected!

To prevent faults of this nature, either a) the state of the physical model needs to be readable by the monitor, or b) extreme care needs to be taken in the design of the physical models to account for off-nominal behavior. This latter option can be difficult in autonomy systems, such as adaptive cruise control, as there are always unexpected occurrences in real life that a developer does not expect.

On the other hand, there are two methods of making the physical state readable by the monitor. The first is by having controllers with access to that physical state broadcast it on the bus, and the second is to use a monitoring solution that can access those physical component interfaces independently, such as an on-chip monitor[13]. The relative drawbacks of these two approaches are discussed in detail in

[11], but can summarized as the expense of additional bus traffic versus concerns about isolation and inconsistencies between the SUT and monitor versions of the physical interface.

One planned extension to PinkySwear is to provide support for additional monitoring solutions, though additional exploration of current practices is needed before a specific target can be chosen.

5.2 Synchronous Model of Execution

PinkySwear uses the NuSMV model checker, which is based on a synchronous model of execution. In this model, all controllers in a system execute at the same time. That is to say, they all observe the current state of the system and then all change their individual state at the same moment. This is a simplification, as in real life processes are interleaved, executing at different times, and often don't maintain a static ordering, where one process comes strictly before another.

While this simplification makes model checking much more tractable, it does mean that the results of PinkySwear analysis could miss faults that are caused by asynchronous execution order. As an example, if two controllers are switching on and off every 200 ms, then not only will they not match 100% of the time (as one controller will switch before the other), but it may not always be the case that controller A switches before controller B.

One way to address this in PinkySwear would be the integration of the SPIN model checking engine, which supports asynchronous execution of processes. The limitation of SPIN is that it only supports Bounded Model Checking (BMC). BMC explores possible system traces in an expanding tree, which means it can only check executions of finite length. Additionally, the number of possible traces grows very fast with the depth, making deep checking of properties very time consuming. For these reasons NuSMV and SPIN provide complementary analyses of a system, detecting different classes of faults, by using different models of execution.

5.3 Small Case Study Size

The final threat to the validity of results obtained from PinkySwear is the lack of empirical evidence supporting them, stemming from the relatively small size of the case study. Since the case study only covers the one system, it is hard to make broad statements about the system's accuracy. Increasing the body of systems that have been analyzed would provide greater confidence in the correctness of the program operation.

Chapter 6: Conclusions

Both formal verification and runtime validation have strengths in detecting the faults described by their particular fault model. To make use of both of these strengths we developed PinkySwear, a unified framework for modeling and monitoring. PinkySwear uses a semi-formal partial specification language to describe system behavior and uses this specification to generate models to detect design flaws and monitors to detect adherence to this specification. It is designed to make formal methods accessible to non-experts by using simple constructs where possible, and templates of formal language expressions where not.

6.1 Contributions

PinkySwear is designed to:

- be easy to use with minimal in formal methods/languages, while still being formal enough to allow for verification and expressive enough to be usable on complex systems.
- act as a testing tool to detect both design faults and implementation faults during development.
- act as a failsafe monitor to detect failures in live systems more effectively than traditional external safety invariant monitoring.

To demonstrate these contributions, we examine a case study of a distributed elevator control system and demonstrate that this framework can be used to specify systems using simple templates, detect design faults using these generated models, and detect runtime faults using the generated monitors. Additionally, we show that performing runtime correctness checking on components using a verified system design can be an effective way to ensure system safety.

6.1.1 PinkySwear is Easy, Formal, and Sufficiently Expressive

Based on the need to provide easy to use, but still formal, descriptions for system behavior, PinkySwear extends some of the templates presented in Kane's template repository[16]. These templates make it easy to describe system behavior in formal language by presenting English language descriptions of the behavior, and allowing users to simply plug in predicates. PinkySwear's extension to these patterns is more natural for system description, as it relates closely to the design concept of behavioral requirements[18]. The formalization of these templates is presented in Sections 3.2.4 and 3.2.5

This same approach is used to help users generate safety rules, by using LTL templates to specify the formal language versions of the safety properties. This has been demonstrated to be an effective way to easily develop formal safety requirements, and is a good fit for our system as discussed in Section 3.2.2.

By applying PinkySwear to a test case system in Chapter 4, we demonstrated that PinkySwear's system descriptions are sufficiently descriptive to be able to encapsulate the behavior of an example complex distributed embedded system accurately. We also demonstrated that these specifications are sufficiently formal as to be able to be used to generate models and monitors.

6.1.2 As a Testing Tool

PinkySwear is designed to be used as a testing tool, to find design faults and implementation faults that may cause systems to become unsafe. In Section 4.3, fault injection experiments demonstrate that PinkySwear can detect injected faults that are confirmed to cause safety violations in the simulation.

6.1.3 As a Failsafe Mechanism

PinkySwear is also designed to be able to be used as a failsafe monitor, to detect potentially unsafe situations and trigger any failsafe actions. The formal argument for its soundness is presented in Section 3.1.1, and empirical evidence of its effectiveness as a monitoring solution for the case study can be found in Section 4.4

6.2 Threats to Validity of the Work

There are several threats to the validity of the work presented in this paper stemming from the case study used to test the tool. Since only a single case study was analyzed, it is possible that PinkySwear is incapable of analyzing systems in the general case. This is unlikely, as PinkySwear is based on generally applicable tools and techniques for system analysis, however a larger body of work will be needed to verify this.

It may also be the case that PinkySwear's specification pattern repository is insufficiently representative to be generally applicable. It is likely, upon further test cases, that we will find that the patterns currently supported by PinkySwear are not capable of representing all the system behaviors that we would like them to. One reason we suspect this is that PinkySwear currently supports only one of the four bounded response patterns from [16]. For example, it does not support the cancel action, which is very important in autonomy systems where planned behaviors may need to be disrupted due to new input. For example, if the driver of a vehicle steps on the brake, this event must cancel a request by an adaptive cruise control to increase engine output.

Another behavior pattern that PinkySwear does not yet support is the minimum hold time for a bounded response, which can be important in control systems to enforce stability or limits. For example, if a thermal sensor for chemical process detects an over temperature condition, cooling will need to be applied for some minimum amount of time to ensure that the vessel does cool below the threshold.

While these behaviors are not currently supported in PinkySwear, integration of additionally is possible. As systems are tested, patterns will be added to the specification repository to increase number of systems PinkySwear is able to represent.

Another threat comes from the fact that the test case is not a real world system, but rather an academic assignment. This means that PinkySwear may not be prepared to address the complexities of real world systems. While the test case is quite complex and based on industry expertise, it is a simplified model[19] and the application of PinkySwear to real systems would be required to demonstrate fully that PinkySwear can analyze them.

6.3 Future Work

Planned extensions to PinkySwear are to integrate additional tools to help patch holes through which faults might slip in the current framework. PinkySwear aims to integrate the SPIN model checking engine, which supports asynchronous execution of processes. This would provide the ability to detect faults that arise due to execution order. PinkySwear also plans to integrate additional monitoring solutions, to find faults which can't be caught through bus monitoring.

Future work also includes increasing the body of work that PinkySwear has been used to test, to provide both empirical evidence towards the accuracy of results obtained from the tool and to help improve the tool's usability by expanding the specification pattern repository.

Appendix A: A More Complicated Safety Pattern

We start with an English text safety rule

After startup, the elevator only goes to a floor after it is called

This safety rule is separated into two parts: the intent (the relation between propositions that we want to hold true), and the scope (when we want it to hold).

In this case the scope is "After Startup" and the intent is "only goes to a floor after it is called."

$$\underbrace{\text{After startup}}_{scope}\text{, the elevator }\underbrace{\text{only goes to a floor after it is called}}_{intent}$$

From here, we look up the intent in the pattern repository[10]. The pattern that fits our safety rule is Precedence, which has the intent "To describe relationships between a pair of events/states where the occurrence of the first is a necessary pre-condition for an occurrence of the second. We say that an occurrence of the second is enabled by an occurrence of the first."

Under Precedence, we now find the scope "after Q." This gives us the LTL formula for the safety rule,

```
[]!Q | <>(Q & (!P W S)),
```

for the pattern "S precedes P after Q."

Now we have to construct the propositions S, P, and Q.

When we try to define these propositions in terms of system messages we realize there is ambiguity in "it is called" and "goes to a floor." To be more specific, what we want to say is, "the elevator only goes to a specific floor k, when it is called to k." For this replication k, we will make k safety rules, just as one would if one were coding an if block.

"The elevator is called to floor 1" *precedes* "the elevator goes to floor 1" $after$ "Startup"

"The elevator is called to floor 2" *precedes* ...

We then turn these English text propositions into logical expressions, again as if we were coding. The variables in these expressions come from the interfaces defined in the system definition document. To refer to a interface a from specific module B, we use the standard dot notation $B.a$. (For the interface definition for the call module, refer to Figure B.1 in the Appendix.)

$$(HallCall1.Call||CarCall1.Call)\ precedes\ (At\_Floor = 1)\ after\ \text{"Startup"}$$

The definition of "Startup" is tricky, as there is no direct relation to a system message. In this case, since the proposition we are restricting is P, "the car goes to floor k", it makes sense to define when "startup" ends in terms of leaving the initial floor.

$$(HallCall1.Call||CarCall1.Call)\ precedes\ (At\_Floor = 1)\ after\ (At\_Floor = -1)$$

We then substitute these propositions into the LTL formula provided by the pattern, and arrive at the LTL expression:

[]!(At_Floor = -1) | <>((At_Floor = -1) & (!(At_Floor = 1) W (HallCall1.Call || CarCall1.Call)))

This LTL expression (and the k others like it) are the formal critical properties used in the PinkySwear system.

Appendix B: Test Case Artifacts

Figure B.1: XML description of the interface and behavior of the Call button in the Test Case Elevator.

```xml
<component>
   <name>Call</name>
   <period>100ms</period>
   <description>Call button for Elevator</description>
   <interface>
      <input>
         <name>Button_Pushed</name>
         <type> <bool/></type>
         <description> Is the button pushed?</description>
      </input>
      <input>
         <name>Door_Closed</name>
         <type> <bool/></type>
         <description> Is the door opened?</description>
      </input>
      <input>
         <name>At_Floor</name>
         <type><enum>{-1,1,2,3,4}</enum></type>
         <description> The floor the elevator is at</description>
      </input>
      <input>
         <name>Target_Floor</name>
         <type><enum>{-1,1,2,3,4}</enum></type>
         <description> The floor the elevator is at</description>
      </input>
      <input>
         <name>Target_Dir</name>
         <type><enum>{up,down,none}</enum></type>
         <description> The direction the elevator will go</description>
      </input>
      <output>
         <name>Call</name>
         <type default="FALSE"><bool/></type>
         <description> Is there a pending call for this floor</description>
      </output>
   </interface>
   <parameter>
      <name>STATIC_Call_Floor</name>
      <type><enum>{1,2,3,4}</enum></type>
      <description> floor the Elevator call is for</description>
   </parameter>
   <parameter>
      <name>STATIC_Call_Dir</name>
      <type><enum>{up,down,none}</enum></type>
      <description> Direction the Elevator call is for</description>
   </parameter>
   <spec type="behavior"> Button_Pushed -&gt; Call</spec>
   <spec type="behavior"> (At_Floor = STATIC_Call_Floor) &amp;
      (Target_Floor = STATIC_Call_Floor) &amp; !(Door_Closed) &amp;
      ((Target_Dir = STATIC_Call_Dir) | (STATIC_Call_Dir = none)) -&gt;
      !(Call)</spec>
</component>
```

Figure B.2: The generated SMV model for the Call button, generated using the *Priority* conflict resolution

```
MODULE Call(time,STATIC_Call_Floor,STATIC_Call_Dir,Button_Pushed,Door_Closed,
       At_Floor,Target_Floor,Target_Dir)
DEFINE CALL_IS_TICK := ((time mod 10) = 0);

VAR
       Call : boolean;
DEFINE

ASSIGN
       init(Call):= FALSE;
       next(Call):= case
              !(CALL_IS_TICK): {Call};
              Button_Pushed: {TRUE};
              ((((At_Floor = STATIC_Call_Floor) &
                (Target_Floor = STATIC_Call_Floor)) & !(Door_Closed)) &
                (((Target_Dir = STATIC_Call_Dir) | (STATIC_Call_Dir = none)))): {FALSE};
              TRUE: {Call};
       esac;
```

Figure B.3: The generated monitor expressions, and propositionalizer expressions, generated using the *Priority* conflict resolution

```
{At_Floor_EQ_STATIC_Call_Floor : "At_Floor = STATIC_Call_Floor",
       Target_Floor_EQ_STATIC_Call_Floor : "Target_Floor = STATIC_Call_Floor",
       Target_Dir_EQ_STATIC_Call_Dir : "Target_Dir = STATIC_Call_Dir",
       STATIC_Call_Dir_EQ_none : "STATIC_Call_Dir = none",
       }

["Button_Pushed -> <0,200>(Call)",
       "!Button_Pushed && (At_Floor_EQ_STATIC_Call_Floor &&
              (Target_Floor_EQ_STATIC_Call_Floor && (!(Door_Closed) &&
              (Target_Dir_EQ_STATIC_Call_Dir || STATIC_Call_Dir_EQ_none)))) ->
              <0,200>(!(Call)",
       "Call -> ([[100,100]](Call) || <<0,200>>(Button_Pushed))"
       "!(Call) -> ([[100,100]](!(Call)) || <<0,200>>(!Button_Pushed &&
              (At_Floor_EQ_STATIC_Call_Floor &&
              (Target_Floor_EQ_STATIC_Call_Floor && (!(Door_Closed) &&
              (Target_Dir_EQ_STATIC_Call_Dir || STATIC_Call_Dir_EQ_none))))))"
]
```

Bibliography

[1] Radu Banabic. *Techniques for Identifying Elusive Corner-Case Bugs in Systems Software*. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2015. 7

[2] A. A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. In *Proc. ICCAD '05*, pages 1052–1059, 2005. 5, 16

[3] Nels E Beckman. A survey of methods for preventing race conditions. 2006. 7

[4] Friedemann Bitsch. Safety patterns - the key to formal specification of safety requirements. In *Proc. SAFECOMP '01*, pages 176–189, 2001. 12

[5] J. Black and P. Koopman. Indirect control path analysis and goal coverage strategies for elaborating system safety goals in composite systems. In *Proc. PRDC '08*, pages 184–191, 2008. 19

[6] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. CAV '02*, 2002. 3, 16

[7] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Improving the Encoding of LTL Model Checking into SAT. In *Proc. VMCAI'02*, LNCS, 2002. 16

[8] Santiago Comella-Dorda, David P. Gluch, John Hudak, Grace Lewis, and Chuck Weinstock. Monitoring distributed real-time systems: A survey and future directions. Technical Report CMU/SEI-2001-TN-018, Carnegie Mellon University Software Engineering Institute, october 2001. 12

[9] J. Duraes and H. Madeira. Emulation of software faults by educated mutations at machine-code level. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 329–340, 2002. 22

[10] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proc. FMSP '98*, pages 7–15, 1998. 12, 32

[11] Alwyn Goodloe and Lee Pike. Monitoring distributed real-time systems: A survey and future directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010. 1, 3, 28

[12] D. Hamilton, R. Covington, and J. Kelly. Experiences in applying formal methods to the analysis of software and system requirements. In *Proc. WIFT '95*, pages 30–43, 1995. 4, 8

[13] 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*, 8, 2014. 27

[14] Mats P. E. Heimdahl. Experiences and lessons from the analysis of TCAS II. In *Proc. ISSTA '96*, pages 79–83, 1996. 12

[15] Constance Heitmeyer, James Kirby, Bruce Labaw, and Ramesh Bharadwaj. SCR*: a toolset for specifying and analyzing software requirements. *Proc. COMPASS '95*, pages 109–122, June 1995. 8, 12

[16] Aaron Kane and Philip Koopman. Monitor based oracles for cyber-physical system testing. *Proc. DSN 2014*, 2014. 4, 8, 12, 13, 15, 16, 17, 29, 31

[17] Moonjoo Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, Insup Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, 1999. 4

[18] Philip Koopman. *Better Embedded System Software*. Drumnadrochit Press, 2010. 30

[19] Philip Koopman. Lessons learned in teaching a complex distributed embedded system project course. *CPS-Ed*, 2013. 19, 31

[20] Leslie Lamport. Real-time model checking is really simple. In *Proc. CHARME'05*, pages 162–175, 2005. 16

[21] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M.Viswanathan. Runtime assurance based on formal specifications. In *Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999. 4

[22] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements specification for process-control systems. *IEEE Trans. Softw. Eng.*, pages 684–707, 1994. 11, 12

[23] R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. *RE '93*, pages 126–133, January 1993. 2, 22

[24] C. Martin and P. Koopman. Representing user workarounds as a component of system dependability. In *Proc. PRDC '04*, pages 353–362, 2004. 19

[25] Stefan Mitsch and André Platzer. Modelplex: Verified runtime validation of verified cyber-physical system models. In *RV '14*, pages 199–214. 2014. 1, 3, 5, 16

[26] Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, and Grigore Roşu. Hardware runtime monitoring for dependable COTS-based real-time systems. *RTSS 08*, pages 481–491, 2008. 4, 8, 17

[27] Amir Pnueli. The temporal logic of programs. In *Proc. SFCS '77*, pages 46–57, 1977. 11

[28] Amir Pnueli, Aleksandr Zaks, and Lenore Zuck. Monitoring interfaces for faults. *Electron. Notes Theor. Comput. Sci.*, pages 73–89, 2006. 9, 17

[29] C.P. Shelton and P. Koopman. Improving system dependability with functional alternatives. In *DSN '04*, pages 295–304, 2004. iv, 19, 20

[30] Hao Wang and Wendy MacCaull. Verifying real-time systems using explicit-time description methods. In *Proc. QFM '09*, pages 67–78, 2009. 16