2020

# Multilevel Runtime Verification for Safety and Security Critical Cyber Physical Systems from a Model Based Engineering Perspective

Smitha Muralidhar Gautham

# Multilevel Runtime Verification for Safety and Security Critical Cyber Physical Systems from a Model Based Engineering Perspective

A dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical and Computer Engineering
at Virginia Commonwealth University.

by
Smitha Muralidhar Gautham

Master of Science
Department of Electrical and Computer Engineering
Virginia Commonwealth University, 2010

Bachelor of Engineering
Department of Electronics and Communication Engineering
PES Institute of Technology, 2007

Director: Dr. Carl R. Elks
Associate Professor, Department of Electrical & Computer Engineering
Virginia Commonwealth University
Richmond, Virginia

# *Dedication*

*In loving memory of my grandparents*

*Nagarathnamma and Narayana Murthy*

*Rathnamma and Bhaskar Rao*

# Acknowledgement

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Carl Elks for his constant support and encouragement over the years. It's has been a privilege to be his student and I have learnt a lot from him. He has been extremely encouraging and an understanding advisor. Without his guidance and help, this dissertation would not have been possible. I would like to thank my PhD committee Dr. Barry Johnson, Dr. Robert Klenke, Dr. Ashraf Tantawy and Dr. Preetam Gosh for their support and valuable feedback on my research.

I would like to thank the Department of Energy, Division of Advanced Sensors and Instrumentation (DOE ASI) and Electric Power Research Institute (EPRI) for funding me on the SymPLe project. I would like to thank Matt Gibson of EPRI for his support and advice on the SymPLe project. I would like to express my sincere thanks to Ken Thomas of Idaho National Laboratories for his support and the opportunity to work on the Bounded Exhaustive Testing project.

I would like to thank the Institute for Software Engineering and Programming Language, University of Lubeck for their help with the TeSSLa tool. I am also honored to work with Dr. Alexander Weiss from Accemic Technologies, and Dr. Richard D Kuhn and Dr. Raghu N Kacker of National Institute of Standards and Technology (NIST). I am thankful to Dr. Alwyn Goodloe from NASA Langley Research Center for his encouragement and insights on runtime verification.

It would like to thank all by colleagues in the VCU Dependable Cyber Physical Systems Laboratory. They are the best team I could have ever had and were always willing to help out. I would like to thank the SymPLe team Athira Varma Jayakumar, Richard Hite, Christopher Deloglos and Dr. Ashraf Tantawy, I have enjoyed every discussion that we had and it is very fulfilling to see the SymPLe project grow. Athira has been a great friend and I have learnt so much from her. Her work on fault injection has been extremely useful for my research and I will cherish all the stimulating discussions that we have had. I would like to thank Dr. Rajagopala for being a great mentor, his guidance has been valuable for my research. I would like to thank Tamara Pena, Aidan Collins, Brandon Simon and Erwin Karincic for their help.

I am very lucky to have my friends Priya, Meera, Shravanthy and Mrugaya who have always been there to cheer me up and make me smile.

My family has stood by me through thick and thin. Their support and encouragement has helped me more than I can express. I would like to thank my parents Latha and Muralidhar for always being there for me. They have taken pride in every little thing I have achieved and their love and blessings has taken me far in life. They have been my role model and my guiding force. I would like to thank my in-laws Anupama and Jayasimha for all their love and blessings. They have been a source of encouragement and support.

I would like to thank my siblings Sapna and Sharath for being my best friends. They are always there rooting for me and wishing me the best. My sister-in-law Nirupama is the smartest and sweetest person I know. She is my ‘go-to person’ for any advice and has been a constant support.  I would also like to thank Hemant, Sudhanva and Ketaki for their best wishes. My little nephews and nieces Siddanth, Lasya, Shourya have inspired me in their own cute ways, always making me happy.

Lastly, I would like to thank my dear husband Atul and my two beautiful kids Advaith and Ananya for their unconditional love and encouragement. I am extremely grateful to my husband Atul for giving me the time to do my research by taking care of the kids and pepping me up with coffee so I could stay up late to finish my work. I am very lucky to have such a loving husband. A big hug to my wonderful kids Advaith and Ananya, for being so understanding and always bringing me a smile!

# Table of Contents

# List of Figures

# List of Tables

# Publications

## Published/Accepted

1. Gautham S., Jayakumar A.V., Elks C. (2020) Multilevel Runtime Security and Safety Monitoring for Cyber Physical Systems Using Model-Based Engineering. SAFECOMP 2020 Workshops. SAFECOMP 2020. Lecture Notes in Computer Science, vol 12235. Springer, Cham.
https://doi.org/10.1007/978-3-030-55583-2_14 *Tier 1*

2. Gautham, S., Bakirtzis, G., Leccadito, M.T., Klenke, R.H., Elks, C.R., 2019. A multilevel cybersecurity and safety monitor for embedded cyber-physical systems: WIP abstract, in: Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems. Presented at the ICCPS '19: ACM/IEEE 10th International Conference on Cyber-Physical Systems, ACM, Montreal Quebec Canada, pp. 320–321.
https://doi-org.proxy.library.vcu.edu/10.1145/3302509.3313321

3. Weiss, A., Gautham, S., Varma Jayakumar, A., Elks, C., Kuhn, D.R., Kacker, R.N., Preusser, T.B., 2020. Understanding and Fixing Complex Faults in Embedded Systems. IEEE Computer Magazine. (*Accepted). Impact factor 3.56*

4. Jayakumar, A.V., Gautham, S., Kuhn, R., Simons, B., Collins, A., Dirsch, T., Kacker, R., Elks, C., 2020. Systematic Software Testing of Critical Embedded Digital Devices in Nuclear Power Applications, accepted to be published in IEEE International Symposium on Software Reliability Engineering (ISSRE). *Tier 1.*

5. Elks, C. R., Bakker, T., Hite, R., Gautham, S., Venkatesh, V., & Moore, J. (2017, June). SymPLe 1131: A novel architecture solution for the realization of verifiable digital I&C systems and embedded digital devices. In 10th Int. Topical Meeting on Nuclear Plant Instrumentation, Control, and Human Machine Interface Technologies, San Francisco, California.

## To be Submitted/Under Review

1. Gautham, S., Rajagopala, A., Deloglos,C, Elks, C.,"Realizing Multilevel Runtime Monitoring for Detection of Hazards in Cyber Physical Systems" Proceedings of the 12th ACM/IEEE International Conference on Cyber-Physical Systems. Presented at the ICCPS '21 *(under review). Tier 1.*

2. S. Gautham, A. B. Rajagopala, A. Weiss and C. Elks " Survey of safety and security monitors using ARM Embedded Trace" *(to be submitted)*

3. Smitha Gautham, Athira Varma Jayakumar, Richard Hite, Christopher Deloglos, Jason Moore, Ashraf Tantawy, Matt Gibson, Carl Elks, "On the Application of Model Based Design and Assurance Methods to Nuclear Power Safety Critical Instrumentation and Control Systems" IEEE Transactions on Nuclear Science. *(to be submitted)*

4. S. Gautham, A. V. Jayakumar, A. B. Rajagopala, R. Hite, and C. R. Elks, "Finding Synergy Between Design-Time Assurance and Runtime Verification by Means of Model-Based Engineering," *(to be submitted)*

5. Co-author: The SymPLe architecture concept: Achieving Verifiable and High Integrity Instrumentation and Control Systems through Complexity Awareness and Constrained Design, International Conference on Dependable Systems and Networks. *Tier 1.*

## Published Sponsored Research Reports

• Major contributor and Co-Author - Achieving Verifiable and High Integrity Instrumentation and Control Systems through Complexity Awareness and Constrained Design. No. 15-8044. Electric Power Research Institute (EPRI), 2019. DOI: 10.2172/1547345

• Major contributor and Co-author “Realizing Verifiable I&C and Embedded Digital Devices for Nuclear Power. Second Annual Technical Report.” Dept. of Energy ASI NEET-2, 2017

• Minor contributor and Co-author “Realization of an Automated T Way Combinatorial Software Testing Approach for a Safety Critical Embedded Digital Device” US department of Energy Idaho National Labs Technical Report, INL/EXT-19-54096, June 2019. DOI: 10.2172/1606019

• Minor contributor and Co-author “Specification of a Bounded Exhaustive Testing Study for a Software-based Embedded Digital Device. ”US department of Energy Idaho National Labs Technical Report, INL/EXT-18-54045, November 2018

# List of Abbreviations

| | |
|---|---|
| CPS | Cyber Physical Systems |
| RV | Runtime Verification |
| MBD | Model Based Design |
| MBDE | Model Based Design Engineering |
| FPGA | Field Programmable Gate Array |
| DepDevOps | Design Development Operation |
| V&V | Verification & Validation |
| STPA | Systems Theoretic Process Analysis |
| FTA | Fault Tree Analysis |
| FMEA | Failure Mode and Effects Analysis |
| RTL | Register Transfer Level |
| HDL | Hardware Description Language |
| VHDL | Very High-Speed Integrated Circuit Hardware Description Language |
| IC | Integrated Circuit |
| IEC | International Electrotechnical Commission |
| FIL | FPGA-In-Loop |
| MIL | Model-In-Loop |
| SIL | Software-In-Loop |
| EC | Event Calculus |
| I2C | Inter-Integrated Circuit |
| SPI | Serial Peripheral Interface |
| UART | Universal asynchronous receiver-transmitter |
| CAN | Controller Area Network |
| DoS | Denial of Service |
| I&C | Instrumentation & Control |
| PID | Proportional Integral Derivative |
| SIL | Safety Integrity Level |
| ECU | Electronic control unit |
| ABS | Anti-lock Braking System |
| AEB | Autonomous Emergency Braking |
| SUO | System Under Observation |
| STL | Signal Temporal Logic |

| | |
|---|---|
| MTL | Metric Temporal Logic |
| LTL | Linear-time Temporal Logic |
| SRV | Stream Based Runtime Verification |
| PHA | Preliminary Hazard Analysis |
| STAMP | System-Theoretic Accident Model and Processes |
| HAZOP | Hazard and Operability Analysis |
| UCA | Unsafe Control Action |
| EDGSS | Emergency Diesel Generator Startup Sequencer |
| LS | Local Sequencer |
| GS | Global Sequencer |
| FB | Function Block |
| DV | Design Verifier |
| ISO | International Organization for Standardization |
| FI | Fault Injection |
| MC/DC | Modified Condition / Decision Coverage |
| FBD | Function Block Diagram |
| ABV | Assertion Based Verification |
| ASIC | Application-Specific Integrated Circuit |
| MBT | Model Based Testing |
| NPP | Nuclear Power Plant |
| ID | Identifier |
| NaN | Not a Number |
| ETM | Embedded Trace Macrocell |
| PTM | Program Trace Macrocell |
| ITM | Instrumentation Trace Macrocell |
| STM | System Trace Macrocell |
| DWT | Data Watchpoint and Trace |
| ELA | Embedded Logic Analyzer |
| SWO | Serial Wire Output |
| SWD | Serial Wire Debug |
| SWV | Serial Wire Viewer |
| TPIU | Trace Port Interface Unit |
| PC | Program Counter |
| ATB | AMBA Trace Bus |

| | |
|---|---|
| PFT | Program Flow Trace |
| TMC | Trace Memory Controllers |
| ETB | Embedded Trace Buffer |
| ETF | Embedded Trace FIFO |
| ETR | Embedded Trace Router |
| ETS | Embedded Trace Streamer |
| FIFO | First In First Out |
| DAP | Debug Access Port |
| JTAG | Joint Test Action Group |
| TDI | Test Data In |
| TDO | Test Data Out |
| TCLK | Test Clock |
| TMS | Test Mode Select |
| CLK | Clock |
| SWCLK | Serial Wire Clock |
| SWDIO | Serial Wire Input Output |
| IO | Input Output |
| CTI | Cross Trigger Interface |
| CTM | Cross Trigger Matrix |
| UAS | Unmanned Aerial System |
| CAPEC | Common Attack Pattern Enumeration and Classification |
| CWE | Common Weakness Enumeration |
| CVE | Common Vulnerabilities and Exposures |
| HW | Hardware |
| SW | Software |

# Abstract

## Multilevel Runtime Verification for Safety and Security Critical Cyber Physical Systems from a Model Based Engineering Perspective

By Smitha Gautham

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering with a concentration in Electrical and Computer Engineering at Virginia Commonwealth University

Major Director: Dr. Carl R Elks
Associate Professor, Department of Electrical and Computer Engineering

Advanced embedded system technology is one of the key driving forces behind the rapid growth of Cyber-Physical System (CPS) applications. CPS consists of multiple coordinating and cooperating components, which are often software-intensive and interact with each other to achieve unprecedented tasks. Such highly integrated CPSs have complex interaction failures, attack surfaces, and attack vectors that we have to protect and secure against. This dissertation advances the state-of-the-art by developing a multilevel runtime monitoring approach for safety and security critical CPSs where there are monitors at each level of processing and integration. Given that computation and data processing vulnerabilities may exist at multiple levels in an embedded CPS, it follows that solutions present at the levels where the faults or vulnerabilities originate are beneficial in timely detection of anomalies.

Further, increasing functional and architectural complexity of critical CPSs have significant safety and security operational implications. These challenges are leading to a need for new methods where there is a

continuum between design time assurance and runtime or operational assurance. Towards this end, this dissertation explores Model Based Engineering methods by which design assurance can be carried forward to the runtime domain, creating a shared responsibility for reducing the overall risk associated with the system at operation. Therefore, a synergistic combination of Verification & Validation at design time and runtime monitoring at multiple levels is beneficial in assuring safety and security of critical CPS. Furthermore, we realize our multilevel runtime monitor framework on hardware using a stream-based runtime verification language.

# Chapter 1

# Introduction

This dissertation advances the state-of-the-art in the development of Multi-level Runtime Monitoring for safety and security critical Cyber Physical Systems. This chapter describes the motivation of this dissertation research, which is the difficulty in ensuring safety and security in Cyber Physical Systems along the design, development and operational context continuum.

## 1.1 Background

As today's embedded systems become ubiquitous, we as a society are relying on their functionality more and more to perform tasks that were once labor intensive, costly, tedious, or even unattainable without the benefit of low-cost embedded computer hardware and software technology. The end result of this technology ascendance is a predominance of advanced embedded system devices that have transformed the way we interact with the world we live in. Examples of such transformative technologies are Edge Computing, Cyber Physical Systems, and Fog Computing, to name a few.

Low cost advanced embedded system and high-performance network (wired and wireless) technology are among the key driving forces behind the rapid growth of Cyber-Physical System (CPS) applications. Therefore, we are observing the rapid uptake of Cyber-Physical Systems  across a number of domains, with a potential economic impact of as much as $11.1 trillion per year globally by 2025, in different settings, including connected semi-autonomous vehicles, healthcare, energy, manufacturing, smart homes, and smart cities  [1].

CPSs are comprised of multiple coordinating and cooperating components, which are often software intensive, interacting with each other and with the physical world around us. Figure 1 shows an overview of such a CPS.

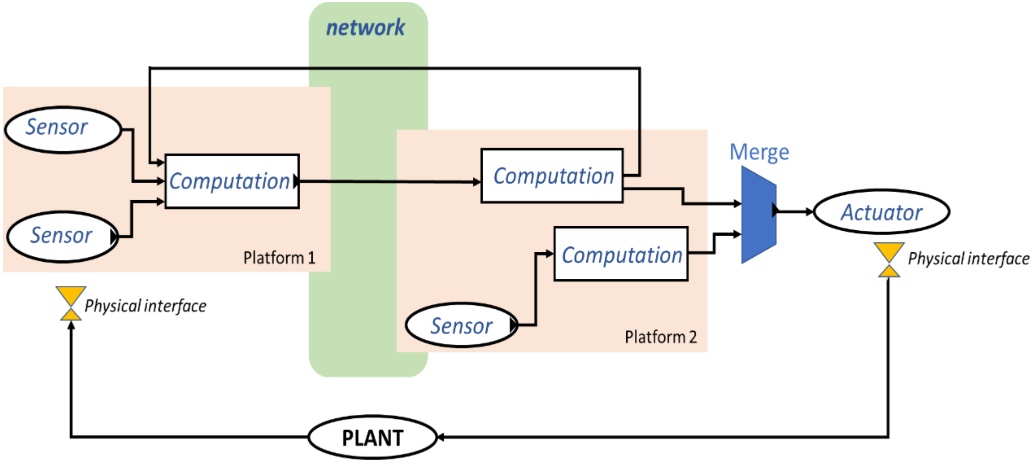


Figure 1: Cyber Physical System comprising of multiple physical devices, computational elements and network (inspired by [2]).

Cyber Physical Systems are used in a number of safety critical applications and their complexity and functionality are on the rise in such applications. Applications such as robotics, autonomous systems, telemedicine and avionics are examples of CPSs becoming more complex, incorporating new technology and with varied features. For example, a CPS such as a modern automobile has environmental and engine control systems, sensors of all types, navigation systems, cameras all working together to achieve enhanced driving experiences  and safety [3]. With the emergence of autonomous vehicle operations, the sphere of control with respect to vehicle is increased to include awareness and sensing of other traffic. As such, the functional and structural complexity of these systems are rapidly increasing, which can have significant engineering and operational impacts.  Specifically, verifying and validating highly complex systems is a known difficult problem with real world cost impacts [4]. With the exponential growth of CPS and their application not only in domestic field but also in critical applications such as avionics, nuclear plants, smart grids etc., security of the CPS is also of paramount concern. Secondly, if the safety or integrity of a CPS is a significant factor (example, energy distribution smart grid) then failure or malicious exploitation of such complex CPSs can result in far-reaching consequences to human life, environment, and financial loss.  Therefore, there is a need to ensure safety and security in increasingly sophisticated CPSs.

## 1.2 Motivation: Example Incidents and Problems

The design of highly-reliable and safety critical systems is driven by the functionality it must deliver while maintaining safety in the presence of faults.  Recent incident and accident reports from various sectors show that even when systems are designed and developed and certified to very high standards (e.g. IEC 61508 SIL4, DO-254, DO-178b), problems can occur that defeat the fault tolerance or cyber security defenses of the said systems. Ref [5] examined the role of software in five aircraft incidents or adverse effects and concluded that in "spite of rigorous design methodologies and reviews, the combination of complexity and unexpected failure mode defeated systemic fault tolerance". Ref [5] identified the critical role of flawed sensor inputs as a key determinant or trigger of "dormant" software defects in recent adverse events. Ref [5] also concluded that in some cases the software was complying with its functional requirements but still placed the aircraft in a hazardous state or contributed to an adverse event.

Ref [6] examined the occurrence of byzantine failures in aerospace systems and concluded that the occurrence of these types of insidious faults probably happen much more frequently than previously suspected. Ref [6] presents evidence of byzantine fault behavior in three different flight control systems and shows the circumstances of byzantine failure manifestation. Ref [7] discovered a byzantine failure in the Draper Labs Fault Tolerant Processor (FTP) which resulted in the quad FTP computer diverging into a "pair-wise" split voting pattern which would have been catastrophic if it had occurred in-flight.

Ref [8] reviewed fifteen catastrophic accidents, and evaluated the causative roles that software played in the accidents. Ref [8] noted that quite often the causes were a combination of software failures triggered by hardware failures and human error compounding the response. Finally, the recent Boeing 737 max Maneuvering Characteristics Augmentation System (MCAS) incidents and accidents only further establish the evidence that complex hazard scenarios and pilot interactions arising from the flawed control system software are challenging to detect during design and testing. As noted in the October 2019 National Transportation Safety Board (NTSB) report; "*pilots' responses to unintended MCAS operation were not consistent with the underlying assumptions about pilot recognition and response that were used for flight control system functional hazard assessments…*" the report further noted that MCAS started out as a modest software system and then evolved to a complex system without full understanding of the complexity and the system dependencies with respect to MCAS [9].

What these examples and survey analysis papers suggest is that even though a system has been thoroughly tested and verified during the design process, expected behavior of the system is difficult to guarantee during unusual off-nominal situations [10]. With the evolution of technology and emergence of critical autonomous systems and Machine Learning, systems today can self-learn and adapt their behavior based on the external environment. This can pose new challenges to Verification and Validation (V&V)

assurance performed at design time and necessitates a means of ensuring that the system requirements and design assurance are carried further down to the implementation stages. We assert that this need may be fulfilled by having runtime monitors that observe system behavior and provide assurance of safety and security without interfering with the system under observation [11]. Therefore, using both comprehensive design assurance methods and run time verification may be beneficial to achieve higher levels of safety and security of the system.

## 1.3 Runtime Verification for CPS

Runtime Verification (RV) is one of the promising techniques for immediate detection of infected states and hazards. It is a mature engineering domain with a vast literature base that has been successfully used in various safety critical industries [12] [13, p. 2] [14]. As stated in [15], runtime verification can be defined as:

"*Runtime verification is the discipline of computer science/engineering that deals with the study, development, and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property."*

Runtime verification sits in between traditional dynamic testing methods and formal exhaustive proof techniques (theorem proving). It is typically classified as a *lightweight* formal verification method. Classical verification techniques such as model checking, dynamic and static testing are employed to provide design assurance. This is complemented by formal, lightweight runtime verification methods. Unlike a formal proof, runtime verification is not complete or exhaustive, but it provides more guarantees of correctness than dynamic testing.

Runtime verification is often referred to as runtime monitoring, trace analysis etc. Checking a system for correctness of a property is referred to as verification, while monitoring only implies observation of the behavior of a system [16]. In this dissertation, the term monitoring is used for both observation and correctness checking. Therefore, runtime verification and runtime monitoring are used synonymously.

At very broad stance, the monitor observes the execution behavior of a target system during runtime while making as few as possible assumptions about the trustworthiness or proper functioning of the monitored system [10] [16] [15] [17]. Referring to Figure 2, a monitor is concerned with the detection of violations (or satisfactions) of properties (e.g. safety, security, functional, timeliness etc.) by analyzing the trace of a system [18]. A trace is comprised of observed variables, memory patterns, an execution control and data flow, protocol sequences, etc. When a violation is observed by a runtime monitor, it typically does not influence or change the program's execution. Hence, it does not repair the observed violation but forms the basis for mitigation of observed problems.

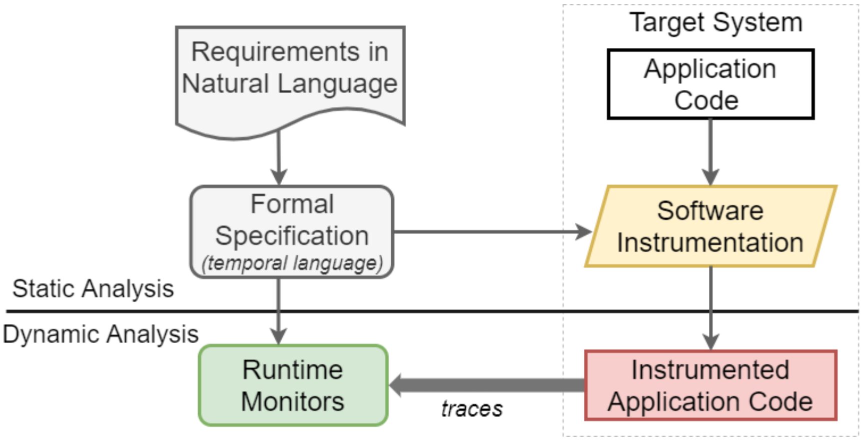


Figure 2: Runtime verification framework (inspired by [19]).

To decide on correctness, a monitor needs a reference of acceptable or correct behavior to compare the extracted trace information with. This *"reference of acceptable behavior"* is often derived from the natural language requirements of the system and then translated into temporal logic formulas– a type of logic that allows reasoning over sequences and time (as seen in Figure 2). The temporal logic formulas reside in the monitor. Execution trace information (i.e. states function variables, decision predicates, etc.) is extracted directly from the target system and forwarded to the monitor where temporal logic expressions are elaborated with the trace data for an on-the-fly validation of system behavior. For example, we can monitor discrete representations of continuous time variables such as sensor readings, system states, and application level variables. In such cases, the monitor uses traces of a set of information states from the CPS to check if the system is compliant with the "*reference of acceptable behavior"* (e.g. a safety property or checking assertion). Acceptable behavior implies there are checking conditions or detection predicates to determine if the system is demonstrating acceptable behavior. These checking conditions are called *specifications* which can sometimes be synthesized on a hardware platform as seen in Figure 3.

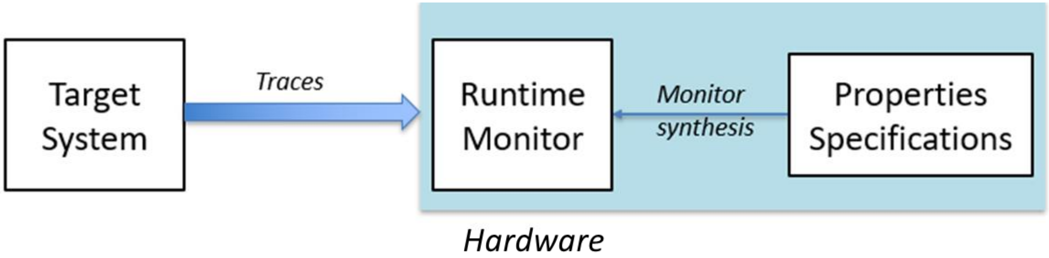


Figure 3: Overview of runtime verification.

## 1.4 Integration of Design and Operational Phases to achieve Dependability: DepDevOps

While subsections 1.2 and 1.3 discussed the need to augment design time verification with runtime monitoring, this subsection emphasizes the synergy between them. Design time verification helps provide an understanding of the system, which in turn helps develop the properties to be monitored at runtime. So, without design time verification, we would not have a clear understanding of what to monitor. On the other hand, many assumptions made at design time may not hold true at run time making assurances based on only design time verification weak. This motivates the need to view design time and runtime synergistically.

This is particularly important as CPSs are evolving towards software intensive systems where functionality, integration, and operations of a given system are largely governed by its complex software interactions. Although software testing methods and practices have undergone tremendous progress over the past 20 years, the evolving nature of software intensive CPSs can create layers of unforeseen failure modes and complex attack surfaces. As we have seen in recent years, design assurance and certification methods may not be sufficient to uncover all design flaws or scenarios that lead to hazardous operation. This is most evident in the Boeing 737-max incidents where the loss of airspeed sensors resulted in unexpected and unanticipated vehicle flight behavior [2]. Such challenges (among others) are emerging drivers for new methods where there is a continuum between design time assurance and runtime or operational assurance. That is, design assurance cannot stand on its own and needs safety and security to be carried forward to the runtime domain creating a shared responsibility for reducing the overall risk associated with the system at operation. On the other hand, runtime verification needs the knowledge gained from design assurance methods to formulate critical properties we want to monitor at runtime. This is the basis of a *"reference of acceptable behavior"* for a monitor. These new methods are sometimes referred to as Dependable Development Operations Continuum (DepDevOps) [20], [21]. The DevOps continuum strives to bridge the gap between the design and operational phases, reduce cost without compromising safety.

### 1.4.1 A Conceptual View of Integrating Design Development and Runtime Verification for CPS

The development process typically begins by formulating the design based on an initial set of requirements. The design is then verified by testing and formal property proving at the unit level, integration level and system level. During the course of verification and design development, we find missing or unclear requirements which get redefined as the design stabilizes during this first phase. This iterative process, as shown in

Figure 4, ensures that we have a complete set of finalized requirements and a verified design. Runtime monitoring properties are derived from these finalized requirements. The iterative process of finalizing requirements and the design is important to ensure that all the critical safety and security properties are captured by the monitor specifications. Further, employing systematic hazard analysis methods such as System Theoretic Process Analysis (STPA) and Hazard and Consequence Analysis (HAZCAD), help analyze the failure modes in a design and refine the requirements to include design considerations to avoid hazards or loss scenarios. Loss scenarios are failures in a system that that can lead to loss of life and property.

The finalized requirements encompass the knowledge gained from each verification step and inform us on critical elements and interactions in a system, which when failed, could lead to a hazard. This knowledge gained during the verification stages help us formulate informed runtime monitor specifications.

In phase II, the runtime monitor is seamlessly integrated with the design. This integration allows us to determine the efficacy of the monitors in detection of anomalies in the CPS for which they were designed. Fault injection campaigns and cyber-attack injection performed on a CPS model can violate a safety property of a system. This violation should be detected by the runtime monitor.

A key contribution of this thesis is the synergetic combination of phase I and phase II which provide tighter integration of design and operational phases of a system. We explore this synergy from a Model-Based Engineering (MBE) perspective.

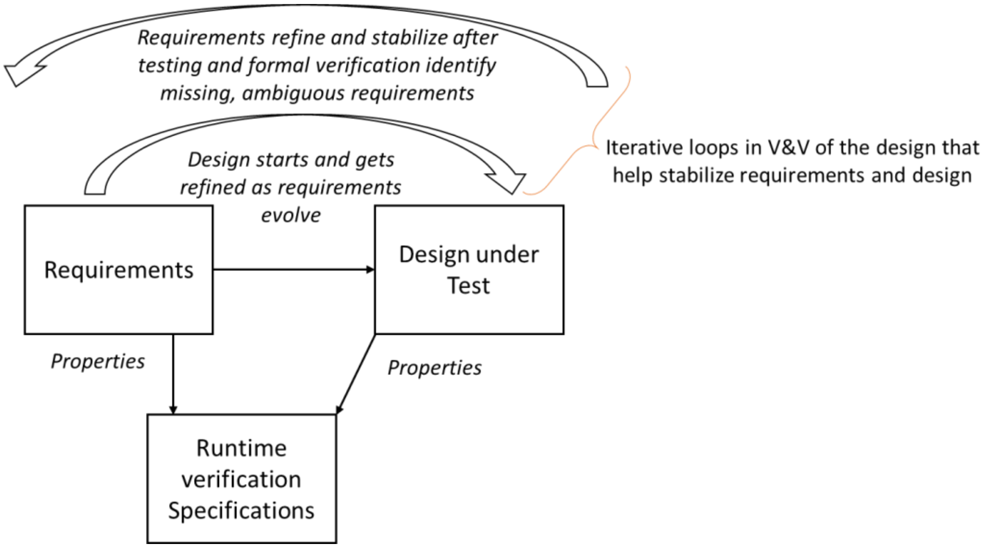

Requirements refine and stabilize after testing and formal verification identify missing, ambiguous requirements
Design starts and gets refined as requirements evolve
Iterative loops in V&V of the design that help stabilize requirements and design
Requirements
Design under Test
Properties
Properties
Runtime verification Specifications

Figure 4: Phase I – Design: Synergy between Design and Runtime Verification Specifications.

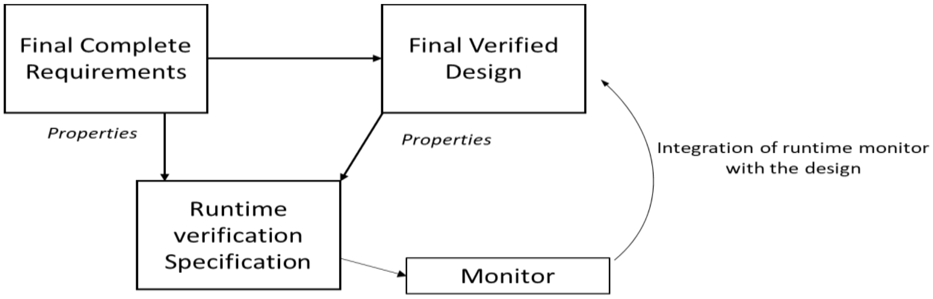

Final Complete Requirements
Final Verified Design
Properties
Properties
Integration of runtime monitor with the design
Runtime verification Specification
Monitor

Figure 5: Phase II – Runtime: Synergy between Design and Runtime Verification Specifications.

# 1.5 Critical Issues

## 1.5.1 Observation of target system

The key precondition for this runtime verification approach is the observability of the system operation such as system states, data and control flow information etc. necessary to access the behavior of the system at runtime, at time intervals of interest. The observability can be related to physical variables or computed states of a system as explained below:

1. An important challenge to runtime monitoring is observability of the value of a physical quantity of the system or its environment that a monitor needs, in order to be able to verify a correctness property for a comprehensive set of scenarios. For example, we can have a simple property "the maximum acceleration of a vehicle should be less than $a_{MAX}$ $m/s^2$", which we can verify with data from an accelerometer (directly) or a vehicle velocity sensor (by differentiating velocity). However, while going up a steep hill or incline, the vehicle should not be able to accelerate as much, let us say acceleration should not be more than $0.5a_{MAX.}$ We cannot change the property to $0.5a_{MAX}$, as on a level road it will produce a false positive. However, if we stick to $a_{MAX}$ this monitor could produce a false negative as an acceleration of say $0.8a_{MAX}$ would not cause the property to be falsified even though such a high acceleration is impossible on a steep hill or incline.

   Therefore, to have a better monitoring condition, the maximum acceleration of the vehicle should be a made a function of the incline of the road. However, if the vehicle does not have a level sensor this quantity is not observable. Unless this level sensor is physically incorporated in the car, we cannot refine the monitoring property due to lack of observability and are confined to a property which does not completely account for all scenarios.  This scenario also emphasizes that integrating such runtime considerations early, during the design phases help us identify deficiencies in the system that need to addressed while implementing runtime monitors.

2. Another observability issue deals with processors where internal variables should be extracted with minimal or no intrusion to the processor in order for a monitor to verify its functionality. For example, the controller of a complex chemical plant may have 5 inputs and 2 outputs that are observable and can be used to formulate a correctness property. But the evaluation of the output may be a complex function of the time history of these 5 inputs and involve calculation of 10 other internal variables/states that are not visible from outside this controller. However, this problem is in principle solvable, as the internal states can be obtained by software instrumentation with its known limitations, namely overly intrusiveness to program execution.  A good monitor should be minimally intrusive to the running program, have low communication overhead, should not have any interference with the program behavior, and must be space efficient (no need to store vast amount of data) and time efficient (no exponential growth in time to form a decision).

### 1.5.2 Safety and Security – How to do both, if possible?

A CPS is often comprised of numerous integrated components and subsystems interacting and communicating with each other to satisfy system (plant) goals. These goals are often related to the functional performance, safety and security of the service a CPS is providing, for example, an automobile cruise control will always disengage when the brake is applied. With the functional safety applications, failure due to a *cyberattack* can lead to situations where the attack can enable a hazardous situation which could affect safety. Therefore, it is important for the runtime monitor to address both safety and security requirements of the system.

In the context of safety critical systems, Cyber Security can be viewed as an extension or expansion of the standard physical threat models (faults, failures) for ensuring system safety [22]. The types of cyber threats and exploits for cyber physical systems are well documented and surveyed in literature [23]. Nonetheless, assuring global safety and security together is a challenging problem as they may be in conflict with each other during certain conditions. For example, if a given security property detects that a cyber intrusion is present, it may mitigate this issue by blocking an input temporarily. In doing so, it may compromise the ability of the safety property to determine if the system is safe during the period of time the input is blocked. In this case, the system is secure but potentially unsafe. However, if we take the somewhat tapered viewpoint that deliberate cyber-attacks that violate safety conditions are of primary concern, then RV provides coverage of security in a limited sense. Moreover, the temporary blocking of input discussed above, is due to the mitigation strategy adopted to counter this attack and not a problem with the monitoring scheme. As such, this dissertation is focused on runtime monitoring for detection of safety and security violations and mitigation strategies are beyond the scope of our work.

Another issue is whether RV or monitoring can distinguish between attacks and faults. In fact, there are classes of cyber-attacks that are indistinguishable from physical faults from an external point of observation [24]. However, in these cases RV can provide timely trace information to offline diagnostics or forensics to determine if the system misbehavior was due to a fault or cyber event. The critical proposition moving forward is to investigate how RV can accommodate both security and safety checking.

### 1.5.3 The Specification of Safe and Secure – Where does it come from?

As discussed above the violations of a safety and security condition may have a common origin: design faults, physical faults, or cyber-attacks that change a combination of system states or variables which could lead to hazardous conditions and safety violations. Consider an autonomous car, where a sharp turn at high speeds can cause skidding or toppling. To assure safety of the car, there should be specific bounds for the steering wheel angle for a given speed of the car. More importantly, at design time such system requirements should be verified (either formally or by testing or both). Design time V&V can provide significant insight into not only what should be monitored, but also how it should be monitored. Nonetheless, most RV frameworks to date gloss over on how design time V&V informs the RV strategies.  It is just assumed that high level safety and security requirements can be translated into monitorable properties and checking conditions, which is not realistic for CPSs. Based on our early work, we noted that there is an interplay between requirements, specifications and design that is rather non-intuitive [25]. High level requirements are often encoded in the design at the lower levels of abstraction (either as hardware or software). It is important to understand the decomposition high-level requirements in lower-level implementations of a design so that we have a complete system's perspective while formulating runtime monitor properties.

Therefore, part of the dissertation is to derive the monitoring properties with System Theoretic Process Analysis (STPA), then investigate the use of Model-Based Design and Engineering (MBDE) to verify these monitor designs before finally implementing them on hardware. This enhances the effectiveness of monitorable specifications and makes them more comprehensive.

## 1.6 Need for Multilevel Monitoring of the Target System

Modern embedded digital devices have evolved to the point where all types of heterogeneous processing and data mobility reside within a single platform or chip from low-level onboard sensor pre-processors to dedicated network communication cores. The integration of customizable system on a chip technology and flexible communication enables tight integration with the physical world. For example, physical components such as sensors and actuators, embedded subsystems and the network to communicate among the components are tightly integrated. The CPS architected from this technology are increasingly vulnerable to design flaws, software flaws, and security threats at multiple levels that span both hardware and software implementations. These multiple levels of integration in such complex systems expose attack surfaces and can potentially be susceptible to attack vectors. Given that computation and data processing vulnerabilities may exist at multiple levels in embedded CPS, it follows that solutions should be present at the levels where the faults or vulnerabilities originate. We assert that a viable approach to this problem is to employ runtime security and safety monitoring at these various levels of processing and integration. Such an approach is beneficial for timely detection of attacks (or failures) and lessen the burden of monitoring overhead. In this dissertation, we have three levels of monitoring as shown in Figure 6:

1) Sensor or Data level - The sensors and actuators constantly interact with the outside environment and the data monitor verifies the integrity of data coming from these devices. The data monitor may also verify configuration modes of the sensors to ensure they are consistent with the given application usage.
2) Computational level - The correct functionality of the computational elements, typically a controller, in a CPS is critical to avoid incorrect actions by the system leading to hazards. We then monitor the input-output relationship of a controller using a functional monitor. Additionally, some attacks may specifically alter the execution sequence of a program on the controller that can be detected by verifying the instruction traces using an execution monitor.
3) Communication Level - Sensors, actuators and computational units in a CPS use communication protocols such as UART, I2C and buses such as CAN.  The flow of information, time delay, etc. can be monitored by a network monitor to detect faults/attacks on communication.

In this dissertation, we demonstrate the need for multilevel monitors for effective and comprehensive detection and isolation of attacks by performing data attack and fault injection on a CPS model.

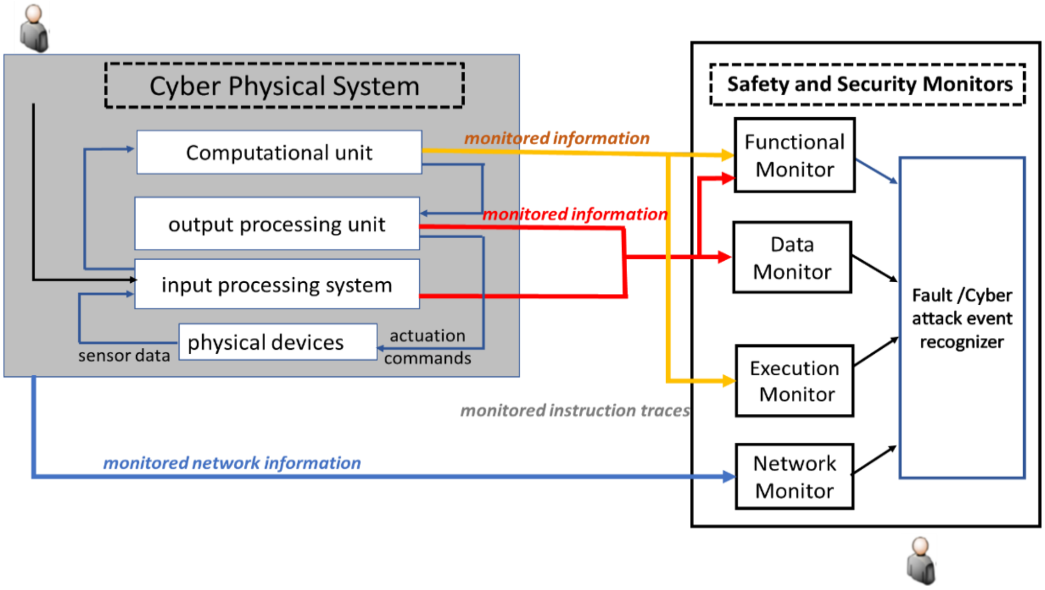


Figure 6: Multilevel monitoring of a CPS.

## 1.7 Problem Statement

A key problem in CPS is that as systems become more complex, failures and exploits are more likely to occur despite rigorous design time Verification & Validation (V&V). The root causes of these failures are very often traced back to poorly understood interdependencies between computer control systems and physical structures or unknown or unfamiliar failure modes that result from the physical and computing relationships [26]. There is a need to ensure that the critical properties still hold true and are consistent at runtime. The gap between design time safety assurance and runtime behavior needs to be addressed. Due to increasing complexity of CPS, safety and security cannot be an afterthought, but has to be considered throughout the design development. *The integration of design and runtime verification is critical to achieve this.*

Most runtime monitoring frameworks for CPSs emphasize the “how to” aspects of realizing monitors in a CPS architecture. This means that methodologies are mostly focused on how to instrument a system, assuming the “what” to monitor is a given. *A comprehensive methodology to determine critical aspects of a design to monitor at runtime needs to be determined.*

Additionally, “where to monitor” and the context of the monitoring need to be addressed. Furthermore, monitoring solutions with minimum intrusiveness is necessary. *Understanding and demonstrating the need for multilevel monitoring and use of non-intrusive monitoring methods is critical to achieve this.* This dissertation attempts to answer the key challenges mentioned above.

## 1.8 Goal and Objectives

The overarching goal of this dissertation is to demonstrate that a *synergistic combination of V&V at design time and runtime monitoring at multiple levels is beneficial in assuring safety of a system as well as ensure its security against vulnerabilities or attacks at multiple levels*.

Towards this goal, we present a Model Based Design (MBD) approach using MathWorks Simulink tools to verify the CPS system by initially performing testing, static verification and formal verification before implementing it on the hardware system. Thereafter, a novel multilevel monitor architecture is used to verify critical conditions at run time. Runtime monitoring properties are derived based on hazard analysis on the system and design time verification. Such monitors observe the system behavior by analyzing streams of information coming from the target system with no or minimal intrusion with its working. Further, we implement the monitors on hardware and verify the efficacy of the monitors in detecting faults/attacks. This approach could potentially improve the safety and security of a CPS.

## 1.9 Contributions and Scope of this Research

This work uniquely combines the benefits of design verification and runtime monitors to ensure safety and security. Our contributions are:

- Investigating and characterizing the synergy between the V&V process at design time and creation of effective runtime verification monitors.
- Development and realization of runtime monitoring framework observing the operation of a target CPS at multiple levels.
- Hardware implementation of the monitors using a stream-based monitoring language.

The scope of this work is in developing and demonstrating a methodology of combining design time V&V and runtime monitoring at multiple levels that can effectively provide protection to a CPS. This work helps bridge the gap between design time assurance and runtime behavior of the system.

## 1.10 Roadmap of Research

This dissertation starts with Chapter 1 that introduces the critical issues in runtime monitoring. It specifically makes a case for the need to synergistically combine design time V&V and runtime monitoring as well as benefits of monitors at multiple levels. Next, Chapter 2 provides a comprehensive literature review of design time V&V and runtime monitors to identify the gaps in this research.

Thereafter, Chapter 3 describes the development of the multilevel monitoring framework and various ways of monitor organization. This chapter presents a comprehensive workflow to design runtime monitors beginning with identifying critical properties to monitor at runtime, placement of monitors, formulating monitor specification and implementation of monitors. Practical consideration for implementing runtime monitors such as data observability, expressiveness of specification by the RV language and executable monitors are discussed. Other important concepts such as monitor completeness and monitor correctness are briefly presented.

Chapter 4 explores the synergy between design assurance and RV using Model based Engineering techniques.  A detailed case study is presented of the verification of an Emergency Diesel generator Start Up Sequencer implemented on a Field Programmable Gate Array (FPGA) overlay architecture, where runtime monitoring properties are derived from each verification stage.   Further the runtime monitor is implemented on a FPGA using a stream-based verification language. In summary, this chapter attempts to describe a methodology that can answer *the "what" to monitor and "where" to monitor questions from a MBE perspective.*

Chapter 5 demonstrates the need for multilevel monitoring in a CPS by injecting faults/attacks at multiple levels on a CPS model. It explains why having localized monitors is beneficial in timely detection of faults/attacks. It clearly shows that a single monitor cannot solve the in-time hazard detection problem. In summary, this chapter answers *the "where" to monitor question*.

Chapter 6 discusses the issues of observability and extraction of data for monitoring at runtime. It gives an in-depth discussion of the ARM Coresight debug and trace capability that can be used to extract data and instruction traces with minimum or no intrusion to the target CPS. It presents design choices while engineering runtime monitors that use the ARM trace capability, along with the challenges in using embedded trace. In summary, this chapter answers *the "how" to monitor with minimum intrusiveness question.*

Chapter 7 presents the concepts of STPA hazard analysis and how this help the design of monitoring properties. An Autonomous Emergency Braking (AEB) system is used to demonstrate the need for multilevel monitoring. Runtime monitors are initially implemented using Simulink library blocks. Further, monitors are implemented using a stream-based verification language called TeSSLa to verify the data, functional and network integrity of the AEB system. The TeSSLa runtime monitors are integrated into the design to ensure that they are able to detect any anomalies in the CPS for which they were designed. Use of ARM processors, coresight capability to extract data traces is explained. In summary, this chapter uses an AEB system to demonstrate *the use of the solutions to "what, where and how to monitor*" questions to develop a comprehensive runtime monitoring solution and implement the monitors on hardware.

Finally, Chapter 8 summarizes and draws a conclusion of the work performed in this dissertation regarding synergetic use of design and runtime verification, multilevel monitoring, use of ARM coresight trace capability to achieve safety and security of a CPS.

# Chapter 2

# Related Work

The verification and validation of a design is critical in identifying design flaws and other safety issues in a CPS. However, there is no guarantee that a fully verified system will function as expected at runtime. Therefore, runtime monitors are especially employed in safety systems, during the operation phase, to complement the V&V efforts performed during design development.

On the other hand, it is not possible to verify every property at runtime. Hence, it is valuable to use design time V&V to understand the CPS vulnerabilities, that guide us to formulate appropriate runtime monitoring conditions. There are a number of challenges and design considerations while engineering a runtime monitor. These are illustrated in the surveys [27], [11] and [3]. Research on how design-time assurance can be carried to runtime is one of the important research areas pointed by the survey [28].

Therefore, a wholistic view of design time and runtime monitoring is required and the two need to work synergistically. However, such an approach is largely unexplored with the exception of a few instances in [29] [19] where this issue has been touched upon.

This dissertation proposes to address this gap by comprehensively studying the combination of design time V&V and runtime verification. Towards this end, the literature review in this section summarizes some of the key work in design time V&V and runtime monitors. Further, we review literature that use runtime monitors in CPSs to ensure safety and security.

## 2.1 Verification and Validation(V&V) of Design

Assurance in design is achieved by comprehensive Verification and Validation methods. As, systems are becoming more complex, traditional design assurance methods with manual coding are becoming difficult. Safety critical industries such as avionics, automobiles and nuclear industry have stringent requirements that the design should comply with standards such as IEC 61508, DO 254, DO 178, ISO 26262. Bidirectional traceability between requirements, design, test cases and formal proofs are some of the many requirements for compliance to these standards. In order to meet these stringent design challenges, V&V and regulatory requirements of safety critical systems, Model Based Engineering (MBE) is widely used. MBE is used for the design and verification of both non-safety and safety critical systems in aerospace, power and automotive sectors [30] [31] [32]. Research on model-based V&V is presented in [33] [34] [35]. Ref [33] presents a verification workflow to designing and verifying medical device software along with the artifacts created in the process to support certification. Ref [36] reports how MBE tools can be used in the aviation industry and recommend areas where the certification guidelines can be improved to reduce failures. In addition, [37] discusses the benefits of V&V technology on avionics systems in reducing development costs and time.

These references indicate the use of MBE in design and verification of safety critical systems in various application domains. Additionally, the structural verification activities using MBE tools serves as an ideal platform to explore synergy between design time and runtime verification. Therefore, we use MBE extensively in this dissertation.

## 2.2 Synergy between design and runtime verification

In this dissertation, we explore how V&V activities in a MBDE environment can help us formulate effective runtime monitoring properties that cover critical design elements in a system. We also explore hazard analysis methods to identify monitoring properties.

### 2.2.1 Design Verification guides runtime monitor design

Ref [3] discusses the importance of considering security parameters in the design phase of a CPS. They explore the challenges in implementing security in CPS and the importance of runtime security monitoring considerations during design development. Ref [38] discusses the benefits in collaboration of design time and runtime verification to increase security in embedded systems. They discuss how the security constraints considered at design time can be monitored at runtime. Combining design time testing and runtime monitors is explained in Ref [39] with the example of an Advanced Driver Assistance System (ADAS). The test cases formulated at design-time are used again at runtime to verify the system by the runtime monitor to ensure that the system is within its safe behavioral requirements. Design time and runtime considerations to ensure safety and security in CPSs and Internet of Things (IOT) is discussed in [40]. Tool support to perform both testing and runtime verification is discussed in Ref [41].

Test driven development where model-based testing guides design development and the benefits of combining testing with static and runtime verification is explained in Ref [42].

In our work, we present our findings on uncovering synergies, exploiting design time artifacts to inform the design and implementation of runtime verification monitors for a representative safety critical system. Specifically, we differ from prior work as we use Model Based Engineering (MBE) tools to guide us on runtime monitoring properties and placement of monitors.

### 2.2.2. STPA analysis guides runtime monitor design

STPA to perform hazard analysis of safety critical systems was proposed by [43] and has been used extensively in literature in the field of avionics and automotive applications to study unsafe interactions among system components and how such interactions can result in UCAs, eventually leading to system failures [23]. Further, STPA has been extended to include security considerations in STPA-Sec and the dependency between safety and security, identification of mitigation strategies using STPA are discussed in [44]. The UCAs identified using STPA analysis of a CPS design to engineer effective runtime monitors to ensure safety and security during operational phase is an important and emerging research area. This approach is explored in [45] where STPA hazard analysis is performed on an Artificial Pancreas System to design a context aware safety monitor and in [46], STPA is used to design monitors for robotic surgery.

Furthermore, bridging the gap between design assurance and runtime safety and security is an important aspect of dependable DevOps continuum for CPS [21, p.].

Our contributions include using STPA hazard analysis to derive multilevel runtime monitoring properties as well as guide the placement of such runtime monitors in a CPS.

## 2.3 Runtime Monitors

With the growth in use of CPSs in a numerous safety critical applications, runtime verification of such systems is becoming an essential and important topic of research [47] [48] [10].

Ref [27] surveys all the challenges in runtime verification such as lack of expressive verification languages, monitor placement, limited observability of data to name a few. Ref [49] presents a taxonomy to classify runtime verification tools based on the type of specifications, deployment, intrusiveness to the system under observation etc. They present an extensive set of design choices to make while engineering a runtime monitor. Ref [11] present a number of challenges in runtime verification of safety critical systems. Some of the key challenges that are presented are deriving monitor properties from system requirements, limited observability of system states, lack of traceability from system requirements to the monitor code, intrusiveness of the monitor with the system under observation, correctness of the monitor code etc.

Ref [50] , [51] perform runtime monitoring of an autonomous research vehicle by passively observing the system without instrumentation of the code. The bus monitor architecture that they implement is as proposed by [35], where the external monitor silently receives messages over a system bus without perturbing it. With the limited information available on the bus, the runtime monitor verifies safe system behavior. Ref [17] provides a comprehensive survey for monitoring distributed real time systems and provides three monitor architectures namely BUS monitor, single-process monitor and distributed-process monitor. Ref [52] integrates run time monitoring in an automotive development workflow and explores using this in autonomous systems. Ref [53] uses safety guards that are inbuilt runtime enforcers that ensure that the system satisfies predefined properties even under malicious attack. Ref [54] present a non-intrusive monitoring approach for multi-core processors based on the execution traces received by the processors. Ref [13, p. 2] presents a real-time and non-obtrusive runtime monitoring framework called R2U2 for Unmanned Aerial Systems. R2U2 performs property monitoring by using runtime observer pairs for linear and metric temporal logic and uses Bayesian networks to detect security threats. Ref [55] shows how anomalies can be detected by observing sequence of data streams from a system.

Further, runtime monitoring using embedded hardware trace from the processors helps achieve non-intrusive monitoring. Ref [56] uses ARM processor’s execution traces to detect code reuse attacks. Ref [57] uses ARM’s embedded trace to detect double free attack on the processor. Ref [58] discusses detection of return-oriented and jump-oriented attacks based on PTM trace from the ARM processor. Ref [59] presents an online control-flow and data-flow error detection mechanism using ARM traces.

A three-layer CPS architecture is proposed in [60] comprising of transport layer, control layer and execution layer and attacks that can occur at each of these layers is surveyed. Since vulnerabilities can exist at multiple levels in a CPS, having monitors at the place of origin would be beneficial for faster detection of attacks/faults.  Prior work also recognizes the need for multiple monitors at runtime for CPSs.

A new development platform called RV-ECU is proposed in [61] where there are multiple monitors, local and global, in a vehicle to ensure. The local monitors are present at the ECU integrated within an existing control unit (e.g., the power steering ECU) to prevent it from taking unsafe actions. Another global monitor is present observing global traffic on the bus.

HECAD (Hierarchical Embedded Cyber Attack Detector) is specialized form of multilevel monitoring that is designed specifically to provide runtime assurance for embedded FCSs and autopilots for cyber-attack events [62]. HECAD is intended to be embedded into the hardware platform of an FCS, but it is a separate physical entity from the sensors, busses and main processor to maintain independence and isolation. It consists of four main functional blocks: Hardware Resource, Information Integrity, Execution, and Functional Monitors (respectively HRIM, I2M, EIM, and FIM). The functional blocks are isolated from one another and operate concurrently. All consensus action for the respective monitors is localized. The data flow within HECAD works in a hierarchical fashion beginning with the HRIM which verifies the hardware resources through monitoring the communication protocols of the on-board sensors and retrieving the raw data from the sensors. The HRIM then transfers the raw data to the I2M which converts the raw data into relevant sensor data and checks for data integrity. Next, the FIM performs system-level functionality analysis to ensure that the system operation is within certain bounds. Finally, the EIM verifies the firmware of the autopilot as well as monitoring its run-time execution to look for anomalies. At present, only the lower level monitors (HRIM and I2M) have been realized in HECAD, however, research is continuing and this dissertation is synergistic to the HECAD architecture.

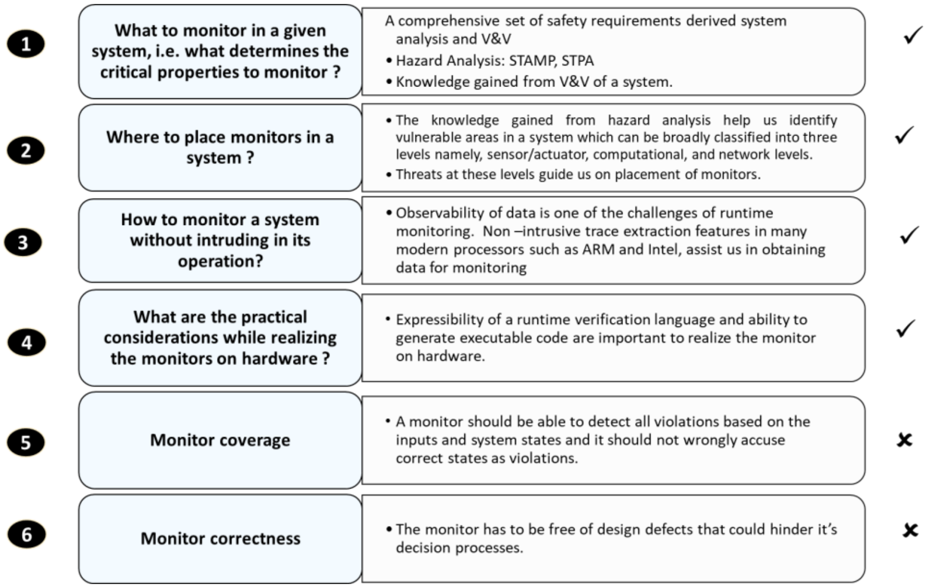
We propose to perform monitoring at multiple levels of a CPS and use ARM embedded trace to obtain data for verification of a correctness property.  We also explore monitoring from a model-based engineering perspective and explore the synergy between design time verification and runtime monitoring. Our key contribution here is demonstrating the benefits of multilevel monitoring framework.

# Chapter 3

# A Framework for the Design and Development of Multilevel Monitoring in Cyber Physical Systems

## 3.1 Introduction

As discussed in Chapter 2, there has been increased usage of runtime monitors in safety critical systems. In order to design, develop and implement effective runtime monitors in critical CPSs, we assert one needs to have a holistic perspective of the CPS and understand the threats a system may encounter in its operational life. Towards this end, this chapter first presents concepts of safety and security, the notion of faults, failures, hazards, and vulnerabilities that exist in a CPS. A taxonomy of faults and threats that can occur across various levels is presented. In addition, this chapter introduces a monitoring framework to reason about monitors in the context of enforcing safety and security in CPSs. In doing so, we explore issues such as monitor coverage, types of monitors, composition, and completeness. We present initial principles for guiding the implementation of monitors. Finally, we present an example of a formal steam-based language that can be used for implementing monitors on hardware.



| | | | |
|---|---|---|---|
| 1 | What to monitor in a given system, i.e. what determines the critical properties to monitor ? | A comprehensive set of safety requirements derived system analysis and V&V<br>• Hazard Analysis: STAMP, STPA<br>• Knowledge gained from V&V of a system. | ✓ |
| 2 | Where to place monitors in a system ? | • The knowledge gained from hazard analysis help us identify vulnerable areas in a system which can be broadly classified into three levels namely, sensor/actuator, computational, and network levels.<br>• Threats at these levels guide us on placement of monitors. | ✓ |
| 3 | How to monitor a system without intruding in its operation? | • Observability of data is one of the challenges of runtime monitoring. Non –intrusive trace extraction features in many modern processors such as ARM and Intel, assist us in obtaining data for monitoring | ✓ |
| 4 | What are the practical considerations while realizing the monitors on hardware ? | • Expressibility of a runtime verification language and ability to generate executable code are important to realize the monitor on hardware. | ✓ |
| 5 | Monitor coverage | • A monitor should be able to detect all violations based on the inputs and system states and it should not wrongly accuse correct states as violations. | ✗ |
| 6 | Monitor correctness | • The monitor has to be free of design defects that could hinder it's decision processes. | ✗ |

Figure 7: Workflow toward the realization of Multi-level runtime monitoring. Correct mark indicates the topics addressed in this dissertation. Cross mark indicates the topics not addressed in this dissertation.

## 3.2 Critical Cyber Physical Systems

Many Cyber Physical System are used in applications that are critical. A system is critical if the services it provides to its end-users or environment are important for their normal function [43]. Additionally, a system is safety critical if loss of service has potential to lead to hazards affecting the safety of the end-users or the environment. In such a system, the loss of data confidentiality, integrity, and availability leads to the disruption of societal services, creates hazards or affects individual privacy. Some examples of safety and security critical systems include healthcare diagnostic and therapy systems, driver assisted automobiles, energy distribution and control systems (smart grid), and nuclear energy protection systems.

Critical CPSs must often provide uninterrupted operation in the presence of faults or become fail-safe until these failures are repaired. These faults can be due to design faults (for example, Software Faults, Specification faults) or physically occurring faults (hardware failures, Single event upsets, etc.). The detection and mitigation of these faults are typically addressed in design and development of such critical systems. Verification and validation of the design with testing and formal methods are employed to identify design faults in a system. Additionally, redundancy and fault tolerance strategies have been the main methods to detect randomly occurring physical faults and failures [63].

Dependable systems concepts and technology have evolved over the past 40 years [64]. They are the underlying technical basis for specifying, designing, and implementing critical systems. A key contribution to the dependability concepts and theory is the work by [65]. This work is widely recognized as a standard for understanding the concepts of critical system attributes. This thesis adopts [65] to provide a consistent set of terminology and concepts for discussing safety and security.

Another factor that is exacerbating the security and safety of CPS is the so-called “Trinity of Troubles”. Gary McGraw, a safety and security expert coined the term the Trinity of Troubles to describe how Connectivity, Extensibility and Complexity are becoming dominate root causes for vulnerabilities in modern computing systems [66]. A CPS can consist (and often does) of several embedded subsystems which are connected through a network where even a small failure can may propagate to other sub-systems. In fact, these failures may accumulate while being masked for some intermediate time in certain components and then be activated under unusual conditions with far reaching consequences to the users of the system. Additionally, contemporary embedded systems are becoming more architecturally complex as evidenced by System-on-a-Chip (SoC) paradigms. While the SoC embedded computing paradigm is beneficial for “reducing integration challenges and time to market”, the drawback is increased complexity. Such a complex system is often designed to be extensible with regular updates which can be a security risk as vulnerabilities may slip in to the system patch updates unknowingly. In short, current trajectories of embedded computing, networking, software engineering which are the drivers for CPSs are concerning from a dependability perspective. Faults, vulnerabilities, and unknown design flaws are considered threats to Dependability.

## 3.3 Concepts of Dependability and Security

According to Avizienis, et al [65] the definition of dependability is stated as: "*The ability of a system to provide its intended and agreed upon functions, behavior, and operations in a correct timely manner and to avoid service failures that are more frequent and more severe than is acceptable.*" The attributes of dependable systems are the means by which the quantitative and qualitative requirements of a system are specified. Following are some basic terms and concepts related to dependable system attributes as used in this dissertation.

**Definition 1:** ***Reliability,*** a conditional probability that the system will perform correctly throughout the interval $[t_0, t]$, given the system was performing correctly at time $t_0$, which is related to the continuity of service [63].

**Definition 2:** ***Availability,*** a probability that a system is operating correctly and is available to perform its functions at the instant time, t [63], which is related to readiness for usage.

**Definition 3:** ***Safety,*** is the freedom of mishaps, accidents or losses that have consequences to environment or people.

**Definition 4:** ***Hazard***, is a system state or set of conditions that, together with some (worst-case) environmental conditions, will lead to an mishap, accident, or loss [43].

**Definition 5:** ***Security***, is the absence of unauthorized access to, or handling of, or alteration of computer system state and data [65].

**Definition 6:** ***Integrity,*** is the absence of improper system state alterations, modifications and damages [65].

**Definition 7:** ***Confidentiality,*** is the absence of unauthorized disclosure of information [49].

A system is reliable if it can provide correct and continuous service at any given time. A system is available when it is functional and ready to provide correct service at all times. A system is deemed to be safe if it can ensure that there are no catastrophic consequences to the environment or the user. A secure system is protected against unauthorized access, able to provide correct service and is ready to provide authorized service even during the presence of failures [3] [4].

The attributes of dependability and security may vary in importance, depending on the application context. As example, robotic arm that is used in the assembly and manufacturing of laptop computer motherboards requires high availability and reliability. A similar robotic arm used in tele-surgical operations requires high degrees of safety, integrity, and reliability to ensure that mishap risks are as low as possible. Availability, integrity is generally required to varying degrees for most systems. Whereas reliability, safety, and confidentiality may or may not be required according to the application. In this dissertation, we are focused on CPS applications that are both safety and security critical.

In this section, we first discuss the vulnerabilities of a CPS due to inadvertent faults, errors and failures. This is followed by a discussion of attacks that are deliberately carried out with malicious intent to exploit the vulnerabilities of a CPS.

### 3.3.1 The Three Universe Model: Faults, Errors, and Failures

*Faults, errors* and *failures* are the threats to dependability and safe operation of a CPS. When the service provided by a system accurately depicts its functionality, it is deemed to be providing correct service. Failure of a system indicates that it has transitioned from providing correct service to incorrect service. A system is considered to be failed if its functionality does not match the system specifications or if the specifications did not sufficiently depict the expected functionality.

Failure of a system can be observed as a deviation in an external state. This deviation is called an error. An error can propagate among the components of a system causing deviation from correct service. For example, there can be an error transferred from system A to system B through an interface.

The cause of an error is due to a fault in the system. A fault can either be active or dormant. Fault activation can occur due unsafe input patterns, a computational process or environmental factors. A fault is capable of manifesting into an error when it is active. The implicit relationship between faults, errors and failures are depicted in the Three Universe Model explained in [67].

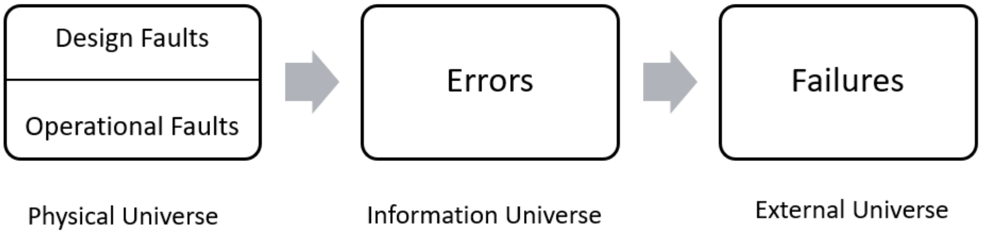


Figure 8: Three Universe Model (adapted from [67]).

The Three Universe Model explains the cause-effect relationship between faults, errors and failures. *Fault* is a defect or a flaw in the system. Manifestation of a *fault* results in an *error* and an error can cause deviation of system behavior which could eventually lead to *failure* of the system. As, faults are the root cause of system failures, it is important to understand the origin of these faults.

The Three Universal Model classifies faults into two categories based on the phase of occurrence: (a) Design faults (b) Operational faults. Design faults occur during the *development phase* of a system due to flaws in design due to incorrect implementation, missing requirements, redundant functionality, dead logic etc. Operational faults occur during the *use phase* of a system when the system is deployed and operated by an end-user. Faults occurring in *use phase* may be due to naturally occurring faults such as hardware defects, semiconductor faults, improper user inputs or faults caused by malicious intruders [65].

Ref [65] presents a taxonomy of faults and provides a comprehensive classification of faults based on phase of occurrence or creation which can be design or use phase, intention of a fault which can be malicious or non-malicious, dimensions of occurrence which can be fault in hardware or software components of a system, etc.

Although software testing methods and practices have undergone tremendous progress over the past 20 years, the evolving nature of software intensive CPSs can create layers of unforeseen failure modes and complex attack surfaces. These unanticipated failure modes and attacks can defeat the fault tolerance or cyber security defenses incorporated in a system. Therefore, the fault tolerance capabilities are in-sufficient to ensure dependability at runtime. In many safety critical application domains, runtime monitors (or runtime verification) are used to enforce operational safety and security – as a complementary defense to design assurance. Runtime monitors can catch both design faults and physical faults thereby providing a complementary method to traditional redundancy and fault tolerance. Runtime monitors enhance the dependability case especially in the context of DepDevOps continuum. To decide on an appropriate placement of runtime monitors in a CPS, it is important to understand the location or origin and cause of faults. Based on the location and cause of faults in a CPS, they can be classified into four categories as described in [68]. Figure 9 summarizes a broad classification of faults in a CPS.

1. **Hardware Faults -** These are physical faults because of component defects, physical deterioration or external disturbances such as natural causes which may not be due to any human interferences [65] [68, p.].
2. **Software Faults** – These are faults created during the development phase of an application. Software faults are caused due to design flaws, incorrect and incomplete requirements and implementation faults.
3. **Communication Faults** – These are the faults that occur in the communication network in CPS that results in missing packets, wrong routing etc.
4. **Interaction Faults** – These are faults created by the user such as inadvertent configuration changes, wrong setting, etc.

Faults can be classified based on their temporal behavior as well:

1. **Transient** – Occur only for a small instant of time and disappears. For example, transient bit flips and signal delays.
2. **Intermittent** – Occur repeatedly in a system. Intermittent faults can occur when there is a loose wire connection or when a component is in the verge of breaking down. [10]
3. **Permanent** – persists indefinitely until repaired. Permanent faults can be stuck at ‘0’ or stuck at ‘1’ faults where a signal is stuck at a ‘0’ or ‘1’ value. A permanent fault can also be a stuck-open fault where the value is ambiguous and cannot be determined. Design flaws or faults, damaged components are classified under permanent faults.

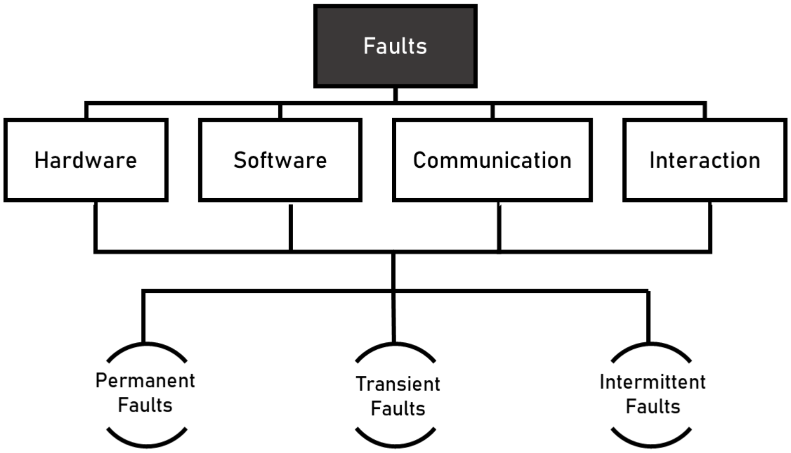


Figure 9: Classification of faults based on their origin in a CPS and based on their duration of occurrence.

In order to maintain the dependability and security of CPSs in the presence of faults, fault tolerance capabilities are incorporated in a design. But, a fault introduced with a malicious intent can defeat the fault-tolerance capabilities causing a threat to the system. Therefore, we study the effects of faults introduced with a malicious intend in the next section.

### 3.3.2 Classification of Attacks in Cyber Physical Systems

Integrity, Availability and Confidentiality are the primary attributes of Security of a CPS. These may be in addition to any dependability attributes such reliability, safety discussed above.

As discussed above, safety engineering has been an important part of industries where hazardous operation of CPS due to faults can have a severe effect on the people and environment. With the rise in cybersecurity threats in the past two decades, safety and security have to both coexist as protection strategies and practices for critical CPSs. In security, the term equivalent to a hazard is vulnerability. This means that a security weakness in a product leaves it open to a loss. Thus, security can be defined in terms of the system state being free of threats or vulnerabilities that can result in potential losses. The security of a system can be viewed as a potential pathway to effect safety, when viewed broadly and beyond the Information Technology perspective of cybersecurity. Therefore, exploiting a vulnerability in such a system can be the causal factor for the activation of a hazardous state with respect to the CPS.

An attack is a deviation of correct service intentionally caused by a human with a malicious objective to cause harm to a system [49]. According to Avizienis in [69], an attack can be called a malicious fault which can be grouped into [65]:

a) **Malicious logic faults** – They can be due to an intentional flaw created during development phase or a virus or a worm introduced in a system during the operational phase. A virus or a worm is a software script that can exploit a system vulnerability, replicate itself and gain access to unauthorized data and violates the attributes of Security.

b) **Intrusion attempts** – They can be due to external operational faults where a threat actor can intentionally alter the behavior of the system to cause fluctuations of expected service, wire-tapping, over heating etc.  to violate security attributes.

In order to ensure safety, timely detection of attacks (security threats) is essential. Security vulnerabilities exist at various dimensions in a CPS. There are a number of classifications of attacks. For example, attacks are classified based on the severity of intrusion in [70]. A four dimensional attack classification if presented in [71]. First classification is based on the attack vectors, second on the attack targets, third on the vulnerabilities in a system and the fourth classification is based on the payload or effects beyond the attack itself [71]. A taxonomy based on attacks at specific layers in a CPS is presented in [72].  Ref [72] shows how security vulnerabilities exist at different levels in a CPS. A taxonomy based on the CPS level that is attacked, is illustrated in Figure 10. Although this is not a comprehensive taxonomy, it provides an overview of attacks that occur at different levels in a CPS. This taxonomy presents the entry points for an attacker. A thorough understanding of the vulnerabilities of the CPS and ways of manipulating the system by an attacker is essential to monitor and defend the system against persistent threats.

Figure 10, depicts an abbreviated attack classification for a CPS that is broadly classified into three domains. First, attacks on low level hardware/firmware-oriented devices.  These include sensor or actuator attacks, for example sensor spoofing, firmware attacks, replay attacks, attacks on the board level buses like I2C and SPI to name a few. Second, attacks on the connection or network layer (e.g. CAN, ProfiBus, Fieldbus, USB, etc..) that include attack on a communication bus such as Denial of Service (DoS), packet injection, eavesdropping. Lastly, attacks on the computational elements such as malware injection, control flow attacks such as code injection, code reuse, buffer overflow etc. that can affect the functionality of the application. This classification is certainly not definitive in its enumeration of possible attacks, but the purpose of it is to stimulate “systems thinking” about how different attack classes can be grouped into different layers of the architecture.

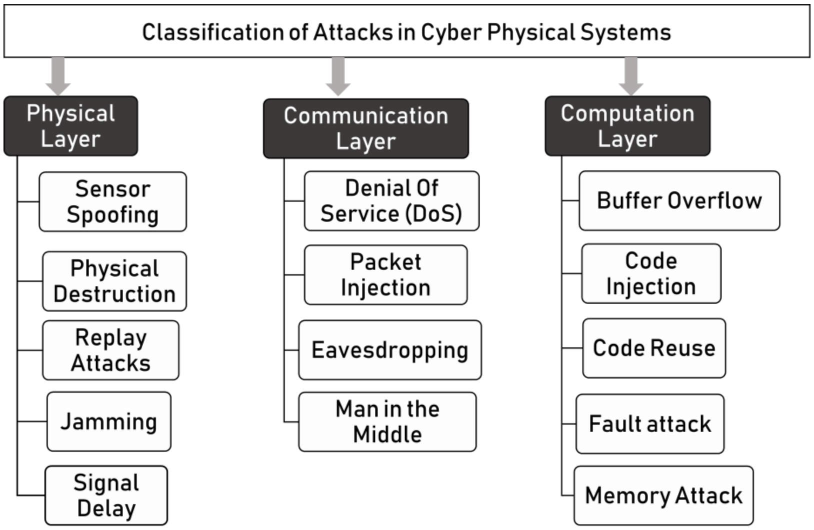


Figure 10: Taxonomy of Attacks in CPS.

A more robust stance on classifying attacks is from the perspective of MITRE/NIST attack ontology database. The US National Institute of Standards and Technology (NIST) along with MITRE Corporation maintain a set of vulnerability/attack databases whose purpose is to aid in the defense of cyber IT systems. These databases are called CAPEC, CWE, and Common Vulnerabilities and Exposures (CVE) [73]. CAPEC (Common Attack Pattern Enumeration Classification) is a pseudo-ontological hierarchy of attacks. It describes attacks based on techniques used to accomplish them, as well as with respect to the goal of the attacks (such as collecting information or manipulating a state). There are over 500 attack patterns contained in this, described in natural language. CAPEC is organized as a hierarchy where high-level instances describe attacks in a general way for classes of systems, and low-level instances describe specific type of attacks for more specific systems. The other companion database is Common Weakness Enumeration (CWE). CWE weaknesses are organized according to multiple views, such as where in development the vulnerability/fault arises, or by abstractions of the software behaviors. CWE is lower level information and is more expansive in its listings – it’s like an encyclopedia. Like CAPEC, it is pseudo-ontological, providing a high-level understanding of each of the concepts leading to vulnerability. CAPEC and CWE are linked. That is, numerous CAPEC attack patterns refer to weaknesses in CWE that they target. The advantage of the CAPEC/CWE perspective is that it is maintained and used by IT professionals worldwide, and there are tools to support it. The disadvantage is that it is lacking entries from the CPS attack/vulnerability perspective. Tools like CYBOK partially help overcome this deficiency, by mapping and associating CAPEC/CWE database information to models of CPSs [74].

# 3.4 Formal Development of Multilevel Monitoring Framework

## 3.4.1 A CPS Reference Architecture

In order to develop a framework for multilevel monitoring we need a “reference architecture” of a Cyber Physical System. A CPS encompasses many computational units and includes physical interfaces to sensors and actuators. A CPS system can be discrete, continuous or a hybrid system, where a discrete system consists of digital signals, continuous system consists of analog signals and hybrid system use both digital and analog signals [2]. Figure 11 depicts a common interpretation of a generalized CPS structure.

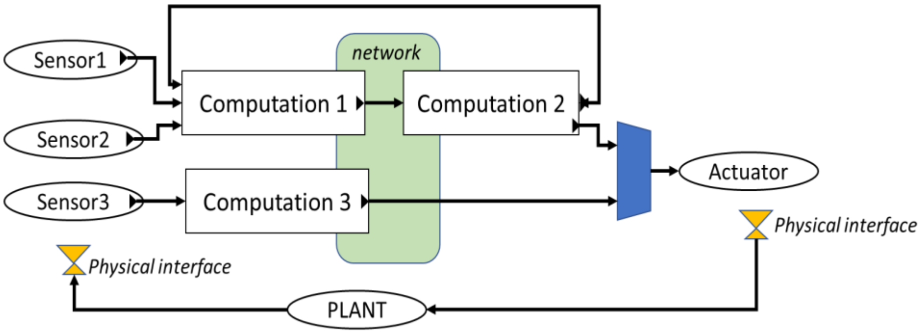


Figure 11: Structure of a Cyber Physical System (Adapted from [2] but modified and redrawn).

Embedded computers are typically governed by their programmed behavior. When an embedded processor executes a program, it effectively becomes the machine it is programmed to be. Thus, the application is encoded using the processor instructions to accomplish the intended goal. However, a CPS has certain computational behavior that go beyond the above explanation.

The Cyber Physical System computation is modeled for applications where the overall behavior is represented as a *reactive system*. A *reactive system* is characterized by its interaction with its environment from which it is continuously accepting requests and continuously producing results [75]. In such systems, correctness or safeness is related to the reactive system’s behavior over time as it interacts with the environment. While functional computations, calculate a value upon termination, reactive programs usually do not terminate unless an exception event has occurred. Further, reactive systems are typically *real-time*, i.e. they deliver services that has a deadline by which the computation should be complete. Applications of reactive systems include building environmental control systems, process control systems, vehicle control systems, and communication protocols.

In the above method, we start with a model for reactive computations. The most appropriate choice of model for runtime monitoring is the finite/infinite automata model. Lee and Seshia [2] present an Extended Finite State Machine Model (EFSM) where a set of trigger conditions define the state transitions in a CPS. CPSs are constantly interacting with the environment and the trace data is defined by updated variables, states, execution traces etc. The elements of the trace data are a set of trigger conditions that are used to verify a change in state in a CPS. The cyclic behavior of the CPS can be defined by the EFSM model, which has three steps : evaluation of trigger conditions, compute the next state and control actions and perform data operations [76].

Analog sensors convert physical continuous time measurements into digital representations of the sensed process variable that serve as inputs to the CPS. In a reactive computation model, these senor inputs are continuous interactions with the computation (e.g. the program running on the processor) and are comprised of data streams or runs of information.  Input can also be digital or modal inputs. Here, digital inputs provide an event to the CPS that could cause it to alter its program behavior due to the occurrence of the event. An event may also be a change in mode, for example, change from cruise control “on” to cruise control “off” in an automobile. An event may also be an unexpected incident, such as indication that fuel is low in an automobile. Events are usually sparsely occurring data streams. Actuators respond to the inputs and states of a system and produce an output. Specifically, the computations in the CPS produce digital commands that are sent to actuators which translate these commands into physical control of the plant or updates for human monitoring. Thus, actuator channels are also viewed as ongoing data streams.

As discussed in the taxonomy of faults/attacks in the last section, faults/attacks occur at various dimensions or levels of the CPS reference architecture. In many safety critical systems, runtime monitors are used to complement design assurance and enforce operational safety and security. In order to detect attacks and complex evolving failures across a CPS, we posit that single monitor solutions are insufficient. Vulnerabilities exist in a CPS at multiple levels that span both hardware peripherals and software implementations which include sensors, actuators, application software, and communication networks etc. These different layers of a CPS have diverse functionalities and are integrated together in a

CPS.  Therefore, we suggest that multiple distinct types of monitors positioned across the different layers are beneficial in providing a more comprehensive detection capability.

## 3.5 Justification for Multilevel Monitors for detection of attacks/failures

The placement of monitors is influenced by the origin of vulnerabilities in a CPS. As seen in Section 3.3, vulnerabilities occur across different dimensions of a CPS. They can be broadly classified into three domains:

(i) Faults/Attacks on low level hardware/firmware-oriented devices.
(ii) Faults/Attacks on the communication or network layer (e.g. I2C, CAN, SPI).
(iii) Faults/Attack on the computational elements.

### a. Placement of Monitors and benefits of Multiple levels of Monitoring

We consider four levels of monitoring across the CPS as seen in Figure 12. Our approach is synergetic to the HECAD architecture [62]. The monitors are described below.

- **Data monitors:** They mainly monitor the hardware/firmware-oriented devices such as sensors and actuators that constantly interact with the outside environment. They check for integrity of the information coming from these devices through the physical interface.

An example of a data monitor is observing the speed of a car that allows detection of a fault or spoofing attack on the sensor measuring the velocity with a condition “rate of change of velocity < 10 m/s$^2$”, i.e. an abrupt change in the measured velocity will lead to a violation of this property.  Such a monitor *will directly receive data* from a sensor node or sensor module (e.g. fusion of radar and camera data to determine velocity) and before it is sent to other units of a CPS such as the CAN Bus or a computational unit. The reason this is important is as follows: Suppose the sensor communicates a packet of information (i.e. value of vehicle speed) every 0.1 seconds, when we get a stream of information starting at some initial time t=0, we can view this information as vehicle speed at t=0, 0.1 s, 0.2 s, 0.3 s…. and so on. However, if we pass it through a CAN Bus that is faulty or attacked, where the first packet arrives at t =0.05 s, the second at t=0.2s, third t=0.25s, etc. and also some packets can be added or lost, then this information cannot readily be used to monitor the variation of the velocity as a function of time.

- **Network monitors:** They mainly monitor the connection or network layer of the CPS. Sensors, actuators and computational units in a CPS use communication protocols such as UART, I2C and buses such as CAN. Network monitor checks for signal faults, incorrect signaling protocol, timing, configurations, etc. in these communication networks.

An example of a network monitor is observing the time taken by a packet of information with a unique ID sent from a certain sensor node to reach another node. This allows detection of an attack on CAN Bus with a condition "time for a packet of information to travel between two specific nodes through a CAN Bus < 100 ms", i.e. any delay more than this will lead to a violation of this property.

In summary, if we are monitoring the *vehicle speed* that is data monitoring while if we monitor the time taken by a packet of information representing the *vehicle speed* to travel between two nodes of a CAN Bus this is network monitoring.

- **Functional monitors:** They mainly monitor the computational units of a CPS to verify the overall system behavior or functionality of a processing unit within the CPS. Safety and security properties are monitored for expected system behavior.

Thus, a functional monitor observes the relationship between different streams of information and concludes whether there is a violation of a monitorable property.

- **Execution Monitor:** They extract out execution traces (variables, execution traces etc.) from the executing code. The methods we are investigating at the present include control flow monitoring, and using watchdog timers to detect change in execution times due to an attack.

We develop a multilevel monitoring approach where "levels" indicate the different architectural layers in a CPS. For example, we consider data, functional, network and execution layers for which the example of what could be monitored are respectively sensor information, controller functionality, CAN Bus communication, and program control flow.

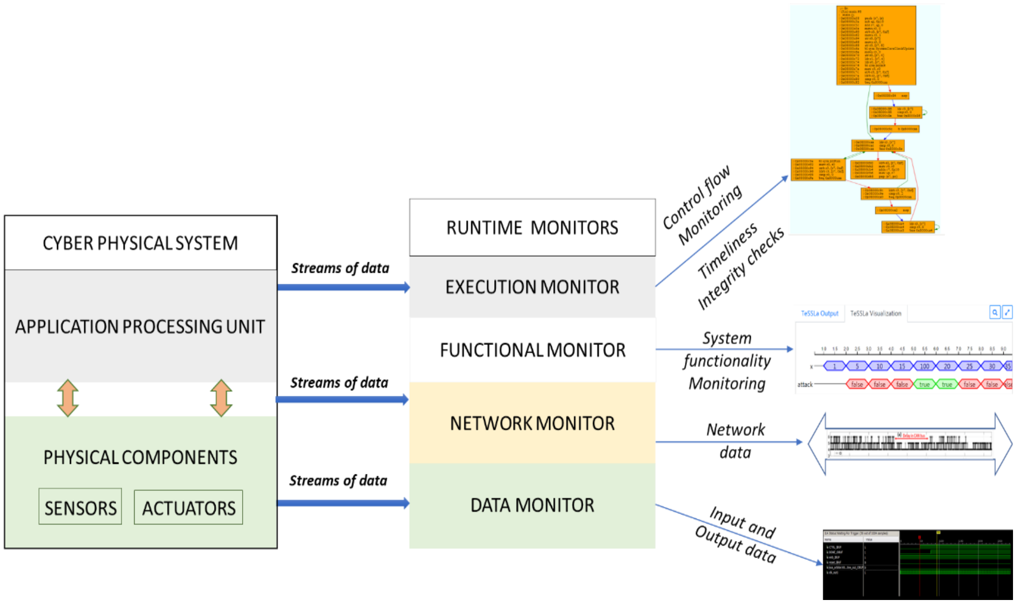


Figure 12: Multilevel monitoring.

Having monitors at multiple levels (data, network, execution and functional monitors) will ensure that more classes of faults/attacks can be detected and isolated early before it propagates and affects the system. Attacks that fall outside the intersection, in Figure 13 can only be detected by having a localized monitor at that particular level in the CPS. Having these local monitors at each critical level in a CPS helps cover one other's blind spot [28]. Furthermore, faults/attacks may be detected by monitors at other levels (than that of their origin). Even in such cases, monitors at multiple levels are needed to find the location of these faults/attacks.

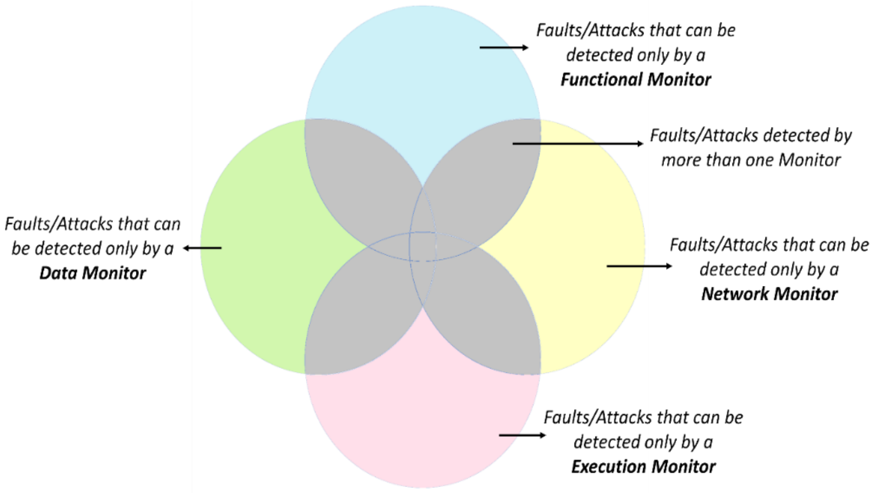

Faults/Attacks that can be detected only by a **Functional Monitor**
Faults/Attacks detected by more than one Monitor
Faults/Attacks that can be detected only by a **Data Monitor**
Faults/Attacks that can be detected only by a **Network Monitor**
Faults/Attacks that can be detected only by a **Execution Monitor**

Figure 13: Need for Attacks/Faults detected by monitors at multiple levels.

It is also important to distinguish three different terms used in this thesis by explaining each of these terms: monitoring multiple properties, multiple monitors and multilevel monitors:

i. **Monitoring multiple properties**: When we are monitoring the functioning of a single controller with multiple inputs and outputs we may check if various properties hold (e.g. when input-1 increases, output-1 decreases; when input 1- begins to increase, output-2 begins to increase 5 seconds later, etc.) If all these properties of the same processor/controller are monitored locally by a trusted hardware, for example a trusted FPGA, then this FPGA acts as a single monitor, though it could be monitoring multiple properties.

ii. **Multiple monitors:** If the functioning of different controllers (e.g. PID controller, ECU, ABS, etc.) are being monitored locally by different trusted FPGAs, then each of these FPGA is deemed as a different monitor. In this case, there are multiple monitors but they are still monitoring a single level, i.e. the functional layer of the CPS. Note that each of the monitor may be monitoring one or more properties corresponding to the correct functioning of a particular controller.

iii. **Multilevel monitors:** For monitors to be considered multilevel monitors they must be monitoring the functionality at different architectural layers in a CPS (e.g. consider data, functional, network and execution layers). Note that at each of these levels, there could be one or multiple monitors and each monitor in turn can be monitoring one or multiple properties.

It should be noted that having a single monitor to observe streams of data from multiple computational units may have lot of challenges such as [27]:

a. CPS can have both synchronous and timed asynchronous components with distributed clocks. Hence, the timing of observations of data streams may vary extensively making monitoring such systems challenging.
b. The latency of various components in a CPS may vary, resulting in unpredictability in global execution behavior.
c. The criticality of components in a CPS may differ, where the consequence of failure or malfunctioning of certain components may be more severe than others. It may be difficult to incorporate precedence in the detection mechanism in a monitor.

d. The size of distributed systems with numerous components and execution interweaving makes it very challenging for implementation of a single monitor.

Therefore, decomposition and placement of monitors in a CPS is an important and challenging research topic. Having monitors distributed across a CPS among multiple execution nodes is suggested in [77]. Distributed monitors can result in increased monitoring overhead. Further, the monitors can incur delay in reaching a verdict if they have to communicate with each other to form a decision. However, monitors at multiple levels is beneficial despite these limitations and helps in faster detection and possible isolation of anomalies. Furthermore, multilevel monitors provide a more comprehensive coverage of the attack surfaces, thereby preventing a potentially dangerous anomaly from going undetected.

### b. Are multiple monitors needed at the same level?

We need many localized monitors at the same level to detect attacks/ faults. For example, in a vehicular CPS there are multiple controllers such, e.g. Engine Control Unit, Speed controller, Autonomous Emergency Braking unit etc. which all have different functionalities. When we say "functional level" we are actually monitoring functionality of many such controllers. In theory, one could observe the inputs and outputs of each of these controllers through the CAN Bus and implement many functional properties on a single FPGA or any hardware platform connected to the CAN Bus. However, as we explain in Chapter 5, the functional monitor implemented using data from the CAN Bus can have limited effectiveness in detecting the origin of attacks. This is because some attacks (e.g. denial of service attack, pack injection, packet delay) on the CAN Bus can also manifest as an error in the functionality of a controller while the actual attack occurred on the CAN Bus. Hence, there is a need for locally monitoring different controllers. Consequently, one can envisage having many monitors at the same level (e.g. the functional level in this case).

## 3.6 Formal Model of Multilevel Monitoring

In this section, we develop a formal model for multilevel monitoring by employing the theories of runtime monitoring developed by [10]. Elks in [10], developed a property based model of runtime monitoring around a single monitor for an embedded system. In this section, we extend it to multilevel monitors. In order to explain our multilevel monitoring approach, we first start with some definitions as stated in [48] [49].

**Definition 1:** ***Trace,*** is a finite sequence of observations that represents (or in some cases approximates) the behavior of interest in the monitor system [49].

**Definition 2:** ***Data Stream,*** is a continuous sequence of data or signals received from the CPS under observation.

**Definition 3:** ***Monitor,*** is a device that reads a finite trace and yields a certain verdict [48].
A Monitor is a runtime object that is used to check properties. The monitor will receive observations from the trace (usually incrementally) and may optionally send information back to the monitored system, or to some other source [49]. A monitor observes streams of data coming from a CPS, verifies them against a correctness property and produces a verdict indicating if the property is satisfied or violated.

**Definition 4:** ***Monitored system or System Under Observation,*** is the system consisting of software, hardware, or a combination of the two, that is being monitored. It's behavior is usually abstracted as a trace object [49].

**Definition 5:** ***Instrumentation,*** is the process of extracting/recording the trace is referred to as instrumentation [49].

### a. *Notion of a Property*

A formal description of 'what to monitor' in a CPS depends on the concept of a *property* [78]. Properties are desirable things that should happen in a system such as fairness, progress and termination and undesirable things that should never happen and result in hazards [10]. Properties are guided by the functional and protection specifications of execution behavior of a CPS which is interacting constantly with the environment. Therefore, properties are used to express and formalize the runtime behavior of a CPS. Some of the definitions of properties related to runtime monitors are expressed below:

**Definition 1:** ***Correctness (functional) Property****,* is a condition that is verified by the runtime monitor against a finite *Trace*. The monitor yields a verdict indicating if the Correctness Property was satisfied or violated. A correctness (functional) property is a specification of desired or acceptable behavior for a CPS, for which the CPS executions are refinements of that property.

**Definition 2:** ***Safety Property***, is a condition that is verified by the runtime monitor against a finite *Trace* and checks to ensure that something bad never happens [10]. A safety property is special property that is refined from the safety or security specification of the system. A safety property defines the acceptable safe behavior for a CPS. A safety property is completely characterized by the safe executions CPS and no others.

**Definition 3: A** ***Safety (Security) Specification,*** of a system defines and describes actions and behaviors of a system which must not occur to achieve a protection policy - irrespective of implementation. A safety specification is a set of statements that define the restrictive behavior of the system [49].

Properties that are monitored at runtime are usually derived from higher level system requirements that are approved by domain experts [11]. But, as the complexity of systems are increasing, it is important to ensure that these requirements encompass all the vulnerabilities in a CPS and all the safety properties. Violation of a safety property can result in a hazard. For example, a vehicle should not move if a door is open is a safety property. When a vehicle violates this property, it can cause an accident or a hazard. Safety properties are typically derived from system requirements, hazard analysis, failure analysis etc. which we discuss in detail, later in this chapter.

### *b. Model for monitoring a single property*

### The concept of traces and streams

As described in [10], a monitor *M* observes streams of information from the CPS and renders verdicts on those streams in terms of safety, correctness, and security. Such information could be discrete or digital representations of continuous state variables, sensor data or other states that is viewed from a historical perspective i.e. starting from the current state of the CPS to states previous in time, or past temporal observations. Each stream can be a sequence of inputs or outputs to and from the CPS or sub-system within the CPS. In this monitoring scheme we assume that the monitor receives or requests a finite trace of information on regular time intervals and these traces are time stamped. We assume these monitor and CPS interactions are ongoing and continuous.

***Trace:*** A trace from a CPS is a finite sequence of states or words that can be denoted as

$r = w_1, w_2, w_3, w_4, \ldots w_n \in S^*$.

Where, $w_1, w_2 \ldots w_n$ are states from the CPS

A collection of traces is r(k) = $r_{k1}$, $r_{k2}$, $r_{k3}$, $r_{k4}$, …$r_k$ $\in$S*.

Where, $r_{k1}$, $r_{k2}$…$r_k$ are finite traces from the CPS at k intervals

S* is the set of all finite traces from the target CPS.

A trace is a "snapshot" of the execution behavior of the CPS at specific time instances. We formulate a stream to be composed of traces, which start from the current state of the CPS to a state previous in time.

***Stream:*** We denote a finite *stream* $S^k$ as the concatenation of past $m$ instances of finite traces, as

$$S^k = \left(r\,(k - m), \ldots, r\,(k - 2), r\,(k - 1), r(k)\right) \in S^* \qquad (1)$$

Where S* is the set of all finite traces from the target CPS

Hence, at time $k$ the stream contains information of the past $m$ instances starting with r($k - m$) and ending at the current instance r($k$).

An *infinite stream* is a concatenation of finite traces occurring repeatedly. This formulation captures the notion of the reactive computing model, where executions are cyclic and ongoing. This is denoted as:

$$\rho = \left(r\,(k - m), \ldots, r\,(k - 2), r\,(k - 1), r(k)\right) \in \inf(\Sigma) \qquad (2)$$

Where inf (Σ) is the set of infinitely occurring executions from the target CPS.

As we noted in section 3.4, a CPS architecture is composed of different technological elements, namely, various types of sensors, computational processing units, outputs of various types, and networks to connect these elements together. Expanding equation (3) we arrive at a set of monitored streams to reflect the diversity of streams with respect to the partitions in the reference CPS architecture. We suggest *L(M)* as a *monitor language* which is composed of the streams that monitors witness during runtime monitoring

$$L(M) = (ins^k, os^k\ \delta^k, ns^k) \qquad (3)$$

Where input sensor stream:

$$ins^k = \left(rin\,(k - m), \ldots, rin\,(k - 2), rin\,(k - 1), rin(k)\right) \in S^* \qquad (4)$$

Where output actuator stream:

$$os^k = \left(ro\,(k - m), \ldots, ro\,(k - 2), ro\,(k - 1), ro(k)\right) \in S^* \qquad (5)$$

Where stream of system states and variables monitored from a given computational unit:

$$\delta s^k = \left(r\delta(k - m), \ldots, r\delta(k - 2), r\delta(k - 1), r\delta(k)\right) \in S^* \qquad (6)$$

Where network stream:

$$ns^k = \left(rn\,(k - m), \ldots, rn\,(k - 2), rn\,(k - 1), rn(k)\right) \in S^* \qquad (7)$$

Where current time observation is denoted as $k$ and $m$ as some past time instance observation, and $S^*$ is the set of all finite traces from the target CPS.

### Monitors and Monitor Language

Following the formulation above and inspired by [10], a monitor *M* accepts all legal words in its language. The natural question to ask is what is the language of a monitor? A monitor observes a sequence of events from the CPS in the context of a monitored stream. These events typically constitute the CPS architecture, such as, program execution behaviors, data states, control states, inputs, outputs, etc. These events are usually encoded in the stream as programing constructs such as variables, time, data-structures and data types of specificity to the monitor. In this context, these streams can be characterized as *language or alphabet* of the safety monitor. Said simply, the monitor is designed to recognize and accept runs of words that are legal in the language.

Following from above, we now develop the relationship between the monitor and the target CPS. Let *L(M)* be the language of the monitor, which is defined as the set of accepted words recognized by monitor *M*. Let *L(A)* be the language of the computer-based CPS. *L(A)* represents the programmed behavior of the processor for a given application, including its interactions with the external environment and the interactions among the different elements of the CPS. Thus, *L(A)* can change if a program is modified (either intentional or maliciously) or becomes faulty or a new program is installed that is different from the original. Hence, there may be some words *w* ~~from *L(A)*~~ that are classified as faulty or unsafe behaviors if $w \notin L(A)$ . Suppose the infinite stream $\Sigma^*$ is the set of all words from the computer-based CPS, and $S^*$ is the set of finite words represented in finite streams. These are composed of *L(A)* and $\overline{L(A)}$ , where $\overline{L(A)}$ denotes the set of $\Sigma^* - L(A)$. Thus, $\overline{L(A)}$ is the domain of unsafe and insecure behaviors – behaviors not encoded or refined from the safety/security specification. The relation between $\Sigma^*$ , $\overline{L(A)}$ , *L(M), L(A)* is shown below as explained in [10].

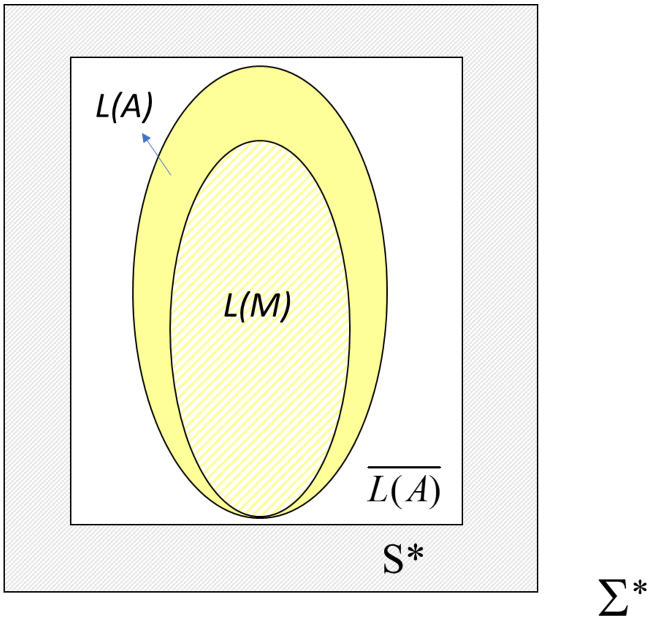


Figure 14: Language based formal model for runtime monitors [10].

In Figure 14, *L(M)* is a subset of words from *L(A).* Ideally, *L(M)* should coincide with *L(A)* so all acceptable language is deemed as safe/secure by the monitor and *L(M)* should have no intersection with $\overline{L(A)}$ to prevent a faulty/insecure state from being missed. However, in reality while the latter can be met, the former condition may not be met completely as *L(M)* is only a subset of *L(A).* In such cases, *L(M)* is more specific than *L(A)* potentially leading to some acceptable behavior being flagged (false positive detection). This however, is better than missing a fault/attack which could be a more dangerous scenario.

The above formal characterization of the relationship between monitors and CPS exemplifies a monitor that accepts streams that satisfy the specifications for safety and security of the system. It detects faults and anomalies that violate such safety and security specifications. In [10], it is shown that a well-formed monitor has a number of properties that characterize its relationship with the target system being monitored. These properties are soundness, accuracy and completeness. The monitor *M* is *complete* with respect to a safety property $P_{safe}$ if for any trace **r** such that **r** |≠ $P_{safe,}$, *M* is guaranteed to reject **r** or raise an alarm. Furthermore, a safety monitor *M* is *sound* with respect to safety property $P_{safe}$ if whenever M raises an alarm, then always **r** |≠ $P_{safe}$ (e.g. no false alarms). Also, a safety monitor *M* is *accurate or reliable* if for any correct trace **r** |= $P_{safe}$, *M* never reports any trace **r** of being unsafe (e.g. **r** |≠ $P_{safe}$). Elks [10], showed that these 3 properties are difficult (if not impossible) to achieve in realistic non-trivial systems. Thus, real world applications, monitors are often categorized in terms *coverage* to indicate how well they detect or protect a system.

Further, the above language characterization of monitors can be extended to multiple monitors with each monitor recognizing a subset of words "*w".*

### c. Multiple Monitors

The language of the computer-based system $L(A)$ can be classified into $L_{io}(A)$, $L_n(A)$ *and* $L_\delta(A)$, based on the kinds of streams that they represent. $L_{io}(A)$ constitute words that belong to the input and output of the CPS. $L_n(A)$ constitute words that belong to the network or communication components of the CPS. $L_\delta(A)$ constitute words that belong to the computational elements of the CPS. The classification of the words is based on the diverse functionality of the different layers of CPS. This is a way of classification of *words* in a computer-based system that we have employed in this dissertation. There can be other ways to classify the words. The word *w* can be stated as:

$$\mathrm{w} = L_{io}(A), L_n(A), L_\delta(A) \in L(A) \qquad (8)$$

In order to monitor these different classifications of words from the target system, we have multiple monitors $\mathrm{M_{io}}$, $\mathrm{M_n}$ and $\mathrm{M}_\delta$ where $M_w = (M_{io}, M_n, M_\delta)$. The language of the monitors $L(M)$ can now categorized based on the words from $L(A)$ that they recognize. $L(M)$ comprises of $L_{io}(M)$, $L_n(M)$ *and* $L_\delta(M)$.

$L_{io}(M)$ be the language, which is defined as the set of accepted words recognized by $M_{io}$
$L_n(M)$ be the language, which is defined as the set of accepted words recognized by $M_n$
$L_\delta(M)$ be the language, which is defined as the set of accepted words recognized by $M_\delta$

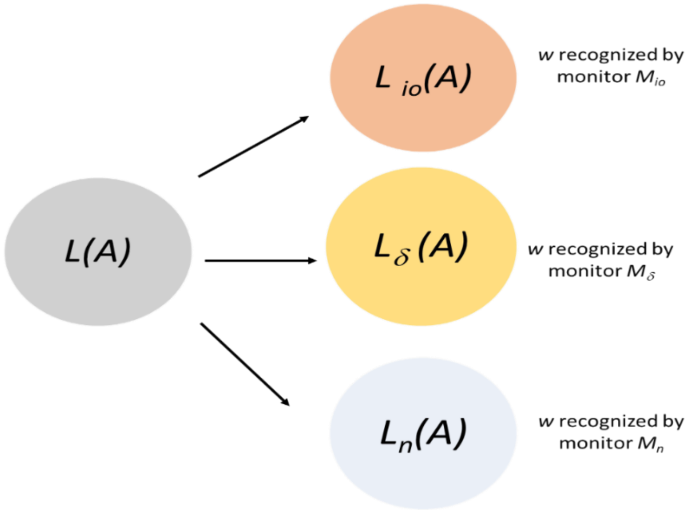

L(A)
L io(A)
w recognized by monitor $M_{io}$
$L_\delta(A)$
w recognized by monitor $M_\delta$
$L_n(A)$
w recognized by monitor $M_n$

Figure 15: Words *w* in the language of the computer-based system *L(A)* classified into $L_{io}(A)$, $L_n(A)$ and $L_\delta(A)$ and are recognized by monitors $M_{io}$, $M_n$ *and* $M_\delta$respectively.

Creating different monitors in a CPS is a natural decomposition, which recognizes a separation of concerns in the partitions of a CPS. This is because, different partitions of the CPS perform diverse functions in the service of the whole CPS objective. Therefore, this decomposition is beneficial in focusing on detection of faults/attacks that are specific to that subsystem. In addition, this enables faster detection of faults/attacks in each partition of the CPS that could help produce faster response to the unsafe behavior of the CPS. Furthermore, streams S* in a heterogeneous CPS, may be represented by different languages from diverse processing systems. Therefore, a single monitor *M* may not suffice or be practical to recognize all the diverse behaviors (words) of a real CPS.

### d. Monitorable Properties

Following the formalism developed by Elks in [10], Monitors ($M_w$) recognizes a set of traces or streams $S^* = ins^k, os^k, \delta s^k, ns^k$ from a target CPS whose computations can be represented as a monitorable property. Assuming that security violations or faults in the system are observable, then we can extend the definition as follows to define a simple monitor:

- Let $P_{observable}$ be a set of all observable traces from a given stream which has events and conditions that are monitorable (from the stream S* and acted upon by monitor defined by $M_w$).
- $P_{observable}$ has one on one correspondence with monitor stream $M_w$.

- Let D be a detection predicate over the streams of data S* and β be the set of all safe runs (traces) in the observed data stream.
- Let $P_{safe}$ be a condition that holds true if there exists only β from the set of all observable runs.
- Let α be a set of bad finite prefixes in a run (violations over $P_{safe}$) from the set of all traces.

  Note: A prefix is a partial sequence of states from a trace or run starting from an origin point and terminating when the first “bad” word occurs.

Then, it follows the condition (as discussed in [10]):

$$D(\beta) \Rightarrow (\ \beta \in P_{safe}\ ) \qquad (9)$$

When the detection predicate witnesses the observable data $P_{observable}$ and the data is safe and has no faults, and then it implies $P_{observable}$ has only safety runs β and the safety property condition $P_{safe}$ holds. In the case of a violation:

$$\neg D(\beta) \Rightarrow (\exists\alpha : \alpha\,\beta \notin P_{safe}) \qquad (10)$$

Then the detection predicate acts on the observable data $P_{observable}$ and has detected a fault or an attack, then there exists at least one bad prefix α with the safety run β. Therefore, the safety property $P_{safe}$ ceases to hold. The above definition is provisioned on two sufficient conditions:

**Condition 1**: The detection predicate(s) *D* that define the safe execution of a run must be defined within the monitor and be under control of the monitor.

**Condition 2**: External processes to the monitor cannot manipulate, gain access or view the detection predicate(s) *D* within monitor $M_w$.

Therefore, the monitor $M_w$ with all the observable data $P_{observable}$, makes real-time safety and security assessments of the CPS using the detection predicate.

At this point, it is important to discuss the relationship between system safety, properties and streams to avoid any misunderstandings between them. System safety is the reduction of exposure to hazards and possible mishaps by a series of safety policies or specifications. These safety specifications are applied to the CPS in order to enforce safe operation. Therefore, *Safety Properties* of a CPS can be defined as a set of execution behaviors that maintain safety specification or protection policies of the CPS [10]. Safety properties for the system are refinements from the higher-level safety specification policies of the CPS. The observed streams from a CPS embody the higher-level safety specifications. During a normal operating condition, the streams S* should comply with the safety specification on the CPS. In the above formulation of a monitor, we associate safety with the β traces of S*.

We next discuss formal design-oriented languages to express safety properties or detection predicates in a monitor.

### e. Expression of monitoring properties using Event Calculus

The above formulations of monitors are abstract models of what a well-behaved monitor should do, but does not address how they do it. The concept of a detection predicate that "checks" to see if an observed stream is safe is a reasonable abstraction, but it provides no context or clue on how to express complex monitoring properties in real CPSs. To transition from abstract models to design models, we need design-oriented tools and methods. In this case, runtime verification languages are often used to express/specify complex properties in complex CPS systems. Research and development in generic runtime verification languages and tools are substantial and constitute a vast literature base [49][9]. Our focus is on design languages that are more tailored for CPSs. There are many popular examples of such logics such as Signal Temporal Logic (STL), Event Calculus, and Metric Temporal Logic (MTL) [79]. These are some of the runtime languages able to express specifications for monitoring complex embedded systems and CPSs. In this work we use Event Calculus as specification language to express monitoring properties of the multi-level monitors.

Event calculus (EC) is a powerful logical formalism that can conveniently express the effect of events or actions in a CPS in a general way [80]. It is particularly suitable in its ability to express high level functional events as well as low level hardware events. For example, one can express the condition that the temperature of the room increases at a certain rate after a heater in turned on. Formally, in the language of event calculus, switching "on" the heater is an *action or an event*, that affects the temperature of the room (a fluent) at certain time points. *Fluent* is variable whose value can change over time [80]. It could be a quantity, such as "temperature of the room", whose numerical value changes over time [80], or a proposition, such as "the temperature is less than 70F", whose truth value changes with time to time. The latter are called propositional fluents and are used in this dissertation.

Happens, Initiates, Terminates, HoldsAt and Clipped  are the basic event calculus predicates defined in [80]. *EC Predicates* are first order logic functions used to describe the events and actions. They are used to express a correctness property in a runtime monitor, where the property describes conditions such as "what happens when" and "what events do" and how an action can affect a fluent. The EC predicates are described below [80]:

- *Happens (α, t)* means that an action α happens at time t.
- *Initiates (α, f, t)* means that an action α occurs at time t and a fluent f starts hold true. Fluent f that holds true at the start of an operation is shown by the predicate *InitiallyP*(f).
- *Terminates (α, f, t),* means the termination of a fluent which signifies that fluent f stops being true after an action α occurs at time t.
- *HoldsAt (f, t)* shows that the fluent f holds at time t.
- *Clipped ($t_1$, f, $t_2$)* indicates that the fluent f is terminated anytime between time period $t_1$ and $t_2$.

Based on observing the system experiment or simulation, if one knows that 60 seconds after the heater is turned on, the room temperature increases; the event calculus predicates (summarized below) allows us to define the effect of an action (switching on the heater) on the fluent (temperature rise) in the following way:

$$\text{Happens}(HeaterOn, \text{t}) \Rightarrow \text{HoldsAt}\big(Temperature(\text{Increases}, \text{t} = \text{t} + 60)\big) \qquad (11)$$

We have used event calculus to express safety and security properties in this dissertation. Although, the EC predicates can give a useful logical formalism to a correctness property, they cannot be synthesized on hardware.  Therefore, event calculus is used to formally express and define monitor properties only and not used for implementation of monitors. To realize a correctness property in hardware, we use TeSSLa, a stream based runtime verification language especially designed for CPS [81]. TeSSLa has a rich set of functions in their library which can be used to describe a correctness property. We discuss about TeSSLa later in this Chapter.

## 3.7 A High-Level View of Monitoring Approaches

In addition to different levels of monitoring, an important aspect of monitors is whether they are in-situ or native to the CPS or if they are external monitors that are implemented on a platform that is physically separate from the CPS. Furthermore, when multiple monitors are involved, their organization and how they interact with each other are important aspects of the monitoring architecture.

### 3.7.1 In-situ vs. External Monitors

The runtime monitor can be placed in the same platform which is integrated with the application. Alternatively, the monitor can be on an external isolated platform. There are advantages and disadvantages to both these approaches.

#### *a. In-situ Monitor*

In-situ monitors are shown in Figure 16. Here, the monitors are embedded or integrated with the System Under Observation (SUO), which denotes the CPS being monitored. Maximum system state observability is one of the biggest advantages of an in-situ monitor. The monitor can be just another thread or a task in the program. But, since an in-situ monitor can modify the source code it should be ensured that it does not introduce any additional vulnerability to the SUO. Sharing resources and memory by both the SUO and the monitor can increase the attack surface [82]. Also, modification of the source code would require recertification for usage in safety critical applications.

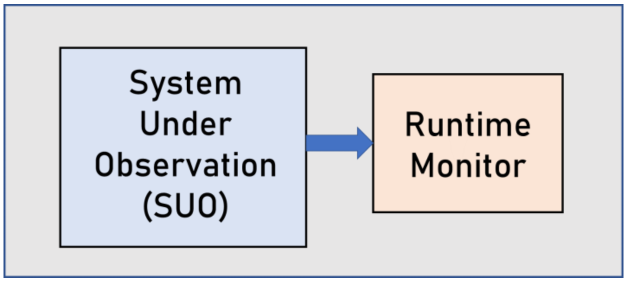


Figure 16: In-situ Monitor.

#### *b. External Monitor*

An external monitor is physically separate from the SUO, and is implemented on a separate platform as shown in Figure 17. Such a monitor has its own memory and resources and would not modify the SUO, thus reducing the attack surface [82]. Even if the target is compromised by an attacker, the behavior of the monitor cannot be altered. Therefore, physical isolation and separation enables stronger arguments for non-interference with SOU. Such a monitor can be independently certified, making it easier to integrate with an existing safety critical system. But, limited observability of system states and internal signals constrains the monitoring properties that can be formulated. Properties can only be checked over external system data as not all internal states will be accessible. Instrumentation of SUO code may be necessary to extract any additional data necessary for monitoring [50].

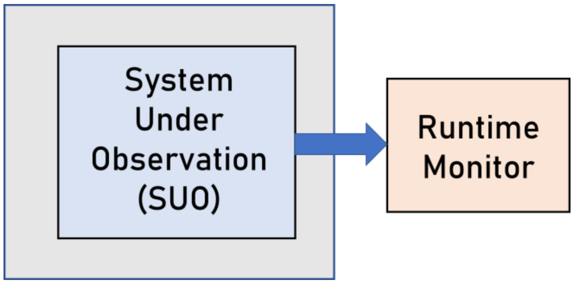


Figure 17: External Runtime Monitor.

### 3.7.2 Organization of Monitors

Organization of monitors is an important research topic and has been discussed extensively in the survey [10], [83] and [17] . Ref [83] classifies the organization of monitors as: (i) Centralized (ii) Decentralized (iii) Orchestrated; and (iv) Choreographed. In the centralized organization the different processes generate a single trace which is observed by one monitor while in the decentralized monitors organization the single synchronized trace is observed by multiple independent monitors [83]. In the orchestrated monitors organization, the traces are produced independently and locally by each processes in a distributed system, but they can be monitored remotely [83]. In contrast, choreographed organization monitors each process locally, obviating the need to transmit all the trace information. Thus, it is more scalable and less vulnerable.

Ref [10] organizes monitors as parallel, sequential, associative and complementary. Parallel monitors verify the properties independently, sequential monitors have a serial composition where the decision of one monitor is defendant on another. Associative configuration of monitors takes monitors that are independent of each other and relate their properties. Complementary configuration of monitors exchange variables and input information to verify a property. Ref [62] presents an attack detection framework called HECAD which follows a hierarchical monitor organization, where there is improved collaboration between the monitors. In the hierarchical organization, the monitor in higher level of hierarchy communicates with the monitor in the lower level of hierarchy for any abnormal behavior of the system, thereby providing a multi-level verification.

In a similar manner as above, we classify monitoring into 3 broad categories:

1. **Monolithic Monitoring** – Here there is a single monitor that looks at the traces from different computational units or processes and is implemented on a single platform (Figure 18). As the complexity of CPS are increasing, such a monitoring scheme may be impractical and inefficient for faster detection of attacks/faults.

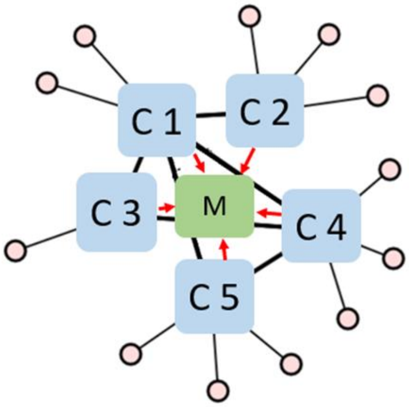


Figure 18: Monolithic Monitoring.

2. **Distributed monitoring** – Here, there is a monitor for each computational unit and the monitors form a distributed network. There is no central decision maker among them and the detection of an attack/fault depends on the consensus based among monitors. In such a scheme, each participant imposes its own requirements ending in a variety of specifications expressed in different formats. Figure 19 shows a distributed monitor organization where C are the computational units and M are the monitors.

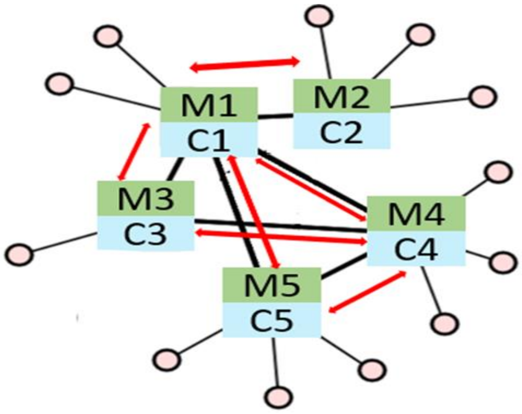


Figure 19: Distributed Monitoring.

3. **Heterogeneous monitoring** – Here, there are localized monitors at each level of computation and integration that do not need to interact with each other as shown in Figure 20. The monitoring consensus is localized and encompasses the diverse nature of the platforms.

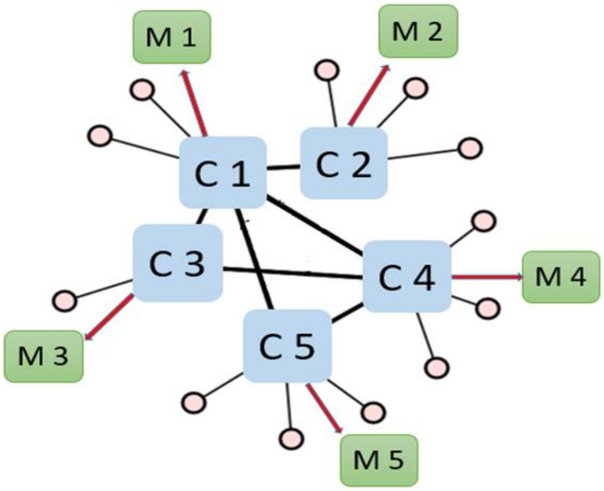


Figure 20: Heterogeneous Monitoring.

The emphasis of this dissertation is to demonstrate the necessity and advantages of having monitors across different levels of a CPS. We chose the heterogeneous monitoring organization with local independent monitors at each level of a CPS as it is more scalable when extended to a large number of computational elements and monitors. But in principle, the concept of multilevel monitors and its advantages over monitoring a single level also holds for another monitor organization.

## 3.8 Practical considerations: realization of monitors

The practical considerations to the design of runtime monitors involve the ability to express the sequence of operations, time dependencies, and complex interactions of a system in the specifications. Runtime verification comes with a rich set of formal specifications languages such as Signal Temporal Logic (STL), Metric Temporal Logic (MTL), past time Linear-time Temporal Logic (LTL) to express complex interactions. However, there is a need for specification languages that allow effective expression with the concept of complex event processing and provide rich verdicts beyond yes/no when a correctness property is verified [18]. One potential way to cater to this challenge is Stream Based Runtime Verification (SRV). SRV combines event processing and runtime verification to handle quantitative data and has the expressiveness to specify complex properties. Additionally, they produce rich verdicts which are valuable.

Furthermore, the tool should allow the synthesis of executable monitors from the specifications, so that they can be implemented on hardware. Tools such as NASA Copilot [84] and TeSSLa [81] allow synthesis of executable code monitors often realized in C or HDL (Hardware Description Language) code, directly from the runtime verification language. In this dissertation, we have used the TeSSLa runtime verification language to realize monitors on hardware.

### 3.8.1 TeSSLa Runtime Verification Language

TeSSLa is a temporal stream-based specification language designed for specifying properties where the timing and sequencing are critical [81]. It supports timestamped events and declarative programming style, which is well suited for expressing specifications. TeSSLa accepts streams of inputs that can be synchronous or asynchronous. Additionally, it can generate the specification in hardware language for implementation on FPGAs.

TeSSLa specification language operates on independent streams of data that are time-stamped. For example, consider three streams of data: p(n), q(n) and r(n). Since these are values of the data at different discrete points in time, we denote a stream as p (1), p (2), p (3) … p(n). Using TeSSLa, we can easily implement conditions such as the following:

$$If\ p(n)\ >\ q(n),\ then\, r(n+2)\ = 5 \qquad (12)$$

This simply means that if at any instant of time the data (value) of stream “p” is greater than that of stream “q”, the value of the data in steam “r” after two instants should be “5”.

Such stream runtime verification (SRV) languages can check logical properties and compute temporal metrics and statistics from the input trace [85]. An example of a TeSSLa specification for a requirement “increase or decrease in position *x* over a time period of one second should be less than or equal to 5 m” is as shown in Figure 21. This condition can be stated succinctly as:

$$|x(n) - x(n-1)| <= 5 \qquad (13)$$

We see that as *x* goes from 1m to 5m between *t* = 0 and 1s and again from 5m to 10 m between *t* = 1 and 2s, this condition is met. Therefore, there is no attack (attack=false). However, as *x* goes from 1 m and 100m between *t* =3 and 4s and again from 100m to 2 m between *t*=4 and 5s, this condition is not met. Therefore, there is an attack detected (attack=true) for two consecutive seconds. A snippet of the TeSSLa specification for this property is in Figure 22. This property was simulated in the TeSSLa simulation tool [86].

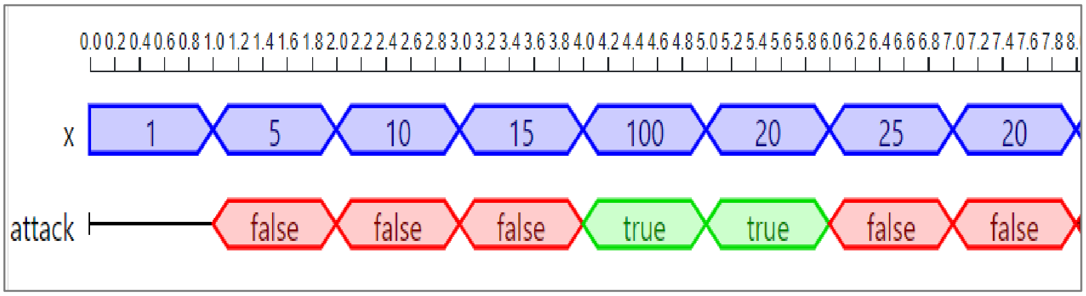
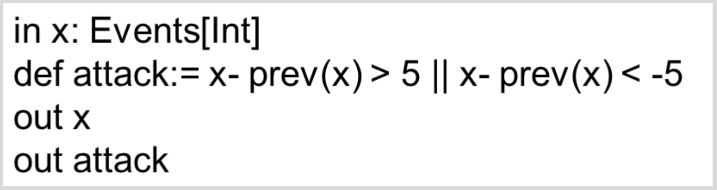


Figure 21: Stream of data indicating position “x” in blue with time stamp (top) and monitor indicating whether an attack has occurred (bottom). Simulated in the TeSSLa simulation tool [86].



```
in x: Events[Int]
def attack:= x- prev(x) > 5 || x- prev(x) < -5
out x
out attack
```

Figure 22: Snippet of the TeSSLa specification.

Consider another safety property to verify the correct functionality of the throttle signal given by an AEB controller “If the Braking signal is non-zero, then the throttle should be zero”. This property ensures that the throttle is not “on” when the braking action is engaged. To verify this property, TeSSLa receives streams of inputs of Brake state and Throttle signals and checks them against the property. TeSSLa output is a stream of the property being evaluated for every event on the input. We use TeSSLa's well defined set of library functions, which can be used to write specifications to verify both timing constraints and event ordering such as an event always preceded or followed by other event(s). The usage of TeSSLa and its library functions are explained in [86] [54] [81].

TeSSLa monitor output verifies the property to ensure the correct throttle action for a given Braking state. As the Braking state becomes greater than zero, the throttle should be zero. However, it fails to do so which is detected by the monitor that indicates "false" implying the specification is violated as seen in Figure 23.

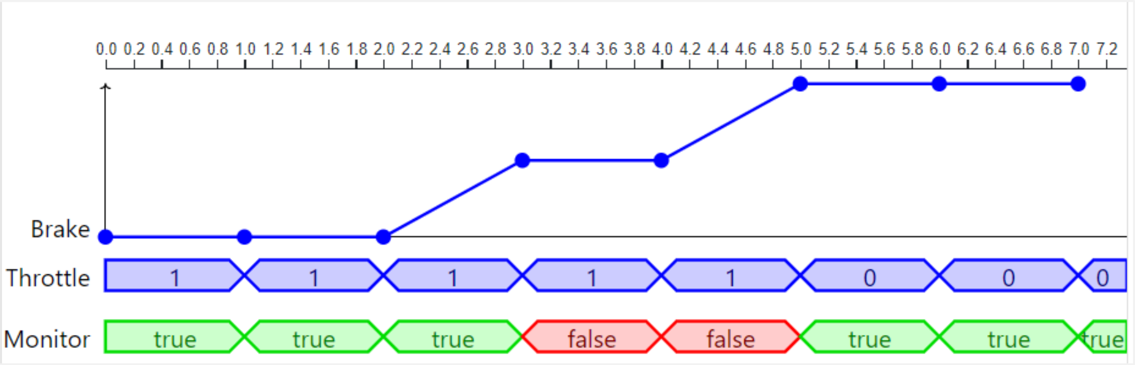


Figure 23: TeSSLa Monitor output that verifies the property to ensure correct throttle action for a given Braking state. As the Braking state becomes greater than zero, the throttle should be zero. However, it fails to do so which is detected by the monitor that indicates "false" implying the specification in violated. Simulation performed in the TeSSLa simulation tool [86].

This property to verify throttle action was implemented on a simulator tool provided by TeSSLa [86]. When the Brake state in not zero, the Throttle has to be zero. But due to a fault in the functionality of the AEB controller, the Throttle remains "1" and its transition to zero is delayed for two clock cycles. The Monitor indicates that the property is falsified for two clock cycles as seen in Figure 23.

We explain a detailed workflow for generating TeSSLa monitors and implementation on hardware in Chapter 7.

## 3.9 Bridging the gap between design time V&V with Runtime Monitoring

While we have described multilevel monitoring, monitor architecture, expressing monitorable properties and a stream-based language for implementation of monitors on hardware, one of the important questions is “what to monitor at runtime”.

In order to understand what to monitor, we need to know what can fail in the system and the failure modes. In t1he survey paper “The Role of Software in Spacecraft Accidents” [87], the author notes that almost all software related failures were due to flaws in requirements. In these cases, although the software behaved as intended, the design behavior was not safe. Such examples emphasize the need to bridge the gap between design assurance methods and system behavior during operational phases which is the main idea of the DepDevOps continuum [88]. Deriving runtime monitoring properties from such incomplete requirements that do not encompass critical system behavior would lead to technically correct but practically worthless monitoring results [50].

Although this has to do with requirements engineering where safety and security requirements are consolidated by systematic methods, we investigate the use of hazard analysis and V&V steps followed during design development to help us refine the initial requirements and guide us on “what to monitor” at runtime.

In this dissertation, we do not perform extensive Hazard Analysis but we do use model-based V&V to derive runtime monitoring properties. Additionally, in Chapter 7 we demonstrate the value of STPA Hazard analysis formulating monitoring properties for an Automatic Emergency Braking system (AEB) for Autonomous Vehicles. As we discuss in later chapters, we have found both STPA Hazard analysis and Model Based V&V to be beneficial in arriving at runtime monitoring conditions (Figure 24).

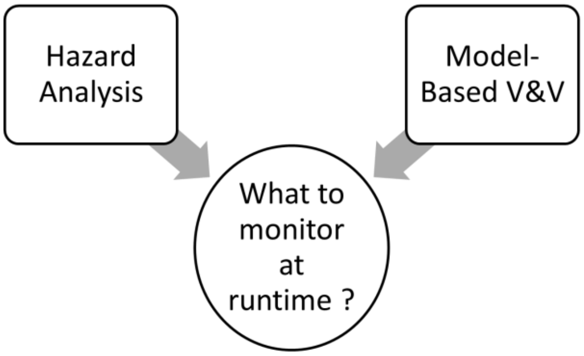


Figure 24: STPA hazard analysis and Model based V&V inform us on "what to monitor" at runtime and placement of monitors.

### 3.9.1 Hazard Analysis

Hazards are events occurring in a system that causes violation of safety property and can thereby lead to a critical failure of the CPS. The risk to a CPS is therefore a combination of [50]:

(i) The probability of a hazard leading to a failure

(ii) The consequence of a hazard (it's criticality)

Typically, risk reduction involves reducing (i) or (ii) or a combination of both. This is achieved through hazard analysis throughout the life cycle of a design.

Preliminary Hazard Analysis (PHA) is performed in the early stages of a design and this involves identifying hazards, ranking them, identifying mitigation techniques, etc. This analysis results in developing a set of system requirements, specification and testing methods that could even impact the choice of system architecture [50] [89] [10].

Coming up with system requirements can be explained with an example of opening/closing of a car's door. If the car starts moving when the door is open, it is a safety issue and one should therefore include a specification, "the car should not move until all the doors are closed". Similarly, another specification could be "the door cannot open when the car is moving". We note that these specifications define safe operating conditions without going into the details of exactly how they can be enforced [10]. Understanding the causal factors of such hazards at design time provides insights on how these causal factors can be monitored in the operational phases, thus bridging the gap between development and operation.

Both forward and backwards search can be used to identify potential faults or failures in a CPS. Example of a forward search technique is Failure Modes and Effects Analysis (FMEA). Here all possible failures in a process/product or in our case, in a CPS are identified. Conversely, a backward search technique named Fault Tree Analysis (FTA) can also be used. Here a hazardous state of the system is specified and the objective is to find all possible paths that could lead to this hazardous or undesirable state [10]. For example, if a car equipped with Autonomous Emergency Braking (AEB) gets very close to a collision, it can be caused by radar/camera sensor error not detecting this or sensor fusion module not working correctly. If all this was found to work well, there may be another path that can result in this failure. That could be, the AEB controller did not function correctly or it did, but the brake physically failed to engage. Such failure modes are analyzed using the FTA method. Other more formal techniques include the Systems Theoretic Process Analysis (STPA) that is based on System-Theoretic Accident Model and Processes (STAMP), which is an accident causality model based on system theory [90] .

**STPA Hazard Analysis:** Systematic hazard analysis methods such as Hazard and Operability Analysis (HAZOPs) or Failure mode and effects analysis (FMEA) assessments are employed at design time to understand the vulnerabilities in a system. However, there is little reporting or research in the literature on how system-level hazard analysis processes are integrated into runtime monitoring design and deployment processes. Such an integrated or coupled framework would play an important role in understanding the safe and unsafe operating conditions in a CPS which in turn provides evidence and insights on runtime monitoring of: (1) what to monitor, (2) where to monitor, and (3) context of the monitoring.

Hazard analysis methods such as STPA emphasizes that safety should not be treated as an after-thought but should be designed into the system [43]. In STAMP and STPA, system safety is reformulated as a system control problem rather than a component reliability problem as in FMEA methods. STPA provides a systematic method to identify unsafe control actions (UCA) (e.g. denial of service or delayed service, the wrong sequence of user inputs, etc.) and understand their effect on the functional safety of a system. Employing such methods not only makes the design safer but also provides key insights on critical elements and properties in a design that we may want to monitor at runtime.

As seen in Figure 25, is an example framework we derived that is anchored by STPA. Referring to Figure 25, the starting point in our workflow is executable models of the CPSs. Models capture the overall system objectives and the interactions among its functions. The first step is Hazard Identification–

Identify threats to safe and secure operations. This is often derived from system requirements formulated by domain experts.

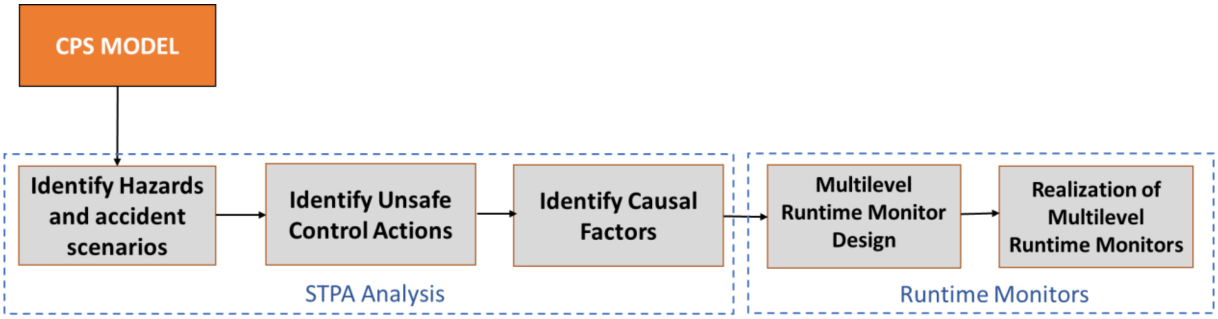

CPS MODEL
Identify Hazards and accident scenarios
Identify Unsafe Control Actions
Identify Causal Factors
STPA Analysis
Multilevel Runtime Monitor Design
Realization of Multilevel Runtime Monitors
Runtime Monitors

Figure 25: Conceptual view of STPA driven Runtime Monitor.

For example, an unsafe distance between two cars can lead to an accident. The next step is to identify Unsafe Control Actions that can lead to hazards, e.g. Delayed Braking can lead to unsafe distance between two cars. The third step is finding causes for the Unsafe Control Actions (UCA). These can be factors that are not related to failures such as fog interfering with LiDAR distance estimation, or they can be due to attacks/faults. Finally, the causal factors of an Unsafe Control Action inform "what" to monitor and "where" to monitor in the CPS hierarchy. Properties are formulated for each of the UCAs in the analysis. We use STPA analysis for identifying causal factors for hazards for an AEB controller example which is explained in Chapter 7.

### 3.9.2 Model-Based V&V

In this thesis we explore how structured verification activities in a Model Based Design and Engineering (MBDE) context help formulate more effective monitoring specifications to cover vulnerable areas in a system operation. We describe how each verification artifact can guide us in refining the initial requirements, from which monitor specifications are formulated.

We show that leveraging synergy between design and runtime verification produces more informed runtime safety monitors in Chapter 4. Although this is not a formal approach for hazard analysis, it nevertheless facilitates systematic design and analysis of systems that can be used to identify potential hazards and faults. This understanding of the underlying factors of the hazards leads to data and information about what monitoring needs to be performed. We demonstrate this approach by verifying an Emergency Diesel Generator Startup Sequencer (EDGSS) implemented on an FPGA overlay. We explain this approach in Chapter 4, where the V&V artifacts expose critical design elements that we want to monitor at runtime.

## 3.10 Other considerations in runtime monitor design

Some of the other aspects of runtime monitoring that are important are monitor coverage and monitor correctness. They answer some important questions pertaining to runtime monitoring such as:

a. Do we have all the properties to maintain safety and enforce security?
b. Can we monitor a property if there is data available?
c. Given a runtime monitor design, does it detect all faults and attacks related to a property?

These questions relate to system safety and security property completeness, data observability by a monitor, and monitor coverage which is essential for effective detection of faults/attacks. Question (a) is a known challenging problem in dependable systems as there may be unknown failure interactions or hazard causal factors in CPS that were not envisioned or considered. We argue the best way to cope with this issue is by employing systematic model-based assessment methods such as STPA coupled with MBDE to arrive at monitoring conditions for a specific system. In this dissertation, we use MBE to derive informed properties to monitor at runtime. It should be noted, in absolute sense question (a) is impossible to guarantee, and most engineering efforts use the ALARA ("as low as reasonably achievable") principle in the management of safety-critical systems - which states that the residual risk shall be reduced as far as reasonably practicable for a given system.

The next question (b) is if we can monitor the properties given the data available (observability). Here, we present an example to show the issues in monitoring properties with limited observability. Let us assume that the maximum acceleration of a car is less than 10m/s$^2$ on a flat surface. When monitoring acceleration based on data from a velocity sensor, we can add a condition “rate of change of velocity < 10 m/s$^2$”. Thus, if we detect a large rate of change in velocity (e.g. velocity changes by 2 m/s in 0.1 s) the data monitor detects that there is either a fault or an attack on the velocity sensor. However, the acceleration could be higher, if the car is on a downward incline, so one may be more conservative and change the condition to “rate of change of velocity < 14 m/s$^2$” to avoid a false positive.

Consider this car going uphill with an incline at which the maximum acceleration is physically limited to 5 m/s$^2$. However, our condition does not detect an error unless the acceleration equals or exceeds 14 m/s$^2$; in other words, the fault or attack that causes the acceleration to be more than twice what is physically possible at that uphill incline will go undetected. To correct this monitored property for the incline of the road, we have to be able to detect the incline of the road, for which there may be no sensor (i.e. the incline is not observable). Of course, one could argue that we just add another sensor or we infer the incline from the calculated torque, but these inclusions come at a cost – they require more integration of data.

Hence, it may not be possible to monitor all of the relevant aspects of properties due to limited sensors or data sources to provide appropriate information. However, one can monitor critical properties to detect most anticipated faults/attacks based on the observable data.

The next question (c) is that given a monitoring property, can we ensure that it can certainly detect faults/attacks pertaining to that property? We can to certain degree by using novel methods such as model checking and property based fault injection as described in [91]. Fault saboteurs can be injected at various points in the system and property-based fault injection can be performed to ensure that all faults can be detected for a given property. Property based fault injection is based on model checking so it is exhaustive if the state space of the model is searchable by the model checker (e.g. the state explosion problem). In this method, fault saboteurs are inserted systematically in the model and faults are injected when the system state matches a particular property. It can be verified that a violation of safety and security condition is detected by the runtime monitor, due to this injected fault. The more comprehensive the property-based fault injection, greater the assurance that designed monitor can detect all faults/attacks for a set of anticipated fault models and attacks. Property based fault injection methods help us verify monitor coverage for large classes of faults/attacks.

## 3.11 Towards a Systematic Framework for Multi-level Monitoring

The goal of this chapter was to study all of the major factors that influence the design and development of well-formed runtime verification monitors. These factors helped us formulate the monitoring framework in Figure 26. Section 3.3 laid the foundation, explaining the characteristics of a critical system, namely dependability, reliability, availability, safety and security. These attributes can be violated due to attacks/faults that occur across different layers in a CPS as explained in section 3.3.

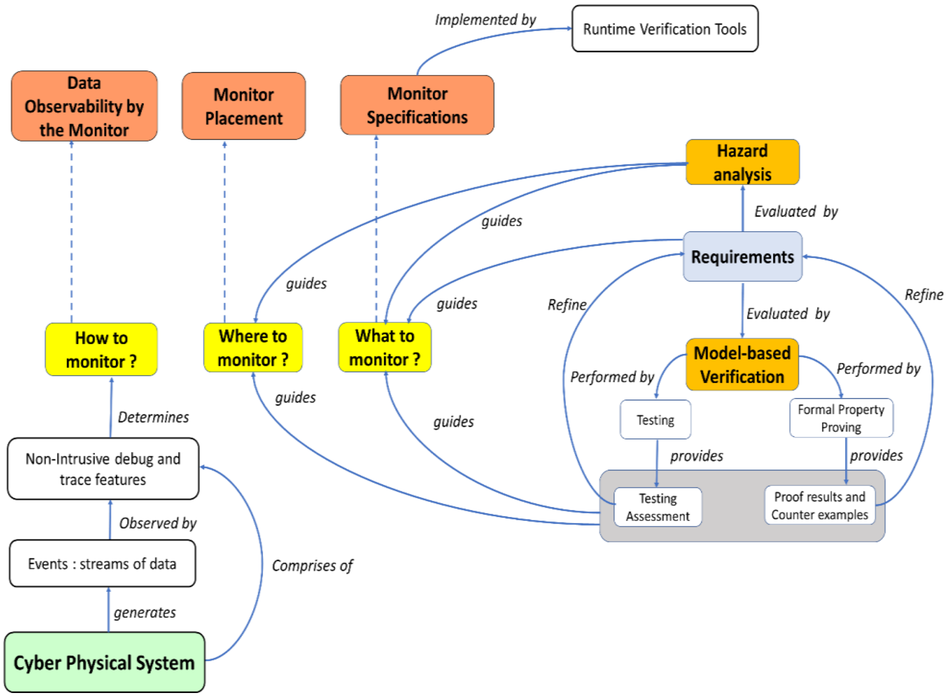


Figure 26: Runtime monitor framework.

Section 3.4 developed a well-formed abstract model of multi-level monitoring that characterizes how monitors interact with the CPS and defines the concepts streams, properties, and monitors. Monitor placement and monitor specifications are guided by verification and hazard analysis of a system. Section 3.9 explains how hazard analysis methods such as STPA and model-based verification help us identify the possible reasons for failure in a system, thereby, they guide us on 'what to monitor' and 'where to monitor' in a CPS. In this dissertation we try to understand the benefits of using model-based V&V and STPA hazard analysis to bridge the gap between design time assurance and runtime behavior of a system. Section 3.8 discussed the practical aspects of realization of monitor specifications on hardware.

Various aspects of this framework are discussed in the future chapters in this dissertation. In chapter 4, we discuss how model-based verification guide us on formulating monitoring properties. In chapter 5, we demonstrate the benefits of multilevel monitors and placement of monitors. In chapter 6, we discuss the non-intrusive debug and trace features in ARM processors which determine 'how we monitor' a CPS, based on instrumentation, embedded trace etc. In Chapter 7, we leverage the knowledge gained by STPA analysis of an Autonomous Emergency Braking (AEB) system to determine placement of monitors and monitoring properties. Additionally, we synthesize runtime monitors from the monitoring specification using a stream-based runtime verification language.

# Chapter 4

# Synergy between Design Assurance using Model based Engineering and Runtime Verification

## 4.1 Introduction and purpose

Ensuring the safety and security of high integrity CPS applications is a difficult task. Rigorous verification and design assurance strategies such as testing and formal verification are especially important for safety-critical embedded real-time systems where the impact of failure or malicious exploits is significant. In these situations, stringent design assurance V&V methods along with standards and regulatory guidelines (e.g. IEC 61508, DO-254, ISO- 26262) are used to provide high levels of assurance evidence to confirm that the systems are safe or security failures are mitigated to "as low as reasonably achievable" standard. One of the challenges with respect to runtime verification as noted in [28], is how to extend (or connect) design assurance to runtime implementation in a more methodical way. This underlies the key idea of DepDevOps, and in the context of this dissertation provides a continuum from design time assurance to runtime monitoring.

In this chapter, we provide specific insights into "bridging the gap" from design time to runtime from a model-based engineering perspective. We believe a model-based engineering approach to addressing this problem may provide evidence towards general approaches for exploiting the synergy between design time and runtime verification activities. We explore how structured verification activities in a Model Based Design and Engineering (MBDE) context help formulate more effective monitoring specifications to cover vulnerable areas in a system operation.

*We assert that leveraging synergy between design and runtime verification produces more informed runtime safety monitors. By "synergy", we mean the following: The combination of design verification and runtime monitors increases the efficiency of runtime monitors by identifying critical ways and places in the design where a system might fail. Such critical conditions and complex interactions have to be monitored at runtime. If the runtime monitors are not informed by design verification, the benefits of employing such monitoring over and above design verification may be incremental. In contrast,* *employing design verification with informed runtime monitors would far exceed the benefits of applying each of these individually.*

This chapter explores such a synergy between design time verification and runtime monitors. Initially, design assurance using MBE is explored using an IEC-61508 compliant verification workflow. Next, we explore how the design V&V can help formulate better runtime monitoring conditions.

Organization of this chapter is as follows: We first talk about Model-Based Engineering (MBE) and some of it's basic elements. Then we talk about the connection between design time verification and runtime monitor development and show synergetic elements between the two, through a workflow diagram in Section 4.3. Our representative system to study synergy was based on the verification performed on an Emergency Diesel Generator Start Up Sequencer (EDGSS) implemented on a FPGA overlay architecture called SymPLe. We explain the details of this representative system in Section 4.4. Section 4.5 presents an IEC 61508 compliant V&V workflow to establish interaction between design verification artifacts and monitor design. In this section and in Appendix A we provide few examples to show synergy between V&V and runtime verification. An example of implementation of runtime monitors for the properties derived after exploiting synergy is explained in Section 4.6. In Section 4.7 we summarize the key findings of our study and how this guided us on 'what to monitor' and 'where to monitor'.

## 4.2 Model-Based Engineering

Model-based design and engineering (MBDE) involves elevating models in the engineering process to a central role in the specification, design, testing, and integration of a system thereby offering a full platform for system development and verification [92]. MBDE tools have been used in many applications including automobile, avionics and other safety critical industries as they offer the promise of end-to-end "concept to design to realization to verification". It is claimed that with MBDE approaches errors are found earlier in the development life cycle, as compared to traditional approaches, where the impact is less consequential to cost. For example, practitioners have reported productivity gains of 4 to 10-fold when using MBDE tools [93]. While these claims have been evidenced and demonstrated in various industries, there is little or no reporting in literature on the synergy between MBDE and runtime verification and its benefits.

The phrase “Model Based Design and Engineering” has been used in many contexts to the point where it’s meaning and purpose is somewhat unclear. For the purposes of this dissertation, we use a more refined definition taken from the systems engineering and the formal methods community.

**Definition:** “*Model-based Design Engineering (MBDE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle”* [94].

To support design, analysis, and verification activities of CPSs the systems engineering community stresses the importance of *executable models (i*.e. models for which timed behavior can be generated). Execution models for Cyber Physical Systems (CPS) and embedded control applications require discrete-time, continuous-time and discrete-event representations to capture all of the possible interaction behaviors between computer elements, sensors, networks and physical elements. Finally, from an embedded software design perspective, a high value proposition is automatically generating corresponding high-level code (C code or VHDL code) from functional model representations in the MBDE environment. In Model Based Design Environments like MathWorks Simulink, an initial executable graphical model represents the software or hardware component under development by including appropriate design details while ignoring the details of the underlying software implementation. The model is then refined until it is sufficiently complete to serve as the design to a deployable implementation through automatic code generation.

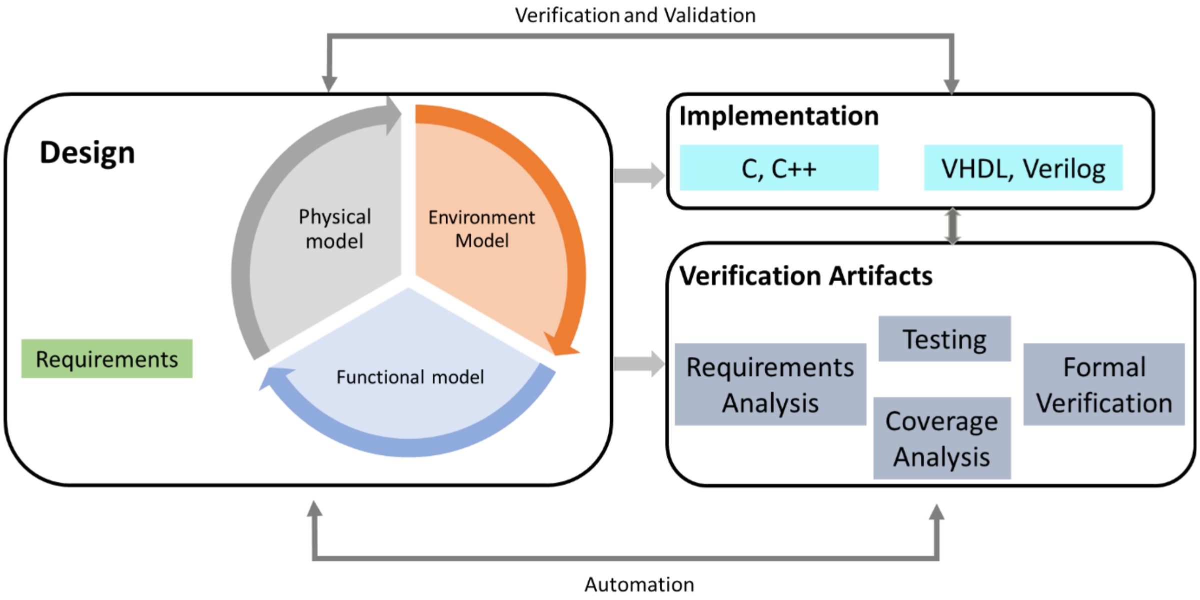


Figure 27: Basic elements of a Model Based Engineering design. (inspired by [95]).

Model Based Design and Engineering as described above offers specification aspects as well as design, analysis and implementation aspects in single framework as seen in Figure 27. In comparison with traditional software development that has a clear separation of phases, in Model-Based Design a continuous integration of specification, design and implementation phases is the norm. Moreover, the same modeling notation is typically used throughout the consecutive development phases. There are various generalized claims on the benefits of MBDE for Software [96], and our position in this dissertation is to provide results and experiential findings that show the synergy between MBDE design artifacts and runtime verification needs.

## 4.3 Connection between Design time Assurance and Runtime Verification

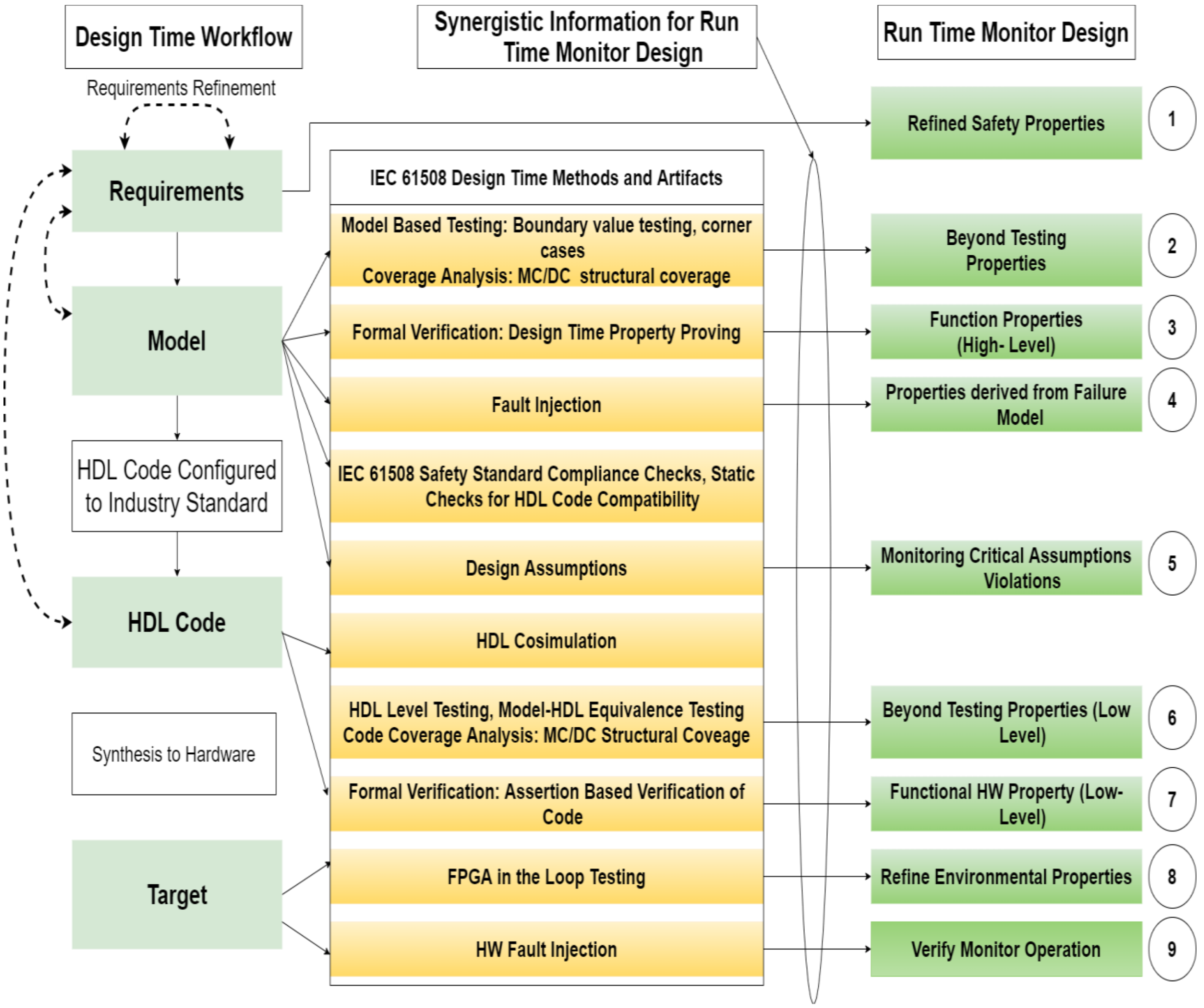
As we discussed above, MBDE is an excellent framework for examining relation between design time and runtime verification activities. This is largely due to the seamless integration between requirements, models, testing, and implementation. In this work, we wanted to reflect on how design and verification artifacts could aid in runtime verification monitors and where those artifacts might arise in the design process. Our representative Cyber Physical System that we are using to study synergy is in the context of Nuclear Power Instrumentation and Control (discussed in Section 4.4). Therefore, the verification artifacts used for design assurance were as per IEC 61508 guidelines – which is used by the Nuclear Industry (I*EC 61508 "Functional safety of electrical/electronic/programmable electronic safety*-*related System*"). In addition, IEC 61508-3 recommends the use of runtime monitoring for SIL 3 and 4.

IEC 61508 is a governing standard for industrial functional safety supporting the design, development, and operation of programmable electronic systems (often referred to as PLCs). IEC 61508 standard is important to our research for several reasons; (1) because it is like a "mother" standard, as many other safety critical system standards like ISO 26262, IEC 61513 (nuclear 61508) are heavily influenced by 61508-3; (2) it presents a realistic representation of design assurance requirements for a large body of critical CPSs. We set out to examine where runtime monitor design can be influenced and aided by the design time artifacts produced from a model-based engineering effort using verification performed as per IEC 61508 guidelines. Additionally, we hoped to see where assumptions made at design time may need to be checked at runtime.  The workflow in Figure 28 links the V&V activities performed on a model to runtime monitor design.

Two key aspects of this workflow are end-to-end traceability, and refinement of high-level models to low level code (HDL). Requirements are checked all the way down to the HDL code. Refinement of high-level requirements occurs when testing and V&V activities find incomplete, ambiguous requirements. In the middle section of Figure 28 are the design time assurance methods we employed to comply with IEC 61508 SIL 3 and 4.  Each of these design assurance activities produced a set of evidence, which was linked to requirements by the traceability tool in Simulink. The right side of Figure 28 depicts how synergistic information flowed from the design time activities to runtime monitor design. The *Runtime Monitor Design* shows how we derive properties to monitor at runtime in a methodical way as we perform V&V steps. The numbers on the Figure 28 represent roughly the workflow of our runtime monitor design activities. Although it depicted as linear flow, the process was iterative, where designs were refined as requirements were consolidated. Note that for each design time V&V activity, various types of runtime monitor properties can be arrived as shown in Figure 28. We classify the monitor properties derived from various stages of V&V into 9 categories:

1. Monitor properties derived from safety requirements.
2. Monitor properties derived from beyond testing conditions.
3. Monitor properties derived from functional behavior of the system.
4. Monitor properties based on insights from fault injection.
5. Monitor properties derived from design assumption.

6. Monitor properties derived from beyond testing conditions at the code level.
7. Monitor properties derived from low level HDL verification of system functionality.
8. Monitor properties refined after consideration of environmental factors and variations.
9. Monitor properties refined after fault injection on hardware.



Design Time Workflow
Synergistic Information for Run Time Monitor Design
Run Time Monitor Design
Requirements Refinement
Requirements
Model
HDL Code Configured to Industry Standard
HDL Code
Synthesis to Hardware
Target
IEC 61508 Design Time Methods and Artifacts
Model Based Testing: Boundary value testing, corner cases
Coverage Analysis: MC/DC structural coverage
Formal Verification: Design Time Property Proving
Fault Injection
IEC 61508 Safety Standard Compliance Checks, Static Checks for HDL Code Compatibility
Design Assumptions
HDL Cosimulation
HDL Level Testing, Model-HDL Equivalence Testing
Code Coverage Analysis: MC/DC Structural Coveage
Formal Verification: Assertion Based Verification of Code
FPGA in the Loop Testing
HW Fault Injection
Refined Safety Properties
1
Beyond Testing Properties
2
Function Properties (High- Level)
3
Properties derived from Failure Model
4
Monitoring Critical Assumptions Violations
5
Beyond Testing Properties (Low Level)
6
Functional HW Property (Low-Level)
7
Refine Environmental Properties
8
Verify Monitor Operation
9

Figure 28: V&V workflow to explore synergy.

1. ***Monitor properties derived from safety requirements*** – These properties enforce a safety requirement dictated by the safety envelope of the application. This is the classic perspective view of runtime monitors where requirements are translated into runtime monitorable properties.

2. ***Monitor properties derived from beyond testing conditions*** – Design time testing may have been limited due to complexity, or where "corner" cases revealed unexpected behaviors or interactions,

which provide insight into establishing the runtime monitor requirements, in addition, these types of properties may cover beyond expected operating conditions or design basis events.

3. ***Monitor properties derived from functional behavior of the system* –** These embody the permissible properties or demanded functionality of the application with respect to its expected operating environment.

4. ***Monitor properties based on insights from fault injection* –** Fault injection is essential to verify that the monitorable properties are able to detect safety violations. They give insights into critical vulnerabilities and assist in deriving properties that are critical and should be monitored at runtime.

5. ***Monitor properties derived from design* assumptions –** These are very important to safety critical systems. Assumption monitors observe conditions that are related to assumptions that were made at design time. Often these assumptions are related to how the application behaves, given the hardware is operating as designed.

6. ***Monitor properties derived from beyond testing conditions at the code level* –** This is similar to point 2 (except that was at the model level) and involves testing that reveals corner cases and critical design elements that could be important to monitor at runtime.

7. ***Monitor properties derived from low level HDL verification of system functionality* –** These observe critical timing and state behavior at the hardware level, which is usually abstracted away or simplified at the higher levels of model functionality. As an example, a functional clock in Simulink is timeless "tick", but when translated into VHDL code it has to be a well-behaved clock signal with appropriate phases and set up times.

8. ***Monitor properties refined after consideration of environmental factors and variations* –** When the design is implemented on a hardware platform such as FPGA, monitor properties can be refined to include factors such as variations due to signal fluctuations, clock glitches, etc.

9. ***Monitor properties refined after fault injection on hardware to detect safety violations* –** This helps verify the effectiveness of runtime monitors in detection of safety violations. If any fault injected is not detected, the monitor properties are refined to include more checking conditions. This is also essential to ensure monitor completeness.

Our investigation was based on the V&V activities performed on an Emergency Diesel Generator Start Up Sequencer (EDGSS) application. The EDGSS was implemented on a FPGA overlay architecture called SymPLe [97] . We discuss more about our representative system EDGSS and SymPLe in Section 4.4. Our investigation is based on the following conditions:

1. We had access to the design models and V&V data for the SymPLe FPGA architecture (e.g. it was designed in-house). As such, we could examine and analyze critical aspects of the architecture with respect to the application (EDGSS).
2. We understood the model-based design workflow that produced the SymPLe FPGA architecture and the EDGSS application. Thus, we had a good understanding of design evidence and artifacts used to verify the SymPLe architecture, and the extent of their validity.
3. An end-to-end verification was performed on both the EDGSS application and the SymPLe architecture. The verification effort began with system requirements, progressed to functional models, testing models, and then finally automatically generated Hardware Description language (HDL) code which was targeted for FPGA implementation. Accordingly, we could bi-directionally trace high level requirements to low level design requirements/specifications.

## 4.4 Representative System to Explore Synergy

While 4.1 describes the general idea and motivation for exploiting the synergy between design time and runtime verification, it is necessary to select a specific example to elucidate this concept further. Towards this end, we use an Emergency Diesel Generator Start-up Sequencer (EDGSS), which is a critical part of Nuclear Power Plant (NPP) emergency power backup systems, responsible for maintaining electrical power to safety functions such as reactor cooling, coolant pump circulation, and preventing Main Control Room blackout. We also briefly describe an FPGA overlay architecture called SymPLe on which the EDGSS was implemented.

### 4.4.1 Emergency Diesel Generator Start Up Sequencer (EDGSS)

EDGSS applications are highly safety critical (IEC 61508 SIL 4). There is a large amount of sequence, priority and time dependent logic related to the diesel generator control and part of it is shown in Figure 29. Figure 29 shows a high-level model of the EDGSS. The complexity of such as system makes it more vulnerable to failures at runtime. This case study is based on high-level design documentation that does not take into account the redundant implementations of the system and related voting logic that would be realized in the final system [97].

The EDGSS has 14 inputs that determine the state of the outputs, Engine Start Signal and Open Air Start and Fuel Valves as seen in Figure 29. The basic operation of the EDGSS is as follows: If none of the restrictive signals are asserted (e.g. engine trouble signal, low water pressure signal etc.), then after the air start valve and the fuel valve are opened, the diesel begins to crank, powered from a compressed air supply. If the diesel engine is successful in starting, the engine speed will exceed the crank speed and the signal to the air start valve is activated. If the diesel engine is not successful in starting, the signal to the air start valve and the fuel valve is de-activated. In this case, after a certain delay, the restart sequence can be initiated for subsequent attempts. A more detailed description of the system can be found in Ref [98].

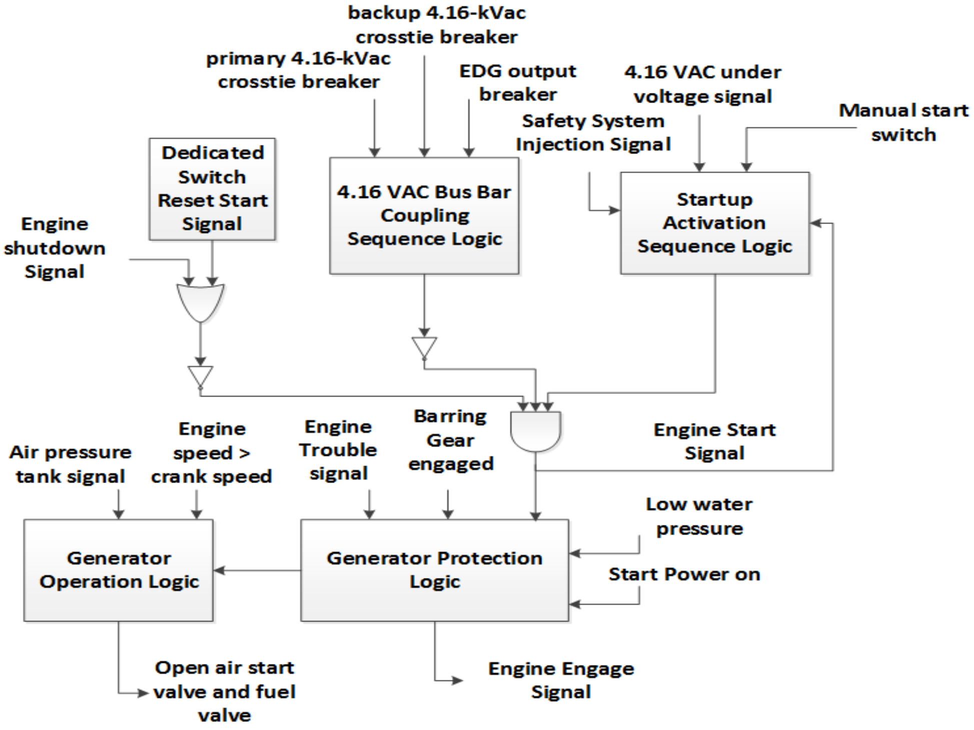
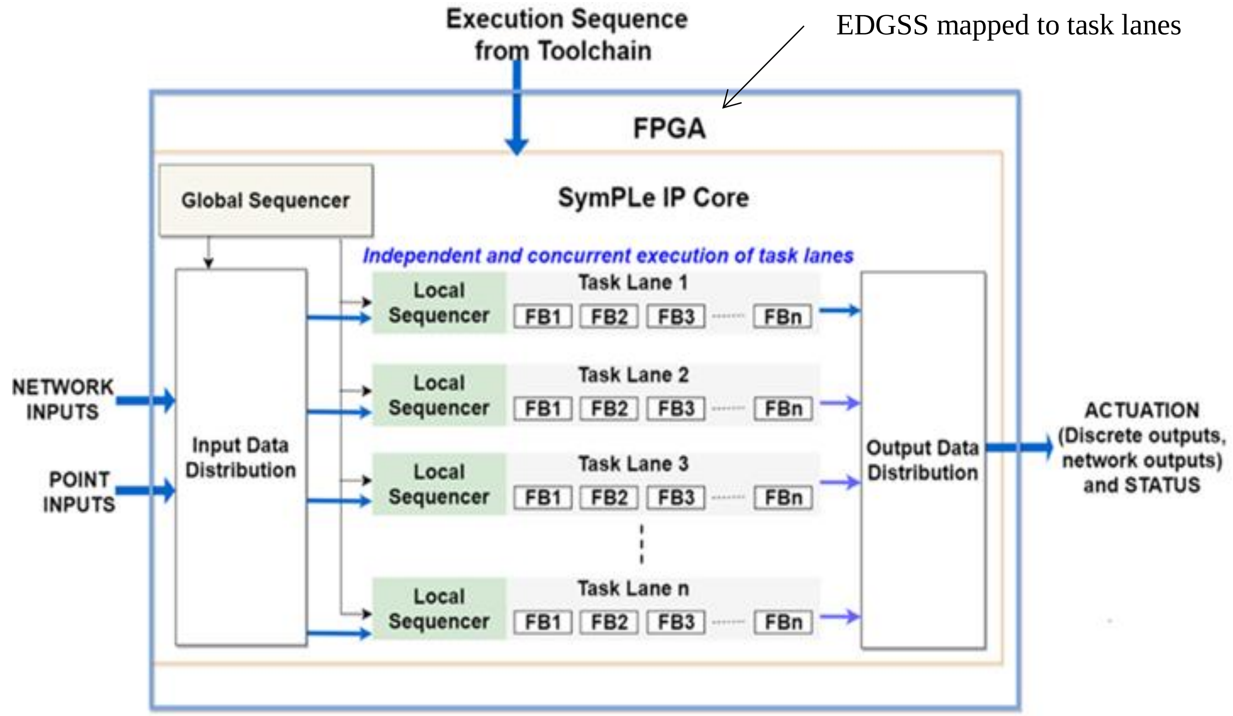

backup 4.16-kVac crosstie breaker
primary 4.16-kVac crosstie breaker
EDG output breaker
4.16 VAC under voltage signal
Manual start switch
Safety System Injection Signal
Dedicated Switch Reset Start Signal
4.16 VAC Bus Bar Coupling Sequence Logic
Startup Activation Sequence Logic
Engine shutdown Signal
Engine Start Signal
Air pressure tank signal
Engine speed > crank speed
Engine Trouble signal
Barring Gear engaged
Low water pressure
Start Power on
Generator Operation Logic
Generator Protection Logic
Open air start valve and fuel valve
Engine Engage Signal

Figure 29: High Level model of the Emergency Diesel Generator Startup Sequencer.

### 4.4.2 SymPLe: An FPGA overlay architecture

The EDGSS was hosted on a FPGA overlay architecture called SymPLe [25]. Referring to Figure 30, the SymPLe architecture is comprised of three basic control hierarchies: the global sequencer, local sequencers or tasks, and a complete set of Function Blocks (FB) per task lane. In SymPLe, all executions occur in task lanes. These are independent processing stations where function blocks organized as function block programs are executed. Functions blocks assigned to a task lane are scheduled via deterministic sequence of task executions. Functions blocks receive data, operate on that data as per the intended functionality, and provide results in their output registers. The global sequencer is mostly concerned with scheduling of task lanes and the marshaling of data I/O in the architecture. The local sequencers' function is to locally coordinate the triggering of a task lane and marshal data to function blocks while executing.



EDGSS mapped to task lanes

Figure 30: EDGSS implemented on SymPLe architecture.

The choice of a FPGA implementation for SymPLe was motivated by the fact that FPGAs are the most suitable replacement choice for older analog NPP equipment and the sequential logic aspects of the EDGSS are a good fit for FPGAs. At the broadest stance, the SymPLe overlay architecture emulates an execution model similar to that of a Programmable Logic Controller (PLC). Meaning, SymPLe is programmed by graphically connecting together pre-verified function blocks in Simulink that constitute a Function Block (FB) diagram program. These FBs are then compiled into an execution sequence depending on the application, which is loaded onto SymPLe. It should be noted that SymPLe architecture design was verified with respect to IEC 61508 SIL 3-4, as such it is a representative example of a safety critical platform [25]. In this work, we wanted to reflect on how design and verification artifacts could aid in runtime verification monitors and where those artifacts might arise in the design process.

In the sections below, we give specific examples from SymPLe architecture that assist us in monitor design. We first explain the V&V activities with example to the SymPLe architecture. We later discuss the synergy points that help in runtime monitoring.

## 4.5 Design Assurance using Model Based Engineering (MBE)

In this section, we describe a V&V workflow using MBE tools to understand and explore how design time assurance can be extended to runtime. We discuss V&V activities performed on the EDGSS and SymPLe architecture to inform the connections between design time assurance and runtime verification. The following features offered in MBE help achieve design assurance:

***Bi-directional traceability*:** Bi-directional traceability refers to the ability (by tools) to relate or connect requirements to lower level artifacts like design specifications, low level requirements, models and code. MBD offers bidirectional traceability at each of these incremental levels in design development (between requirements, model, code, hardware implementation) thereby offering a chain of evidence to support verification. Traceability is one of the important criteria to ease the burden of qualification and offer design assurance for usage in safety critical systems.

***Semi-formal and formal methods* -** Formal verification does not need any inputs like simulation-based verification. Critical conditions in the design are written as properties which are formally verified. Formal verification checks the entire state space and proves or disproves the property. They mathematically ensure that there is no undesirable behavior due to system flaws. Usage of semi-formal and formal methods are some of the highly recommended measures for SIL level 3 and 4.

***Static Verification*:** Static verification of the model and code help identify issues such as overflow/underflow errors, array out of bounds errors etc. early in the development cycle. These design flaws are difficult to detect in manual coding and can manifest as security vulnerabilities at runtime.

We have used MathWorks Simulink tools such as Design Verifier (DV), Simulink Test and HDL coder which are IEC 61508 certified and are used extensively in safety critical industries[99] . Additionally, we extend the formal verification of the model to further verify the HDL code using Mentor Questa [100]. The Questa tool provides formal verification at lower levels of the design (HDL code) - typically synthesizable to FPGA or ASIC. Questa has been widely used in verification of safety critical systems [101]. These two tools are seamlessly used to provide end-to-end verification.

### 4.5.1 Design Assurance Workflow

An overview of the V&V workflow (Figure 31) that describes verification at each stage in the design process is explained below. V&V efforts explained in this section are not application specific and can be followed for any model-based design that can eventually be implemented on a hardware platform (e.g. processors, FPGAs). We use the IEC 61508 standard to formulate the V&V workflow, due to its widespread use in the safety community. But these verification steps can be generalized for other safety standards such as ISO 26262, DO 178 for usage in safety critical applications.

The workflow is divided into two sections, V&V at the model and V&V at the code. If the code is automatically generated by the model, the V&V workflow followed for the code helps establish equivalence with the model.  The V&V efforts seamlessly connects verification efforts of the model with the code. These efforts help formulate a methodology that can be used to comply with IEC 61508 safety standard.

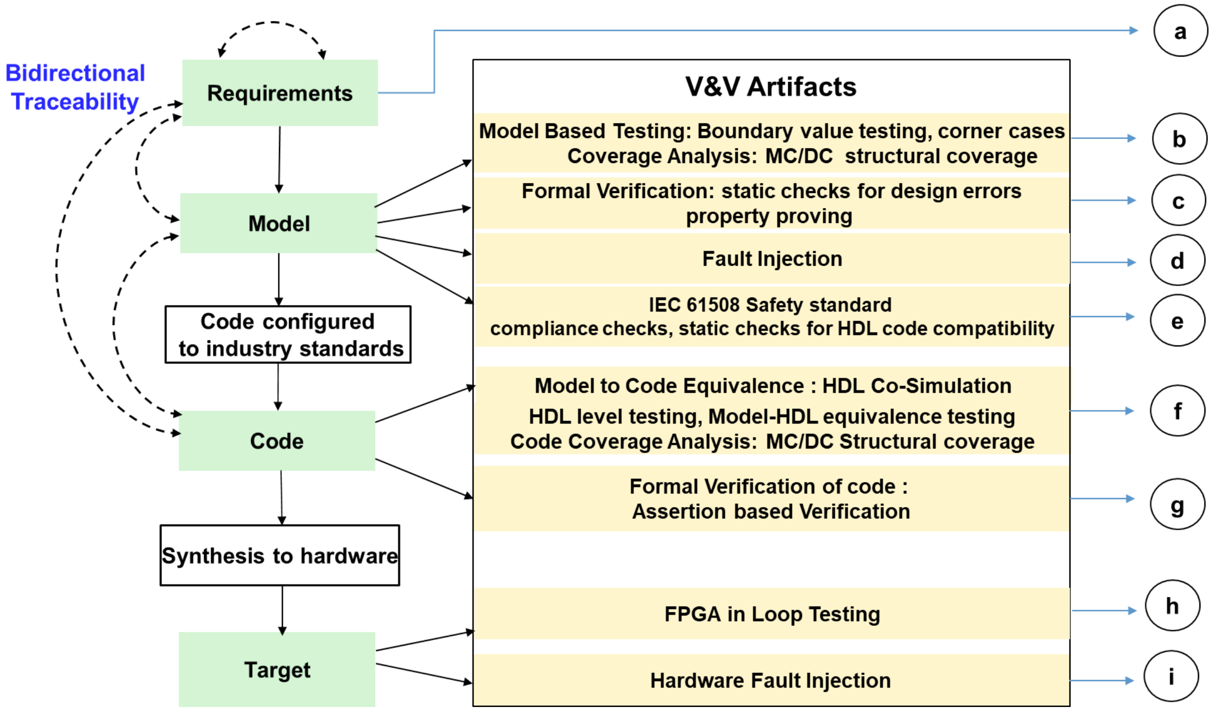

Bidirectional Traceability
Requirements
Model
Code configured to industry standards
Code
Synthesis to hardware
Target
V&V Artifacts
Model Based Testing: Boundary value testing, corner cases
Coverage Analysis: MC/DC structural coverage
Formal Verification: static checks for design errors
property proving
Fault Injection
IEC 61508 Safety standard
compliance checks, static checks for HDL code compatibility
Model to Code Equivalence : HDL Co-Simulation
HDL level testing, Model-HDL equivalence testing
Code Coverage Analysis: MC/DC Structural coverage
Formal Verification of code :
Assertion based Verification
FPGA in Loop Testing
Hardware Fault Injection
a
b
c
d
e
f
g
h
i

Figure 31: V&V workflow.

In Section 4.3 we identified nine instances in the V&V process, where we could leverage the lessons from design assurance to design runtime monitors. We provide examples for a few of those instances below. More examples of instances for synergy and details of the V&V process are in Appendix A.

### a. *Example 1: Requirements Analysis*

*Provides an example of monitor properties derived from safety requirements as shown in point-1 of Figure 28.*

Requirements provide a comprehensive set of characteristics that have to be satisfied by a design. Thus, the V&V process at the model starts with the requirements analysis and they are iteratively consolidated during the process of Model-Based Testing (MBT) and coverage analysis. As requirements serve as the backbone for design and development and the base reference for verification and validation activities, they need to be easily understandable. In addition to being free from ambiguities, the safety requirements need to be free from having contradictions and adverse interactions with the non-safety function requirements. Critical safety requirements are formulated as runtime monitors.

The design requirements are classified into high level requirements and mid/low level requirements. High level requirements depict the general properties of the design. Mid/low level requirements indicate the detailed implementation necessities of the design. Verification is performed on the model based on these requirements. Therefore, it is important to have well defined, clear and complete set of requirements that include safety and security requirements of the design so that some of the more critical requirements can also be monitored at runtime.  The bi-directional traceability feature in MBE help link each requirement to the implementation.

Here we state a few safety requirements of the SymPLe architecture and EDGSS.

***Requirements for the components in SymPLe Architectures***

1. The *Done* and *Busy* signals cannot be high at the same time.
2. *Latch_input, Latch_output, Execute* can be high only for one clock cycle.
3. *Latch_input* and/or *Latch_output* is never high simultaneously.
4. When execute signal is high, *Write_output* cannot be high.

***Requirements for EDGSS***

1. When *Engine Shutdown* signal is True, the *Engine Start* Logic should go False within T seconds.
2. When *Engine Shutdown* signal is True, the *Open Air Start Valve* and *Fuel valve* should go False within T seconds.

3. When *Low Jacket Water Pressure* is True, the *Open Air Start Valve* and *Fuel valve* should go False within T seconds.
4. When *Engine Speed* > *Crank Speed* is True, the *Open Air Start Valve* and *Fuel valve* should go False within T seconds.
5. When *AirTank Pressure* < *Specified Value* is True, the *Open Air Start Valve* and *Fuel valve* should go False within T seconds.

### *b. Example 2: Model Based Testing (MBT) and Coverage Analysis*

*Provides an example of monitor property derived from beyond testing conditions as shown in point-2 of  Figure 28 and monitor properties derived from design assumption as shown in point-5 of  Figure 28.*

High level requirements are often decomposed and encoded into a design at lower abstraction levels. These lower level specifications are usually what we want to check with respect to runtime verification. However, it's easy to miss or overlook a checking condition if we don't have a complete perspective of the decomposition of a high-level requirement. Testing and formal verification helps us give better understanding of the design and thereby helps write informed runtime monitoring specifications.

MBT involves five major steps namely: Requirement and Model Analysis, Test Plan Preparation, Test Specification, Test Execution and Test Results Analysis. Initially, the requirements of the design and the model are analyzed and the behavior of the system is studied. A test plan is prepared next to identify the properties of the system that we want to verify by writing test cases. Thereafter, test cases are specified. The test case specification defines the detailed test procedure that involves describing the sequence of test steps. The test inputs to be fed into the component and outputs to be verified for testing the requirements are identified. The test procedure gives the sequence of test input values and the timing of feeding them into the component. Lastly, the test cases are executed and the results are analyzed. Simulink offers features such as Test Manager, Test Harness and Test Sequence/Assessment to assist in testing the model and analyzing coverage. Figure 32 shows the system under test with test sequence blocks where we define the test cases and test assessment block where we verify the results of testing.

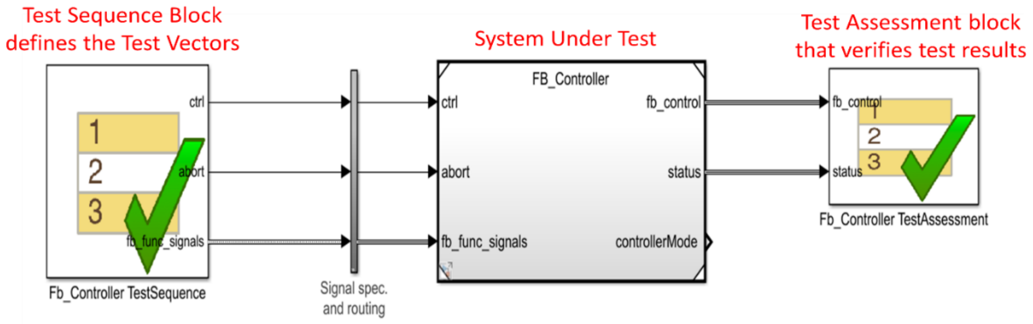


Figure 32: Simulink System Under Test with Test Sequence block and Test Assessment block.

Test cases are formulated from the system requirements for each component of the model, test cases are executed and verified for the model behavior [102]. MBT involves unit testing and integration testing. In unit testing, each component is tested to ensure that they adhere to the requirements. Integration testing is performed to ensure that the interaction among the components is as expected and the interface between components have the same data type and signal ranges. A coverage report is generated showing the model coverage metrics. Coverage helps measure the degree of execution of a model or a code when a test case is run [103]. Model Coverage metrics include execution coverage, decision coverage, condition coverage and MCDC coverage [103]. The main motive behind measuring the test coverage are:

- to determine dead logic branches
- to determine if sufficient test vectors have been created
- to determine if existing requirements are sufficient.

More details of model-based testing and coverage analysis of SymPLe are in Appendix A.1. Below are two cases where MBT guided runtime monitoring.

### *Unexpected interaction found during MBT that we want to monitor at runtime (point-2 of Figure 28).*

During the integration testing performed on the SymPLe architecture, a dead lock situation was found between two components in the architecture. The Local Sequencer and the FBController in SymPLe have a close interaction where a state flow transition in one component affects the other. There are two signals “ctrl” and “state” which are used to co-ordinate the interaction between the Local Sequencer and the FB Controller. The timing and sequencing of these signals has to be correctly coordinated to ensure proper functioning of these components. During integration testing performed in Ref [91] it was found that the system goes to a deadlock situation where each component waits for a trigger from the other component to traverse through all the states. The Local Sequencer is stuck at Fetch state and FB Controller is stuck in Execute state and they don't transition further to other states due to timing inconsistency between the "ctrl" and "state" signals. The details of this testing scenario is explained in Ref [91]. Figure 33 summarizes this deadlock scenario. Although this problem was addressed in the design, such complex interactions revealed during MBT have to be monitored at runtime.

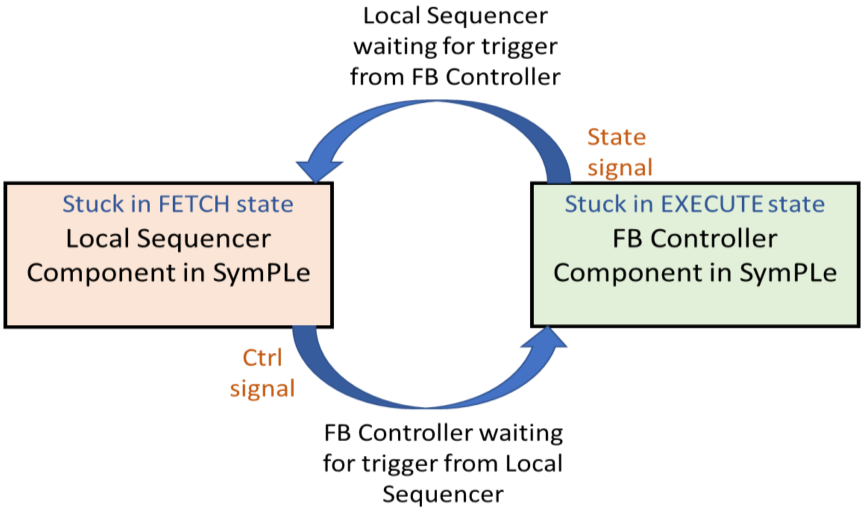

Local Sequencer waiting for trigger from FB Controller
State signal
Stuck in FETCH state
Local Sequencer Component in SymPLe
Stuck in EXECUTE state
FB Controller Component in SymPLe
Ctrl signal
FB Controller waiting for trigger from Local Sequencer

Figure 33: Model Based Testing detects complex interactions that result in Deadlock scenario (based on [91]).

### *Design time assumption made during MBT to be monitored at runtime (point-5 of  Figure 28)*

One of the vulnerabilities found during testing of the SymPLe architecture was a fault in the *control signal* which handles writing outputs after execution of a task (Figure 34). This fault can result in writing output values before execution of a task is complete, thereby violating higher level requirements at the application level. Although such vulnerable areas can be addressed at design time, it may be important to also monitor them at runtime. The implementation of runtime monitors for this property is explained in Section 4.6. This example is a typical case where an assumption held at design time can potentially be violated due to fault at runtime. The assumption made was that the *ctrl* signal will not change and is held constant until an execution of task is complete, thereby ensuring that output is written only after task execution is complete. This vulnerability (violation of assumption that *ctrl* signal is held constant) in the architecture was detected during MBT and emphasizes the importance of monitoring such assumptions at runtime.

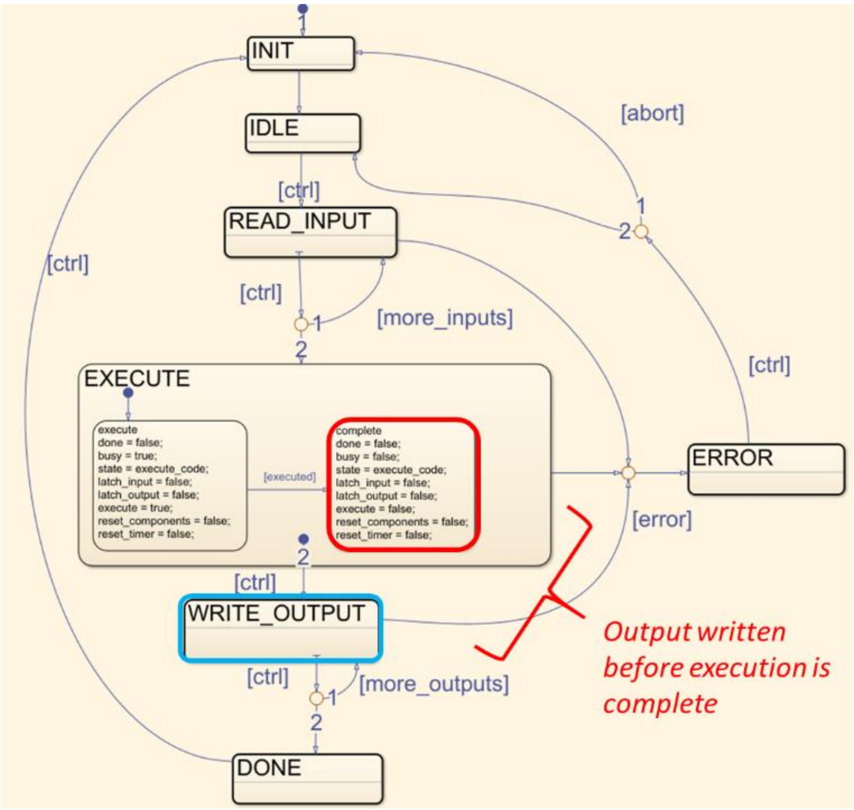


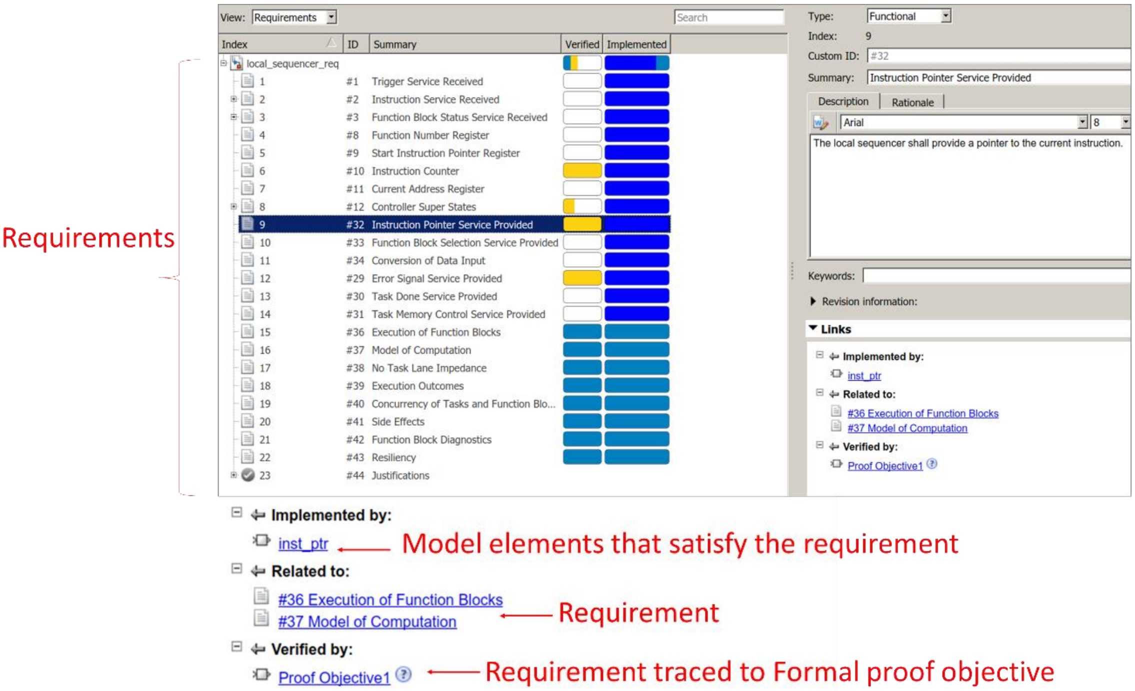
Figure 34: State Flow diagram showing transition to *write_output* state before execution is complete.

### c. Example 3: Formal Verification using Design Verifier

*Provides an example for monitor property derived from functional behavior of the system as shown in point-3 of Figure 28.*

Although testing can expose many design flaws, it is impossible to test a system exhaustively. Formal verification does not need any inputs like simulation-based verification. Critical conditions in the design are written as properties and formal verification checks the entire state space and proves or disproves the property. They mathematically ensure that there is no undesirable behavior due to system flaws. If a property fails, a counterexample is generated showing the input conditions that resulted in the property failure. We perform formal verification using Simulink Design Verifier (DV). Explanation of Simulink Design Verifier is in Appendix A2.

Simulink Design Verifier (DV) was used to formally verify the SymPLe model. DV offers comprehensive model verification along with traceability between requirements, model and the properties. Figure 35 shows the traceability of a requirement to a proof verified with DV. The traceability feature help gives us a complete picture of the design and verification artifacts while designing runtime monitors.


Requirements
Model elements that satisfy the requirement
Requirement
Requirement traced to Formal proof objective

View: Requirements | Search

| Index | ID | Summary | Verified | Implemented |
|---|---|---|---|---|
| local_sequencer_req | | | 🟦🟨⬜ | 🟦 |
| 1 | #1 | Trigger Service Received | ⬜ | 🟦 |
| 2 | #2 | Instruction Service Received | ⬜ | 🟦 |
| 3 | #3 | Function Block Status Service Received | ⬜ | 🟦 |
| 4 | #8 | Function Number Register | ⬜ | 🟦 |
| 5 | #9 | Start Instruction Pointer Register | ⬜ | 🟦 |
| 6 | #10 | Instruction Counter | 🟨 | 🟦 |
| 7 | #11 | Current Address Register | ⬜ | 🟦 |
| 8 | #12 | Controller Super States | 🟨⬜ | 🟦 |
| 9 | #32 | Instruction Pointer Service Provided | 🟨 | 🟦 |
| 10 | #33 | Function Block Selection Service Provided | ⬜ | 🟦 |
| 11 | #34 | Conversion of Data Input | ⬜ | 🟦 |
| 12 | #29 | Error Signal Service Provided | 🟨 | 🟦 |
| 13 | #30 | Task Done Service Provided | ⬜ | 🟦 |
| 14 | #31 | Task Memory Control Service Provided | ⬜ | 🟦 |
| 15 | #36 | Execution of Function Blocks | 🟦 | 🟦 |
| 16 | #37 | Model of Computation | 🟦 | 🟦 |
| 17 | #38 | No Task Lane Impedance | 🟦 | 🟦 |
| 18 | #39 | Execution Outcomes | 🟦 | 🟦 |
| 19 | #40 | Concurrency of Tasks and Function Blo... | 🟦 | 🟦 |
| 20 | #41 | Side Effects | 🟦 | 🟦 |
| 21 | #42 | Function Block Diagnostics | 🟦 | 🟦 |
| 22 | #43 | Resiliency | 🟦 | 🟦 |
| 23 | #44 | Justifications | | |

Type: Functional
Index: 9
Custom ID: #32
Summary: Instruction Pointer Service Provided

Description | Rationale
Arial 8
The local sequencer shall provide a pointer to the current instruction.

Keywords:
Revision information:
Links
Implemented by:
inst_ptr
Related to:
#36 Execution of Function Blocks
#37 Model of Computation
Verified by:
Proof Objective1

Implemented by:
inst_ptr
Related to:
#36 Execution of Function Blocks
#37 Model of Computation
Verified by:
Proof Objective1

Figure 35: Bi-directional traceability between requirements, model and formal proofs.

***Formal Verification of the SymPLe architecture for Higher-level Requirement (case-1)***

Example Design time property verification helps derive properties related to system functionality that can be monitored at runtime.  Figure 36 shows a property modeled in Simulink and verified by Simulink Design Verifier. The property asserts mutual exclusivity between *done* and *busy* signals. Only one of these signals can be high at a given time. An output could be *done* only when it is not *busy* performing a task and vice versa. In the Figure 36, if the sum of the two signals *done* and *busy* is not zero and is less than or equal to ‘1’, the property holds true. However, more than one signal being high violates the mutual exclusivity and hence falsifies the property. This property satisfies a higher-level requirement of the SymPLe architecture to verify system functionality. Such properties can be monitored at runtime.

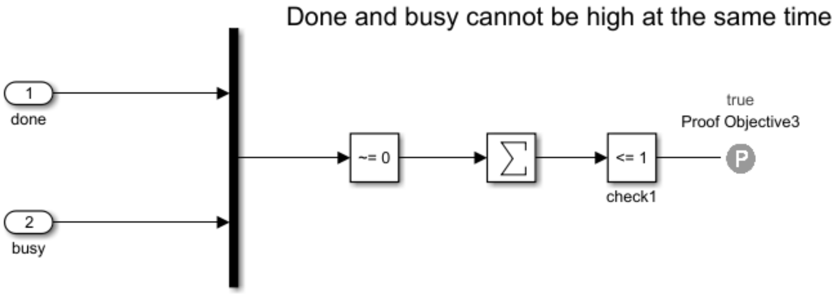


Figure 36: Simulink Design Verifier property.

***Formal Verification of the SymPLe architecture for a lower-level requirement (case-2)***

Below is an example that shows formal verification of a property related to the error handling functionality of Local Sequencer, which is a component of the SymPLe architecture. MBT provides insights into how to construct proofs, and how to organize them in a way that they are synergistic to testing. *As such, knowledge gained during MBT helped identify the critical areas in the design that may need to undergo a rigorous formal verification.* Knowledge gained during MBT helped better understand the behavior of the system as MBT deals with dynamic execution and find vulnerable/weak areas in design that could be a potential source for design flaws. Particularly, the failed test cases directed us to formally verify certain important requirements of the system. Properties were written for critical requirements and failed test cases. The failed test cases would fail again during formal verification (as expected). But, careful analysis of the counterexamples provided insights on other potential design flaws that could cause failures. More properties were formulated to check for design flaws after counterexample analysis.

For example, MBT exposed a configuration error in Local Sequence (Local Sequence is a component in SymPLe architecture). A configuration error "Initialize Outputs Every Time Chart Wakes Up" found during MBT exposed other vulnerabilities in the design that needed rigorous formal verification. This wrong configuration setting resulted in unexpected behavior in a state flow implementation. Careful study of this error exposed a design flaw related to the error handing in Local Sequencer. A property was modeled to formally verify this error handling behavior as shown in Figure 37. The property failed and the counterexample showed us how this failure could lead to invalid transition, thereby leading to non-deterministic behavior among the components in SymPLe. The state flow in Figure 38 has to transition to INIT state if task_error=1 and it has to transition to FB_SELECT state if task_error=0 which was violated as seen in Figure 38. Property proving guided by model-based testing proved to be extremely useful in uncovering such vulnerable/weak areas in the design. Such vulnerable areas can be additionally monitored at runtime.

When we are in FETCH,FB_EXECUTE or WRITE states and error =1 and the error is recoverable then the state flow must transition to FB_SELECT after 3 clock cycles

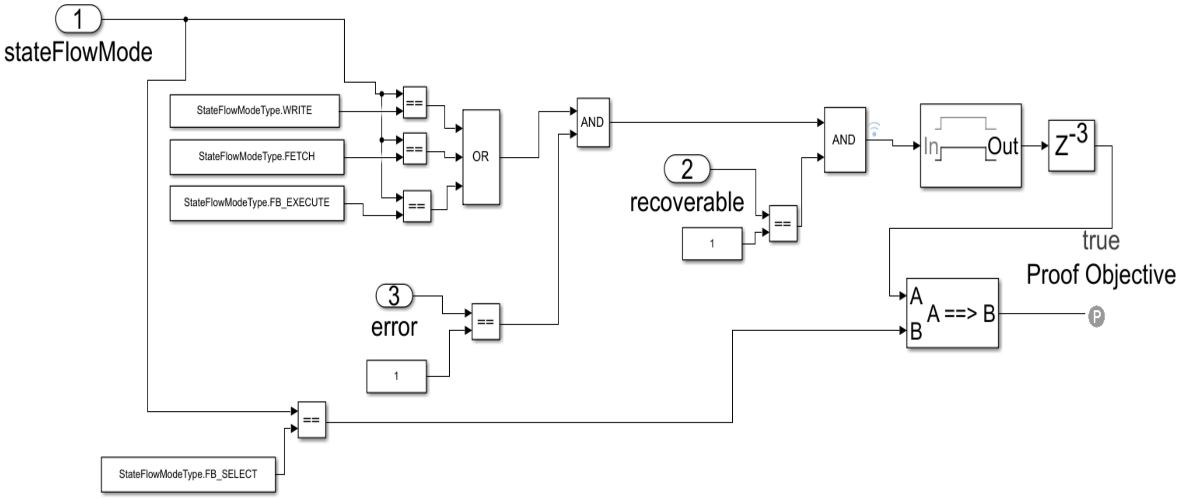
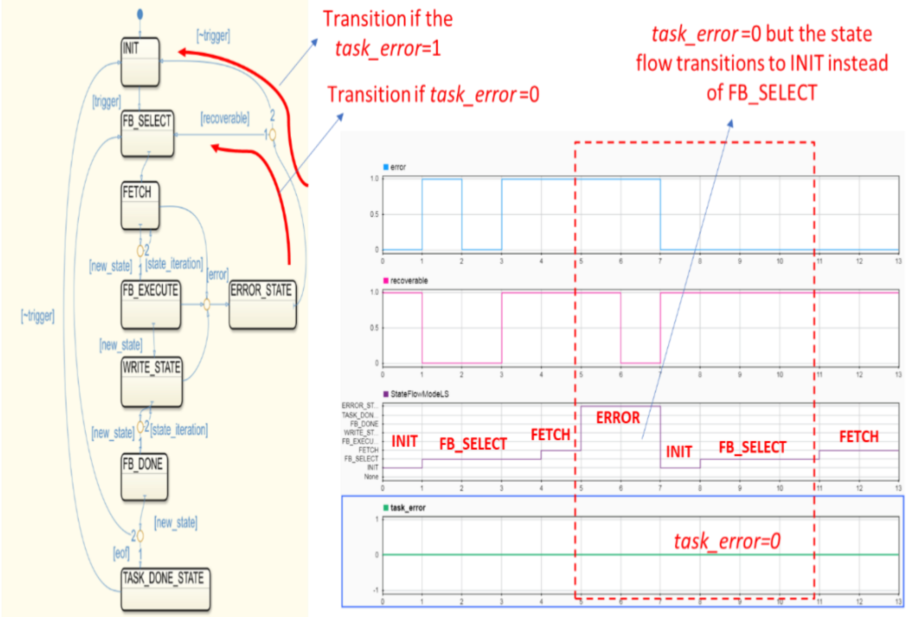


Figure 37: Design Verifier property to verify valid transition in the state flow diagram when there is an error.



Local Sequencer State Flow diagram

Counter example

Figure 38: Counter-example showing invalid transition due to design issue in error handling.

Furthermore, Simulink DV can be used to perform static verification of the model to detect dead logic, divide by zero, integer overflow, out of bound array access, NaN floating point violations, specified minimum and maximum value violations.  More about static verification checks is discussed in Appendix A 3.

### *d. Example 4: Fault Injection*

*Provides an example of monitor property based on insights from fault injection as shown in point-4 of  Figure 28).*

Fault injection testing is one of the important steps in the verification of a design. It helps verify the fault tolerance capabilities of the design and understand system behavior in the presence of faults. In the SymPLe architecture, fault tolerance capabilities are incorporated at the lowest level of the architecture which are the Function Blocks. In Figure 39, GT and GT1 are the Dual Modular Redundant blocks that compute the 'greater than' operation.  The State Comparator block compares the outputs of the two blocks and flags a "stateError" if the output signals don't match. A *True* value in the stateError output indicates the mismatch between the two redundant blocks due a fault in the system.

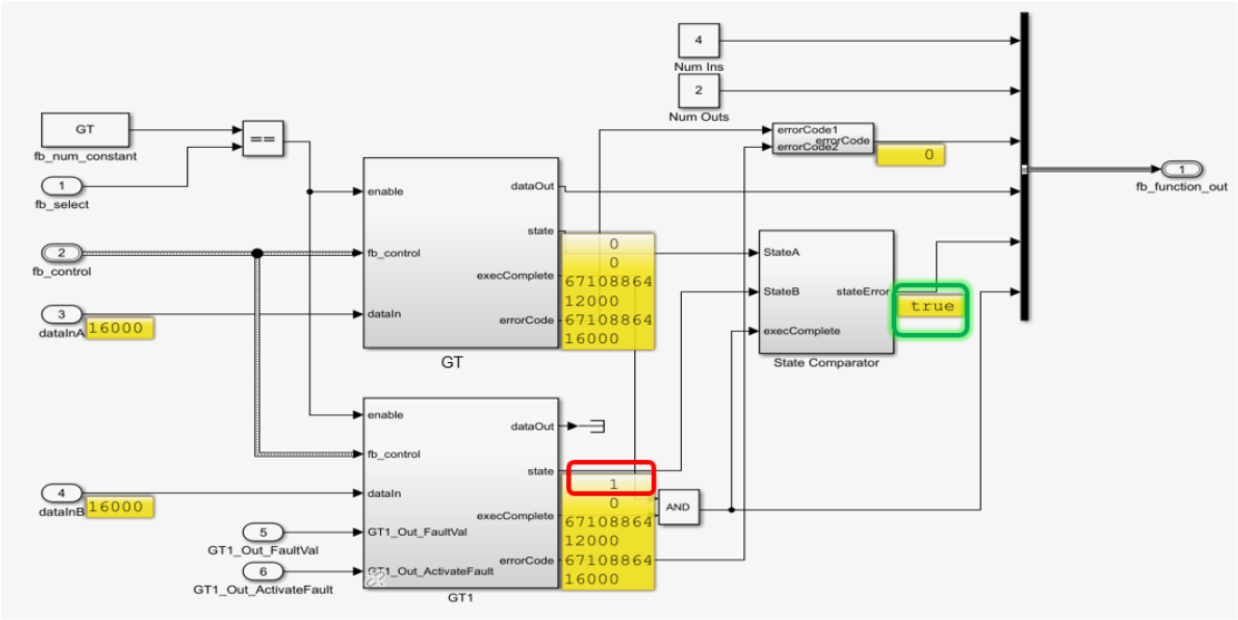


Figure 39: Fault Injection on Greater Than Functional Block in SymPLe architecture.

In another example, faults were injected at the system level on the EDGSS inputs using fault saboteurs as described in [91] and Simulink assertion blocks were used to model runtime monitor properties in Simulink.

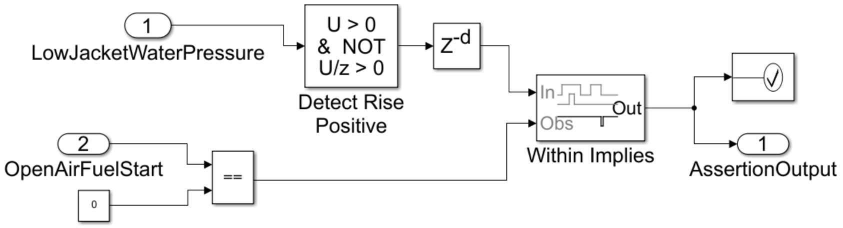

When Low Jacket Water Pressure signal is True, the Open Air Start Valve and Fuel valve should go False within T seconds.

Figure 40: Runtime monitor property modeled using Simulink verification blocks.

Figure 40 shows a runtime monitor property modeled for an EDGSS safety requirement using Simulink assertion blocks. This property verifies that the *OpenAirFuelStart* input goes to False, within T seconds when a *Low Jacket water pressure* is detected. Modelling runtime monitor properties at design time verifies the fault tolerant capabilities designed in the system and also understand system response in the presence of faults. We explain the runtime monitors implemented using formal runtime specification language in Section 4.6 for this property.

In the sections below, we summarize our findings on synergy from the verification artifacts performed on a model-based design. Another important task of the V&V at the code level is to prove the equivalence between the model and the code an ensure that the properties verified at the model holds true at the code level. Code is generated for the final version of the model after addressing the issues identified during MBT and formal verification. Simulink tools can be used to generate both C code or HDL code, depending on the target hardware platform for the application. The sections explaining the V&V steps of the code and synergy points for designing runtime monitors are explained in Appendix A.

Since the EDGSS on the SymPLe architecture is based on a FPGA platform, we explain the V&V steps of the code based on the HDL code generated from Simulink tools. Details on how similar V&V steps can be carried out for a C code is also provided in Appendix A 8 and A 9.

## 4.6 Implementation of Monitors Based on Synergy

We set out to design the runtime monitors that would ensure operational safety of the EDGSS based on the synergy investigation. We implement runtime monitors for two properties in this section. We use a stream-based runtime verification tool called TeSSLa [81] to write monitoring specifications for the EDGSS application. TeSSLa is a temporal specification language which is ideally suited for applications such as EDGSS where sequencing and timing of events is critical. TeSSLa specifications are written using a wide range of library functions provided by the tool which can be synthesized on hardware (FPGA). The specifications can be verified against a sample input trace using a simulator provided by the tool before generating a Verilog code (monitor). We implemented the runtime monitors generated by TeSSLa on a Zynq XC7Z020-1CLG400 FPGA [104].

The TeSSLa runtime monitors can be introduced into the Simulink MBDE environment by FPGA in loop (FIL) implementation [105]. Simulink interfaces with FPGA tools (e.g. Xilinx Vivado) to synthesize and generate bitstream for FIL implementation. In FIL mode, the Simulink and FPGA are synchronized. i.e. Simulink waits for the FPGA to finish executing and provide an output for every simulation timestep. The workflow for generating TeSSLa monitors from specifications and integrating them with Simulink as well as more details of synchronization in FIL implementation is explained in detail in Chapter 7.

In Figure 41 TeSSLa runtime monitors are implemented on the FPGA and are used to verify the SymPLe EDGSS model. To verify the detection properties of the monitors, fault injection was performed to introduce transient and stuck-at faults at various places in the design. The TeSSLa runtime monitors were able to detect violations of properties in the presence of faults.

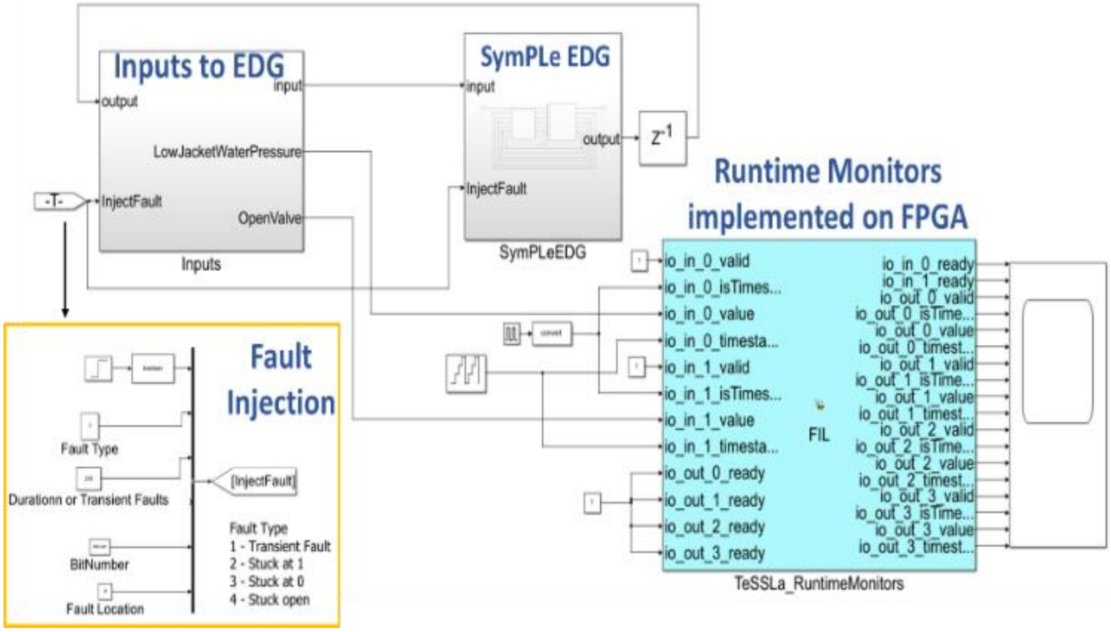


Figure 41: EDGSS model verified using TeSSLa Runtime monitors implemented on the FPGA.

We provide two important cases:

(1) Monitoring system level property for the EDGSS at design time, considering faults that can occur at runtime

(2) Runtime monitoring property derived from design time assumptions, which are critical.

**Case 1.** Here we monitor system level properties for the EDGSS, considering faults that can occur at runtime; one such application level property for the EDGSS is "When *Low Water Pressure* is True, the *Open Air and Fuel Valve* should go False within $T_m$ seconds". The property is expressed in Event Calculus, an event based formal language [80]:

$$\begin{aligned} &\text{Happens}(LowWaterPressure, \mathrm{T_a}) \Rightarrow \\ &\neg\, \text{HoldsAt}\,(OpenAirandFuelValve, \mathrm{T_b})\ \wedge\ (\mathrm{T_b} < \mathrm{T_a} + \mathrm{T_m}) \qquad (1) \end{aligned}$$

Here $T_a$ is the time when the *Low Water Pressure* goes True, $T_b$ is the actual time at which the state of *Open Air and Fuel Valve* goes false (indicates valve is closed). $T_m$ is the expected time after $T_a$ within which the *Air and Fuel Valve* should close. In the EDGSS system, when a water pressure sensor detects a low value (1100th time step in this example in Figure 42 a), the air and fuel valve in the EDGSS must close (should go False) by the 2,475th time-step.

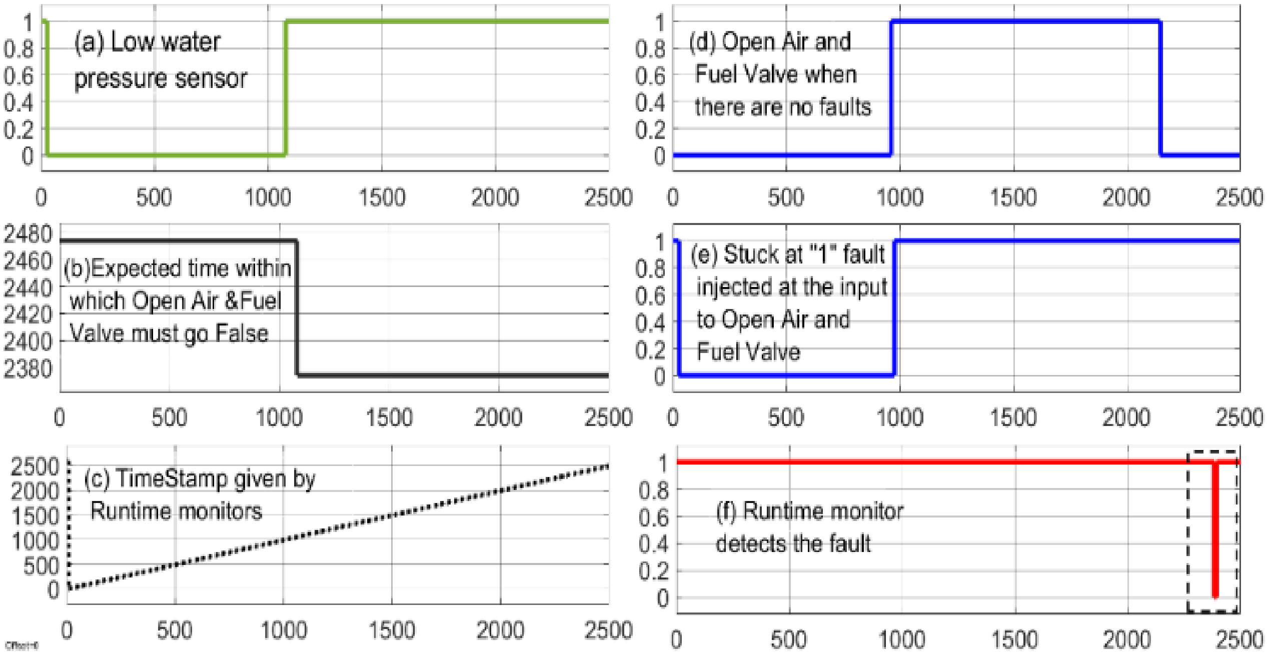


Figure 42 : Stuck-at "1" fault injected on EDGSS model and is detected by the runtime monitor (a) Low water pressure sensor goes high at 1100th time-step (b) Runtime monitor calculates and estimates that the value should close by 2475th time step (c) Time stamp information given by the runtime monitor (d) The valve closes within 2475$^{th}$ time step when there are no faults (e) stuck-at 1 fault injected and keeps the valve open all the time (f) Run monitor detects the fault.

A stuck-at “1” fault at the input to the *OpenAirFuelValve in* Figure 42 (e) keeps the valve open all the time causing a safety requirement violation and thereby falsifying the property as detected by the monitor in Figure 42 (f). Note that Figure 42 5 (d) is the correct response when no faults are injected and is merely added here to show what the correct response should look like in comparison to the response due to stuck at “1” fault in Figure 42 e).

**Case 2:** This example relates to monitoring a critical design assumption of the FPGA SymPLe architecture. In SymPLe, the requirement “If and only if the *executed* signal goes high at time $T_a$, the *Latch Output* should go high at the next time instant $T_b$=$T_a$+1” ensures that the output is written only after the application (e.g. EDGSS) execution is complete. This property is written in Event calculus as:

$$\text{Happens}(executed, \text{T}_\text{a}) \Leftrightarrow$$

$$\text{HoldsAt}\left(latch_{output}, \text{T}_\text{b}\right) \wedge (\text{T}_\text{b} = \text{T}_\text{a} + 1) \qquad (2)$$

Here $T_a$ is the time when the *Executed* goes High (True), $T_b$ is the next time instant when the *Latch_Output* should hold true. As described earlier in Figure 28, runtime monitoring properties are guided by various stages of verification. One example is a vulnerability found during testing of the SymPLe architecture, where there was a flaw in the way SymPLe handles writing outputs after the execution of a task. For certain conditions, this flaw resulted in writing output values *before* execution of a task was complete, thereby violating higher level requirements at the application level. Although this flaw was addressed at design time, it is classic example of a system design assumption that may affect application integrity if violated for any reason. Thus, we constructed a runtime monitor to observe this functionality. Transient faults were injected on the *latch output* signal which results in writing the output before execution is complete as shown in Figure 43. The TeSSLa runtime monitor we designed was able to detect these types of faults.

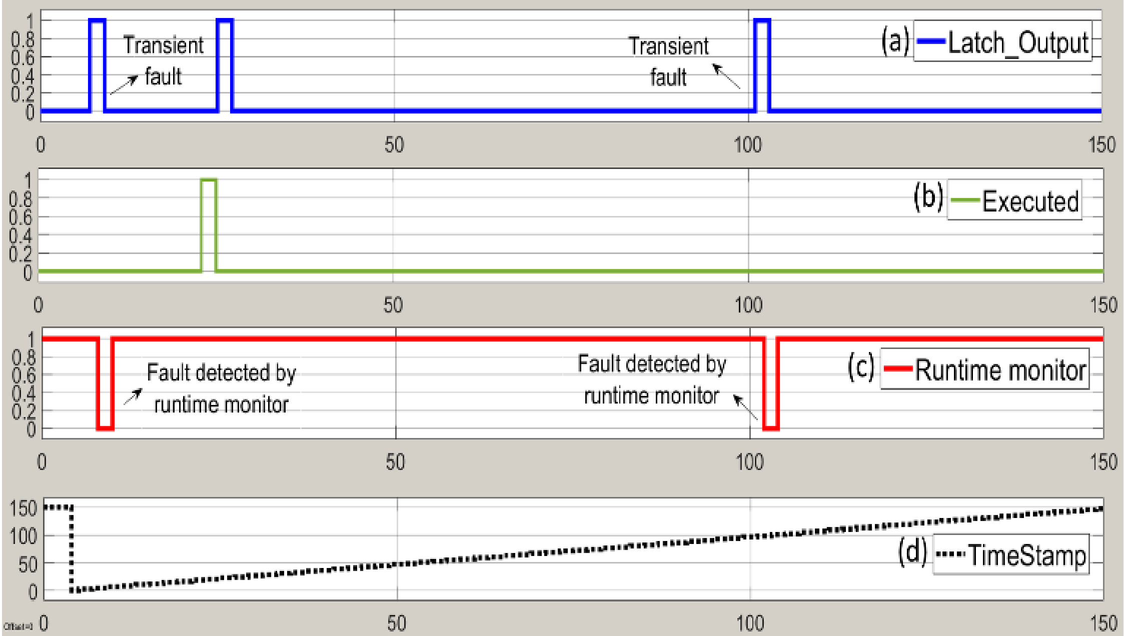
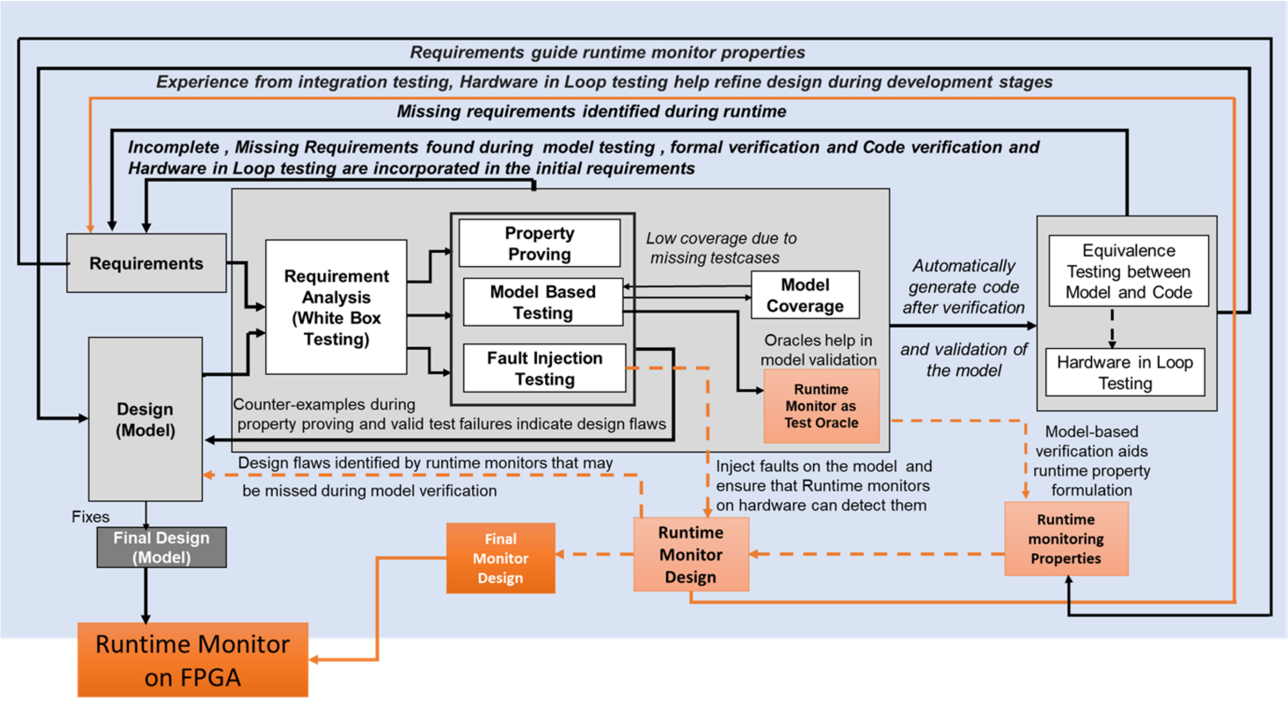


Figure 43: Transient Faults injected on SymPLe was detected by the monitor (a) Transient faults injected on the Latch Output in two places which results in writing output before execution is complete (b) Executed signal indicates completion of processing a task (c) Fault detected by runtime monitor as transient fault results in latch output before execution is complete(d) Timestamp information given by the runtime monitor.

## 4.7 Findings on Synergy from V&V of Model Based Designs

Figure 44 summarizes the instances where we looked for and exploited synergy that are shown in orange. Our investigation was aimed at characterizing what types of information, evidences, and artifacts manifest in MBE to help in designing runtime monitors, to support a more inclusive safety case. We summarize below the key findings on synergy between runtime verification and design processes. Figure 44 shows the revised "map" of the design time workflow and V&V activities that we formulated as part of this investigation.



Requirements guide runtime monitor properties
Experience from integration testing, Hardware in Loop testing help refine design during development stages
Missing requirements identified during runtime
Incomplete , Missing Requirements found during model testing , formal verification and Code verification and Hardware in Loop testing are incorporated in the initial requirements
Requirements
Requirement Analysis (White Box Testing)
Property Proving
Model Based Testing
Fault Injection Testing
Low coverage due to missing testcases
Model Coverage
Oracles help in model validation
Runtime Monitor as Test Oracle
Automatically generate code after verification
and validation of the model
Equivalence Testing between Model and Code
Hardware in Loop Testing
Design (Model)
Counter-examples during property proving and valid test failures indicate design flaws
Design flaws identified by runtime monitors that may be missed during model verification
Inject faults on the model and ensure that Runtime monitors on hardware can detect them
Model-based verification aids runtime property formulation
Fixes
Final Design (Model)
Final Monitor Design
Runtime Monitor Design
Runtime monitoring Properties
Runtime Monitor on FPGA

Figure 44: Iterative workflow showing synergy between Design time verification and runtime monitors.

*Model based design and V&V provide multiple points on "where" and "what" to monitor* – we found that evidences from model-based design help us understand the critical signals.

**Finding 1:** *Model based testing aids in the runtime property formulation*. We found that evidences from model-based testing where "corner" cases revealed unexpected behaviors or interactions provided insight into establishing the monitor requirements (e.g. properties to monitor at runtime). This answers the questions, what to monitor and where to monitor?

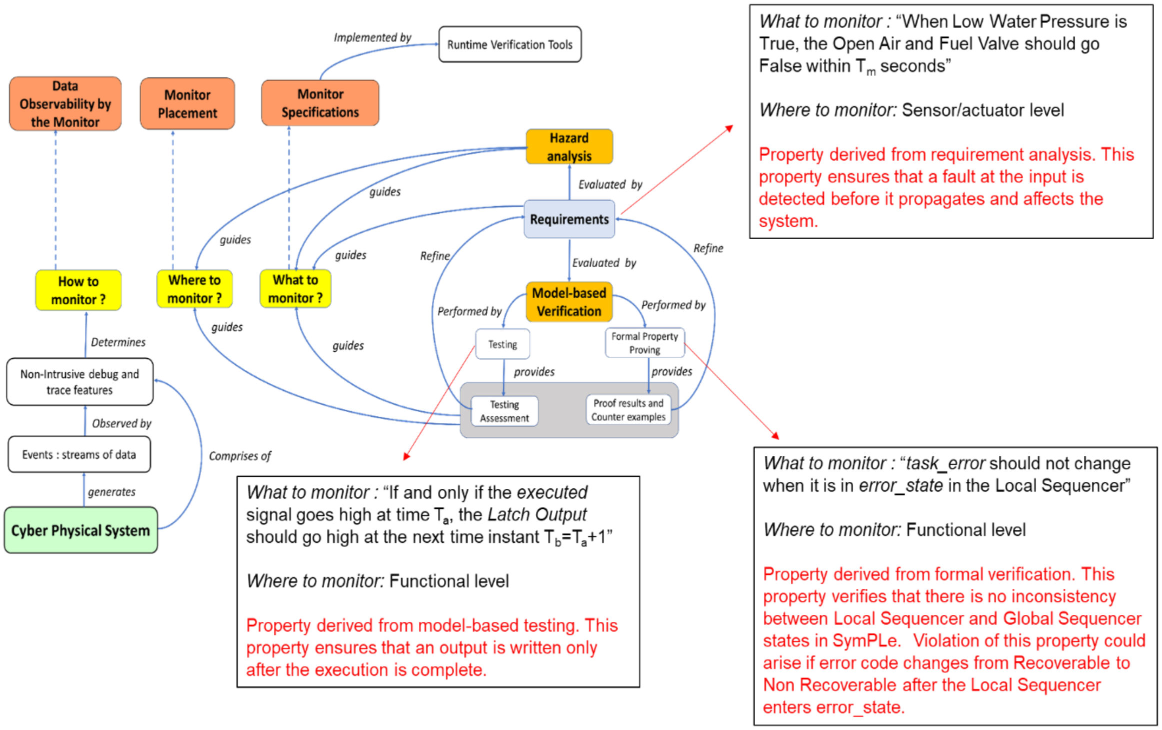
**Finding 2:** *Requirements traceability helps unite runtime monitor design with V&V artifacts*. Model based engineering environments often have powerful bi-directional traceability tools linking high level requirements to models to code. We found this type of functionality is extremely useful for navigating the chain of V&V evidence from requirements to sub-requirements to test cases and code artifacts. This capability enabled a more complete picture of the "what to monitor" with respect to crafting runtime monitoring properties.

**Finding 3:** *Finding critical design integrity assumptio*ns. Very often a design assurance is based on assumptions that are discharged at design time. However, if these assumptions (for any reason) are violated at runtime then guarantees of system behavior may not hold. We found that design time property specification and proving revealed critical design assumptions if falsified (for any reason) could result in significant consequences affecting EDGSS safety. Again, this answers the question of what to monitor (property is critical) and where to locate to monitor (where the behavior is critical).

**Finding 4:** *Iterative stabilization of model helps with monitor consistency*. Most V&V workflows are expressed as a linear process (as it is depicted in the V-model [106] , [35]) but the actual process was found to be more complex with each stage reaching maturity over time. There was an iterative feedback at each stage in the V&V process as show in Figure 44**.** This results in a set of requirement clarifications and feedback queries on ambiguous and contradictory requirements which can lead to disconnects between monitor design and system design. We found that refinement of models and monitors need to go hand in hand. As design specifications/requirements and properties are refined due to testing, monitor design is benefitted by these iterations. It informs us of the critical properties, i.e. 'what to monitor?'.

**Finding 5:** *Model based fault injection is useful*. We found that injecting the same faults at model level and the FPGA level is necessary to ensure the monitors implemented on hardware are consistent with their model detection specification. Fault injection testing was performed on the EDGSS application, by inserting fault saboteurs at various points in the design to inject stuck-at and transient faults and ensures that runtime monitoring properties can detect them. This helps verify that the monitors can detect all the anticipated fault/attacks for which they were designed.

In summary, in this chapter we discussed various instances in the V&V process where the lessons learnt could be used to design efficient runtime monitors. Figure 45 summarizes some of the instances discussed earlier in this chapter, which guided us on properties to monitor at runtime and monitor placements.



*What to monitor :* "When Low Water Pressure is True, the Open Air and Fuel Valve should go False within $T_m$ seconds"

*Where to monitor:* Sensor/actuator level

Property derived from requirement analysis. This property ensures that a fault at the input is detected before it propagates and affects the system.

*What to monitor :* "If and only if the *executed* signal goes high at time $T_a$, the *Latch Output* should go high at the next time instant $T_b$=$T_a$+1"

*Where to monitor:* Functional level

Property derived from model-based testing. This property ensures that an output is written only after the execution is complete.

*What to monitor :* "*task_error* should not change when it is in *error_state* in the Local Sequencer"

*Where to monitor:* Functional level

Property derived from formal verification. This property verifies that there is no inconsistency between Local Sequencer and Global Sequencer states in SymPLe. Violation of this property could arise if error code changes from Recoverable to Non Recoverable after the Local Sequencer enters error_state.

Figure 45: Model-based verification guides 'what to monitor' and 'where to monitor'.

We discussed our preliminary findings of synergy between design time verification in MBE and runtime monitors via an IEC 61508 methodology with an example EDGSS safety critical system implemented on FPGA architecture. The evidence suggests that MBDE methods and tools are supportive for integrating these synergy connections. We demonstrated some key verification activities at design time that help runtime verification and also explained how runtime monitoring scenarios can be considered at design time. This provides a continuum from design time assurance to runtime monitoring. Finally, we integrated monitors generated from a stream-based runtime verification tool called TeSSLa, into a model-based engineering workflow to explore and demonstrate such synergy.

# Chapter 5

# Design and Evaluation of Multilevel Runtime Monitoring using Model-based Engineering Methods

## 5.1 Introduction and Purpose

In this chapter we implement a multilevel monitoring scheme using Model Based Engineering tools (MathWorks Simulink) to ascertain the benefits and challenges of evaluating multilevel monitors with respect to security and safety considerations. The purpose of this chapter is to access design and evaluation of multilevel monitoring in the context of a CPS example. We demonstrate the benefits of multilevel monitors for comprehensive, faster detection and isolation of attacks by performing data attack and fault injection on a Simulink CPS model. In this chapter we have 3 levels of monitoring: data, network and functional monitors. They are implemented using Simulink blocks.

## 5.2 MathWorks Simulink Verification Blocks

Simulink is a graphical programming tool from MathWorks. It is used to design an executable graphical model that represents the hardware and software components under development. It comes with a variety of pre-verified library blocks that can be used to model the behavior of a system and the operational semantics of the underlying modelling language in Simulink is formal in nature. The model includes relevant design details, ignoring the complex details of the underlying software implementation. This initial model is refined till it is suitably complete and serves as the implementation that can be deployed through automatic code generation. Simulink comes with a number of tools that can be used for conducting analysis and verification of both the model and the automatically generated code.

The MathWorks Simulink tools have a number of modeling constructs (known as function blocks) such as, proof blocks, constraint blocks and temporal operators that we have used to model runtime properties. Figure 46 summarizes these Simulink library blocks and we briefly describe them below. Note that Simulink has a number of blocks that can be used to formulate properties and the ones we describe are examples of blocks which we have used in this dissertation.

**Temporal Operators -** Simulink library consists of three temporal operator blocks namely Detector, Extender and Within Implies that can be used to model temporal runtime properties [107].

***Detector* -** When an input signal is true for a specified number of time steps, the Detector block constructs an output signal which is based on a selected output type that can be:

a. Delayed Fixed Duration - When the input is true for a specified duration, an output signal is constructed with a specific delay w.r.t to when the input has a rising edge and this output is true for a fixed number of time steps (independent of the input falling edge).
b. Synchronized - When the input is true for a specified duration, an output signal is constructed which is true as long as the input is true. The output signal is synchronized with the input signal.

**Extender -** This block produces an output signal which is an extension of the input signal. This block works in two modes:

a. Finite – The output is extended by a fixed number of time steps.
b. Infinite – The output is extended indefinitely.

**Within Implies ('Within' In) => Obs -** This block ensures that the output (obs) is ’True’ for at least one-time step within each duration for which the input is ‘high’. If this condition fails, the Within Implies block outputs a ‘False’ signal after the true duration of the input.

- **Proof assumptions** - This Simulink block is used to constrain the inputs while proving a property.

- **Assertion Block -** This block outputs a true signal when the property holds true. It outputs a false signal along with an error message showing the time step, when a property fails.

- **Implies Block -** This block checks the value input B when input A is ‘True’. When an input A holds ‘True’ it verifies that the input B is also ‘True’. The Implies block gives a ‘False’ signal if this condition fails.

- **Detect Blocks -** Simulink has Detect blocks such as Detect Change, Detect Increase, Detect Decrease, Detect Rise positive, Detect Rise Negative, Detect Rise Non-negative and Detect Fall Nonpositive that can be used to model properties. The output of the Detect block is a ‘True’ or ‘False’ signal depending on whether the input satisfies the condition. Ref [108], summarizes all the Simulink library blocks that can be used to formulate properties.

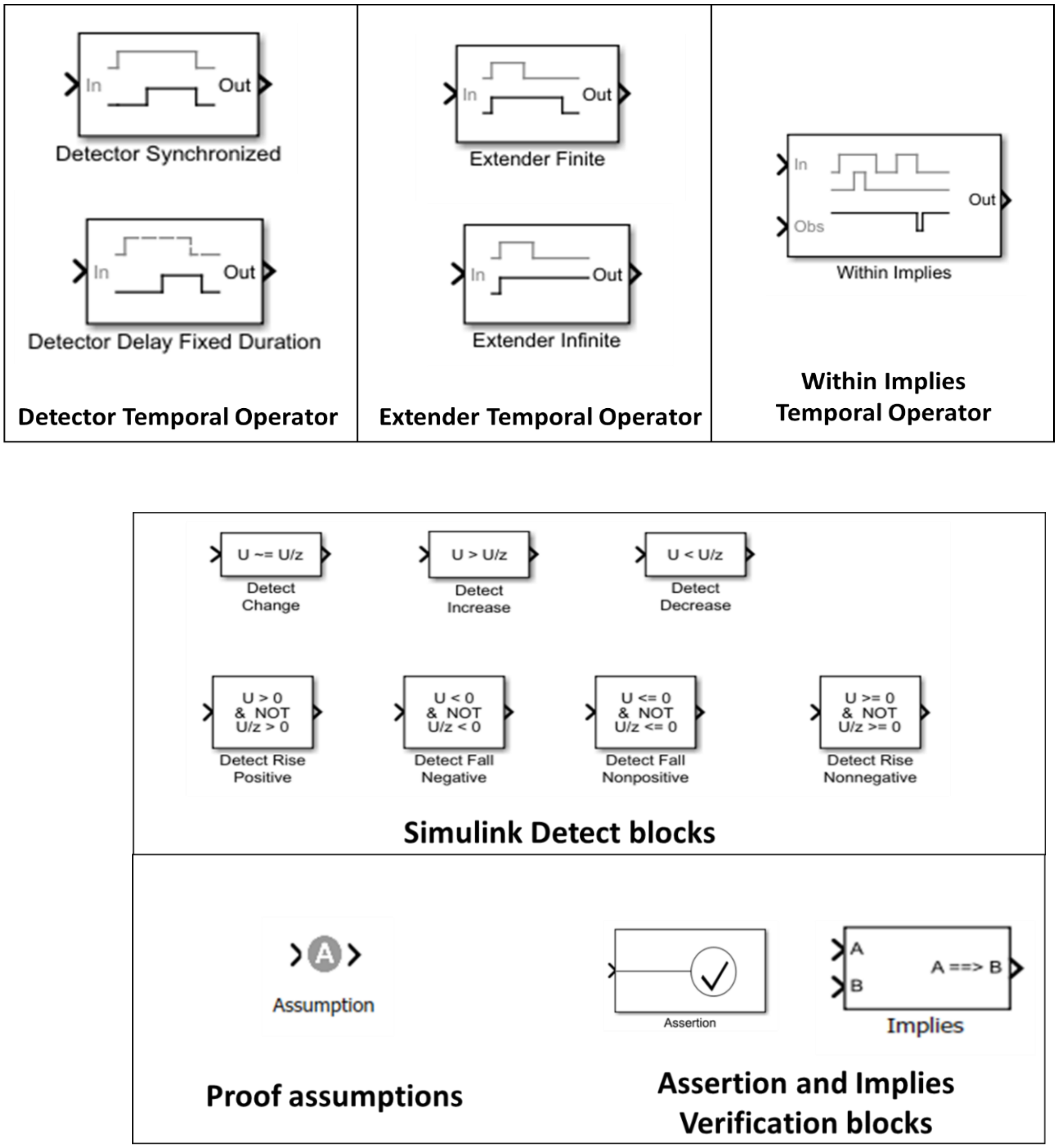


Figure 46: Simulink temporal operator blocks, proof assumptions, assertion and Implies verification blocks.

## 5.3 Specifying monitoring properties using Event Calculus

We express the safety and security properties to be monitored using the *Event Calculus* formal language and implement the properties using Simulink blocks. This process is explained in this section.

***Expressing natural language in Event Calculus and translating it into Simulink Blocks***

Requirements expressed in a natural language can be ambiguous and can have uncertainties. Therefore, it is helpful to derive runtime monitoring properties from requirements that are expressed in a formal language, which are typically free of ambiguities. Event Calculus provides a logical framework to express natural language requirements, in a formal way. Additionally, it is well-suited to describe CPS events and their temporal properties. Thus, Event Calculus is a formal method that provides a convenient way to translate natural language conditions to formal checking conditions. It should be noted that Event Calculus is not a necessary intermediate step, but is helpful in translating expressions in a natural language to Simulink blocks in an unambiguous way. We provide an example below.

Consider a heater where we observe that the heating element is turned on and there is a delay of 60 seconds due to various factors such as thermal inertia, sensor response, etc. before the temperature sensor records an increase in temperature. Thus, the property of the system in natural language is "*If the Heater is turned on at time T sec, Temperature should increase at time T+60 sec*".

In Event Calculus, this condition is expressed as *an event* "heater turns on" causing the *fluent* "temperature increases" to hold true after a time delay, of 60 seconds. Such a condition is expressed as: when the *event* "$Heater\_TurnOn$" happens at T=0, it implies that a *fluent* "$Temperature\_Increase$" holds true after 60 seconds.

$$\text{Happens}\ (Heater\_TurnOn, \text{T=0}) \Rightarrow \text{HoldsAt}\ (Temperature\_Increase, \text{T=T+60}) \qquad (1)$$

Figure 47 implements the property using Simulink blocks. Here we use the *Extender, Detect Increase* and *Implies* blocks to model the property. When an event HeaterOn occurs, the Simulink *extender* block introduces a fixed delay of 60 seconds at which time the *Implies* block verifies if the condition "$Temperature\_Increases$" is true. The increase in temperature is determined by the *Detect Increase* block in Simulink library as shown in Figure 47.

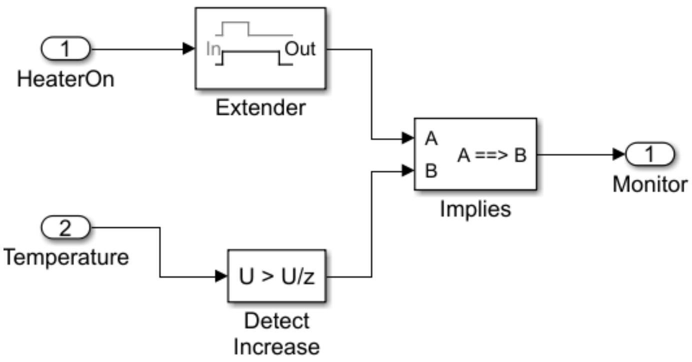


Figure 47: Implementation of the property "*If the Heater is turned on at time T sec, Temperature should increase at time T+60 sec*" using Simulink blocks.

In a similar manner, all event calculus monitoring conditions discussed in the subsection 5.4.2 are implemented using Simulink library blocks.

## 5.4 Example CPS: Anti-lock Braking System (ABS)

We use a Simulink model of an Anti-lock Braking System (ABS) from MathWorks examples as a target CPS to demonstrate multilevel monitoring framework [109]. The ABS system is summarized in the Figure 48. ABS is a safety critical unit in a car that helps prevent the locking of brakes thereby preventing an uncontrollable skid. The slip in a car is calculated based on the wheel rotation speed and actual vehicle speed measured by sensors in the plant (modeled by the vehicle dynamics in the Simulink ABS example). This slip value is communicated to the ABS controller through the CAN bus. The ABS controller compares the measured slip to a pre-set threshold slip (chosen so that the slip is below this threshold, which is acceptable for the safe operation of the car) and determines if the brake has to be on or off. The brake state (on/off) output, determined by the ABS controller, is communicated back to the plant through the CAN bus. Some important considerations while designing the monitoring framework are discussed in sub-sections 5.4.1 and 5.4.2.

### 5.4.1 Rationale for monitor placement in the ABS

Considering the heterogeneous nature of CPS and the attacks that can occur at various levels, we consider three monitors (Figure 48) to detect attacks/faults: Functional monitor M1 at the ABS controller and slip calculation unit, Data monitor M2 at the wheel speed sensor, vehicle speed sensor and brake actuator and Network Monitor M3 at the CAN bus. The rationale for the choice of monitors and their placement are as follows: The data from sensors of dynamic quantities such as vehicle speed or wheel speed can be attacked or corrupted, hence a data monitor (M2) is needed there. At the ABS controller and slip calculator modules, there are various faults that can compromise the functionality of the controller/computational element, hence a functional monitor (M1) is necessary. Finally, by injecting spurious traffic into the CAN bus, genuine data being transmitted between the ABS controller and the plant can be delayed or even distorted. Therefore, it is necessary to have a network monitor (M3). ABS functionality can be monitored even from the information in the CAN bus. Although, functional monitoring on the CAN bus can offer effective bolt-on solution to existing CPS, it is important to note that the CAN bus has limited observability, all data and functionality we want to monitor may not be available of the CAN bus.

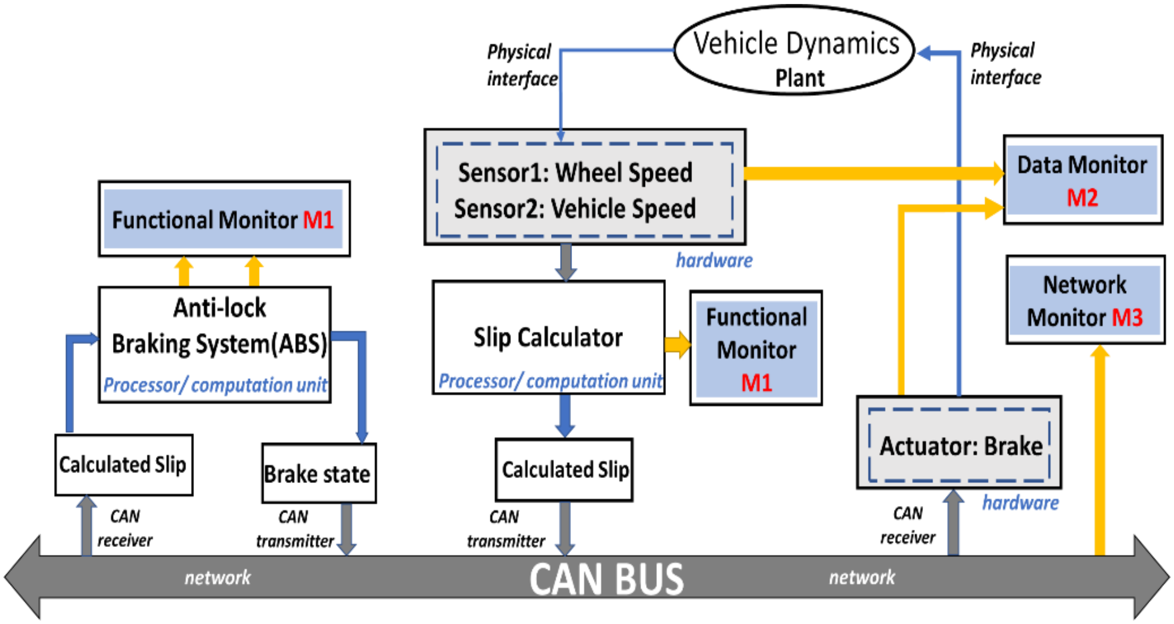


Figure 48: Anti-lock Braking System showing (a) Functional Monitor M1 at the computational units (b) Data monitor M2 at the hardware sensor /actuators level (c) Network monitor M3 at the CAN bus network level.

## 5.4.2 Monitoring properties for ABS controller expressed using Event Calculus

All properties were derived from system level requirements for the specific ABS application. In chapter 3, we introduced the basic formalisms of event calculus. In this example, we use the Happens and HoldsAt predicates to specify properties of the ABS system. The semantics of these two predicates are as follows:

*Happens (α, t)* means that an action or an event α happens at time t.
*HoldsAt (f, t)* means that the fluent f holds at time t.

**Property 1 verified by Functional Monitor M1:** If the calculated *Slip* is greater than a permissible threshold of $Slip_{safe}$ at time T, then the brake should be off at time T. Here *Slip* is the event and state of the $Brake_{off}$ is the fluent.

$$\text{Happens}(Slip, \text{T}) \wedge \left(Slip > Slip_{safe}\right) \Rightarrow \text{HoldsAt}\left(Brake_{off}, T\right) \qquad (1)$$

**Property 2 verified by Data Monitor M2** *:* If there is an event on wheel speed *WheelSpeed_A* at time $T_a$ and another event on wheel speed *WheelSpeed_B* at time $T_b$ where $T_b=T_a+T_d$, then the rate of change of wheel speed $R_w = \frac{(WheelSpeed_B - WheelSpeed_A)}{Td}$ should be less than *Rw_safe* (rate of change of wheel speed for safe operation).

Here $T_d$ is time elapsed between successive wheel speed measurements. $WheelSpeed\_A$ and $WheelSpeed\_B$ are the events and the rate of change of wheel speed being less than the permissible rate of change of wheel speed is the fluent:

$$\text{Happens}(WheelSpeed_A, \text{Ta}) \wedge \text{Happens}(WheelSpee\text{d}_B, \text{Tb}) \wedge (\text{Tb} = \text{Ta} + \text{Td})$$
$$\Rightarrow \text{HoldsAt}\ (\text{Rw} < Rw\_safe, \text{Tb}) \qquad (2)$$

**Property3 verified by Network Monitor M3:** If there is a packet arrival in the CAN bus ($Packet_A$) at time $T_a$ and another packet arrival ($Packet_B$) at time $T_b$ then the rate of packet arrival $T_p = T_b - T_a$ should be less than $T_{safe}$ which is the delay in the CAN bus when there is normal traffic for all time T. Here Tp is time elapsed between successive packet arrivals. Arrival of $Packet_A$ and $Packet_B$ are the events and rate of packet arrival $T_p$ is the fluent:

$$\text{Happens}(Packe\text{t}_A, \text{Ta}) \wedge \text{Happens}(Packet_B, \text{Tb})$$
$$\Rightarrow \text{HoldsAt}\ (\text{Tp} < Tsafe,\ \text{Tb}) \qquad (3)$$

The Event Calculus formalisms above combined with Simulink modeling allows designers/modelers to precisely capture monitoring properties.

## 5.5 Evaluation of Multilevel Monitors

The ABS controller, sensors and the CAN bus were injected with attacks/faults and the efficacy of the monitors in detecting these attacks/faults were evaluated. We used the data injection toolbox in [110] to inject sensor attacks on the model. We have performed fault injection by inserting fault saboteurs in the model, at various points in the system as explained in [91]. The fault saboteurs that we have used in this dissertation was from Ref [91].

Saboteurs are fault injection components inserted in VHDL designs. We have used the fault saboteurs on the Simulink model in the manner discussed in [91]. Fault saboteurs alter the value and timing behavior of the system when a fault is injected. The fault saboteur remains inactive during other times, when the system is under normal operating conditions. The yellow blocks in Figure 49 are the saboteurs inserted in the ABS controller. The fault injection control signals help characterizes the fault saboteurs i.e. define the type of fault to be injected, duration of the fault etc. *TransientFaultActivation, FaultType, FaultDuration, BitNum* and *FaultLocation are* the control signals of the fault injector block as seen in Figure 49. The control signals are described below:

- *TransientFaultActivation* – This control signal sets the time at which the fault becomes active.
- *FaultType* – This control signal sets the type of fault to inject, the faults can be transient faults, stuck at “1” / stuck at “0” faults or stuck open faults. A numeric value corresponding to the fault type we wish to injected is set.
- *FaultDuration* – This control signal sets the duration of fault i.e. how long we want the fault to be active. This is applicable to transient faults.
- *BitNum* – This control signal selects the number of bits and bit number we want to be faulted in a signal. For example, if we intend to have a single bit flip at bit location 30 of a 32 bit number, then we set BitNum (30) = 1 and set the rest of the numbers to zero.
- *FaultLocation* – Each fault saboteur that is inserted is assigned a unique ID. This control signal indicates the location of fault that we want to simulate.

Excessive information packets of higher priority from a malicious node flooding the CAN bus emulated a “Denial of Service” attack. The monitoring conditions were modeled using Simulink assertion verification blocks. We discuss below some examples to demonstrate that (1) there are attacks/fault scenarios that can only be detected if there are localized monitors at each level (data, functional, network)

(2) Some attacks/faults may be detected by a monitor at another level (other than the level of its origin), but monitors are nevertheless needed at each level to locate the origin of the attack/fault in such scenarios.

**Table 1** summarizes some of the attacks/faults that were injected in the CPS to demonstrate the need for a multilevel monitoring framework.

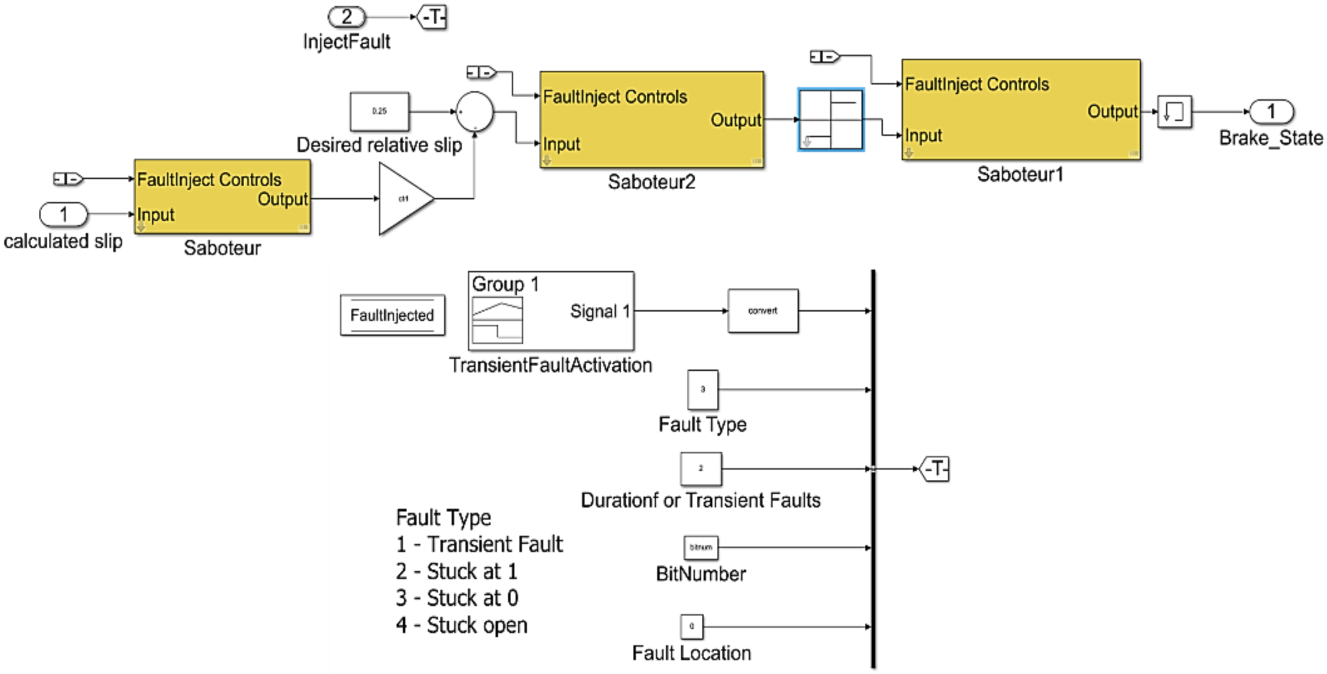


Figure 49: Fault Saboteurs injected in the ABS.

Table 1: Attacks/faults injected on the CPS.

| **No.** | **Attack/Fault** | **Attack location** | **Monitors that detect** |
|---|---|---|---|
| 1. | Stuck-at 0 fault | ABS controller | M1 only |
| 2. | Denial of service attack | CAN bus | M3 only |
| 3. | Sensor measurement injection attack | wheel speed sensor | Attack-1: M2 only<br>Attack-2: M1 and M2 |

**Case-1. Attacks/faults needing localized monitors at each level:**

Consider the Figure 50 where the slip, vehicle speed and wheel rotation speed are plotted as a function of time without the attacks/faults mentioned in

Table 1.

When there is no attack/fault, the ABS is able to ensure that the vehicle speed slows down to under 15 m/s at 12 seconds by appropriately releasing the brake whenever the slip exceeds a threshold. In many cases, where there is an attack/fault as shown in Figure 52, Figure 54 and Figure 56, the vehicle speed is ~20m/s or higher in 12 seconds (thus rendering the braking ineffective). The ABS controller decides whether the brake should be on/off depending on the slip. When the slip is greater than 0.25 (a threshold value) the brake should be off and when the slip is less than 0.25, the brake should be on.

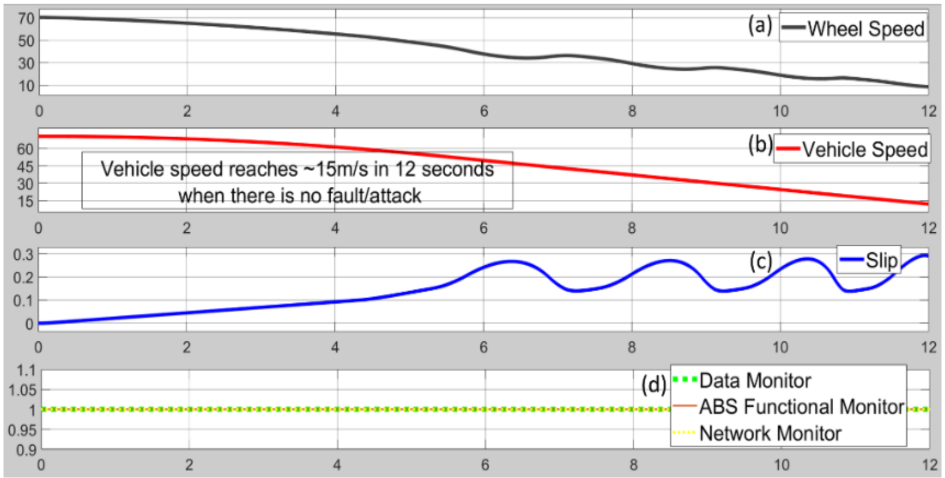


Figure 50: (a) Wheel Speed (b) vehicle speed (c) slip (d) monitor state: when there is no attack/fault on the CPS.

We first consider a fault on the ABS controller which can be critical for the system safety. A “stuck-at zero” fault was injected on the slip at about t=5 seconds and hence the controller never turns the brake off and is always on. Therefore, the property, “the brake is turned off when the slip (“s”) is greater than 0.25” is violated. This property (Property 1, eq. 1 in Section 5.4.2) modeled using Simulink blocks is shown in Figure 51.

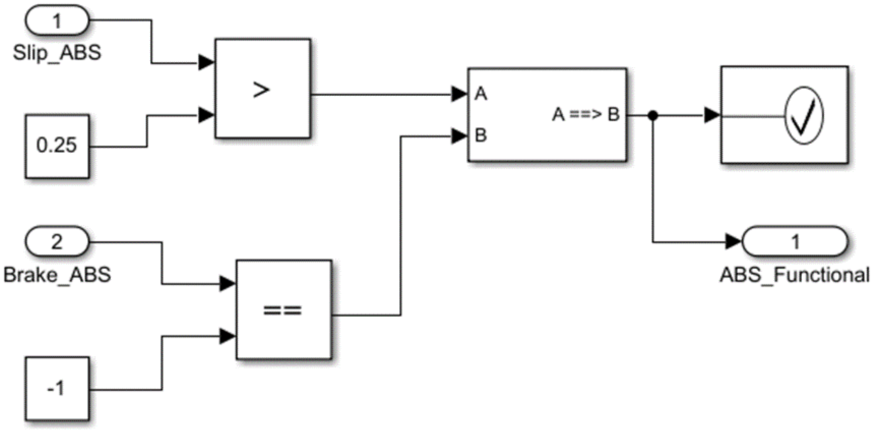


Figure 51: Property 1 modeled in Simulink, verified by the Functional Monitor M1.

It can be seen in Figure 52 that around t=6 seconds, the true slip communicated to the controller exceeds 0.25 and the functional monitor (M1) expects the brake to turn off. However, due to the fault (slip seen by the controller is zero) the controller still keeps the brake on. Hence, the property is violated and fault is detected by the ABS functional monitor. However, since this does not affect the signal transmission through the CAN bus or other sensor properties, the monitors at the network and data levels are unable to detect this. Hence, one specifically needs a functional monitor here to detect the fault.

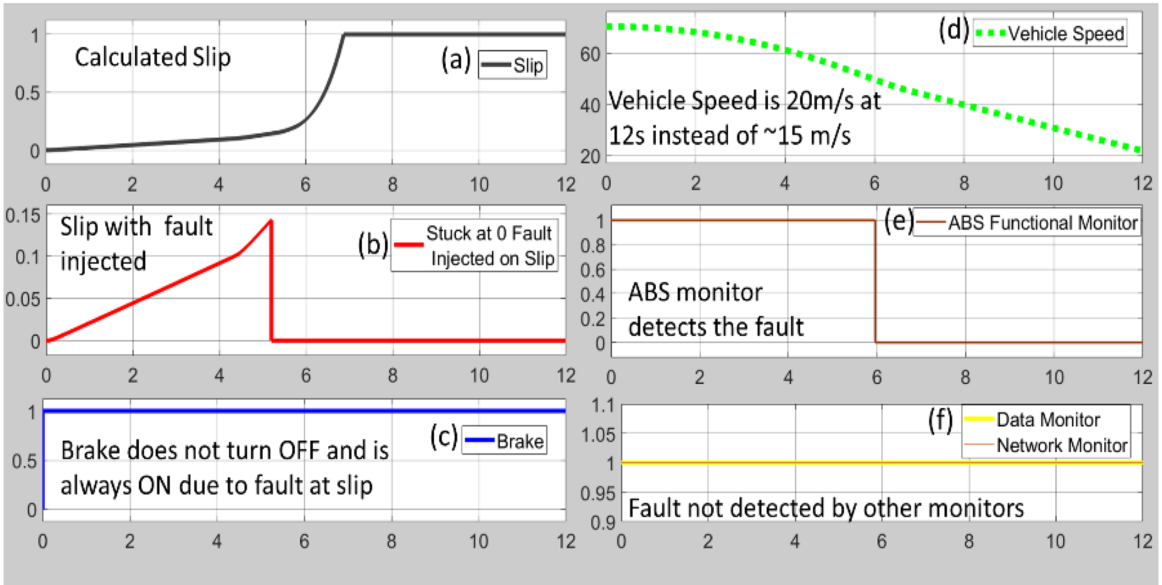


Figure 52: For a stuck at 0 fault on the ABS controller, (a) correct slip calculated (b) slip as seen by the ABS controller due to the fault at its input (c) the brake state which is always "on" as even though the true slip exceeds 0.25, the ABS only sees the slip=0 (d) vehicle speed that is affected by the ABS not correctly functioning (e) the ABS expects the brake to go "off" when slip exceeds 0.25, and thus detects a fault (f) other monitors do not detect this fault.

Likewise, the CAN bus is prone to number of attacks: packet insertion, packet erasure, packet payload modification, to name a few [111]. These lead to Denial of Service (DoS) attack that changes the packet frequency on the CAN bus. Time interval between CAN packets is usually periodic and has a fixed delay. A malicious node can change the time interval between successive packets by injecting extra packets causing delay in the bus. Property 3 (eq. 3 in Section 5.4.2) is modeled using Simulink blocks is shown in Figure 53. In Figure 53, CAN node with ID =7 is the silent node on the monitor end, that listens to all CAN bus messages.

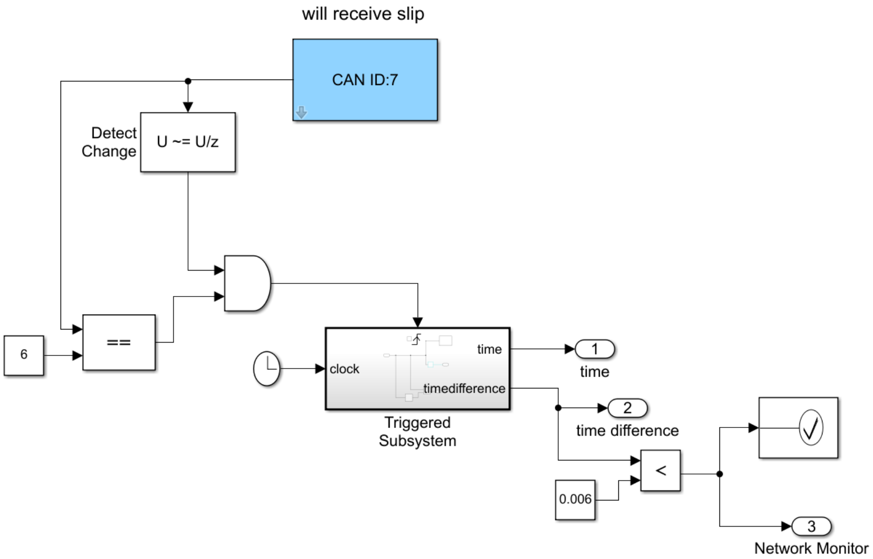


Figure 53:  Property 3 modeled in Simulink, verified by the Network Monitor M3.

An attack on the CAN packet frequency was performed by introducing a malicious node that delays the communication to and from the ABS controller. This was not detected by either the functional monitor (M1) at the ABS or the data monitor (M2). The fixed delay for normal traffic was identified and the network monitor (M3) verifies at runtime that the time interval between subsequent packets is within bounds.  When the time interval exceeds the normal levels the monitor M3 indicated an attack on the network as shown in Figure 54. When the system has no faults/attacks, the ABS controller receives the slip value, approximately every 0.006 seconds through the CAN bus. However, when there is more than a certain level of network traffic due packet injection by a malicious node, the delay in the CAN bus increases, which is detected by the monitor as shown. Flooding the CAN bus with many packets can lead to huge delay as seen in  Figure 54 (b) between 11th and 12th second. This affects the braking and the vehicle speed. The vehicle speed was 30m/s instead of 15 m/s during normal conditions with no fault/attack. While we used this approach as a proof of concept, there are alternate ways of monitoring the bus traffic discussed in [111].

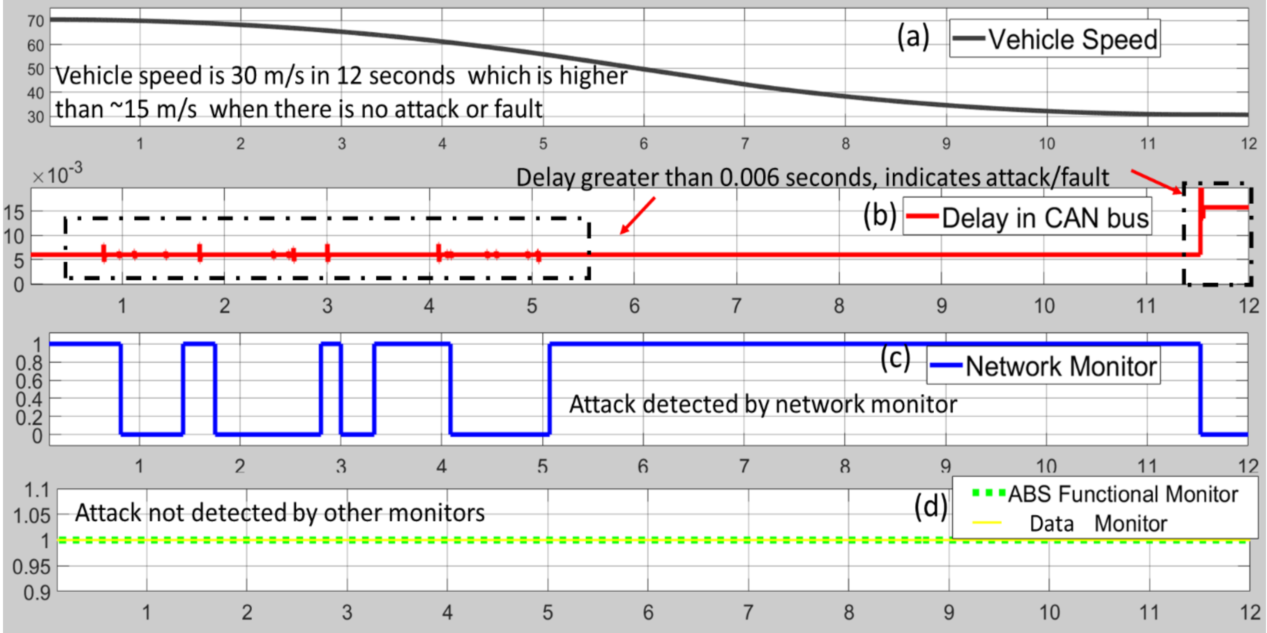


Figure 54: Bus traffic delay detected by Network monitor For a DoS attack, (a) vehicle speed that is affected due to delay in CAN bus (b) delay in CAN bus is greater than 0.006 seconds (c) Network monitor detects the attack (d) all other monitors do not detect the attack.

Next, Property 2 (eq. 2 in Section 5.4.2) that checks if the condition "the absolute value of the rate of change of wheel speed in below a certain threshold" was modeled in Simulink as shown in Figure 55.

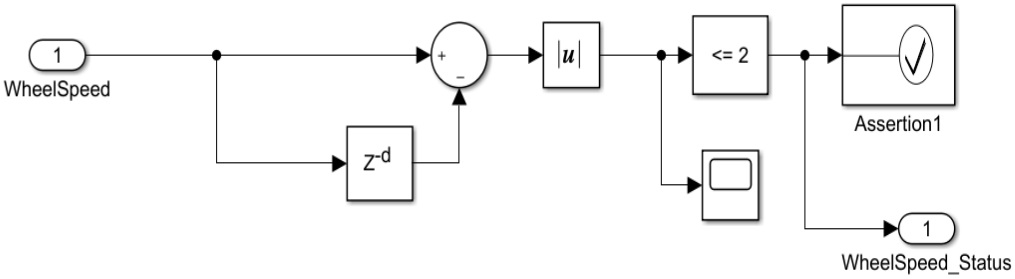


Figure 55: Property 2 modeled in Simulink, verified by the Data Monitor M2.

A sensor attack, "attack-1" on the wheel speed sensor that is detected by the data monitor (M2) is showed in Figure 56. It monitored the safety property "the absolute value of the rate of change of wheel speed should not be greater than $T_w$ rad/sec" where $T_w$ is a threshold rate of change of wheel speed for safe operation. However, *none of the other monitors* were able to detect this attack.

Hence, in all the above cases multilevel monitors are needed as faults/attacks at one level cannot be detected by monitors at the other levels as demonstrated by the above examples. *Hence, we show that having monitors at multiple levels are beneficial (and sometimes required) to detect attacks/faults that span multiple levels and systems.*

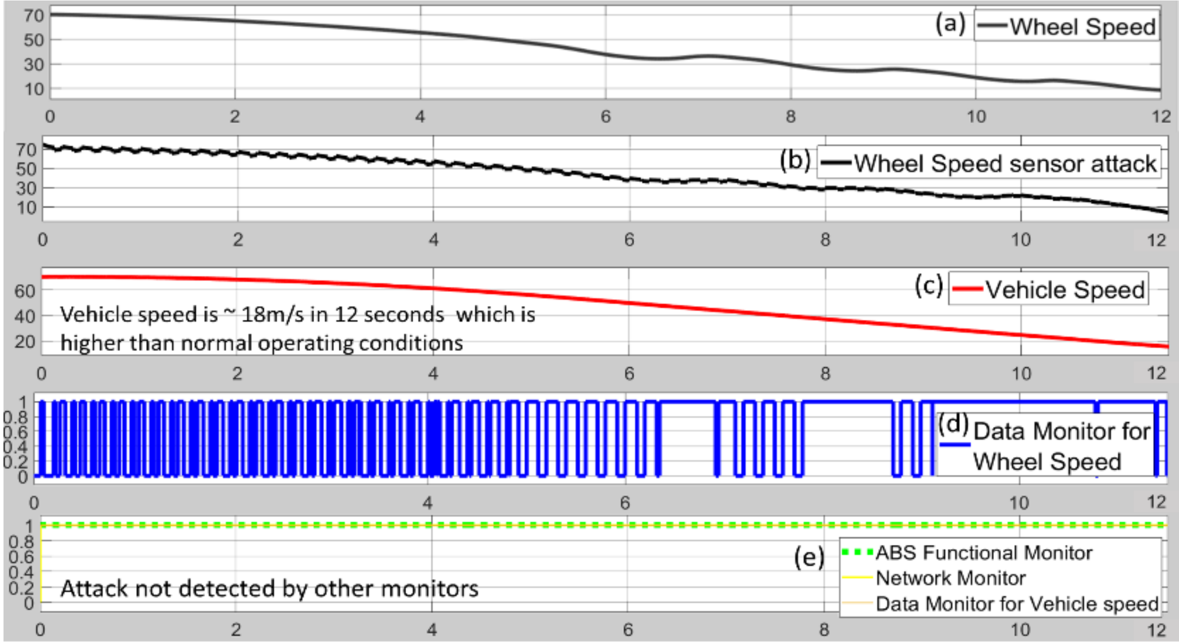


Figure 56: For an attack on wheel speed sensor, (a) Wheel speed when there is no attack/fault (b) wheel speed with an attack (c) vehicle speed affected by attack on the wheel speed sensor (d) Data monitor for wheel speed detects the attack(e) other monitors do not detect this attack.

**Case-2. Attacks/faults detected at more than one level but still needing multiple levels to find to location of the attack:**

When there is sensor measurement attack (discussed earlier) of a much higher magnitude (attack-2), it could cause the rate of wheel speed to change so drastically that it briefly affects the functional relation between the slip and break state monitored by M1. Hence it is detected by the functional monitor in addition to the wheel speed data monitor as shown in Figure 57. Note that this example has less number of disruptions to the wheel speed and does not significantly change the eventual vehicle speed reached at 12 seconds. However, it is still important to detect any attacks on the CPS.

We argue *both* of these monitors are probably needed, as even though the functional monitor detects this data attack, we cannot be sure where the attack/fault originated if we *only* had one functional monitor. We would use the fact that both the wheel speed data monitor and functional monitor detected this attack to pinpoint it was at the wheel speed sensor; while if only the functional monitor had detected the attack (not the data monitor) we would probably conclude the attack was on the ABS controller.

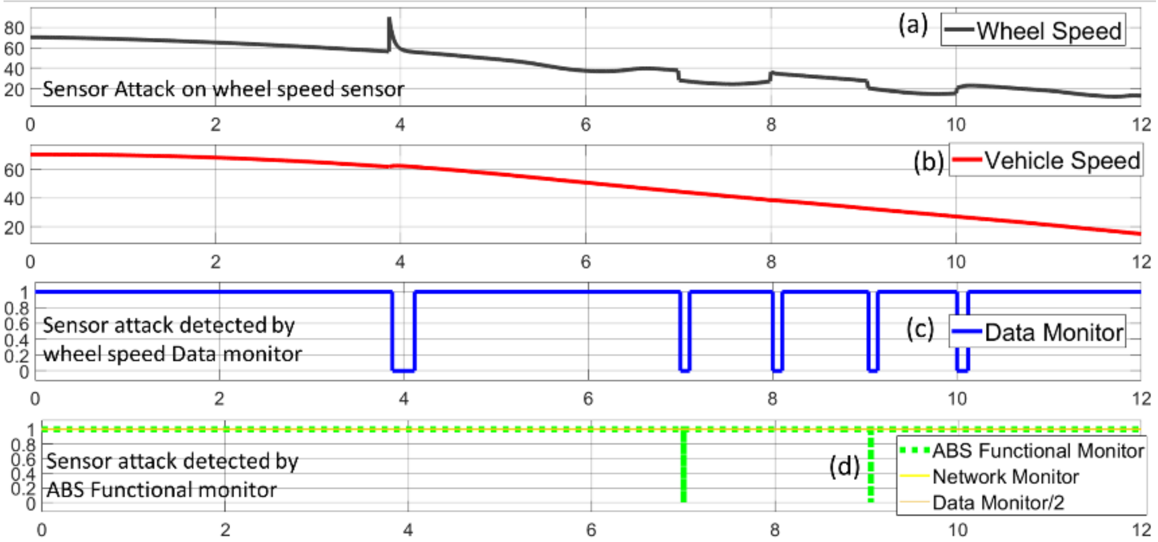


Figure 57: Attack detected by multiple monitors (a) Wheel speed when there is sensor attack (b) vehicle speed not affected significantly by attack on the wheel speed sensor (c) Data monitor for wheel speed detects the attack (d) functional monitor detects this attack.

Another issue to be considered is whether the ABS functionality (M1) and sensor data (M2) can be monitored from the information in the CAN bus. One issue is the CAN bus has limited observability as all data and functionality we want to monitor may not be available of the CAN bus. The other issue is as follows: Suppose the slip and brake state, are available on the CAN bus, we could have implemented the same ABS functional monitor on the slip and Brake ON/OFF state from information in the CAN bus (not shown here) rather than locally as we did earlier. While such a monitor would have detected a fault in the ABS controller action, it would have also been affected by excessive network traffic. So, this monitor alone would not be able to specifically pin point the origin of the attack.

## 5.6 Discussion: Monitor Organization Patterns

While multilevel monitoring has been shown to be beneficial and required for some attacks and faults in the above example, that in itself is not satisfactory for helping make decisions on how to organize the monitors so that they form a stronger defense against such faults and attacks. Model based design allowed us to justify placement of the monitors, what to monitor, and evaluate the monitors with respect to attacks/faults. From the above results, we can reason how monitors can be organized to support stronger checking.  Accordingly, we use classification schemes first postulated by [10] for certain types of *monitor configurations* with respect to the above example. Elks [10] postulated that monitors can be organized within a system context into four elemental types: *parallel*, *sequential, associative, and complementary*. These are explained in detail in [10] but briefly discussed here in the context of organization of multilevel monitors as shown in Figure 58.

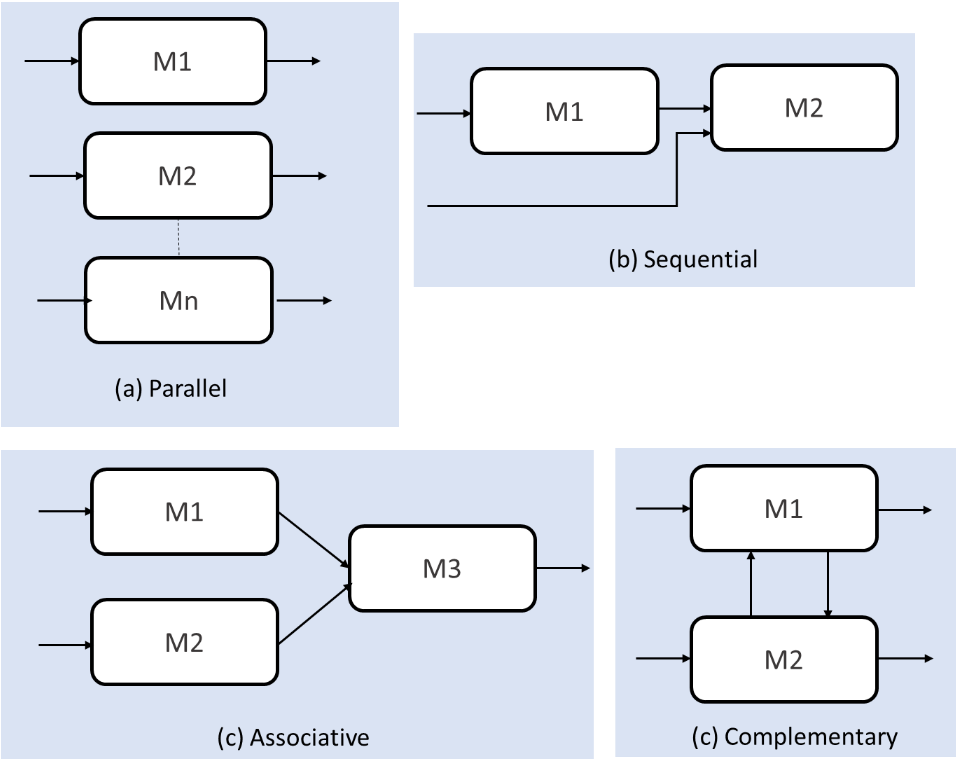


Figure 58: Parallel, Sequential, Associative and Complementary monitor organization postulated by Elks (Figure source [10] with modifications).

**Parallel Organization:** As shown in Figure 58 (a), such organization of monitors check properties in parallel and independent of each other. This provides coverage of checking property violations with respect to the CPS as each monitor verifies different aspects of the CPS architecture [10].

**Sequential Organization:** As shown in Figure 58 (b), in this configuration $M_2$ receives input from the system being monitored as well the decision of $M_1$ and is therefore conditioned on $M_1$'s decisions. $M_1$ is like pre-condition check for $M_2$'s properties. Thus, $M_2$ has the benefit of seeing the decision of $M_1$ and together they can check properties in stronger manner than $M_1$ and $M_2$ acting independently.

**Associative Organization:** As shown in Figure 58 (c), $M_3$ relates inputs from $M_1$ and $M_2$ that make their decisions independent of each other. Associative organization is a special form on sequential patterns. This also allows stronger or extended checking of properties as it can accommodate the decision of both monitors.

**Complementary Organization:** As shown in Figure 58 (d), both monitors $M_1$ and $M_2$ exchange information about each other's decision as well receiving the input from the systems they are independently monitoring. This allows them to consult with each other and reach an agreement on the monitoring decision. This type of monitoring is evident in distributed protocols like byzantine agreement algorithms.

In the ABS study, the monitors at the sensor, communication and functional levels detect faults/attacks as follows:

1. CAN Bus (network level) monitor ($M_N$): Only detects faults/attacks at the network level.
2. Data monitor ($M_D$): Only detects faults/attacks at the sensor level.
3. Functional monitor ($M_F$): Detects faults/attacks at the sensor level and functional level.

Hence, if the network monitor indicates a property is violated ($M_N$ =0) we can be sure there is an attack at the network level; likewise, for a property violated at the sensor level ($M_D$ =0). However, if the functional monitor indicates a property is violated ($M_F$ =0), we cannot determine if the source of the fault/attack is at the sensor or the functional level unless we also consult the monitor at the sensor level. By considering the *sensor level monitor in conjunction with the functional monitor*, and assuming the fault/attack occurs at only one level at a given time (probability of it occurring simultaneously at more than one level is very small) we can conclude the following as described below. Note that when monitor M = '0', it means that the property is violated and M='1' means that the property is satisfied.

a. If $M_F = 0$, $M_D = 0$; fault/attack is at the sensor level

b. If $M_F = 0$, $M_D = 1$; fault/attack is at the functional level

Also, to have complete set of conditions, we add:

c. $M_F = 1$, $M_D = 0$; fault/attack is at the sensor level

d. $M_F = 1$, $M_D = 1$; no fault/attack at either sensor or functional level

Hence, sensor and functional monitors should have an *associative* relationship to locate and isolate the source of these faults/attacks in the example of the ABS controller in this chapter.

Note that there may be other properties, faults/attacks in the same ABS system or in other more complex CPS that merit a different organization for example parallel, serial or complementary. Also, if we use an associative configuration for this CPS, that does not infer all of our multilevel monitoring organizations will be associative, it just means that for this CPS, there is at least one associative monitor organization.

While these findings are preliminary, they suggest the use of a different approach to distributed runtime monitoring than is currently practiced. With most distributed runtime monitoring schemes, the monitor organization or collaboration scheme is established early in the design process and it is therefore statically fixed in the architecture. Our results suggest that an iterative data driven approach may be an alternative approach. In the above ABS example, we started out with a parallel composition (all monitors operating independently). After fault injection campaigns, we evaluated the error propagation patterns and found that fault isolation or identification was not possible for some faults/attacks under the initial parallel monitor composition. However, these faults could be detected by a different monitor organization – the associative organization pattern. These results suggest at least three things:

1. Multilevel Monitor organization is dependent on fault/attack patterns that can manifest in the CPS. Without some form of assessment (e.g. fault/attack injection), initial monitor organization design may be insufficient in detecting complex fault/attack scenarios.
2. The critical CPS properties embodied in monitors have influence on the monitor organization.
3. Monitor organizations for critical CPSs should be guided by empirical methods early in the design process before the monitor organization is set into the architecture.

In short, the key difference with our runtime monitor design framework to other runtime monitor design frameworks is that ours is a model and data driven approach. That is, the critical CPS properties, their relationships to each other and empirical fault/attack simulations guide the monitor organization design.

Model based design and assessment methods worked particularly well in the evaluation of the monitors in this phase of the research. As one can imagine, if you had to build a physical prototype of the monitors, integrate them into the CPS, and then instrument the system for fault injection it would be very time consuming and costly to establish similar conclusions that we reached with model-based design methods. In this particular case, the evaluation results would not have been favorable with respect to the initial monitor organization in the physical prototype approach. Such findings would have pointed to a potential redesign, which would have been very costly at late stage system development.

In the context of multilevel monitor design, we provide confirmatory evidence that model-based engineering methods is particularly useful for evaluating; (1) assumptions on monitor design and fault detection, and (2) monitor organization patterns. By confirmatory, we mean our results fall in line with the boarder evidence that model-based design and engineering methods facilitate finding design deficiencies earlier in the design process.

Finally, we ponder on the farther inferences for future research. First, the results in this chapter preliminarily suggest that multiple monitor organization patterns may be a necessary condition for detecting complex fault and attack patterns in a CPS of the type found in the reference architecture. Intuitively, this makes sense as a CPS like an Automatic Braking System has tight interactions between the various sub-systems and the monitoring properties reflect this tight integration. As example, fault tolerant architectures that are required to detect and tolerate byzantine faults (e.g. lying faults). It has been shown that such a collaborative fault detection scheme is a necessary condition [112] [113]. However, proving multiple monitor organization patterns is a necessary condition for more general classes of CPSs is a challenging research endeavor – beyond the scope of this dissertation.

While our formalism uses such associate (and potentially other) relationship between multilevel monitors, the HECAD formalism [62] follows a hierarchical monitor organization. In HECAD, the monitor in higher level of hierarchy communicates with the monitor in the lower level of hierarchy to detect any abnormal behavior of the system. In other words, it is statically designed to follow sequential organization.

## 5.7 Summary

In this chapter, we have developed and implemented a multilevel monitoring framework and demonstrated the need for monitors at multiple levels to detect various attacks/faults for an ABS controller CPS. The results and broader implications of the research suggest that a model based and data driven approach to multi-level monitoring enables the identification of monitor organization patterns that are required for detecting complex faults/attacks. Without such early monitor organization identifications, we may be susceptible to gaps in protection of fault and attacks. We showed that existing MBE tools (Simulink) are effective and beneficial at modeling, designing, and assessing such monitoring architectures and integrate safety and security considerations early in the design process. We demonstrated the need for organized multilevel monitors for comprehensive detection and isolation of attacks by performing data attack and fault injection on an Anti-lock Braking System (ABS) Simulink CPS model. In the next chapter we will explore how such a multilevel monitoring framework can be implemented in hardware.

# Chapter 6

# ARM Processor Debug and Trace Capability to Assist Multilevel Runtime Monitoring

## 6.1 Introduction and Purpose

The previous chapters have provided a foundation of “what” and “where” to monitor in CPSs, but the remaining question to answer is “how”. The key prerequisite for runtime verification is the observability of the streams (e.g. during system operation) at time intervals of interest. Naively, this is often attained by software instrumentation[1] with its known limitations, namely being overly intrusive to program execution. An ideal monitor should be minimally intrusive to the System Under Observation (SUO) and should not impose any restrictions to the running program. The monitor should be space efficient and time efficient, meaning, it should not need to store large data and should be able to provide a timely decision on the behavior of a system. To that end, we began investigating the possibility and extent of using processor debug and trace capabilities found on many modern processor families to support runtime verification.

Many current processors have debug and trace capability, such as ARM Coresight [115] in ARM architecture and Intel Processor Trace [116] in Intel CPU, which assist us in software instrumentation with minimal intrusion. The debug features help us set both hardware and software breakpoints, watchpoints, examine internal registers and modify them when the system is running. A developer can control the execution behavior while debugging a code. Although traditional debug features are intrusive, ARM provides non-intrusive debug capability. Trace is streams of real-time information of instructions and data transfer provided non-intrusively by the processor. Ideally, this embedded trace feature can immensely help gather system information to make safety and security assessments at runtime. The purposes of this chapter is to help characterize and understand the complex technology of Trace debugging, and provide insights to the runtime verification community that can help guide analysis of these technology options. In this chapter, we explain briefly various ARM processor families and the ARM Coresight debug and trace capability. We discuss the difference between software and hardware traces, and the benefits of using embedded hardware traces for runtime monitoring. Furthermore, we

[1] The process of extracting/recording the trace is referred to as instrumentation [114].

present some of the challenges in using these features, and characterize the options and tradeoffs that arise from the technologies.

## 6.2 ARM Processor Family

The ARM processor family is very representative of the types of processors you would typically find in a safety and security critical CPS. ARM family holdings are one of the leading companies in the world of embedded processors. For this reason, we choose to use ARM processors for our investigation.

The challenge associated with extracting information from a processor that is being monitored is the bandwidth or rate at which information can be extracted from such a processor. A technique called semi-hosting [117] can be used on a wide range of processors where at each step the processor has to halt its execution and output information to be monitored to one of the output pins. Semi-hosting slows down the processor as it stops executing the program/instructions while it outputs data by "moving" information from, for example, a register or memory to the output pins and then restarts executing the program. Secondly, it also greatly limits the information that can be output. ARM debug and trace capability helps in non-intrusive extraction of embedded trace for monitoring. We provide a brief discussion of the ARM processor family below.

The ARM processor family were named ARMv1, ARMv2 and so on to the current ARM v8 processors. After ARM v5, the processors were named based on the market that they were targeted towards. ARM designed a variety of processors (Cortex processors) with varying performance and power levels to cater to a wider market from mobile computing to defense, aerospace applications etc. ARM v6 had Cortex M microcontrollers and from ARM v7, they were broadly classified as Cortex A, Cortex M and Cortex R. Cortex A, Cortex M and Cortex R are the most important classifications of ARM processors based on its performance and computational capabilities. They are briefly described below.

***Cortex A (Application)*** – Cortex A are high performance ARM processors which can run operating systems such as Linux, Android or Windows. They are used in applications such as smart phones, tablet computers etc. They are also used in functional safety applications in autonomous vehicles [118] [119]. Cortex A can deliver up to 2GHz frequency in advanced process nodes [114].

***Cortex R (Real-time)*** – Cortex R are used for applications with high performance real-time requirements. They are used in networking equipment, automotive systems used for braking, airbags etc. Cortex R  has an interrupt system that has low latency [119] .

***Cortex M (Microcontrollers)*** – Cortex M are small and power efficient devices. They are embedded microcontrollers and do not have a Memory Management Unit (MMU) which makes it impossible to install standard OS on a Cortex M. They are mainly used in IOT devices, industrial control applications etc. [120]. Additionally, Cortex M are also used in sensor data processing, body electronics and other embedded applications in automobiles.

A trace is a sequence of instructions or streams of data and program states extracted by the processor for debugging or monitoring. Trace capability can differ in each of the ARM processor cores. In other words, a Cortex M can have a trace module called ETM and a cortex A may have a trace module called PTM. This difference makes monitoring heterogeneous processors particularly challenging. For example, there can be a multi-core application where there is a Cortex M with ETM module and Cortex A with PTM module with different kinds of traces that we may want to monitor at runtime.

The Trace capability is an important feature in the Arm processors (Cortex A, M and R) where information can be extracted for monitoring. Trace analysis can help detect difficult issues in a program as listed in Ref [121]:

1) Illegal pointers, instructions and misaligned data writes.

2) Code overwrites: This includes incorrect writes to Flash, peripheral registers (SFRs) and corrupted stacks.

3) Data that is out of bounds, for example uninitialized variables and arrays.

4) Stack overflows.

5) Runaway programs and incorrect control flow.

6) Communication protocol and timing issues.

7) Spending excessive time in certain functions than required.

The Coresight components that source trace data, collect trace data and the configuration signals that assist in collection of trace non-intrusively are discussed in the next section.

## 6.3 ARM Coresight Architecture Components

ARM processor produces different kinds of traces that reflect the operation and performance of the program. They can be instruction traces, data traces, traces that indicate memory read/writes, events counting, instrumented data, etc. Figure 59 gives an overview of the Coresight components. They can be broadly classified into ARM trace infrastructure and ARM debug infrastructure which are discussed in the next section.

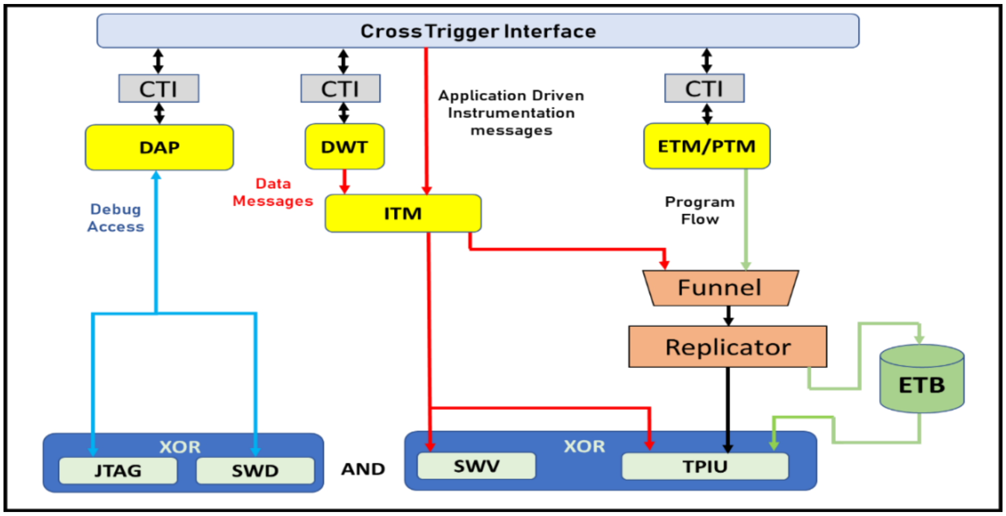


Figure 59: ARM Coresight Architecture.

### 6.3.1 ARM Trace Infrastructure

The ARM trace infrastructure consists of trace source, sink and link components. We discuss the coresight trace sources below:

#### a. Coresight Trace Source Components

Instrumentation Trace Macrocell (ITM), System Trace Macrocell (STM), Data Watchpoint and Trace (DWT), Embedded Trace Macrocell (ETM), Program Trace Macrocell (PTM) and Embedded Logic Analyzer (ELA) are the main trace sources in ARM processors [122]. These trace source modules may not be present in all cortex processor families and their availability differs with the Cortex core and the vendor implementation. For example, the ITM hardware trace is available only in Cortex M processors, Armv8 designs have an ETM, whereas most of the Armv7 designs have a PTM [122].

Further, the ETM/PTM modules have different versions, which have evolved over time starting from ETMv1, ETMv2, ETMv3, PTMv1 and ETMv4. The characteristics of each of these versions may have slight differences, for example ETMv3 provides tracing each instruction whereas ETMv4 provides instruction trace, only for branch executions.

The coresight source modules are discussed below.

***Instrumentation Trace Macrocell (ITM)* -** The ITM module is an application driven trace source in ARM that supports software trace through *Printf* style debugging and provides diagnostic system information based on application events[123]. Due to limited observability of data in a program, instrumentation may be necessary to extract traces to verify a correctness property. ITM provides traces of internal variables that can be used to access system behavior.  They are mainly used for *Printf* style debugging, to obtain application driven traces and diagnostic information about the system [123]. Although ITM can print trace data similar to how one would use a *Printf* function in C programming to send data over a serial bus, it is very sophisticated compared to the serial UART protocol. It takes a few clock cycles to write trace data to the memory mapped registers, which is less than 0.1 microsecond for a processor with ~100 MHz clock. Compared to this, other debugging methods take several milli seconds.

ITM trace can be classified into software, hardware and timestamp information which are explained below. The software trace has the highest priority, followed by hardware trace and timestamp data. ITM arbitrates between these trace packets based on their priority level, if they are generated at the same time [123].

i. **Software Instrumentation Trace** - The ITM unit has 32 stimulus channels (32-bit memory mapped addresses from Port 0 to Port 31) for transmitting the trace. Data can be written into these memory mapped ITM registers to generate trace packets [120]. ITM is minimally intrusive to the program, taking very few clock cycles (100's of nanoseconds) to emit custom trace data. The software instrumented trace in ITM can be used to perform *Printf* style debugging and the ITM data is output to the external pins either through the Serial Wire Output (SWO) or the Trace Port Interface Unit (TPIU) pins. These external interface pins are discussed in the Trace Interfaces section.

ii. ***Hardware Instrumentation Trace* –** ITM can output profiling information generated by the Cortex hardware such as exception traces, event counts, etc. [124]. ITM works with another

Cortex module, Data Watchpoint and Trace (DWT) to output these trace packets in an ITM/DWT trace stream. DWT provides a low bandwidth focused data trace using comparators to detect memory accesses and then generate trace packets with information about memory accesses. This is discussed further in the section on DWT.  DWT also provides timestamping to the ITM trace.

The ITM hardware trace packet has 1-byte Header and 0-4 bytes of Payload [124]. ITM transmits additional synchronization and overflow packets along with the system information that is being extracted in certain scenarios. Synchronization packets are forty-seven '0's followed by one '1' and is emitted during bit to byte alignment during trace transmission. An overflow packet (a number '70') is generated when an trace data is written to the stimulus registers but the FIFO is full [124].

iii. **Timestamping -** Trace packet generated by ITM can be associated with a timestamp. ITM has a 21-bit counter to provide timestamping which are relative to the trace packets. The ITM timestamp counter is clocked by the core clock of the Cortex processor or the bit rate clock of the trace interface (e.g. Serial Wire Viewer interface, explained later the chapter) [124]. The time stamping can also be generated by a Global timestamp generator. For example, Cortex M4 processor has a 48-bit counter that can be used to generate timestamps [125, p. 4].

***System Trace Macrocell (STM)*** – STM provides non-intrusive, time stamped software instrumentation and system trace [126]. They were designed for high performance Cortex A processors and are not present in Cortex M cores. Both ITM and STM provide instrumentation traces. But there are key differences between them in their performance, system awareness etc. as discussed in [126]. A few of them are listed below:

i. STMs were designed specifically for high-speed multicore microprocessor-based systems, whereas, ITMs, were designed for low speed microcontroller trace.

ii. STM has a dedicated Advanced Peripheral Bus (APB) for the incoming data from multiple sources (software, hardware and timestamp). This data is buffered to prevent incoming data from being lost. Furthermore, STM is capable of signal back pressure to resist incoming data to avoid overflow. ITM on the other hand, has a single APB bus and cannot signal back pressure. When the ITM buffer is full, the incoming trace data is lost.

iii. STM has system awareness, meaning the trace data can be associated with the system state. System state can indicate that the system is in a low-power state, give a memory transaction Error Correction Code (ECC) flag etc. This information would be useful to understand the system state when a trace is generated. ITM does not have this feature.

***Data Watchpoint and Trace (DWT)*** – DWT generates debug events for ITM such as data exception tracing, PC sampling etc.[127]. Timestamps and CPU cycles are emitted by the DWT relative to the packets. It has a 32-bit free running counter that counts CPU clock cycles. The DWT registers are configured to (a) enable the debug events such as PC counter, exceptions etc. (b) provide cycle counter for PC sampling (c) configure preset value which determines the sampling rate of the PC etc. [124]. DWT can be used to generate trace for the following events:

i. **Program Counter (PC) sampling** – The program counter can be sampled periodically to perform a coarse grain control flow analysis. It is not possible to sample every instruction using this method due to the high speed of the processors and limited speed for reading the trace. The debug adapter or a trace sink component reading the PC samples may not be able to keep up with the sampling rate, resulting in loss of packets. Furthermore, when interrupt requests are enabled in the processor, the PC sampling may provide the samples of the interrupt service routine which could be an issue when there are multiple threads in a program [124]. ITM/DWT register has a Prescalar value to select the sampling rate of the PC. The fastest PC sampling rate is every 64th clock [124].

ii. **Data read and write cycles** – Data read and written to/from a memory address can be traced by this feature. The PC address of the instruction that resulted in the read/write operation, the data that was read/written, the memory address where data was read/written can be traced.

iii. **Variable and peripheral values** – When a read or write occurs on a peripheral address by a global or static variable, we can generate real-time trace of the peripheral register values [121].

iv. **Event counters** – Time spent by the processor executing a particular function in a program can be obtained by this feature. The time spent will be obtained in terms of cycle counts. An event is generated every time the counter increments. The events that can be traced with the event counter as discussed in [121] are below.

- Number of CPU clock cycles (32-bit counter) - Used to track the number of clock cycles spent in a function or a piece of code.
- Number of folded instructions count (8-bit counter).
- Number of instruction cycles (8-bit counter).
- Number of cycles processor is sleeping (8-bit counter).
- Number of cycles spent in interrupts (8-bit counter).
- Number of cycles spent in load/store operations.

v. **Exception entry and return** – This creates an event when an exception is entered and while leaving an exception. A number is associated with each type of exception in the ARM processor that can also be traced. For example, a non-maskable interrupt exception is exception number 2, memory protection errors is exception number 4, and so on for a Cortex M3 processor [128].

***Program Trace Macrocell (PTM)*** **-** PTM is a trace module in the Arm Cortex architecture which provides high speed instruction traces. It is non-intrusive, meaning it can provide traces of all the instructions executed without interfering with the working of the system. Since, PTM can capture traces at processor clock speed, we need special debug and trace probes to analyze the PTM information without loss of packets. *PTM instruction trace* is generated when there is a change in control flow in the program execution. Instructions that cause a change in control flow are called the waypoints. Waypoints can be caused by indirect branches, conditional and unconditional direct branches or exceptions such as interrupts, system reset, etc. Examples of an indirect branch that causes change in control flow as discussed in [129] are:

- Load instructions with PC as the destination address.
- Data operations such as MOV and ADD instructions that have PC as the destination address.
- Branch instructions that move register values to PC.

PTM data are highly compressed and sent as packets comprising of a header and a payload as per the Program Flow Trace (PFT) protocol. In the PFT protocol, the header is 1 byte while zero or more bytes are allocated for the payload. A header indicates the type of trace generated, for example, branch packets, synchronization information. etc. The PTM packet header for an instruction trace emits a sequence of *Atoms* that indicate the execution of instructions. The Atoms are of three types [130]:

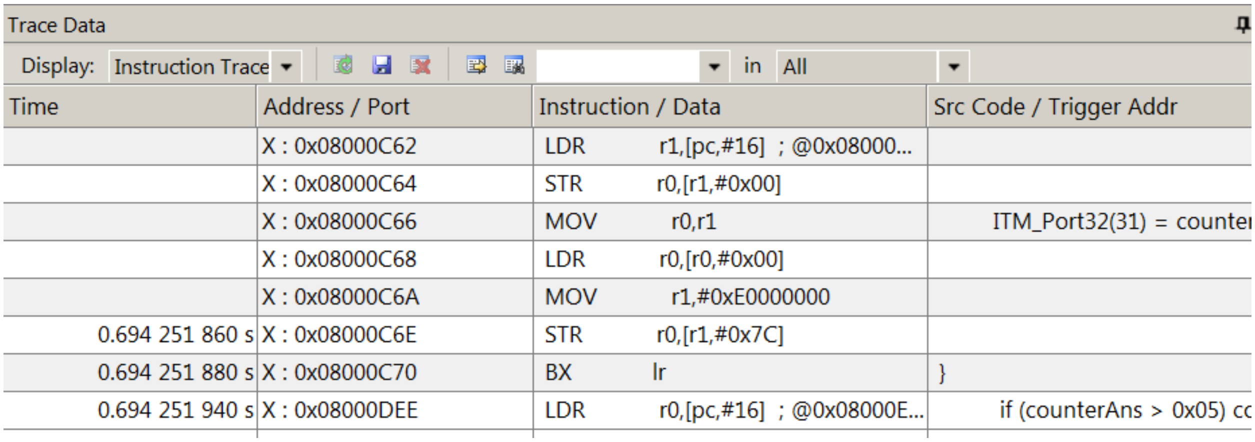
1. E Atom – This indicates that the branch instruction was Executed, and the instruction passed the conditional code.

2. N Atom – This indicates that the branch instruction was Not taken, and the instruction failed the conditional code.

3. W Atom – This indicates cycle boundary and occurs when cycle accurate mode is enabled.

The packets generated by PTM include synchronization information (A-sync for alignment synchronization, I-sync for instruction synchronization), branch address, waypoint update (indicates change in waypoint) etc. When a direct branch occurs, PTM emits an *Atom*, indicating if a branch was taken or not. Similarly, when an indirect branch occurs and if the condition code check fails, an N *Atom* is generated. But, for an indirect branch where a conditional code is satisfied, PTM traces the destination address by generating a branch address packet along with the *Atom*.

***Embedded Trace Macrocell (ETM)* –** ETM is a trace module similar to PTM, which provides high speed traces. Unlike PTM which produces only instruction traces, certain versions of ETM can produce both data and instruction traces. However only one of these traces (i.e. data trace or instruction trace) can be accessed at a given time [129]. The trace capability, timestamping, trace packet configuration is similar to PTM. Figure 60 shows a sample ETM trace viewed on a Keil Micro Vision IDE.



Trace Data

Display: Instruction Trace in All

| Time | Address / Port | Instruction / Data | Src Code / Trigger Addr |
|---|---|---|---|
| | X : 0x08000C62 | LDR r1,[pc,#16] ; @0x08000... | |
| | X : 0x08000C64 | STR r0,[r1,#0x00] | |
| | X : 0x08000C66 | MOV r0,r1 | ITM_Port32(31) = counte |
| | X : 0x08000C68 | LDR r0,[r0,#0x00] | |
| | X : 0x08000C6A | MOV r1,#0xE0000000 | |
| 0.694 251 860 s | X : 0x08000C6E | STR r0,[r1,#0x7C] | |
| 0.694 251 880 s | X : 0x08000C70 | BX lr | } |
| 0.694 251 940 s | X : 0x08000DEE | LDR r0,[pc,#16] ; @0x08000E... | if (counterAns > 0x05) cc |

Figure 60: ETM trace viewed on a Keil Micro Vision IDE.

The ETM *data* and *instruction* traces as encoded as per the ETM signal protocol. The details of the protocol can be found in [129]. *ETM data trace* generates information about the data accessed from a processor. For example, when a load or store instruction is executed, the data trace emits the load/store address along with the associated data.

*ETM instruction trace* is similar to PTM. ETM trace is highly compressed and sent as packets with a header and a payload with the *Atom* information. The *Atoms* have special encoding values to efficiently indicate the trace execution. The details can be found in [129] . The packets generated by ETM include alignment and instruction flow synchronization (A-sync and I-sync) information, branch address, data, timestamp to name a few. The ETM packets are explained in detail in [129].

ETM and PTM can generate cycle accurate trace and timestamp information along with the trace. Although this is very useful for calculating the execution time of a code, it may be an issue with systems with limited communication bandwidth and/or processing capability as the trace generated by PTM/ETM can be in the several GBits for a few seconds of trace recording. This issue is discussed further under the section on trace decoders.

To address large data, trace modules have filtering capability where only a part of the code is traced. The ETM has a better data filtering capability than PTM [122]. ETM filters and triggers can be used to reduce the trace generated and obtain ETM trace only for a region of interest in the code.

***Embedded Logic Analyzer (ELA)*** - ELA provides low level visibility of signals in an ARM processor and monitors the bus signals and memory issues. It enables hardware assisted debugging to detect data corruption and deadlock scenarios. Additionally, ARM ELA-600, which is a current generation ELA can output the ELA trace data to an external interface through a AMBA Trace Bus (ATB) protocol [131]. ELA helps debug design and detect hard to find bugs by providing good observability and control of the system at runtime.

### b. Coresight Trace Sink Components

The trace emitted by the coresight source modules are captured by Trace Memory Controllers (TMC) and trace interfaces which drive trace data to external pins on a target. Trace can also be streamed to High-Speed Serial Trace Port (HSSTP). HSSTP is a serial port which can be used to read trace data at almost 12 Gbits/s [132].

Trace using TMC are discussed below.

***Trace Memory Controllers*** - The TMCs can be an Embedded Trace Buffer (ETB), Embedded Trace FIFO (ETF), Embedded Trace Router (ETR) or Embedded Trace Streamer (ETS). We briefly discuss the TMCs below:

i. **Embedded Trace Buffer (ETB) -** ETB is a small SRAM memory used to store ETM/PTM traces natively on-chip, for later analysis. The ETB acts as a circular buffer that wraps around, replacing the old trace data with the new one, when it is full. The size of the ETB is small, ranging between 4KB to 64 KB. Therefore, the filtering capabilities of the ETM/PTM trace would be helpful in reducing the amount of traces that need to be stored and ensuring that the trace data is not lost due to buffer wrapping [122].

ii. **Embedded Trace FIFO (ETF)** – ETF can be configured to be used as a circular buffer like the ETB or as a hardware or software FIFO. The hardware FIFO mode can be used to remove fluctuations in the incoming trace data. In the software FIFO mode, ETF can be used to read out trace data over a debug interface.

iii. **Embedded Trace Router (ETR)** – ETR can be used to route trace data to system memory or any AXI slave over an AXI interface. It is similar to ETF and ETR, having a circular buffer and software FIFO modes but has a much larger trace storage capability.

iv. **Embedded Trace Streamer (ETS)** – ETS sends trace data to a streaming device through an AXI-Stream interface. The features in an ETS are a subset of the ETR, where those features not related to trace data streaming are excluded.

***Trace Interfaces* –** PTM, ETM, ITM, DWT and Debug Access Port (DAP) are accessed through the following interface units in the ARM processor. DAP is a debug port in the ARM processor which is used for non-intrusive debugging. We discuss more about DAP in the section on ARM debug infrastructure. Many of the interface units discussed below are a part of both debug and trace infrastructure in ARM.

1) **Joint Test Action Group (JTAG):** JTAG is a serial port, industry standard referred to as IEEE standard 1149.1 for testing Printed Circuit Boards (PCB). They are used in a number of hardware including FPGAs, processors to perform boundary scan tests. The Coresight modules can be accessed through the JTAG interface. JTAG has 4 dedicated pins and an Test Reset pin which is optional [133]. The functionality of the 4 pins are as follows:
    - Test Data In (TDI) – serial data input
    - Test Data Out (TDO) – serial data out
    - Test Clock (TCLK) – Clock signal used for JTAG communication
    - Test Mode Select (TMS) – select the device in JTAG chain

2) **Serial Wire Debug (SWD)** – SWD is a debug port which is an alternate to JTAG pins which provide the same functionality as the JTAG but with fewer pins [134]. The overlay connections of SWD with JTAG are shown below [135]. The SWO pin provides data trace capability through the SWV interface.
   TCLK => Serial Wire Clock (SWCLK)
   TMS => Serial Wire IO (SWDIO)
   TDO => Serial Wire Output (SWO)

3) **Serial Wire Viewer (SWV)** – SWV is used to get high speed data trace output from the ITM/ DWT modules of the Cortex processor non-intrusively. SWV uses the SWD and SWO port to provide real-time trace [134]. The data trace is provided through the SWO pin of the SWV interface. SWO is a single pin asynchronous output channel. This output can be read through debuggers such as St-link, Keil U-link etc.

4) **Trace Port Interface Unit (TPIU)** – This is a 4-bit parallel interface to output traces from ARM trace sources. TPIU can accept trace directly from the coresight modules (e.g. ETM, ITM) or through a trace funnel. The TPIU manages stopping of race capture from the coresight source when it receives a trigger. It inserts a unique ID based on the trace source into the trace stream output to TPIU external pins. This helps a trace analyzer associate the trace data with its corresponding source [122]. Many coresight modules output the instruction traces through the TPIU interface.

### c. Coresight Trace Link Components

***Funnel* -** Coresight trace funnel is used when there are multiple trace sources in a processor. The funnel combines multiple trace streams coming from an AMBA Trace Bus (ATB) onto a single ATB. The trace from the single ATB is eventually sent to a trace sink such as ETB or to the external pins, for example, through a TPIU interface.

***Replicator* –** The replicator splits a trace on a single ATB bus into multiple channels to route it to different trace sinks.

***Cross Trigger Interface and Cross Trigger Matrix (CTI, CTM) –*** CTI is used as communication interface between various coresight components. One of more CTMs connect the CTIs with each other. For example, in a multi core system, the CTI and CTM can be used by the debug unit to send trigger and debug commands, and communicate about debug events among them [122] [136].

### 6.3.2 ARM Debug Infrastructure

The traditional approach for debugging is the start/stop debugging method. In this method, the program's execution is halted when it reaches a breakpoint and the user analyzes the system state. Such an approach is highly intrusive and can introduce timing and synchronization issues while debugging concurrent programs. Furthermore, it is extremely time consuming and requires running the software multiple times to detect the root cause of an error. In other words, the program execution has to be halted (started and stopped) at multiple brake points in order to perform this type of debugging.

The ARM coresight debug and trace components help address these limitations of traditional debugging approaches. The debug components and the trace components help provide a non-intrusive method of observing the system states through the Debug Access Port (DAP).

DAP is used to perform debugging such as setting break points, watchpoints, single stepping through the code, reading register contents and memory access. The memory read/write can be performed on the fly without processor intervention, taking no CPU cycles. DAP allows an external debugger to use load and store instructions to access registers and memory locations without the processor entering the debug mode. The DAP module can be accessed with the JTAG or Serial Wire interface which are discussed in section on *Trace Interfaces* [121].

## 6.4 Use of ARM Embedded Trace for runtime monitoring

The coresight trace modules that were discussed in the previous sections can be configured to generate data traces and used for Functional monitoring of a CPS. The instruction traces from PTM/ETM can be used for execution monitoring. Additionally, the PC sampling and Cycle count trace can be used for preliminary assessment of the execution behavior of a system in Cortex M devices.

Figure 61 summarizes the design choices that one has to make while using embedded hardware trace in ARM processors for execution monitoring. These design choices will depend on the ARM processor core used for implementing the application, need for real-time trace analysis or offline analysis of trace data. Note that this is not a comprehensive set of choices but provides an overview of things to consider while using Coresight trace for execution monitoring. In this section in addition to summarizing each of these decoders we briefly discuss the work we have performed with some of the decoders.

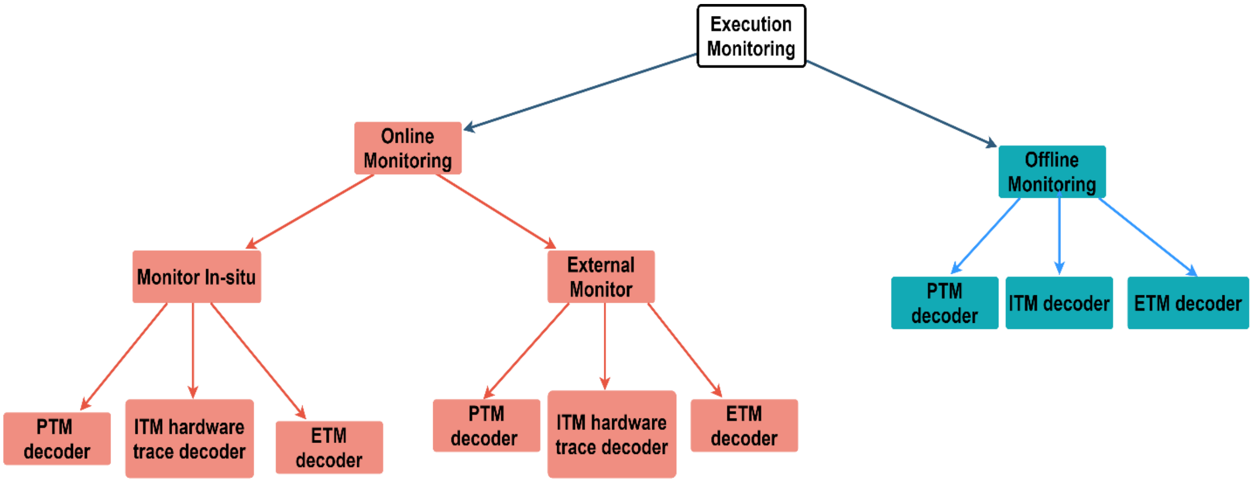


Figure 61: Design choices for execution monitoring that use ARM Coresight Debug and Trace.

The first choice is to decide if we want to perform online monitoring or offline monitoring.

## 6.4.1 Online monitoring

Online monitoring is real time analysis of trace data and on-the-fly validation of system behavior. The trace bandwidth of the embedded trace units can exceed 10 G bits/s, while the processing bandwidth of the trace analysis devices are typically several magnitudes lower [137]. This mismatch makes online monitoring of trace data challenging. But, there are novel tools proven to perform on-the fly evaluation of trace, which have very high bandwidth processing capability [138]. Additionally, the trace data can be filtered to monitor only a part of the code, which reduces the trace data.

### *Online monitors In-situ and external*

The runtime monitor that performs online monitoring can be on the same hardware integrated with the system under observation (Monitor in-situ) or it can be on an external platform (External Monitor). We have explained the advantages and limitations of both these choices in Chapter 3. The challenge with external monitor is the porting of high-speed trace data from the monitored system to the monitor. The interface ports on board of the System Under Observation (SUO) through which the trace data is read, can have a bottleneck on how fast the data streams can be read. Additionally, if we configure enabling the trace registers through a debug adapter, we are limited by the speed of this adapter. For example, if we enable an ITM hardware trace through an St-link debug adapter, we are limited to a maximum speed of 2 MHz to read the trace.

The advantage of in-situ monitor is that we can read trace directly from the trace sink such as ETB or the high speed AMBA trace bus, thus avoiding the limitations imposed by the external board interface pins. But this method may not be preferred in certain applications as the sharing of resources and memory with a monitor can bring additional safety and security concerns.

### 6.4.2 Offline Monitoring

Offline monitoring is collection and storing trace data for later analysis. This type of monitoring is minimally intrusive and communicates with the SUO while logging of events. Once the processor stops execution, this data is sent to an external monitor to verify a property or execution behavior [139]. This type of monitoring can detect a violation only after the processor has completed a run. Hence, this only double checks the system behavior and cannot be used for CPS that need real time monitoring and mitigation.

Once these choices are made, the coresight modules are configured to generate traces. Thereafter, the following steps are necessary to monitor the trace: (a) Trace Decoding (b) Trace reconstruction for instruction traces.

In this dissertation, we have explored multiple trace decoders for online and offline analysis of instruction traces. Furthermore, we have also used the ITM data that can be used to perform functional monitoring. These decoders are explained in detail in the next section. We have not explored trace reconstruction for the instruction traces.

## 6.4.3 Trace Decoders

Runtime monitors use the trace obtained from the coresight modules to verify a correctness property or execution behavior of a program. The raw trace data is usually not human readable, it can be highly compressed and is encoded in a protocol specific to a trace source. Therefore, as seen in
Figure 61, based on the Coresight trace module present in the processor that is monitored, we would need a PTM, ETM or ITM hardware trace decoder for execution monitoring.

For example, the PTM follows the PFT format, ETM trace is in the ETM signal format and the ITM hardware trace has an ITM trace format. Therefore, the first step after obtaining the trace is to decode them. In this dissertation we use trace generated by ITM, but we have also explored ETM/PTM modules. The trace decoders for these modules are discussed below.

### 6.4.3.1 ITM trace decoder

In our dissertation, we have explored online ITM trace analysis using an external monitor. We also perform offline ITM hardware trace analysis.

An external monitor can collect ITM trace from the SWO interface pin or the TPIU pins. ITM trace can be decoded using a simple UART decoder when the trace is extracted from the external SWO pin. When the ITM trace is collected from the parallel TPIU interface, we get higher bandwidth for trace collection. We would need an ITM TPIU decoder to decode the trace coming from a TPIU interface. Our monitor is implemented external to SUO on a  Xilinx XC7Z010-1CLG400C FPGA [104].

**Analysis of maximum rate of transfer for ITM traces from the ARM Processor to the FPGA monitor**

Our analysis was performed for a Cortex M Processor with an external monitor, on an FPGA. An important consideration in online external ITM trace decoder, is how fast this data (about any variables being monitored) can be transferred and whether this rate of data transfer for monitoring suffices for the application of interest.

### *a. ITM trace through SWO pin*

We perform this analysis keeping critical rate of data transfer limitations by processor, SWO pin and the FPGA in mind.

Consider the ARM processor has a clock speed of 168 MHz and is capable of sending a 4-byte (32-bit) value of a variable (ITM trace value) in parallel in one clock cycle to the SWO. The SWO serializes the 32-bit ITM into four 8-bit words that are sent to the FPGA where it is decoded by a UART ITM Trace Decoder (for our application). The SWO has a maximum transmission speed of 2 MHz that appears to be the rate limiting step. This limitation is due to the speed of the st-link debugger through which we collect trace data. There may be other ways to collect trace at higher frequency through the SWO pin by by-passing the debugger. Furthermore, the FPGA we chose for implementing the monitor was configured at a clock speed of 125 MHz.

Now let us suppose that the 32-bit ITM transmitted as four 8-bit words by the SWO and has a format: (Start Bit | 8-bit word | stop Bit). This takes about 40 clocks to send 32 bits, with clock period of 500 ns (2 MHz limited by the SWO frequency due to the st-link debugger). Hence, the time taken to transmit one ITM trace value should be about 40*500 ns = 20 micro-second.

However, we observed that transmitting information higher than 1 MHz with our single wire pin-to-pin connector was not feasible. This effectively limited the SWO frequency we could use to 1 MHz and consequently increased the transmission time to *40 micro-second per ITM data of 32-bits length.* For the ARM processor running at 168 MHz (clock period 5.95 ns) this setup is capable of transmitting ITM data approximately every 6722 clock cycles (For 168 MHz, one clock period of 5.95 ns. Since we need 40 micro-second per ITM data of 32-bits, each ITM data can only be transmitted once in 6722 processor clock cycles; 6722 =*40 micro-second*/5.95 ns).

We next discuss why these limitations are not a concern for functional monitoring of applications such as automobiles and aircrafts. In these systems, the mechanical response rate (e.g. for braking, controlling elevator, flaps, ailerons, etc.) are of the order of 10-100 milli-seconds. So even if multiple variables ~25 (32-bit ITM traces) need to be monitored and are transmitted serially these values can be monitored 40*25 = 1000 micro-second =1 milli-second, about 10-100 times faster than the 10-100 milli-second updates typically needed. With proper transmission cables allowing higher baud rates ~ 1 MHz, this is sufficient for functional monitoring.

The frequency of extracting traces is possibly more critical in execution monitoring where branching attacks, etc. may be completely missed if the traces are available only every 6722 clock cycles with the 1 MHz SWO output frequency.

### *b. ITM trace through TPIU pins*

The TPIU pins provide higher bandwidth trace compared to the SWO pin which may be particularly useful for execution monitoring. In our dissertation we have explored obtaining ITM PC sampling trace in Cortex M processors through a TPIU interface. The DWT control registers can be configured to provide PC samples every 64th clock of the processor. That would be a PC sampling rate of 2.625 MHz (168MHz / 64). When trace is extracted through st-link debugger, we are limited by the speed of the debugger. Therefore, it would result in significantly lower sampling rate.

## 6.4.3.2 ETM/PTM trace decoder

The instruction traces generated from the ETM/PTM coresight modules are highly compressed and are generated at processor clock speed. Decoding the instruction traces generated from ETM/PTM have a number of challenges. Especially, trace decoding while performing online runtime monitoring is particularly challenging. Some of the key concerns include *communication* and *processing* of the large amount of data generated during online trace-data analysis. One needs to use specialized ports and interfaces to transfer data at high speeds for online monitoring when the monitor is external to the target. While this is feasible for an external monitor, it needs special equipment. Such interfaces may not be available on all boards. Next, we require large FPGAs to allow parallelism to manage decompression and interpretation of trace at the rate it is being received.

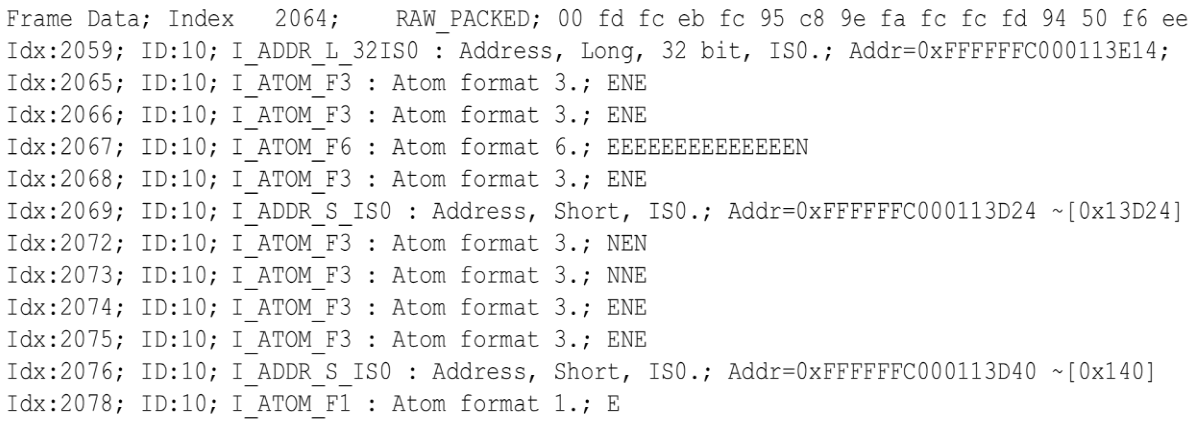
The ETM/PTM trace decoders that we have explored can be classified as software, hardware and hybrid decoders.

### a. Software Decoder

OpenCSD library from Linaro is a trace decoder that can be used for offline trace decoding and analysis [140] [141]. Online decoding of trace would require large FPGAs which would allow parallel processing of trace. OpenCSD is designed to be installed on a processor with Windows or Linux OS and would could therefore serve as an ideal tool for offline trace analysis.

OpenCSD can also be used natively on the ARM processor that is being monitored or on a separate host processor. It allows a client application to assist decoding of the trace generated by an ARM coresight trace source module. The client application acts as an interface between the OpenCSD library and the target system that is being monitored. The OpenCSD library has a decoder management component called a *decode tree*. The *decode tree* helps decode the trace in a series of stages. The client application helps configure the decode tree for a given trace sink and the trace sources feeding into that sink. Additionally, the client application provides the OpenCSD library with raw traces, region traced and memory image of the target CPS program so that trace decoder correctly follows the traced instruction sequence. The OpenCSD decoded trace consists of series of output packets which provide information such as the state of the target CPS, core events etc. The client application interprets the output packets and provides the program execution sequence of the target CPS [141].

Figure 62 shows sample decoded output from OpenCSD for an example trace provided by the installation of OpenCSD [141]. OpenCSD was installed on the host PC and was used to decode a recorded ETM trace provided by OpenCSD. In Figure 62, IDx is the byte index within the ETB buffer, for the ETM trace protocol that generated the output packet. ID denotes the ID of the trace source. I_Atom indicates the *Atom* output which is E or N in the example in the Figure 62, E indicating that the branch was executed and N indicating that the branch was not taken. For an indirect branch destination address is generated.



```
Frame Data; Index   2064;    RAW_PACKED; 00 fd fc eb fc 95 c8 9e fa fc fc fd 94 50 f6 ee
Idx:2059; ID:10; I_ADDR_L_32IS0 : Address, Long, 32 bit, IS0.; Addr=0xFFFFFFC000113E14;
Idx:2065; ID:10; I_ATOM_F3 : Atom format 3.; ENE
Idx:2066; ID:10; I_ATOM_F3 : Atom format 3.; ENE
Idx:2067; ID:10; I_ATOM_F6 : Atom format 6.; EEEEEEEEEEEEEN
Idx:2068; ID:10; I_ATOM_F3 : Atom format 3.; ENE
Idx:2069; ID:10; I_ADDR_S_IS0 : Address, Short, IS0.; Addr=0xFFFFFFC000113D24 ~[0x13D24]
Idx:2072; ID:10; I_ATOM_F3 : Atom format 3.; NEN
Idx:2073; ID:10; I_ATOM_F3 : Atom format 3.; NNE
Idx:2074; ID:10; I_ATOM_F3 : Atom format 3.; ENE
Idx:2075; ID:10; I_ATOM_F3 : Atom format 3.; ENE
Idx:2076; ID:10; I_ADDR_S_IS0 : Address, Short, IS0.; Addr=0xFFFFFFC000113D40 ~[0x140]
Idx:2078; ID:10; I_ATOM_F1 : Atom format 1.; E
```

Figure 62: An example OpenCSD decoded trace for an ETMv4 trace provided by OpenCSD [141].

The decoded execution trace can be used to reconstruct a control flow graph and perform offline analysis of execution trace.

### b. *Hardware Decoder*

The instruction trace generated from the coresight modules can either be stored in the system memory using the buffers such as ETB or directly sent to the external pins which can be captured by a high-speed trace debugger such as Keil Ulink Pro, DSTREAM, etc. for later analysis. The former method of storing trace in an ETB may have limitations due to limited system memory available on devices such as Cortex M microcontrollers to store large volume of trace. Therefore, large FPGAs are used to read data from the external pins for online monitoring of trace.

Ref [142] implement an FPGA based PTM decoder to capture and analyze trace and instrumented data generated by an ARM Cortex A9 processor. They use the data generated by code instrumentation to detect double free attacks. Freeing the resources more than once results in memory leaks and may corrupt the memory introducing a security flaw. Such a vulnerability could allow an attacker to execute arbitrary code. Ref [143] implement a hardware decoder on a Xilinx Virtex xc6vcx75t-2ff784 FPGA device to decode ETMv4 traces. The decoder has less resource utilization for both instruction and data trace and could be generalized to fit other Coresight standards with minimum alterations.

*Although the above methods are useful for online monitoring, they cannot perform assessment of the entire program behavior*. This is due to the requirement of high processing power needed to perform monitoring of large-volume trace. This is feasible if there are resources to perform online decompression and analysis. If not, there are filtering methods and triggers which are often used to reduce the trace data and extract trace only for a region of interest in the target CPS program. Also, buffering trace into trace sink components such as ETB are employed. But filtering and buffering of data have their own drawbacks. *Firstly, the cause of an observed anomaly (attack/fault) must fit into the trace window that is buffered into the ETB.* Otherwise, multiple reruns may be necessary for different regions of the code to reconstruct an execution flow. *Secondly, the anomaly must be reproducible during the multiple runs to capture trace.*

### *c. Hybrid Decoder*

An online execution monitor should be able to provide a rapid full behavioral assessment of the system operation. To achieve this, the traces must be decompressed and analyzed instantaneously. Such an approach is employed by commercial tools such as CEDAR tools® [138] that perform online trace decompression as described in [144] and performs analysis using the TeSSLa runtime verification tools [81]. The CEDAR tools® supports high speed serial interfaces and parallel interfaces to read trace data [145].

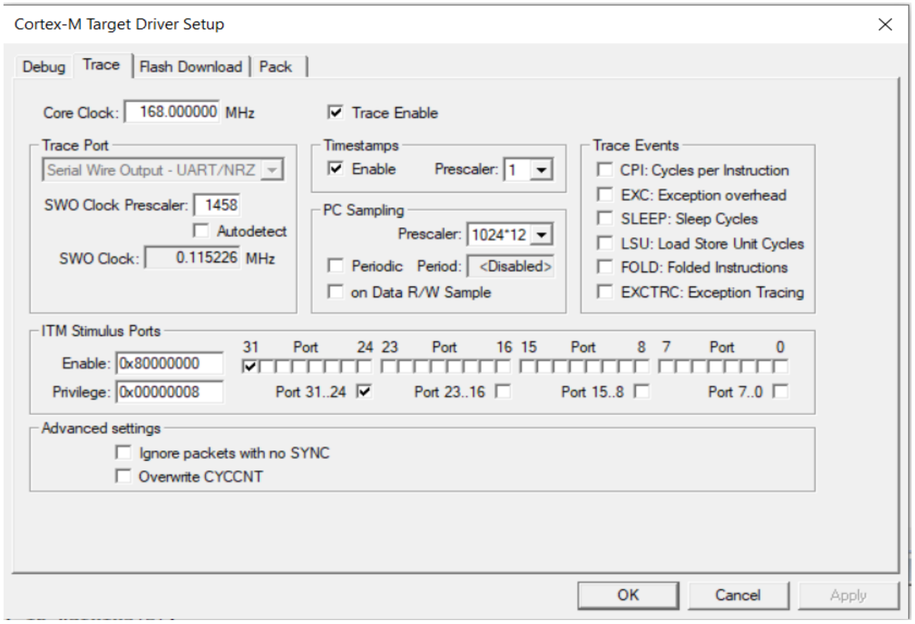
CEDAR tools® platform uses large FPGAs to decompress and interpret traces at runtime. Processing modules decode the trace data stream into messages, extract a stream of relevant higher-level events from the reconstructed control flow, and verify them against a property. This is achieved by TeSSLa specification language. Property violations detected by TeSSLa, are used to extract the most recent trace and event history for analysis. The CEDAR tools® perform monitoring with instruction traces and property verification using TeSSLa for on the fly full behavioral verification of a system. Portability of a monitors built using such tools may be an issue for some of the applications such as Unmanned Aerial Vehicles (UAVs) as runtime monitors for aerial applications due to a premium on Size, Weight, and Power (SWaP) requirements.

## 6.5 Keil and openOCD ARM development tools

We have used Keil Micro Vision[146] and an open source tool called openOCD to configure the ARM processor [147].

### 6.5.1 Keil Micro Vison

Keil IDE supports only Cortex M (Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4 & Cortex-M7) and Cortex-R4 processor development [148]. The debug and trace can be enabled by choosing the options through the Keil IDE as shown in Figure 63. Keil communicates with the processor through a debugger. We can set the prescalar value for the ITM/DWT PC sampling, enable the trace events supported by ITM and set external interface clock (e.g. SWO clock) through the IDE. Additionally, we can also enable the ITM stimulus ports though the Keil IDE.



Cortex-M Target Driver Setup

Core Clock: 168.000000 MHz ☑ Trace Enable

Trace Port
Serial Wire Output - UART/NRZ
SWO Clock Prescaler: 1458
□ Autodetect
SWO Clock: 0.115226 MHz

Timestamps
☑ Enable Prescaler: 1

PC Sampling
Prescaler: 1024*12
□ Periodic Period: <Disabled>
□ on Data R/W Sample

Trace Events
□ CPI: Cycles per Instruction
□ EXC: Exception overhead
□ SLEEP: Sleep Cycles
□ LSU: Load Store Unit Cycles
□ FOLD: Folded Instructions
□ EXCTRC: Exception Tracing

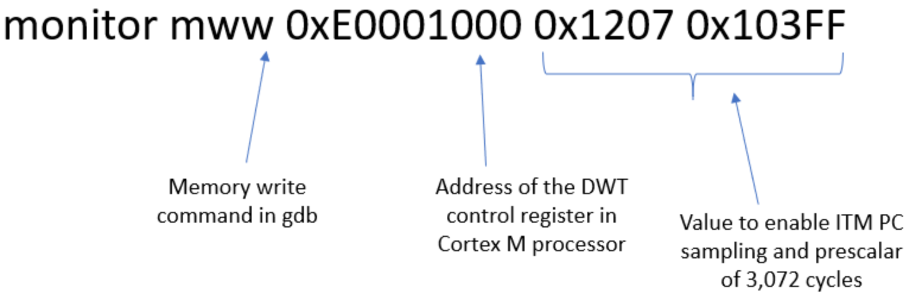
ITM Stimulus Ports
31 Port 24 23 Port 16 15 Port 8 7 Port 0
Enable: 0x80000000 ☑□□□□□□□ □□□□□□□□ □□□□□□□□ □□□□□□□□
Privilege: 0x00000008 Port 31..24 ☑ Port 23..16 □ Port 15..8 □ Port 7..0 □

Advanced settings
□ Ignore packets with no SYNC
□ Overwrite CYCCNT

Figure 63: Enabling debug and trace features in Cortex M processor using Keil IDE.

### 6.5.2 OpenOCD

OpenOCD performs debugging and trace of various ARM devices with GNU Debugger (gdb). From the GDB, we can send commands to OpenOCD to enable trace clock, configure system events and registers etc. Using OpenOCD we write directly into the registers to enable a debug or trace feature. For example, the command “monitor mww 0xE0001000 0x1207 0x103FF” from the gdb to the OpenOCD enables the DWT register (Figure 64).



```
monitor mww 0xE0001000 0x1207 0x103FF
```

Memory write command in gdb
Address of the DWT control register in Cortex M processor
Value to enable ITM PC sampling and prescalar of 3,072 cycles

Figure 64: Configuring DWT/ITM with OpenOCD. Register values from Ref [124].

## 6.6 Example of a Functional monitor using ITM trace

ITM instrumented trace can be used to extract variables to verify a correctness property. For example, if the altitude of an Unmanned Aerial System (UAS) is monitored to ensure that there is no abrupt change in value, we need to get traces of the altitude variable. Further, in an adaptive cruise control system in a car, there may be a need to monitor the speed of the vehicle to ensure that it is below a specific threshold value that is a function of the distance from the vehicle in front of it. In such cases, the variable values can be written to stimulus ports of the ITM module.

We instrumented the source code to enable the ITM trace registers and generate a trace. We have performed Printf style debugging using the ITM module which require a simple UART NRZ decoder at the monitor end. This ITM trace can be read through TPIU or SWO interfaces. We use the SWO interface for monitoring ITM trace. The SWO clock rate is derived from the processor clock rate and trace data can be transmitted similar to a UART transmitter. The SWO UART style debugging is much faster than the standard UART [117]. The output speed of SWO depends on the configured SWO speed.

Figure 65 shows an overview of ITM trace decoding and monitoring. The ITM traces generated from the ARM processor were ported to an external FPGA for decoding and monitoring.

The design was implemented on an STM32F407GTx processor with ARM Cortex M4 32-bit core. The core clock of the processor was 168 MHz and the FPGA clock was configured to be 125 M Hz. The SWO clock though which the ITM instrumented trace is read, was configured at 1M Hz. The agreed baud rate between the two devices was 1 M Hz. The FIFO size on the FPGA had a depth of 16 entities, which are 32 bits in size.

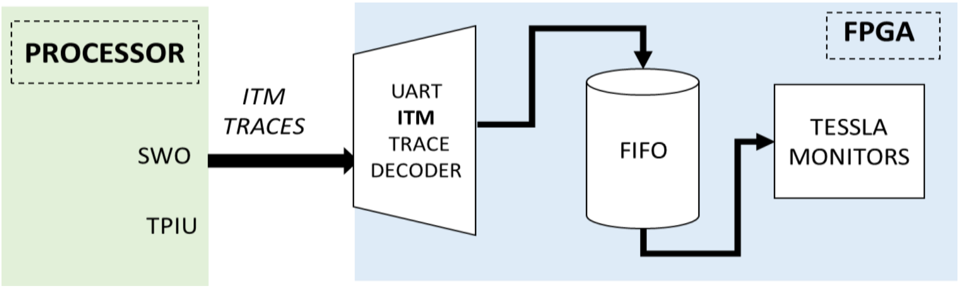
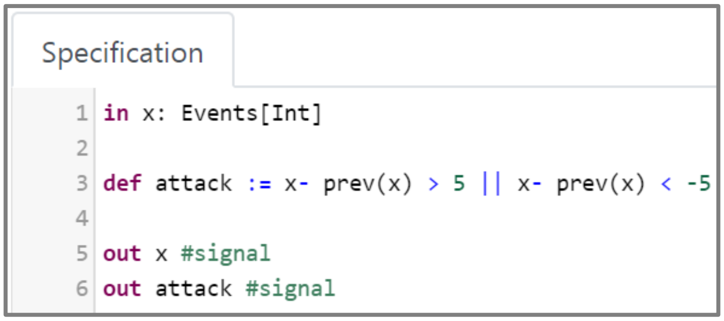


Figure 65: Sending ITM traces from ARM Processor acting as controller via the SWO to the monitor implemented on FPGA.

Runtime time monitors were implemented on an FPGA using the TeSSLa stream-based specification language. TeSSLa was used to write checking conditions to verify system behavior at the runtime monitor end. The ITM software trace received from the SWO interface were decoded by the UART decoder and then input to TeSSLa for assessment of behavior. The SWO UART protocol consists of 1 Start Bit, 8-bit word and 1 stop Bit. The SWO frequency was 1 MHz and the FPGA clock was 125 M Hz. The agreed baud rate for UART transmission was 1 MHz.

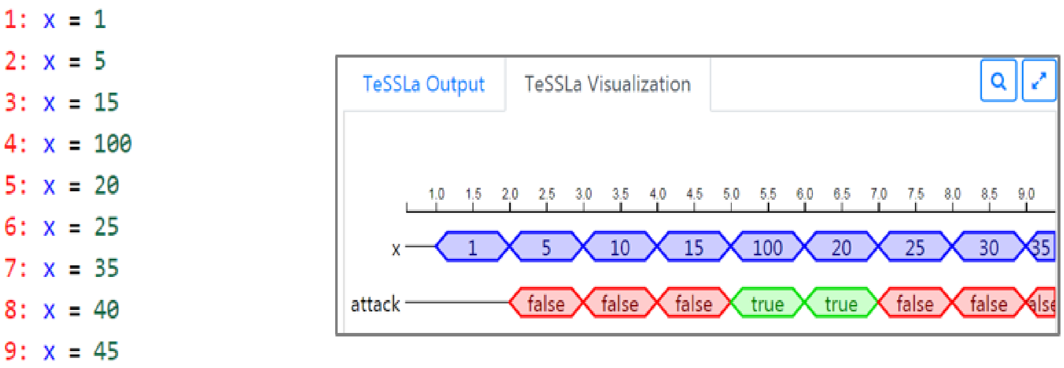
The specification in Figure 66 observes stream of variable *x* and checks if the value x changes by not more than 5 units. If an input trace violates the property, the monitor output "*attack"* indicates that the property is not satisfied.



```
in x: Events[Int]

def attack := x- prev(x) > 5 || x- prev(x) < -5

out x #signal
out attack #signal
```

Figure 66: TeSSLa specification to check for change in x.

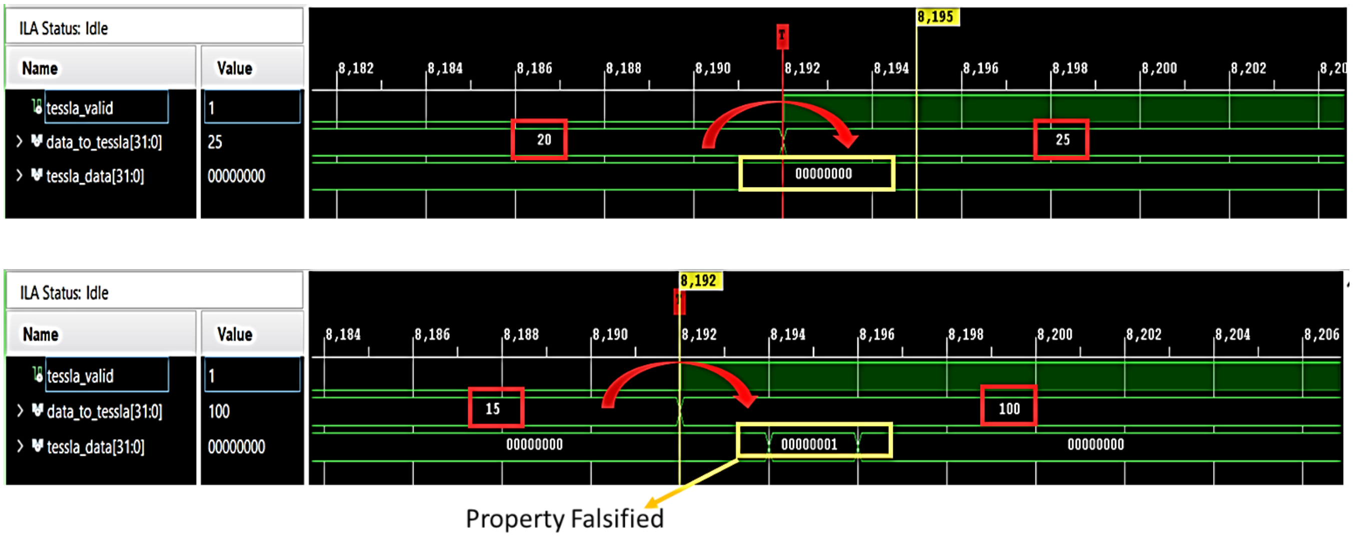
Figure 67 (a) shows an example input trace for x which are timestamped and Figure 67 (b) shows the output of the monitor. These results are from simulation inputs that are monitored by TeSSLa specifications. TeSSLa assumes that all inputs have timestamps coming from a globally synchronized clock and every event in TeSSLa consists of a timestamp and a value. However, these timestamps are not required to have any specific meaning. Depending on the specification, the timestamps can be simply increasing sequence indicators as seen in Figure 67(a).



```
1: x = 1
2: x = 5
3: x = 15
4: x = 100
5: x = 20
6: x = 25
7: x = 35
8: x = 40
9: x = 45
```

Figure 67: (a) Input trace for variable x (b) TeSSLa monitor output to detect change in x. Simulated in the TeSSLa simulation tool [86].

The decoded input traces are stored in a FIFO on the FPGA and input to the TeSSLa monitors for verification. TeSSLa, reads streams of information along with its timestamp values and verifies the correctness property. The TeSSLa results on the FPGA are observed using the Integrated Logic Analyzer in Xilinx Vivado tool. It can be seen in the Figure 68 that when the property is violated, TeSSLa indicates that it is falsified.



ILA Status: Idle

| Name | Value |
|---|---|
| tessla_valid | 1 |
| data_to_tessla[31:0] | 25 |
| tessla_data[31:0] | 00000000 |

ILA Status: Idle

| Name | Value |
|---|---|
| tessla_valid | 1 |
| data_to_tessla[31:0] | 100 |
| tessla_data[31:0] | 00000000 |

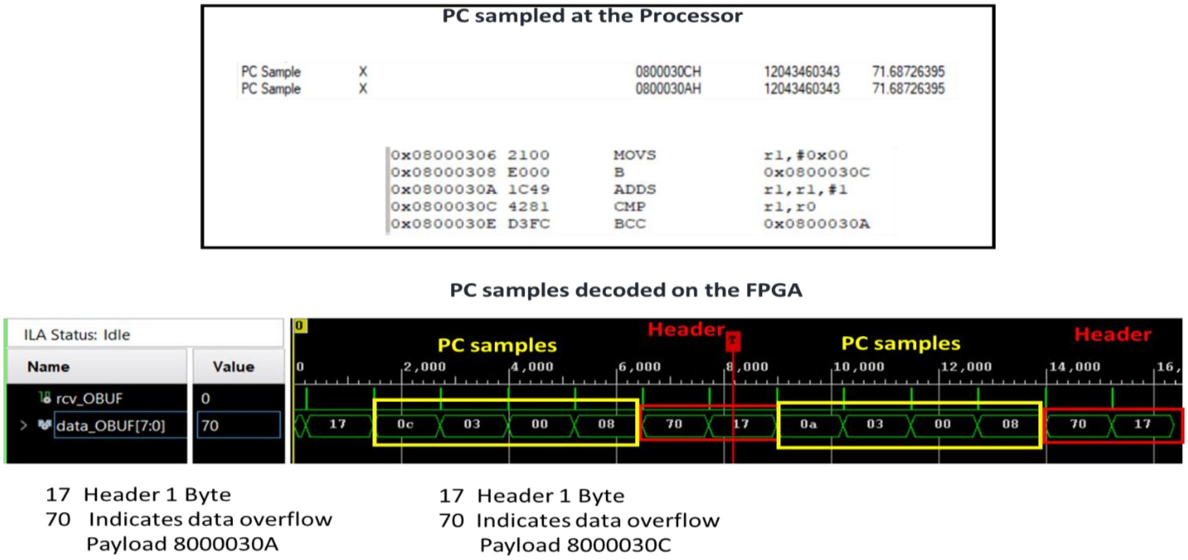
Figure 68: TeSSLa monitor output on the FPGA.

## 6.7 Example of ITM hardware trace decoding which can be used for Execution monitoring

The SUO was implemented on an STM32F407GTx processor with ARM Cortex M4 32-bit core. We experimented with both online PC sampling and decoding and offline PC sample analysis. The ITM module have registers that have to be configured to emit system diagnostic information through the hardware trace.  This was configured through the Keil IDE.

Since we extracted ITM hardware trace through the st-link debugger, our sampling frequency was very low. Online sampling resulted in data overflows as the rate at which the data was read through the st-link debugger did not match the sampling rate of the PC. Figure 69 shows the PC samples decoded on the FPGA and ‘70’ in the decoded Header of the PC sample indicates data overflow.

The data flow can be avoided if the ITM data trace is read without the st-link debugger. In this case, the processor clock was 168M Hz, SWO clock was 1 M Hz. The PC was sampled every 1024th clock.  The FPGA was configured to run at 125M Hz. When the PC was sampled at a much lower rate, there was no overflow. But, such coarse grain PC samples may not be useful to make assessment of the execution behavior of a system



PC sampled at the Processor

```
PC Sample      X                     0800030CH      12043460343    71.68726395
PC Sample      X                     0800030AH      12043460343    71.68726395
```

```
0x08000306 2100       MOVS         r1,#0x00
0x08000308 E000       B            0x0800030C
0x0800030A 1C49       ADDS         r1,r1,#1
0x0800030C 4281       CMP          r1,r0
0x0800030E D3FC       BCC          0x0800030A
```

PC samples decoded on the FPGA

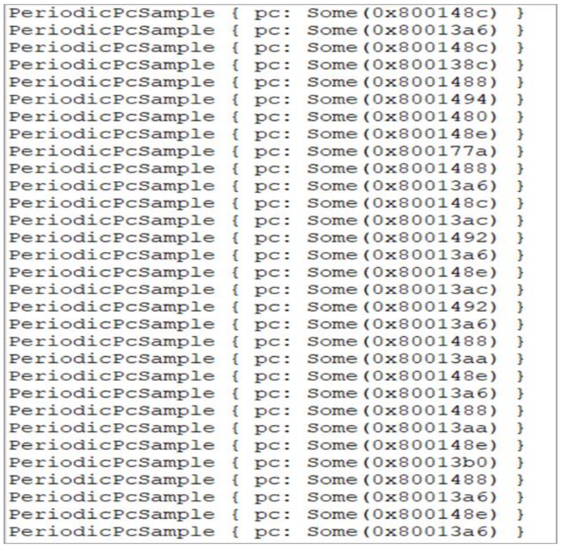
17  Header 1 Byte
70  Indicates data overflow
Payload 8000030A

17  Header 1 Byte
70  Indicates data overflow
Payload 8000030C

Figure 69: PC sample decoded on the FPGA. PC samples extracted from st-link debugger results in data overflow.

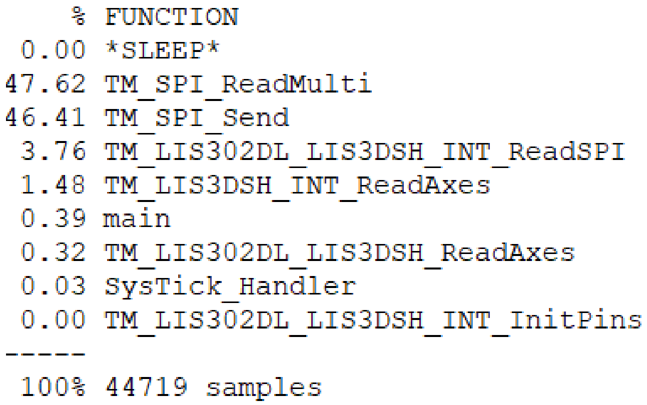
Next, we experimented with offline decoding of ITM PC samples extracted through the TPIU interface. We sampled the PC every 3,072th clock cycle, which would be about 50KHz for a 168 MHz processor clock. This approach was as explained in [124]. As mentioned earlier, these limitations can be overcome by by-passing the st-link debugger for trace collection. By avoiding the st-link debugger, we should be able to collect PC samples every 64th clock. It should be noted that execution monitoring with ITM PC samples can be performed only on Cortex M processors and we would need high speed interconnects to transfer trace from the Cortex M processor to the external monitor on an FPGA. We performed this analysis as described in [124] and used the 'itm-decoder' in [149] to perform offline decoding of ITM trace. The itm-tools can be used to get profiling information for the application through the ITM/ DWT modules as shown in Figure 71. Figure 70 shows the PC samples decoded offline.

We performed ITM PC sample decoding of an application from [150, p. 4] which reads accelerometer data on STM32F4 Discovery board. The *TM_SPI_ReadMulti()* function in the application reads accelerometer data. If the profiling information indicates execution time which significantly differs from the normal execution time for a function, we can determine that there may an issue with the execution behavior or the system is stuck in a function.



```
PeriodicPcSample { pc: Some(0x800148c) }
PeriodicPcSample { pc: Some(0x80013a6) }
PeriodicPcSample { pc: Some(0x800148c) }
PeriodicPcSample { pc: Some(0x800138c) }
PeriodicPcSample { pc: Some(0x8001488) }
PeriodicPcSample { pc: Some(0x8001494) }
PeriodicPcSample { pc: Some(0x8001480) }
PeriodicPcSample { pc: Some(0x800148e) }
PeriodicPcSample { pc: Some(0x800177a) }
PeriodicPcSample { pc: Some(0x8001488) }
PeriodicPcSample { pc: Some(0x80013a6) }
PeriodicPcSample { pc: Some(0x800148c) }
PeriodicPcSample { pc: Some(0x80013ac) }
PeriodicPcSample { pc: Some(0x8001492) }
PeriodicPcSample { pc: Some(0x80013a6) }
PeriodicPcSample { pc: Some(0x800148e) }
PeriodicPcSample { pc: Some(0x80013ac) }
PeriodicPcSample { pc: Some(0x8001492) }
PeriodicPcSample { pc: Some(0x80013a6) }
PeriodicPcSample { pc: Some(0x8001488) }
PeriodicPcSample { pc: Some(0x80013aa) }
PeriodicPcSample { pc: Some(0x800148e) }
PeriodicPcSample { pc: Some(0x80013a6) }
PeriodicPcSample { pc: Some(0x8001488) }
PeriodicPcSample { pc: Some(0x80013aa) }
PeriodicPcSample { pc: Some(0x800148e) }
PeriodicPcSample { pc: Some(0x80013b0) }
PeriodicPcSample { pc: Some(0x8001488) }
PeriodicPcSample { pc: Some(0x80013a6) }
PeriodicPcSample { pc: Some(0x800148e) }
PeriodicPcSample { pc: Some(0x80013a6) }
```

Figure 70: Offline decoded ITM PC sampling data in Cortex M devices.



```
    % FUNCTION
 0.00 *SLEEP*
47.62 TM_SPI_ReadMulti
46.41 TM_SPI_Send
 3.76 TM_LIS302DL_LIS3DSH_INT_ReadSPI
 1.48 TM_LIS3DSH_INT_ReadAxes
 0.39 main
 0.32 TM_LIS302DL_LIS3DSH_ReadAxes
 0.03 SysTick_Handler
 0.00 TM_LIS302DL_LIS3DSH_INT_InitPins
-----
 100% 44719 samples
```

Figure 71: Profiling information provides percentage of time spent in all the functions in an application.

Additionally, ITM/DWT module can be used to measure the clock cycles spent while executing the code. This could be used to measure execution time. The execution time can be measured offline and this can be compared with the online execution times that can be derived from DWT/ITM modules and run time checks can be performed to ensure timely execution of the program. But the DWT counter is 32 bits and for a 168M Hz clock it cannot measure execution time longer than 25 secs.

The execution behavioral assessments using PC sampling and profiling information can be good for preliminary assessment of the execution behavior of a system.

## 6.8 Discussions and Summary

While Chapter 4 and 5 discussed the ‘what to monitor’ and ‘where to monitor’ aspects of runtime monitoring, this chapter talked about ‘how to monitor’ a CPS using the non-intrusive debug and trace features in ARM processors. The contributions of this chapter are establishing a comprehensive perspective on the use ARM processor debug and trace capabilities to facilitate real-time runtime monitoring. The use of debug and trace features for implementing runtime monitoring for CPSs is a relatively new approach for runtime monitoring, where the research and practice is still evolving. This chapter provides a basis for researchers and practicing engineers working in the field of runtime execution monitoring to understand the complex technology of processor debug and trace, and how it aids in runtime monitoring. In addition, this chapter serves as a good introduction about ARM embedded trace and choices available while performing execution monitoring.

The various types of ARM Coresight debug and trace components we investigated suggest some innovative directions for security protection and safety monitoring. As example, functional monitoring via ITM Trace provides an effective and direct path for real-time monitoring of safety and security properties at the expense of limited execution observations. That is, functional monitoring typically only observes and extracts the variables and parameters of interest via ITM trace port. Meaning, if the property fails you don't really know why – to know why, you need lower level forensics of execution behavior to ascertain the "why". It may be possible to exploit the high-fidelity execution trace collection (such ETM) as a "on demand" limited service to support forensics when functional monitoring detects something is "wrong". It may be that such a service can be done mostly off-line – but in a way that is much quicker and responsive than current metrics of days or even weeks to complete a forensic analysis of a CPS. Such an approach is non-trivial and will require sophisticated HW and SW monitor design akin to the Cedar-tools [138]. However, such an approach would be transformational in the way we think about protecting our systems – Runtime Security as a Service in the cloud computing parlance.

We explained the ARM Coresight debug and trace components that can be used to obtain information to perform runtime monitoring. We discussed how ITM software trace can be used to extract traces of internal variables to verify system level properties. Further, we presented a number of choices one has to make to verify the execution behavior (control flow, execution time etc.) of the system based on ARM embedded trace. Depending on the requirements of an application, for example, online or offline, in-situ or external monitor, trace source (ETM, PTM or ITM hardware trace), a corresponding trace decode has to be used to decompress and decode the execution traces. We discussed some of the decoders that we have explored such as OpenCSD for ETM/PTM traces and itm-tools for ITM PC sample traces. We also discussed few references that perform online execution monitoring using ETM/PTM traces.

Finally, we discussed our implementation of runtime monitors that use ITM software trace to verify a functional property. We also provided some results on online and offline decoding of ITM traces and the challenges that we faced and how they can be addressed. This chapter provided an overview of 'how to monitor' i.e. how we can obtain information about the system non-intrusively to perform runtime monitoring.

As a standalone, this chapter serves as a reference on understanding the effective use of ARM embedded trace. This chapter has potential for future research direction on multilevel monitoring.  Each of the choices presented for executive monitoring have specific benefits and challenges and one has to determine the optimal choice for a given application.

# Chapter 7

# Realization of Multilevel Monitors on Hardware and use of ARM Coresight trace for Functional Monitoring

## 7.1 Introduction

This chapter address the ‘what to monitor’ and ‘how to monitor’ questions of runtime monitoring. We derive the hazard monitoring properties for multilevel monitors by leveraging STPA methodology. Further, we realize multilevel runtime monitors using a stream-based runtime verification tool called TeSSLa.

In chapter 5, we used the example of an Anti-lock Braking System (ABS) with monitors at three levels, all implemented in Simulink, to demonstrate the benefits of multilevel monitors. In Chapter 6, ARM Coresight architecture and extraction of embedded trace with minimal/no intrusion was described. This gives us a framework to implement the controller on an ARM processor and we also aim to implement all the monitors on hardware in this chapter. Furthermore, we also aimed to work with a CPS that was sufficiently complex to allow multiple runtime properties at different levels, particularly for the controller which included a state machine in addition to functional relationships between inputs and outputs.

Towards this goal, we used an Autonomous Emergency Braking Systems (AEB) in this chapter. We first implemented monitoring conditions derived from STPA analysis in MBE: the AEB and monitors are simulated on Simulink. We then implemented the monitors on Zynq FPGA using a stream-based verification language while the AEB was still simulated on Simulink. Finally, we also implemented the AEB controller on ARM Coresight processor and functional monitor on Zynq FPGA while only the plant and CAN bus were simulated on Simulink. We showed that various faults/attacks injected into the system were detected by the multilevel runtime monitors using a stream-based runtime verification tool called TeSSLa.

## 7.2 CPS for Multilevel Monitoring: Autonomous Emergency Braking (AEB) controller

Our representative system, which we monitor at multiple levels, is a simplified AEB system from the MathWorks Simulink examples library [151]. Figure 72 shows an abstract view of the AEB system. Here the output of the AEB controller determines the braking state that decelerates the ego car (Ego car is the car with autonomous features). The dynamics of the car under this braking condition is modeled by the Vehicle Dynamics PLANT module whose output along with the scenario under consideration (explained later) determines the inputs to the radar and vision sensors. The outputs of these sensors are fused to estimate the relative distance and relative velocity between the ego car and Most Important Object (MIO). Note the MIO is not always the lead car. For example, if a pedestrian comes in front of the ego car, this would be the MIO.

Based on these inputs (such as relative distance and velocity to the MIO), the AEB controller estimates the braking state as summarized by Figure 72 a. When the ego car is still at a safe distance but gets closer than a threshold for safe operation, an alert is issued. If the driver does not brake or the braking is insufficient, then at a certain critical relative distance, the AEB engages the stage I partial braking. If this does not suffice, a closer relative distance stage II partial braking is applied and then full braking is engaged. This decelerates the car to avoid a collision. Avoiding near-collision or a collision with a pedestrian is characterized by having a minimum headway distance when the velocity of the ego car reaches zero as shown in Figure 72 a.

### 7.2.1 Simulation Scenario

The simulation scenario we consider is as follows (from the Simulink library in [151]): The model plant (ego car) follows a lead car with a vehicle(s) on the side lane as shown in Figure 72b. The vehicle(s) covers a pedestrian from the ego car's sensors till he/she crosses into the lane of the ego car. The sensors: both camera and radar locate the pedestrian and the sensor information is fused to calculate the time needed to stop the car. When this is below a certain threshold, the driver is alerted and if sufficient braking action is not taken, as the object draws closer, the AEB initiates a first, then second stage partial braking, followed by full breaking. Thus, AEB prevents collision with the target.

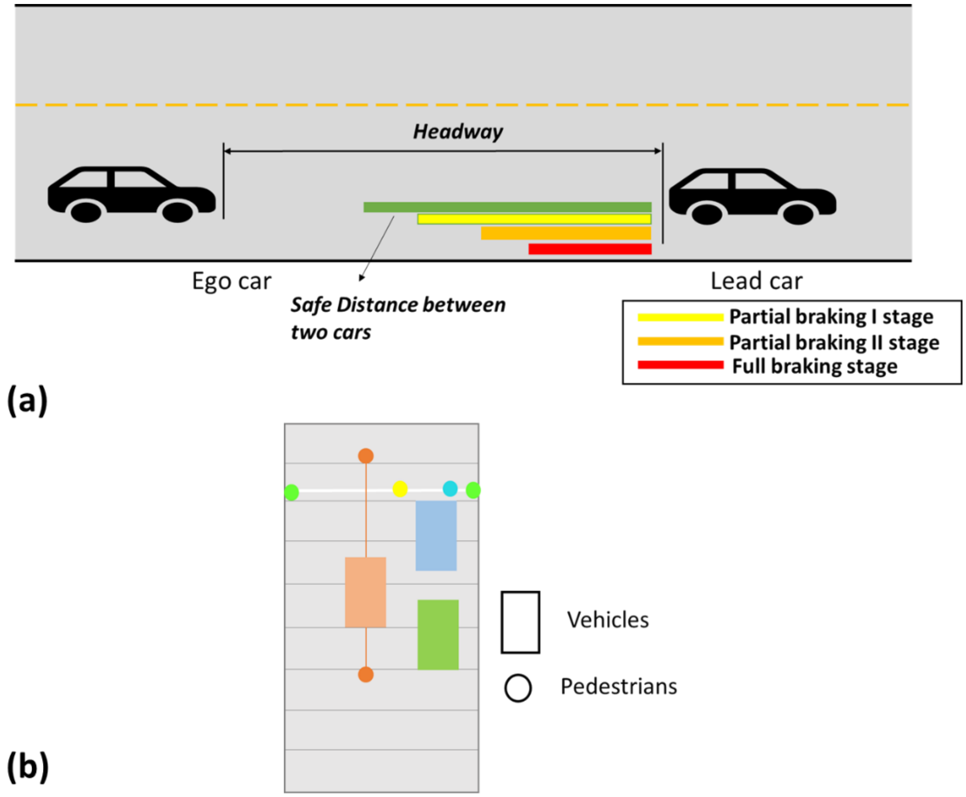


Figure 72: (a) Schematic showing the headway (distance between the ego car and lead car) and various stages of autonomous breaking initiated (b) (b) Depiction of a scenario in the Simulink simulation (an example in their library that we used) where there are two vehicles ahead of the ego car but on the next lane to the right. They conceal a pedestrian who is crossing from right to left until he/she is in on the same lane as the ego car. Both figures are based on illustrations in Simulink that have been adapted and modified here.

## 7.2.2 Examples of threats in an AEB system

Autonomous vehicles which rely heavily on a variety of sensors and network connectivity for their self-driving intellect, create several attack surfaces that can be exploited by attackers to compromise their safety. GPS, LiDAR and Camera sensors, Bluetooth connectivity, Infotainment systems, On Board Diagnostics (OBD) port, and the vulnerable Electronic Control Units (ECUs) themselves are potential attack surfaces in Autonomous vehicles [152].

Autonomous vehicles rely on LiDAR and Camera sensors for sensing their environment and making driving decisions and hence these attack surfaces are a major threat to Autonomous Vehicle safety. LiDAR spoofing which is possible with an attacker's precisely controlled light source can misguide the vehicle about obstacles around it thereby resulting in unexpected vehicle behavior including emergency braking and frozen vehicle conditions [153]. Spoofing the camera sensors could cause vehicles to misinterpret traffic signs and speed limits resulting in dangerous driving conditions for passengers.

Man-in-the-middle attacks can exploit the On-board diagnostics (OBD) port in the cars to make the ECUs operate incorrectly or erase the stored information in ECUs by sending special commands through CAN bus and even reprogram the ECUs with malicious code. Such ECUs are embedded computational elements in a vehicle and it is critical that their functionality is not compromised.

Attackers can accomplish a Denial of Service attack by disabling the ECU’s participation in the CAN bus communication through unauthorized usage of commands on OBD II port. Bluetooth interface along with an insecure infotainment system in a vehicle is another potential attack surface that allows attackers to access private information in devices connected or paired with the vehicle Bluetooth.

## 7.3 Formulating Properties to Monitor at Runtime

We formulate runtime monitoring properties from the STPA model related to braking conditions of the Autonomous Emergency Braking (AEB) system. Our approach builds on the work of [154] and extends it to include more detail about the UCAs, and we use fault injection to simulate the UCAs. For the specific example of our AEB system, the headway between two vehicles being very small is identified as a key hazard. The unsafe control actions (UCAs) that can lead to a small headway are (i) Insufficient braking action (ii) Incorrect braking (iii) Untimely braking (we limit our analysis to these UCAs in this dissertation).

Table 2 illustrates STPA with enumerated UCAs related to braking conditions. Following STPA analysis, we identify the causal factor for each of the above unsafe braking actions. Further, we study what attacks or faults can trigger each specific causal factor. With this analysis, we find that such attacks/faults occur at different levels as follows:

Table 2: Snippet of STPA analysis of UCAs in AEB

| Control Action | Insufficient or No control action results in a Hazard | Incorrect control action results in a hazard | Delayed control action results in a Hazard |
|---|---|---|---|
| Braking | AEB controller provides insufficient or no PB2 braking when a MIO is approaching the ego car.<br>(UCA 1) | AEB controller provides incorrect deceleration and no PB2 braking when a MIO is approaching the ego car.<br>(UCA 2) | AEB controller provides delayed braking when the distance from the MIO is less than safe ego car distance.<br>(UCA 3) |

i. Sensor or Data level: Wrong relative velocity or relative distance estimate due to sensor failure (for example, hardware fault on vision/LiDAR sensor) or attack (for example, sensor spoofing) leading to inappropriate or no control action due to incorrect sensor data.

ii. AEB Controller or Functional level: Insufficient control action or functionally wrong control due to an attack/fault in functioning of the controller.

iii. CAN Bus or Communication Level: Delay in communication (e.g. delay or denial of service due to injecting spurious information into the CAN Bus). leading to delayed or untimely control action.

We formulate properties to monitor at runtime for the above identified causal factors.

**i. Runtime Monitor to detect sensor inputs (causal factor) that can lead to UCA-1, where UCA-1 is: AEB controller provides insufficient or no PB2 braking when a MIO is approaching the ego car.**

The sensor fault/attack that causes UCA-1 can be detected by monitoring the properties pertaining to relative velocity and relative distance as follows:

**Property 1 (P1 at the Data Monitor):** The rate of change of relative velocity (i.e. relative acceleration) should be less than $a_{safe}$ m/s2. The relative velocity, $V_A$ and $V_B$ at time $T_a$ and $T_b$ respectively, where $T_b=T_a+T_d$, should satisfy the condition $a_r = (V_B-V_A)/T_d$, is less than $a_{safe}$ (i.e. rate of change of relative velocity for safe operation). This is based on the condition that a car has limited thrust, which restricts its acceleration (rate of change of velocity). Thus, if a LiDAR sensor is spoofed and the relative velocity changes drastically (could cause UCA-1), the attack is detected by this property being falsified.

**Property 2 (P2 at the Data Monitor):** The. rate of change in relative distance (i.e. relative velocity) should be less than $V_{safe}$ m/s$^2$. The relative distance, $S_A$ and $S_B$ at time Ta and Tb respectively, where $T_b=T_a+T_d$, should satisfy the condition $V_r = (S_B-S_A)/T_d$ is less than $V_{safe}$ (i.e. relative velocity for safe operation). Thus, if a vision/camera sensor is spoofed and the relative position changes drastically (could cause UCA-1), the attack is detected by this property being falsified.

**ii. Runtime Monitor to detect incorrect controller behavior (causal factor) that can lead to UCA-2, where UCA-2 is: AEB controller provides incorrect deceleration and no PB2 braking when a MIO is approaching the ego car.**

The fault/attack on the AEB controller that causes UCA-2 can be detected by monitoring the following properties:

**Property 3 (P3 at the Functional Monitor):** If the Time To Collision (TTC) is negative and its absolute value is less than the Partial Braking-2 stopping time (PB2), then the AEB braking status must be "AEB>=2". This property ensures that sufficient braking control action is applied in the system. If there are faults/attacks on the AEB controller, it fails to apply the required braking condition leading to incorrect control action, which is detected when this property is falsified.

**Property 4 (P4 at the Functional Monitor):** If AEB_Status is not zero (i.e. AEB=1, 2, or 3), the Throttle should be zero. This ensures that the car's throttle is not "on" when the brake is engaged by the AEB. This ensures that braking action is provided when an MIO is approaching.

**Property 5 (P5 at the Functional Monitor):** If AEB_Status is greater than or equal to 2, indicating Partial Braking-2 or Full Braking, the deceleration should be greater than 5 m/s$^2$. This ensures that when the brake is engaged by the AEB, the car should actually be decelerating faster than 5 m/s$^2$, ensuring sufficient braking action.

iii. **Runtime Monitor to detect network delay (causal factor) that can lead to UCA-3, where UCA-3 is: AEB controller provides delayed braking when the distance from the MIO is less than safe ego car distance.**

The attack on the CAN bus that causes UCA-3 can be detected by monitoring the property pertaining to the rate of arrival of information packet at a node in the network.

**Property 6 (P6 at the Network Monitor):** The time interval between two successive packet arrival in the CAN bus should be less than $T_{safe}$. This condition ensures that the consecutive packet arrival at a node from another given node, $Packet_A$ and $Packet_B$ at time $T_a$ and $T_b$ respectively, where $T_d$=$T_B$-$T_A$, should satisfy the condition $T_d < T_{safe}$. Thus, if spurious information is injected in the CAN bus leading to a delay in transmission of information (that could lead to untimely control action), the attack is detected by this property being falsified.

## 7.4 Faults/attacks and their detection by the monitoring conditions in Simulink

Each of these monitoring properties were first implemented in Simulink with local monitors at the data, computation (functional) or communication/network levels as indicated above. Then faults/attacks were injected at various levels which were detected by monitors as listed in Table 3. Table 3 shows that most faults/attacks were detected only by monitors located at the same level. We performed fault/attack injection on the Simulink AEB model. Fault injection was performed by inserting fault saboteurs, which are fault injection components that can alter the behavior of the system when activated [155]. Fault injection control signals can help characterize the fault saboteurs by defining the type of fault to be injected, duration of the fault, etc. Transient and Stuck-at faults were injected using these fault saboteurs. Similarly, data attack injection was performed by Simulink library blocks. Furthermore, the CAN bus attack was performed by injecting sporadic noise by a malicious node, thereby delaying the transmission of genuine data packets in the CAN bus. This would lead to a Denial of Service (DoS) attack due to packet injection on the CAN bus.

Table 3 summarizes the detection of the various faults/attacks that were injected. Attack on the velocity in the sensor fusion module is detected by a local monitor at the sensor level that checks for the property

(P1) that the rate of relative velocity change does not exceed a safe value. An attack on the position is similarly detected by a local monitor that checks the appropriate position property (P2) at the sensor level.

Table 3: Fault/attack injection in AEB system and detection by multilevel monitors. "Y" indicates fault/attack detected and "N" indicates fault/attack not detected.

| **Location of Fault or Attack** | **Fault/ attack Injection Type** | **Monitor property which detected the fault/attack** | | | | | |
|---|---|---|---|---|---|---|---|
| | | **Data Monitor** | | **Functional monitor** | | | **Network Monitor** |
| | | **P1** | **P2** | **P3** | **P4** | **P5** | **P6** |
| Sensor Fusion Module | Data Injection attack on velocity | **Y** | N | N | N | N | N |
| Sensor Fusion Module | Spoofing attack on position | N | **Y** | N | N | N | N |
| | | | | | | | |
| AEB controller | Stuck-at '1' fault on AEB braking | N | N | **Y** | N | N | N |
| AEB controller | Transient Fault on throttle | N | N | N | **Y** | N | N |
| AEB controller | Stuck at '0' fault on deceleration | N | N | N | N | **Y** | N |
| | | | | | | | |
| CAN bus | Denial of Service (DOS) attack due to packet injection –Attack 1 with sporadic noise of 230 msg/sec | N | N | N | N | N | **Y** |
| CAN bus | Denial of Service (DOS) attack due to packet injection – Attack 2 with sporadic noise of 180 msg/sec<br>*Note: To detect this violation, property P6 was made more sensitive to noise | **Y** | N | N | N | N | **Y*** |

At the AEB controller, stuck-at and transient faults affecting the AEB braking status, throttle, and deceleration are all detected by a localized monitor at the controller level by detecting the violations of the properties P3, P4, and P5.

Finally, a DOS attack on the CAN bus with sporadic noise at an average of 235 packets per second leads to a delay in the transmission of signals from the sensor fusion module to the AEB controller and is only detected by the local monitor (property P6) at the communication level. However, for a different sporadic noise condition (even a slightly smaller average of 180 packets per second), the delay occurring at a specific time could lead to the violations in the property P1 as well (i.e. also detected at the sensor level). The reason for data violations is as follows:

The data monitoring condition that the absolute value rate of change of relative velocity is less than 1.5 $m/s^2$ is arrived at by looking at the normal functioning of the system (ego car) with the AEB controller. However, an appropriate delay in the relative velocity reaching the controller at a critical operating condition can cause the vehicle to accelerate/decelerate at an atypical rate, leading to a violation of that property. Therefore, we see violation of property P1.

Note that sometimes properties formulated for a system may be more general and may not be able to capture the exact input output relationship in a complex system. Such a property can have false positives which can be addressed by making the property more specific for a particular scenario.

## 7.5 Hardware implementation of Runtime Monitors

### 7.5.1 Workflow for generation of monitors

We follow the workflow described in Figure 73 to generate runtime monitors which are synthesizable on hardware and can be integrated with the Simulink AEB model. We formulate the properties we want to monitor at runtime in the TeSSLa specification language which is similar to the Scala language. The TeSSLa compiler converts the specifications into Verilog code. We import the Verilog code into MathWorks Simulink and integrate with the AEB system to create an FPGA in the loop (FIL) design [105]. Simulink invokes Xilinx Vivado tools to generate bitstream for the monitors and provides the interfaces between the model and the runtime monitor on FPGA. The rationale behind the above approach i.e. bringing the runtime monitors into the Simulink are:

(1) It helps connect the development and operation phases, thus providing a seamless integration to achieve dependability of the CPS. This is the main intention of the dependable DevOps continuum.

(2) This setup also helps us study the efficacy and correctness of TeSSLa monitors implemented on hardware (FPGA) in detecting the anomalies in the CPS for which they were
designed.

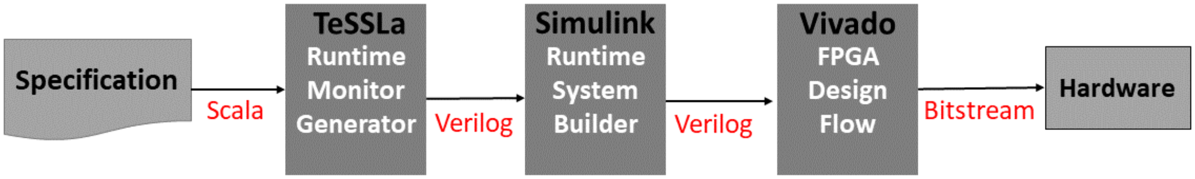


Figure 73: Workflow for generation of TeSSLa monitors and integrating them with the Simulink AEB model.

Thus, Simulink Model-Based Engineering (MBE) tools help connect design with runtime monitoring for accessing security and safety considerations early in the design development process.

### 7.5.2 Implementation of Monitors

The implementation of monitors involves designing the properties and verifying with the specification. We implemented the runtime monitors using the TeSSLa stream-based verification language. TeSSLa accepts streams of inputs and verifies it against the specifications. For every event on the input, TeSSLa verifies the specification and produces a stream of output. TeSSLa can be used to verify both timing constraints in a system and properties related to event ordering where certain events may always be followed or preceded by other events [54] [81].

Our system design consists of three levels of runtime monitors (Data, Functional, and Network) for the AEB controller system running on Simulink. As illustrated in the

Figure 74, the monitors are implemented on the FPGA (XILINX Zynq-7000 XC7Z010- 1CLG400C [104]), which runs  at 125 MHz clock.

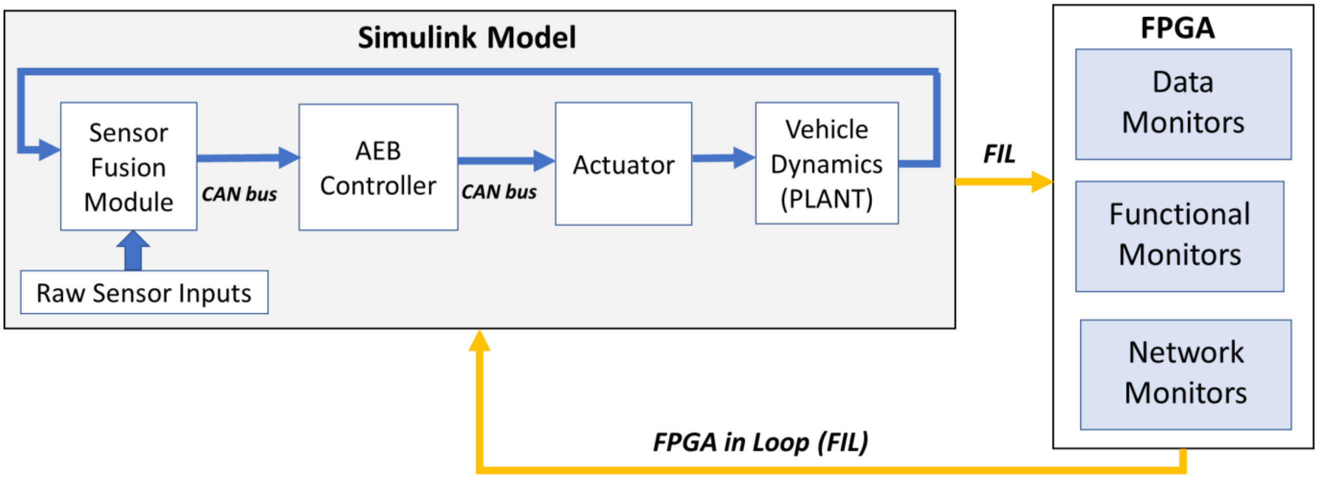


Figure 74: Schematic showing the AEB controller, plant, and sensors simulated on the Simulink Model and Data Integrity of Sensors, Functionality of AEB Controller and Integrity of Network monitored by FPGA in the loop (FIL).

### a. Simulink FPGA in Loop (FIL)

In FIL mode, the Simulink and FPGA are synchronized. i.e. Simulink waits for the FPGA to finish executing and provide an output for every simulation timestep or “clock tick”. We verify that a fault /attack injected on the CPS model is detected by the monitors on the FPGA. FIL does not consider the time taken by the monitor for processing and detection of a hazard. Therefore, when deployed on a real CPS, there may be an issue where the monitor is not able to detect a hazard in real-time for the same property since the FIL is not sensitive to the FPGA processing time. But this issue would not arise as the FPGA is usually much faster than the typical response time of a CPS that it is monitoring. To explain this better we explain some terms and give examples of different scenarios with numerical values of sensor updates, FPGA processing time, etc. below.

***Simulink fundamental time step or tick time:*** Simulink solves the state equation of the dynamical system represented by the various blocks. This is an ordinary differential equation solved numerically by various techniques such a Euler or Runge Kutta scheme. The time-step used in the numerical scheme is usually critical to the accuracy of the solution and is the Simulink “fundamental time step” or the “tick” time.

***Tick time and sample time vs. wall clock time***: The fundamental time step for a fixed time step solver is the time-step “Δt” used in solving the state equation for the Simulink model. This can be very small if accurate solution is needed. But outputting the data for each time step may be excessive. For example, for a car with typical responses in 100 milliseconds, knowing the velocity every 1 microsecond is excessive. Thus, a sample time is introduced that ensures the data is sampled for only certain multiples of the fundamental time step. For the example of a car, knowing the relative velocity every 1 ms is more than enough. Thus, the sample time (1 ms) is the fundamental time step or tick time (1 micro-second) multiplied by one thousand.

*Note that all the above is just an artificial simulation time*. It does not matter whether the Simulink program is solved in the computer in 10 microsecond or 1 second of "wall clock" time for every 1 ms tick clock.

***FPGA behavior in FIL:*** When the FPGA is implemented in the FIL mode, the time taken by the FPGA to monitor a certain monitoring condition is not important. This is because the entire plant is not a real plant physically running in real time but a Simulink simulation. Thus, if the FPGA receives an input every 1 ms but takes 2 ms to process it, the Simulink will just wait long enough that the operation is synchronized.

***Utility of FIL:*** So how does implementing "runtime" monitors on FPGA that evaluates the simulated plant dynamics (Simulink) with controller that may also be simulated on Simulink or implemented on a processor run in the processor in loop (PIL) mode, bring us one step closer to runtime monitoring of a real system (i.e. AEB controller on an actual not simulated car)?

First, it helps us find all the issues going from model-based engineering (MBE) where the monitor is implemented on Simulink to monitors actually implemented in hardware (on an FPGA in this case). This helps ensure the monitor properties are implemented correctly and as anticipated on hardware. More importantly, special stream-based languages (e.g. TeSSLa used here) are restrictive in how certain conditions and relationship can be defined given the syntax/functions available in that language. Solving all this in FIL, does bring the monitors implemented on FPGA closer to working well while monitoring an actual physical system.

***Considerations for full hardware implementation:*** Once the runtime monitors have been implemented at MBE and FIL level, what considerations must be made for implementing them on a physical system? Some important issues that can arise are (i) Noisy data that are typical of sensors in the real world, (ii) Time taken by FPGA to perform monitoring.

While (i) would involve the use of appropriate filtering and other signal processing we discuss (ii) in detail next. If an FPGA takes 100 ns, for example, to evaluate trace data being sent to it every 1 ms for checking a property, this will work well. One may only need to synchronize the very fast FPGA execution with the slow updates to the trace data by incorporating appropriate periodic delays on the FPGA.

However, if one uses a very complex monitoring condition with many internal loops, etc. and the property evaluation times in 10 ms, when data is being sent to it every 1 ms, this would not allow run time monitoring of a physical system. In this scenario, we would need a faster FPGA clock.

For the systems (AEB and ABS) we simulate in Simulink, the monitoring conditions are such that the FPGA processing times are much smaller than the physical response time of the plants and hence the above issue is not anticipated.

### 7.5.3 Observations

Our observation of the monitor response for various scenarios are as follows:

**No Fault or Attack:** We first start with the case where there is no attack so it serves as a reference. As seen in the Figure 75b, the AEB status switches from “0” to “1” at t=2.5 seconds and further changes to “2” at t=2.6 seconds as the scenario involves a pedestrian (MIO) in front of the ego car. Application of the brake decreases the relative velocity from -6m/s to -2m/s by t=4 seconds (Figure 75a) and the headway is 2.2 m at t=3.5 seconds (Figure 75c) showing effective functioning of the AEB controller. As expected, none of the monitors detect an attack as there is none (Figure 75d).

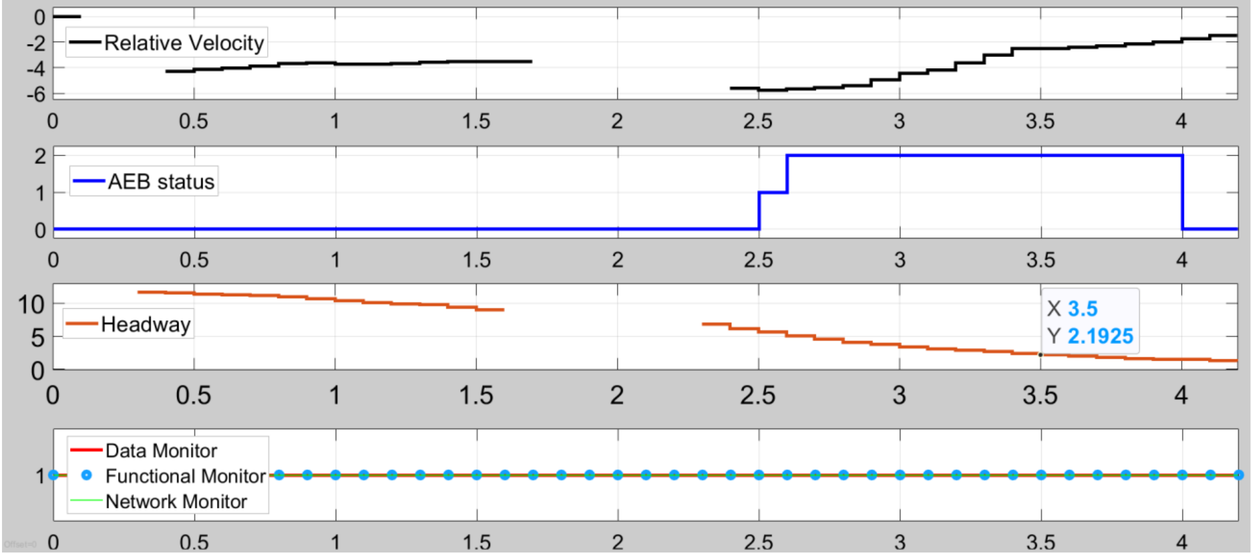


Figure 75: (a) Relative velocity (b) AEB status (c) Headway and (d) Monitor state; when there is no fault/attack.

### Fault/ Attack at Data level

We have implemented Property 1 discussed in Section 7.3 on an FPGA using the TeSSLa specification language. The property verifies that the "The rate of change of relative velocity (i.e. acceleration) should be less than $a_{safe} = 15m/s^2$".

In Figure 76a, data injection attacks on the sensor at t = 2.6, 2.8, and 3.2 seconds results in the relative velocity (wrongly) input to the AEB controller. The AEB controller, which functions correctly, does not change the AEB status from "1" to "2" at t=2.6 seconds (Figure 76 b) as it did in the no attack condition. This due to the wrong data (input) received. Consequently, the braking is less effective and the headway distance is 1.9 m instead of 2.2 m at t=3.5 seconds (Figure 76c). The TessLa monitor continuously estimates the acceleration from the stream of information for the estimate of relative velocity and detects any abrupt change in velocity (increase or decrease) of more than v=1.5 m/s between two successive data points that are t=0.1 seconds apart in this stream. This corresponds to an acceleration (or deceleration), $a = Dv/Dt > 15m/s^2$ that is indicative of an attack on the system. Figure 76 d shows that the Data monitor detects all three attacks on the relative velocity. However, none of the attacks are detected by the Functional monitor and the Network monitor as an attack on the relative velocity does not change the functionality of the AEB controller or the delay in the network.

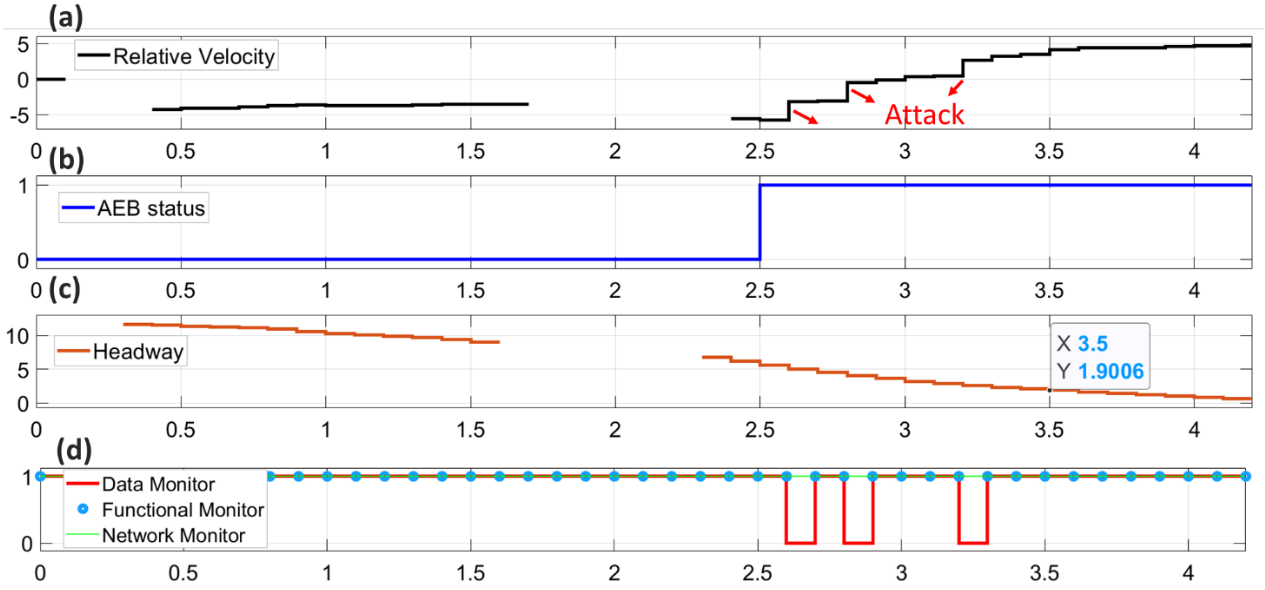


Figure 76: (a) Attacks on the relative velocity between ego car and MIO (b) AEB status with these attacks and (c) Headway with these attacks (d) All three of the attacks on relative velocity are detected by the Data monitor but are not detected by the Functional monitor or Network monitor.

### Attack at Function level

We implemented Property 3 discussed in Section 7.3 on an FPGA using the TeSSLa specification language. The AEB controller receives inputs from the sensor fusion and uses it to calculate quantities such as the Time To Collision (TTC), partial breaking-1-time, partial breaking-2, etc., and calculates the level of braking (Braking Status 1, 2, 3), if any, needed to prevent a collision. In this case, the AEB controller fails to apply the braking condition-2 (more deceleration) or complete braking due to a fault, when the partial braking condition-1 (less deceleration) does not suffice to prevent a collision as shown in Figure 77. When the condition: TTC < 0 & |TTC|< PB2 stopping time is met at t=2.6 second (Figure 77d), the controller should be applying partial braking-2. However, the brake state fails to go from "1" to "2" as shown in Figure 77c. Consequently, the braking is less effective and the headway distance is 1.7 m (Figure 77e) instead of 2.2 m at t=3.5 seconds. The fault is successfully detected by the Functional monitor due to violation of property as shown in Figure 77f. Since this is a failure in the functional relationship between the input and output of the AEB controller and not a data fault/attack (sudden change in relative velocity for example), or a change in network delay, it is not detected by the Data or Network Monitor.

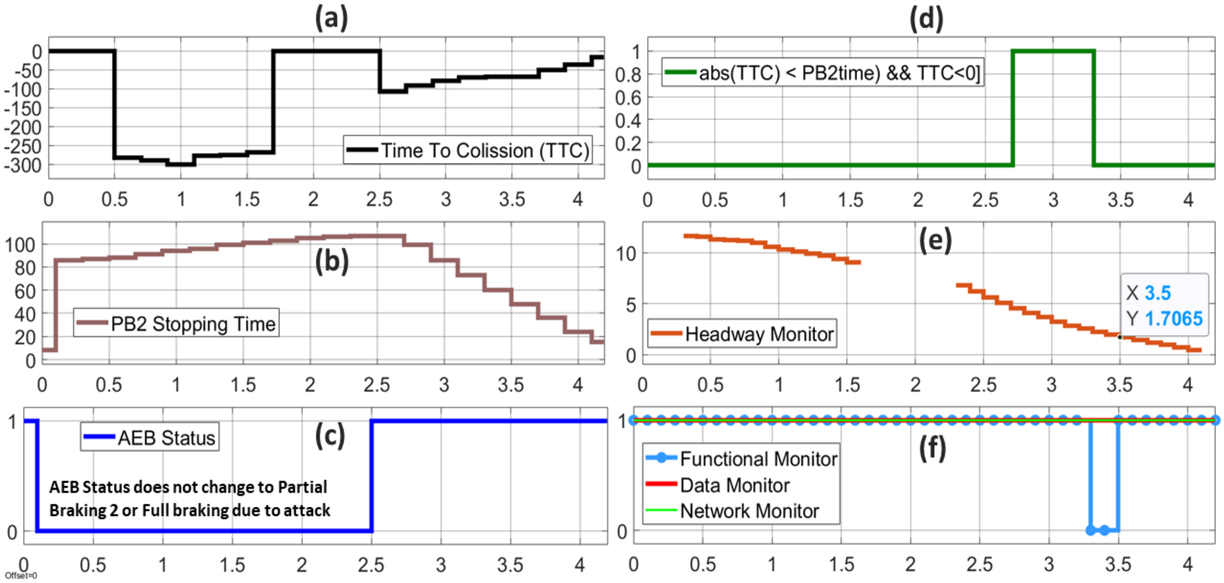


Figure 77: (a) Time to collision (TTC) between ego car and MIO (b) Partial Breaking-2 (PB2) stopping time (c) AEB controller output that determines braking status (d) Checking if a relationship between TTC and PB2 holds (e) Headway (f) Fault on AEB controller detected by the Functional monitor but are not detected by the Data or Network monitor.

**Attack at the Network Level**

We implemented Property 6 discussed in Section 7.3 on an FPGA using the TeSSLa specification language. This property monitors the rate at which the relative velocity information is being sent from the sensors to the AEB controller. When there is no noise in the channel this information (denoted by ID #6 in Figure 78a) is received every 0.006 seconds. However, when sporadic noise is injected into the CAN bus and it takes up the limited bandwidth, the time between receipt of two packets of information can exceed 0.006 seconds, sometimes reaching as high as 0.8 seconds as shown in Figure 78a. The monitoring condition is violated when this delay exceeds 0.31 seconds, which is why Figure 78d shows an attack detection at t=2.6 seconds. Figure 78b shows that this large delay in the transmission of information through the CAN bus delays the application of the brake. Such untimely braking action causes the headway in Figure 78d to be 2 m instead of 2.2 m at t=3.5 seconds.

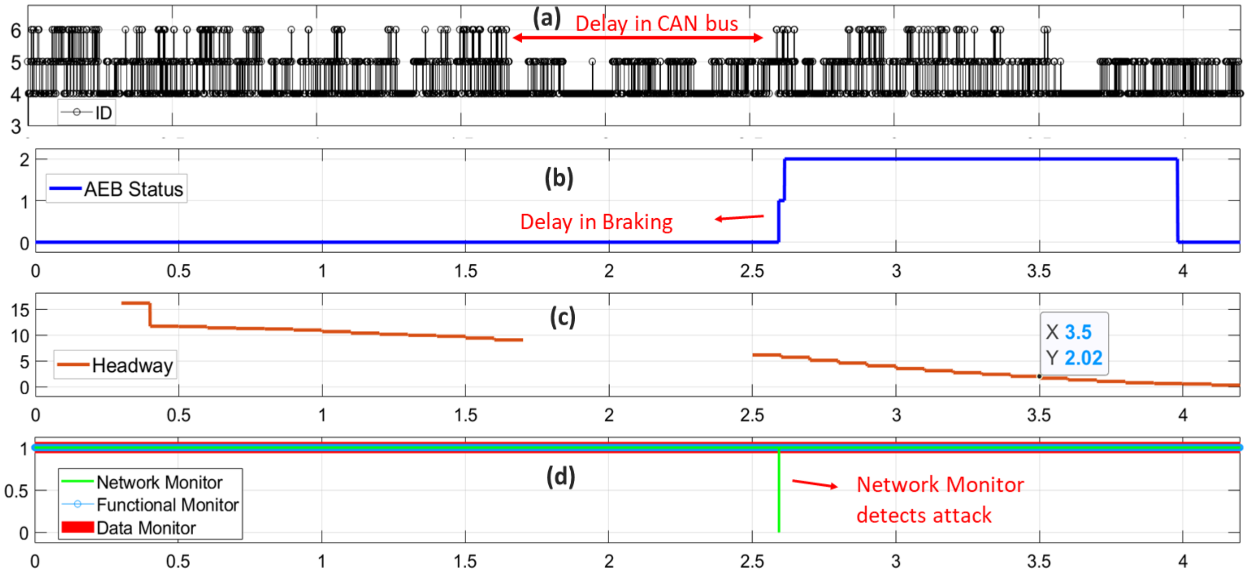


Figure 78: (a) Message ID in the CAN bus (b) AEB controller braking status (c) Headway (d) Network attack is detected by the Network monitor but not detected by the Data or Functional monitor.

In summary, local monitors at multiple levels are needed to detect attacks/faults across a CPS.

## 7.6 Functional Monitoring using ARM Embedded Trace

We implemented the AEB controller on a STMicroelectronics STM32F407VGTx ARM Cortex M4 processor [156] and the TeSSLa monitors an FPGA (XILINX Zynq-7000 XC7Z010- 1CLG400C [104]), while the plant and sensors continued to be simulated in Simulink. The hardware setup is as shown in Figure 79. ARM Coresight Architecture’s Instrumentation Trace Macrocell (ITM) was used to obtain streams of data from the processor. We performed ITM *printf* style debugging where the data to be monitored was written to one of the 32 ITM stimulus channels and was output to the external SWO pins. The core clock of the STM32 board was 168M Hz and the SWO frequency was 1 M HZ. The FPGA was configured for 125 MHz clock. The data obtained from the ITM module can be read either through a 4 pin Trace Port Interface Unit (TPIU) or through a Serial Wire Output (SWO) pin of a 2 pin Serial Wire Debug (SWD) port in the ARM processor. We used the SWD interface to read data traces due to ease of availability of decoders for this interface. But, one can always use the TPIU interface or the Aurora Gigabit Trace to get better performance. The Aurora trace unit can be used to get data of higher bandwidth more securely as it has Cyclic Redundancy Checks (CRCs) incorporated in them [157].

The STM32-MAT feature from MathWorks Simulink [158, p. 32] enables us to extract trace data for monitoring while the processor is running in Processor in Loop (PIL) mode in Simulink. In this setup, the processor communicates with Simulink through UART ports. The UART ports are used to receive sensor inputs and send controller output to the plant in Simulink.

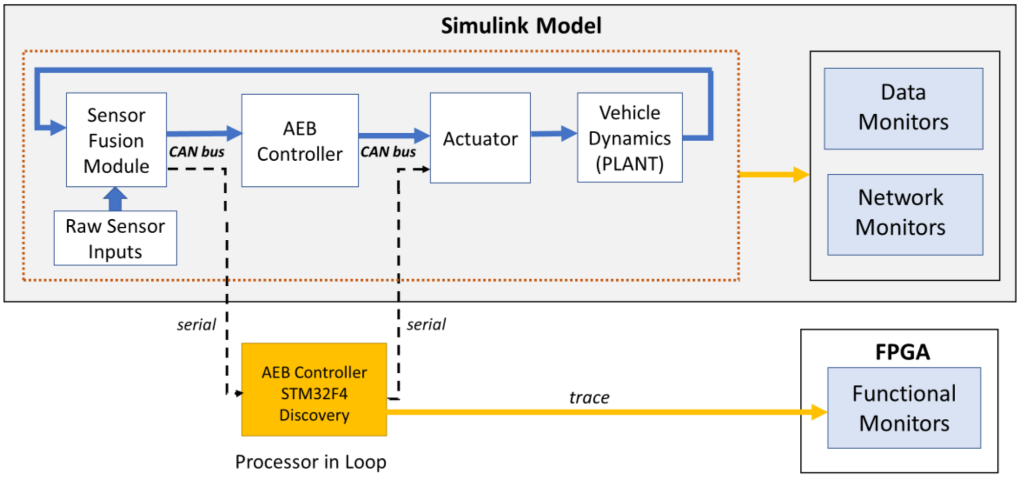


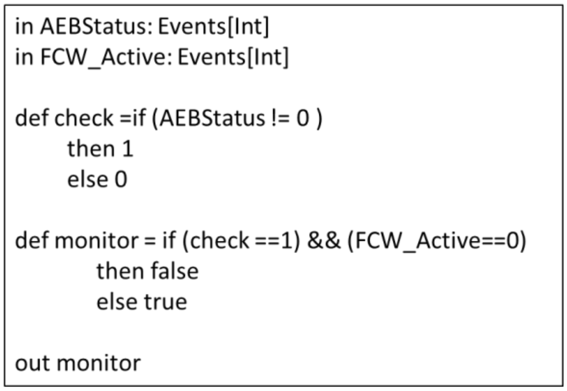
Figure 79: Schematic showing the AEB controller implemented on an ARM Cortex M4 processor and verified by TeSSLa monitors on the FPGA. Plant and sensors are simulated on the Simulink model.

### *Example of property monitored on the AEB controller using traces obtained from ARM processor*

Functionality of the AEB controller was verified by checking the following property at runtime:
“If the AEB status is not equal to zero, then the Forward Collision Warning (FCW) has to be High”.

$$(AEB\_status\ !{=}0)\ \ {=}{>}\ \ FWactive{=}1$$

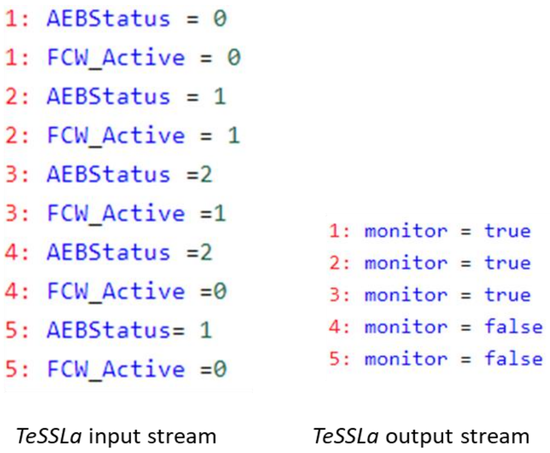
This property is based on the understanding that when the AEB controller is in any stage of partial or full braking (AEB status =1, 2, 3), the Forward Collision Warning (FCW Active) should be “1”. This property was written in TeSSLa specification language as shown in Figure 80. Verilog code can be generated from this TeSSLa specification and can be used as a runtime monitor.



```
in AEBStatus: Events[Int]
in FCW_Active: Events[Int]

def check =if (AEBStatus != 0 )
       then 1
       else 0

def monitor = if (check ==1) && (FCW_Active==0)
          then false
          else true

out monitor
```

Figure 80: TeSSLa property that verifies the relationship between AEB_status and FCW active signal.

TeSSLa accepts a stream of inputs of AEB Status and FCW Active and produces a stream of outputs at each instant of time. Figure 81, shows the TeSSLa input and output steams that were simulated using the TeSSLa simulation tool [86].



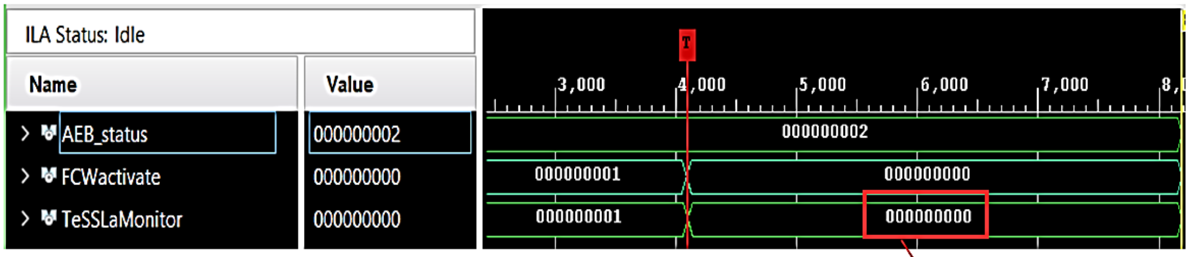
```
1: AEBStatus = 0
1: FCW_Active = 0
2: AEBStatus = 1
2: FCW_Active = 1
3: AEBStatus =2
3: FCW_Active =1
4: AEBStatus =2
4: FCW_Active =0
5: AEBStatus= 1
5: FCW_Active =0
```

```
1: monitor = true
2: monitor = true
3: monitor = true
4: monitor = false
5: monitor = false
```

Figure 81: TeSSLa input and output streams, Simulated in the TeSSLa simulation tool [86].

TeSSLa specification was implemented on the FPGA and the results of the monitor are seen in Figure 82. When the AEB Status = ‘2’ and FCW Active = ‘1’, the TeSSLa monitor output is also ‘1’ indicating that the property was satisfied. In the next case, when the AEB Status = ‘2’ and FCW Active = ‘0’, the TeSSLa monitor output is ‘0’ indicating that the property was violated.

This property was verified based on AEB status and FCW active data streams extracted from ITM. This data was decoded by the SWO UART decoded and the property was verified by the TeSSLa monitor implemented on the FPGA. The results of the TeSSLa monitor were viewed using Xilinx Integrated Logic Analyzer.



ILA Status: Idle

| Name | Value |
|---|---|
| AEB_status | 000000002 |
| FCWactivate | 000000000 |
| TeSSLaMonitor | 000000000 |

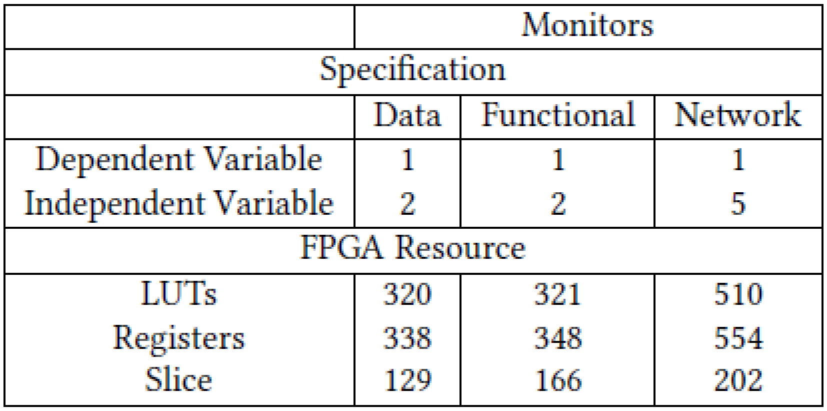
*Property violation detected by the monitor*

Figure 82: TeSSLa monitor verifies the property “If AEB Status is 1, 2, 3 (not equal to “zero”), then the warning FCW Active must be “1”. When his property is violated, the TeSSLa monitor indicates that the property has failed.

## 7.7 Preliminary assessment of scalability of TeSSLa monitors and FIL system components

The FPGA design of the monitor consists of two main components: a) the monitor design and b) the system components. The monitor design is the design description of the specification and the system components are introduced by Simulink for FPGA-in-the-loop (FIL) design. Since the system components remain constant, the scalability focus is on the monitor design which varies based on the level of the monitor. The hardware design size of the monitor is dependent on the number of independent variables, the number of dependent variables, and the logic of the monitor specification. In order to understand the scalability factor, we consider a single dependent variable in our specification and varying independent variables. Table 4 confirms that when the variables are increased from 2 to 5 (2.5×) the resource utilization increases by 1.6×. Furthermore, this resource utilization when compared with the entire FPGA chip (most utilized part considered) increased from 2.93% to 4.59%.

Table 4: Resource utilization of multilevel monitors.



| | Monitors | | |
|---|---|---|---|
| Specification | | | |
| | Data | Functional | Network |
| Dependent Variable | 1 | 1 | 1 |
| Independent Variable | 2 | 2 | 5 |
| FPGA Resource | | | |
| LUTs | 320 | 321 | 510 |
| Registers | 338 | 348 | 554 |
| Slice | 129 | 166 | 202 |

## 7.8 Summary

This chapter provides insights on how hazard analysis methods such as STPA can guide us on 'what to monitor' and 'where to monitor' in a CPS. In this chapter, we leveraged systematic hazard analysis (STPA) to assist in deriving monitoring properties from the Unsafe Control Actions at various levels in the CPS architecture. STPA analysis helps us identify the causal factors for hazards, which in turn guides us on 'what to monitor' and placement of monitors. We then designed, simulated and tested multilevel monitors in a model-based design environment (Simulink) to ascertain the effectiveness of our approach for hazard detection due to various attacks/faults. To show viability of this approach, we synthesized and implemented the monitors on FPGA and used TeSSLa, a stream-based language for runtime monitoring. We then integrated the hardware monitors into the AEB model to demonstrate the efficacy of hardware-based monitors localized at multiple levels of the CPS application.

We later used ARM cortex ITM instrumentation feature to extract data streams for monitoring. We provided an example of functional monitoring of the ITM data trace. Finally, we provided preliminary estimation on the scalability of the multilevel Monitoring approach with respect to CPS applications.

Below are our preliminary findings:

(1) We find that the systematic nature of STPA hazard analysis is beneficial in deriving and refining multilevel monitoring properties that are related to hazard causal factors. While the process of STPA is largely manual, the use of Simulink assists in the simulation and evaluation of the monitors with respect to threats and faults and loss scenarios - before implementation.

(2) Realization of multilevel monitoring appears to be scalable with respect to “at-scale” Cyber Physical Systems of the type found in critical application domains.

(3) We find Model Based Design and Engineering methods and tools significantly improve productivity in the evaluating runtime monitoring schemes with respect to hazard coverage and refinement. Furthermore, integrating runtime monitors at the model level supports the DevOps continuum to bridge development and operation phases.

(4) Hardware monitors implemented on FPGA are able to monitor streams of data using TessLa, a stream-based language that can be used to evaluate a wide range of monitoring conditions.

# Chapter 8

# Summary, Contributions and Future Work

The overarching goal of this dissertation is to demonstrate that a synergistic combination of V&V at design time and runtime monitoring at multiple levels is beneficial in assuring safety and security of a system against faults and attacks at multiple levels. This dissertation took a broad perspective on multilevel runtime monitoring of CPSs in order to develop a deeper understanding of the design time and runtime monitoring synergisms that exist at multiple levels of a typical CPS. The motivation of this broader approach was twofold; (1) a significant body of work in Runtime Monitoring is related to the development of powerful monitoring languages that can express monitoring properties of CPSs, but there is a scarcity of work related to "what" to monitor, "where" to monitor, and "how" to monitor from a systems perspective, (2) the need to uncover and characterize the system level challenges that assuredly exist when designing, developing, and implementing Multilevel Runtime Monitoring.

Towards this goal, we first presented a Model Based Design and Engineering (MBDE) approach using MathWorks Simulink tools to verify the CPS system by initially performing testing, static verification and formal verification on an Emergency Diesel generator Start Up Sequencer (EDGSS) system implemented on a Field Programmable Gate Array (FPGA) overlay architecture. We demonstrated some key verification activities at design time that help runtime verification and also explained how runtime monitoring scenarios can be considered at design time. This provides a continuum from design time assurance to runtime monitoring.

Due to the prevalence of MBDE methods in safety critical systems, we believe that our initial findings of synergy between model-based design assurance and runtime verification is impactful. One of the challenges in designing runtime monitors is to formulate informed specifications that cover vulnerable areas in a design. MBDE helps identify such design flaws and complex interactions that can cause failures in a system. Model-based verification methods can be used to address these issues and they guide us in formulating properties to monitor critical design elements and assumptions at runtime. The MBD approach can be used to access monitor organization patterns for an application, early during the design process. Our findings suggest that multiple monitor organization patterns may be an essential condition for detecting complex fault and attack patterns in a CPS. While we have shown the benefit of using MBD to arrive at the multiple monitor organization for a specific application and set of faults/attacks, this approach can be generalized to any other system.

Next, we demonstrated the benefits of multilevel monitoring framework using the example of an Antilock Braking System (ABS) in a Model Based Engineering (MBE) environment. Specifically, we used the Simulink model of an ABS and injected faults/attacks at the various levels of the ABS system and demonstrated that multilevel monitors were able to detect these faults/attacks. Specific levels of monitoring are: data (wheel and vehicle velocity) sensors, functional (ABS controller) and network (CAN Bus). We also expressed these monitor properties in Event Calculus.

While the above ABS system with monitors all modeled in Simulink, demonstrated the benefits of multilevel monitors, we felt the need to choose a more complex system, to derive monitoring conditions in a methodical manner and implement our monitors on hardware. Towards this end, we chose a more complex Autonomous Emergency Braking (AEB) system. We derived multilevel monitoring conditions at run time related to causal factors for hazards, leveraging systematic hazard analysis (STPA) to assist in deriving monitoring properties from the Unsafe Control Actions (UCA) at various levels in a CPS. We then tested the multilevel monitors in a model-based design environment (Simulink) to evaluate their effectiveness in detection of attacks/faults. To show viability, we also synthesized and implemented the monitors on FPGA. We then used TeSSLa stream-based language to integrate such hardware monitors with the CPS model to demonstrate the efficacy of hardware-based monitors localized at multiple levels of a CPS.

Finally, we implemented the AEB controller on an ARM processor, which receives input from and gives output to the Simulink model simulating the network and plant as we do not have a physical car to instrument. Such monitors observe the system behavior by analyzing streams of information coming from the target system with no or minimal intrusion to its working.

## 8.1 Key contributions of this dissertations

The key contribution of this dissertation research is to demonstrate the benefits of multilevel runtime monitoring both at the model level and implement them on hardware in detecting faults/attacks at multiple levels. We also demonstrate the benefits in using synergy between design time V&V and runtime monitoring in the design of such multiple monitors. In this process, we also made the following other contributions:

- Synergy between the V&V process at design time and creation of effective multilevel runtime verification monitors.
- Demonstrated multilevel runtime monitoring framework that observes and detects faults/attacks in a CPS at multiple levels.
- A theory of multilevel runtime verification and use of event calculus to describe such monitoring conditions.
- Ability to derive complete monitor specification from rigorous V&V process and hazard analysis.
- Hardware implementation of such monitors using a stream-based monitoring language.
- The ability to perform hardware in the loop evaluation of runtime monitors and target systems with simulated plant and network but controller implemented on an ARM processor.
- Presented design choices for usage of ARM Coresight trace capability in runtime monitoring.

## 8.2 Future work

While this dissertation lays some of the foundations for multilevel monitoring that includes their design from hazard analysis, synergistic use of design time MBE and runtime monitors, their hardware implementation and ability to detect faults/attacks at multiple levels there are several directions on which future research could be performed. These include:

1. Synergy between of design-time and runtime verification can be explored further with a tighter integration between model-based engineering, STPA and runtime monitoring. Specifically, tools can be developed to connect the workflows together in a more unified way.
2. Rigorous fault injection and attack injection experiments can be performed to collect more comprehensive data towards achieving better monitor coverage. Additionally, novel approaches such as property-based fault injection can also be performed to achieve higher monitor coverage.
3. Execution monitoring using PTM/ETM instruction traces to detect sophisticated control flow attacks can be performed.
4. Generality and scalability of multilevel monitors can be explored by using them in diverse CPSs and evaluating the resources needed for implementing such monitors on an actual system (e.g. autonomous car or UAV).

# Appendix A

## A 1. Model Based Testing and Coverage Analysis of SymPLe architecture

Figure 84 shows the coverage metrics of FBController which is a component in SymPLe.  The reason for low coverage was due to two blocks in model, Simulink HDL Coder and S-R Flip Flop. These two blocks are Simulink Library blocks that can be configured to operate in different modes. For example, the HDL counter can be configured to be free-running counter or count-limited mode where the HDL counter counts for a fixed value at a predefined sampling rate. In the FBController model, since we used the counter in the count-limited mode, the blocks in the HDL counter pertaining to other modes become dead logic in the design. Such components are justified, explaining the reason for low coverage. Simulink provides a color-coded feature where blocks with missing coverage are red and blocks with full coverage are green in color as seen in Figure 83. This helps to visually identify the coverage metrics of each block and address the low coverage blocks.  Additionally, test vectors are formulated to account for low coverage due to missing test cases.

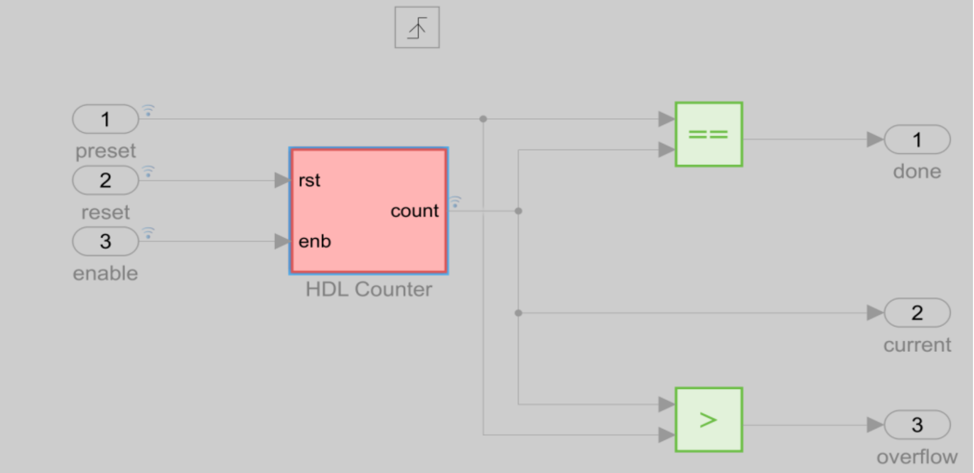
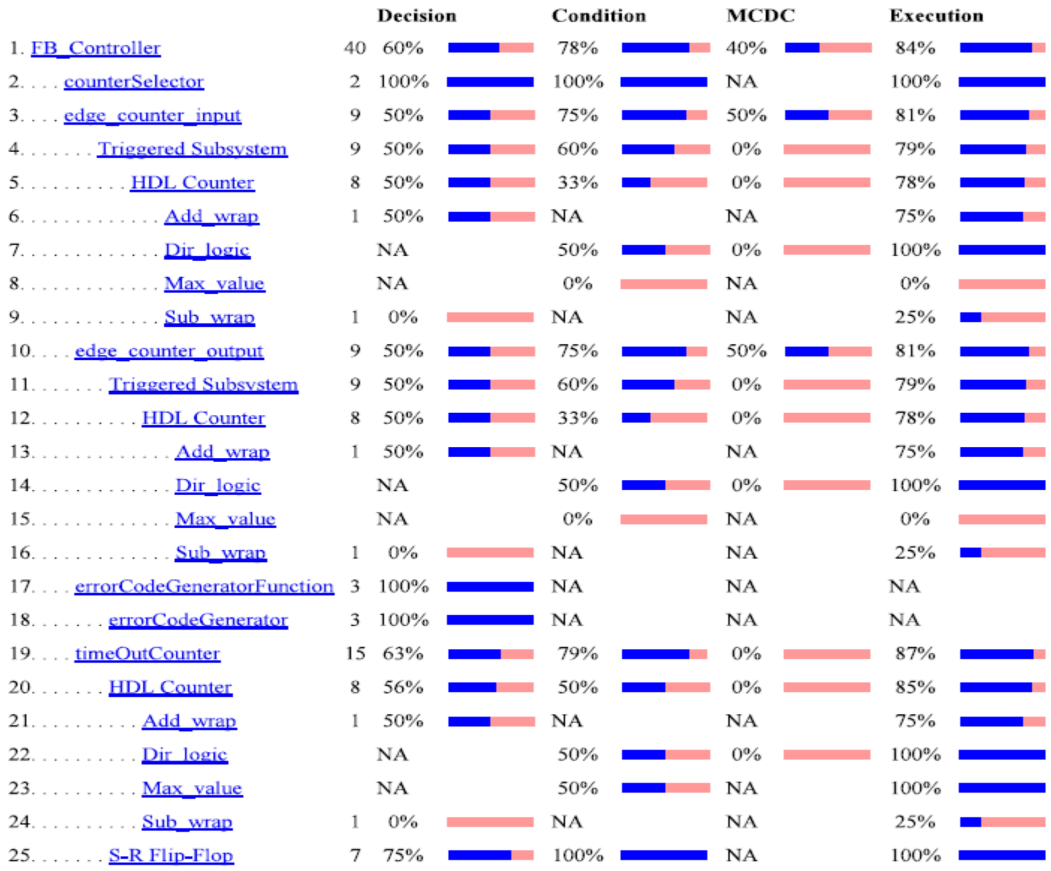


Figure 83: Low coverage indicated in Simulink blocks.



| | | Decision | Condition | MCDC | Execution |
|---|---|---|---|---|---|
| 1. FB_Controller | 40 | 60% | 78% | 40% | 84% |
| 2. . . . counterSelector | 2 | 100% | 100% | NA | 100% |
| 3. . . . edge_counter_input | 9 | 50% | 75% | 50% | 81% |
| 4. . . . . . . Triggered Subsystem | 9 | 50% | 60% | 0% | 79% |
| 5. . . . . . . . . . HDL Counter | 8 | 50% | 33% | 0% | 78% |
| 6. . . . . . . . . . . . . Add_wrap | 1 | 50% | NA | NA | 75% |
| 7. . . . . . . . . . . . . Dir_logic | | NA | 50% | 0% | 100% |
| 8. . . . . . . . . . . . . Max_value | | NA | 0% | NA | 0% |
| 9. . . . . . . . . . . . . Sub_wrap | 1 | 0% | NA | NA | 25% |
| 10. . . . edge_counter_output | 9 | 50% | 75% | 50% | 81% |
| 11. . . . . . . Triggered Subsystem | 9 | 50% | 60% | 0% | 79% |
| 12. . . . . . . . . . HDL Counter | 8 | 50% | 33% | 0% | 78% |
| 13. . . . . . . . . . . . . Add_wrap | 1 | 50% | NA | NA | 75% |
| 14. . . . . . . . . . . . . Dir_logic | | NA | 50% | 0% | 100% |
| 15. . . . . . . . . . . . . Max_value | | NA | 0% | NA | 0% |
| 16. . . . . . . . . . . . . Sub_wrap | 1 | 0% | NA | NA | 25% |
| 17. . . . errorCodeGeneratorFunction | 3 | 100% | NA | NA | NA |
| 18. . . . . . . errorCodeGenerator | 3 | 100% | NA | NA | NA |
| 19. . . . timeOutCounter | 15 | 63% | 79% | 0% | 87% |
| 20. . . . . . . HDL Counter | 8 | 56% | 50% | 0% | 85% |
| 21. . . . . . . . . . Add_wrap | 1 | 50% | NA | NA | 75% |
| 22. . . . . . . . . . Dir_logic | | NA | 50% | 0% | 100% |
| 23. . . . . . . . . . Max_value | | NA | 50% | NA | 100% |
| 24. . . . . . . . . . Sub_wrap | 1 | 0% | NA | NA | 25% |
| 25. . . . . . . S-R Flip-Flop | 7 | 75% | 100% | NA | 100% |

Figure 84: Coverage Analysis of SymPLe component. Low coverage due to dead logic and missing test cases.

While performing testing, there may be requirements that are 'beyond testing', meaning that a requirement cannot be tested due to complexity of the design, numerous interactions with other components etc. It may be important to monitor such requirements at runtime. Additionally, we have to ensure that assumptions made during design verification hold true during runtime.

## A 2. Simulink Design Verifier

Design Verifier is a type of hybrid formal methods tool where model checking and constrained theorem proving are combined. The Design Verifier tool works on the principles of Bounded Model Checking and K-induction technique for solving the state reachability problem and satisfiability (SAT Solver) to check if the mathematical representations of the properties of a model are satisfiable. Counterexamples that encompasses n states are detected by the Bounded Model checker by checking if the property P fails in any of the first n states during the function's execution. Further, the induction rule generalizes the property P for the entire state space by showing that if it holds true for every n states of execution, it should hold true for the successive states that follow (n, n+1, …...Infinity) [159]. This type of formal method can be tailored to verify complex systems. Model checking handles finite state problems well, while K-induction handles more complex infinite space problems.

In Simulink DV, a proof objective is generally specified as illustrated in Figure 85. Suppose we have a function F for which we would like to prove a certain property P, as shown in Figure 85, the output of function F is specified as input to block P. Property P is a predicate, which should always return true when hypotheses H set on the input data of the model are satisfied. Often properties in DV are expressed as implication form, "if there is some X, Y, Z conditions then, property P follows. P is connected to an Assertion block where the results of the verification effort are reported. H is connected to a Proof Assumption block. Proof Assumption blocks constrain the input data of the model so the state space of the search can be managed. A user often starts with constrained forms of the proof (e.g. intermediate proofs) and then progressively relax the constraints as the proof becomes more global [159].

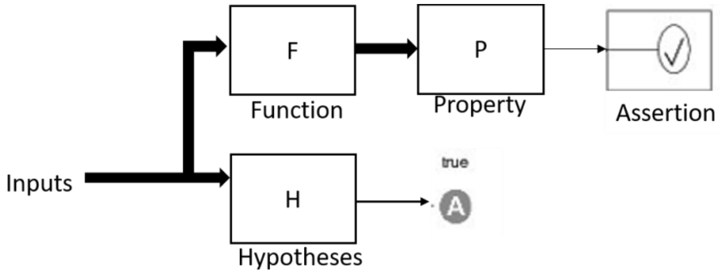


Figure 85: General Proof Outline for Simulink Design Verifier (inspired by [159]).

## A 3. Safety Standard Compliance checks

Static verification is the process of verifying whether the model adheres to modelling guidelines and various industry standards without executing the model. Static verification checks are also performed as a part of the standard compliance checks. Safety standard checks such as IEC 61508, ISO 26262 checks etc. can be performed on the model to ensure that the design meets the stringent requirements for usage in safety critical applications.

We have used Simulink Check tool to verify compliance to standards of the model. We performed compliance checks to the IEC 61508 standard for the SymPLe. IEC 61508 checks can be performed on the final model (after MBT and formal verification) to ensure that the design meets the stringent" requirements for usage in safety critical applications. Design errors and model settings that cause generation of inefficient code or code unsuitable for safety-critical applications can be identified by using Model Advisor tool in Simulink. Additionally, configuration settings that include optimization, code generation and solver settings in the models can be checked using IEC 61508 static verification to ensure that they are safety compliant.

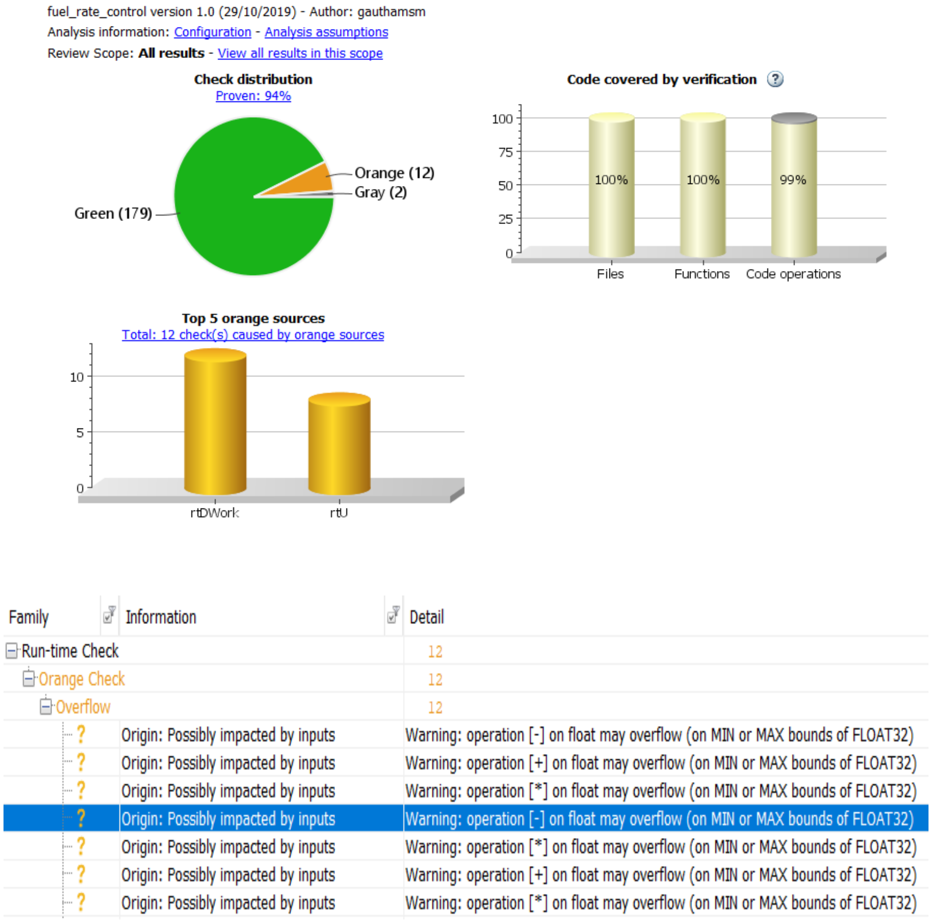
It is also possible to perform model compliance checks with standards such as CERT C, CWE, and ISO/IEC TS 17961 Standards [160]. These checks ensure that the Simulink blocks used in the design can generate code that comply with the secure coding standards. This helps address the security requirements early in the V&V workflow before generation of the code automatically for the design. Some of the characteristics of the design that have to be followed at the model level in order to comply with security standards such as to CERT C, CWE are [160]:

1) Switch case Simulink blocks always have a default case.
2) The output ports are always assigned to a valid value and is not left unassigned
3) Checks for inequality or equality of floating-point numbers are excluded in the design
4) The word length of the integers, match the bit size of its hardware implementation

No Dead logic and operations performed in the design should not result in integer overflows, divide by zero and array out of bound errors. Static analysis of the model (e.g. using Simulink Design Verifier) help detect many of these flaws.

***Static verification of code using Polyspace***

MathWorks Simulink uses Polyspace tool to perform static analysis of C code. Polyspace is a static analysis tool that uses formal methods to analyze C code and identify safety and security vulnerabilities. Polyspace Code prover is used to detect potential runtime errors that could result in unreachable code, unsafe type conversions such as long to short and float to int etc. [161]. Polyspace Bug Finder checks the code for safety standard checks such as MISRA, ISO 26262, IEC 61508, DO-178 and security standard checks such as CWE, CERT-C, ISO/IEC 1796 [162]. Figure 86 shows Polyspace static verification report which indicates warnings on overflow vulnerability.



fuel_rate_control version 1.0 (29/10/2019) - Author: gauthamsm

Analysis information: Configuration - Analysis assumptions

Review Scope: **All results** - View all results in this scope

| Family | Information | Detail |
|---|---|---|
| Run-time Check | | 12 |
| Orange Check | | 12 |
| Overflow | | 12 |
| ? | Origin: Possibly impacted by inputs | Warning: operation [-] on float may overflow (on MIN or MAX bounds of FLOAT32) |
| ? | Origin: Possibly impacted by inputs | Warning: operation [+] on float may overflow (on MIN or MAX bounds of FLOAT32) |
| ? | Origin: Possibly impacted by inputs | Warning: operation [*] on float may overflow (on MIN or MAX bounds of FLOAT32) |
| ? | Origin: Possibly impacted by inputs | Warning: operation [-] on float may overflow (on MIN or MAX bounds of FLOAT32) |
| ? | Origin: Possibly impacted by inputs | Warning: operation [*] on float may overflow (on MIN or MAX bounds of FLOAT32) |
| ? | Origin: Possibly impacted by inputs | Warning: operation [+] on float may overflow (on MIN or MAX bounds of FLOAT32) |
| ? | Origin: Possibly impacted by inputs | Warning: operation [*] on float may overflow (on MIN or MAX bounds of FLOAT32) |

Figure 86: Polyspace static verification report.

## A 4. Model to Code Equivalence and Code Coverage Analysis

Model Based Engineering have executable models where synthesizable code is automatically generated from a model. It is important to ensure that this code exactly depicts the model.

### *A 4.1 Co-simulation*

Co-simulation is a way verifying equivalence between the model and Code. Co-simulation helps to run the model and the generated code simultaneously as shown in

Figure 87. Since our EDGSS application is on a FPGA platform, we use the co-simulation feature of Simulink with Mentor Modelsim tool. Mentor Modelsim is an HDL simulation tool which can be used to run the HDL code generated for the model [163]. The test cases formulated for the model can be reused

and run on the HDL code and HDL code coverage can be analyzed using the Mentor Modelsim tool. Performing Software in Loop (SIL) testing for C code is explained in **Appendix A 8**.

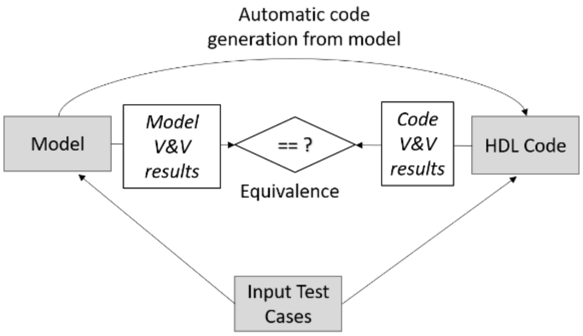


Figure 87: Modelsim Co-simulation Equivalence Testing.

Simulink offers a feature called Equivalence Testing (Figure 88) where the model and the HDL code can be run at the same time, and the scope results show the differential error between the model and the code results.

Figure 89 (a) shows the test cases that were run on the co-simulation model and the results are seen in Simulink Test Manager where we perform equivalence testing.

Figure 89(b) shows the equivalence testing results where the results from the model and code are equivalent.

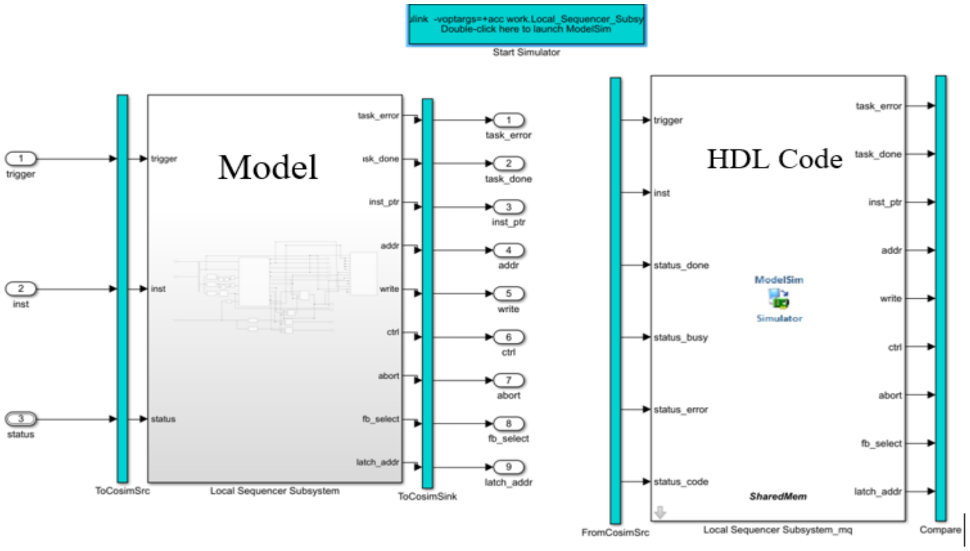
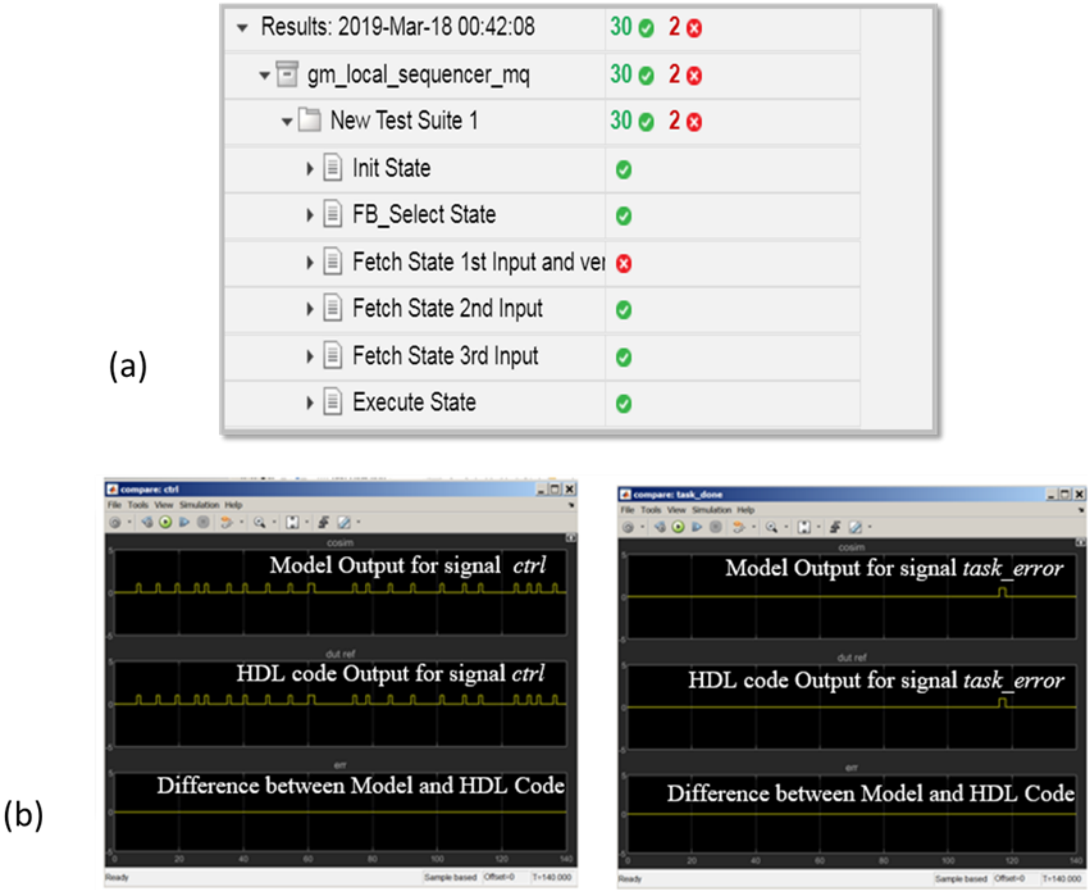


Figure 88: Modelsim Co-Simulation.


(a)
(b)
compare: ctrl
File Tools View Simulation Help
cosim
Model Output for signal *ctrl*
dut ref
HDL code Output for signal *ctrl*
err
Difference between Model and HDL Code
0
20
40
60
80
100
120
140
Ready
Sample based
Offset=0
T=140.000
compare: task_done
File Tools View Simulation Help
cosim
Model Output for signal *task_error*
dut ref
HDL code Output for signal *task_error*
err
Difference between Model and HDL Code
0
20
40
60
80
100
120
140
Ready
Sample based
Offset=0
T=140.000

| | |
|---|---|
| Results: 2019-Mar-18 00:42:08 | 30 ✓ 2 ✗ |
| gm_local_sequencer_mq | 30 ✓ 2 ✗ |
| New Test Suite 1 | 30 ✓ 2 ✗ |
| Init State | ✓ |
| FB_Select State | ✓ |
| Fetch State 1st Input and ver | ✗ |
| Fetch State 2nd Input | ✓ |
| Fetch State 3rd Input | ✓ |
| Execute State | ✓ |

Figure 89: (a) Test Manager Modelsim Results (b) Modelsim Co-Simulation comparing the Co-sim and Model output for ctrl and task_error.

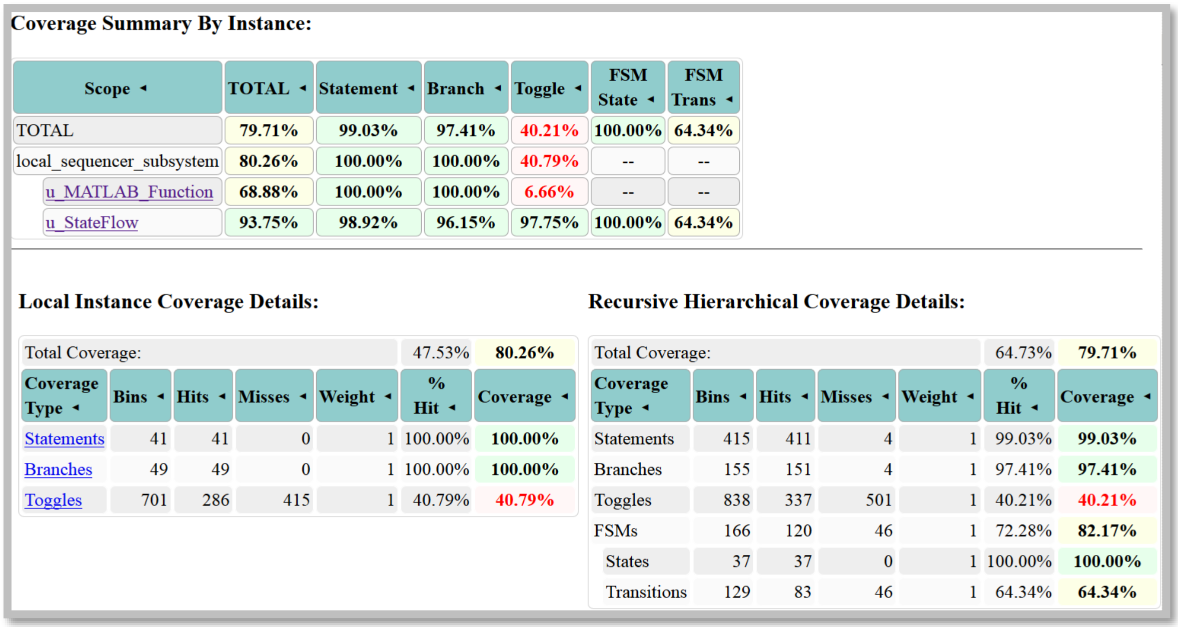
### *A 4.2 HDL Code Coverage*

Code coverage analysis was performed on the HDL Code that was generated by Simulink HDL Coder. HDL Code coverage was achieved using Mentor graphics Modelsim tool. Code coverage involves coverage metrics such as Statements, Branches, Conditions, Expressions, Toggle, FSM States and FSM Transitions that are explained below:

- Code coverage metrics checks the following on the HDL code[163]:
- Statement coverage: ensures that each HDL statement is used in the design and there are no redundant HDL lines
- Branch coverage: ensures that all the decision branches are covered.
- Conditional coverage: ensures that each condition and their opposites are covered

- Expression coverage: ensures that all expressions are coverage and any don't care expressions are excluded from coverage analysis
- Toggle coverage: ensures that bit transition in a register i.e. transition from 1 to 0 and 0 and 1 are covered
- FSM states: ensures that all states in a Finite State Machine (FSM) are reachable
- FSM transitions: ensures that all possible transitions between states in a FSM are covered

Figure 90 shows the summary of the code coverage for the Local Sequencer component in SymPLe when the test cases were run in Simulink.



**Coverage Summary By Instance:**

| Scope | TOTAL | Statement | Branch | Toggle | FSM State | FSM Trans |
|---|---|---|---|---|---|---|
| TOTAL | 79.71% | 99.03% | 97.41% | 40.21% | 100.00% | 64.34% |
| local_sequencer_subsystem | 80.26% | 100.00% | 100.00% | 40.79% | -- | -- |
| u_MATLAB_Function | 68.88% | 100.00% | 100.00% | 6.66% | -- | -- |
| u_StateFlow | 93.75% | 98.92% | 96.15% | 97.75% | 100.00% | 64.34% |

**Local Instance Coverage Details:**

| Total Coverage: | | | | | 47.53% | 80.26% |
|---|---|---|---|---|---|---|
| Coverage Type | Bins | Hits | Misses | Weight | % Hit | Coverage |
| Statements | 41 | 41 | 0 | 1 | 100.00% | 100.00% |
| Branches | 49 | 49 | 0 | 1 | 100.00% | 100.00% |
| Toggles | 701 | 286 | 415 | 1 | 40.79% | 40.79% |

**Recursive Hierarchical Coverage Details:**

| Total Coverage: | | | | | 64.73% | 79.71% |
|---|---|---|---|---|---|---|
| Coverage Type | Bins | Hits | Misses | Weight | % Hit | Coverage |
| Statements | 415 | 411 | 4 | 1 | 99.03% | 99.03% |
| Branches | 155 | 151 | 4 | 1 | 97.41% | 97.41% |
| Toggles | 838 | 337 | 501 | 1 | 40.21% | 40.21% |
| FSMs | 166 | 120 | 46 | 1 | 72.28% | 82.17% |
| States | 37 | 37 | 0 | 1 | 100.00% | 100.00% |
| Transitions | 129 | 83 | 46 | 1 | 64.34% | 64.34% |

Figure 90: Code Coverage Summary.

Although we were able to get almost 100% code coverage at the model level for the Local Sequencer, we were not able to get the same coverage at the code level. Additional signals are added in the code by the Simulink HDL coder that were not in the model. Furthermore, coverage metrics at the code such as toggle coverage are not present at the model level. All this would need more test cases written to get better coverage.

### *A 4.3 Runtime Monitor properties derived from beyond testing conditions at the code level*

The additional signals generated which are not present in the model must be thoroughly analyzed. These signals resulted in low coverage metrics at the code level indicating the need for additional test vectors. Testing the code can reveal corner cases and complex interactions that we may want to monitor at runtime.

## A 5. Formal Verification of Code

*Example of monitor property derived from low level HDL verification of system functionality as shown in point-7 of Section 4 Figure 28.*

Formal verification of the model has significant benefits in detecting design flaws early in the development life cycle. We additionally would like to extend formal verification to the HDL level. Once the HDL (VHDL) code is generated from MathWorks model, we port the code into Mentor Questasim environment. We used Mentor Questasim tool to verify formal assertion properties on the HDL code. We perform Assertion Based Verification to formally verify functional properties on the code.

Assertion Based Verification (ABV) offers a powerful formal verification paradigm for checking properties on the HDL code. Property specification is the key step in ABV and through these specifications we state the requirement in the design that we want to formally verify. The two most widely used ABV languages are, Property Specification Language (PSL) and System Verilog (SVA).

### *A 5.1 Assertion Based Verification (ABV)*

ABV offers a powerful verification paradigm by checking critical conditions which are important for the correct functionality of the system and flags an error if the condition fails to hold true. Assertions are similar to active comments which can either be inserted in the RTL file describing the design to be verified or in a separate bind file used in the verification process. Assertions capture the design behavior at each clock cycle and they can consequently be used to verify intermediate behavior of the signals. Therefore, ABV helps to detect the time and place of occurrence of the design flaws.

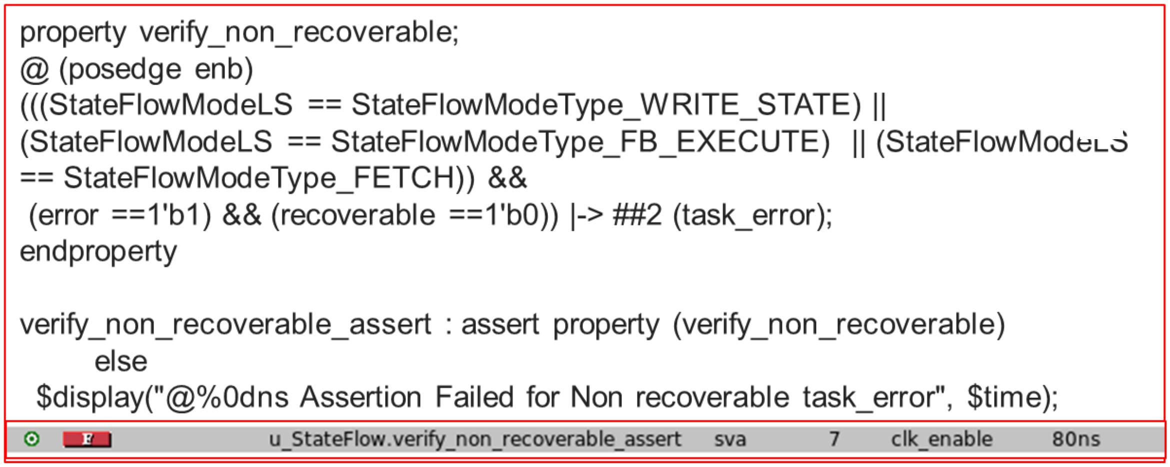
Property specification is the key factor in ABV and states the purpose of the design. Properties comprise of four distinct layers namely: Boolean layer, Temporal Layer, Verification layer and Modelling layer.

Boolean layer describes the design in a Boolean expression. The Temporal layer provides a description of the relationship between the Boolean expression over time while the Verification layer states how the property should be used, for example, if it should be asserted and checked or if it should be assumed as a constraint to the signal [164]. The Modelling layer models the behavior of the inputs in the design and gives behavior to the signals and variables in the design. It also gives names to the properties from the temporal layer [165].

Property Specification Language (PSL) or System Verilog are the two predominantly used assertion languages. System Verilog inherits the expression language, syntax and semantics of Verilog and can be directly written as a part of the Verilog design code. PSL is a separate language and is designed to work with different layers of a HDL language. Therefore, it cannot be directly written as a part of the HDL code. Bind files are used to attach the PSL properties to a HDL code or PSL statements can be included in the HDL model via comments. Although the two languages are very similar, on a technical level there are fundamental differences. The choice between the two languages is made depending on the verification requirements of the design and methodology used [166].

### *A 5.2 SymPLe example of formal verification of HDL code*

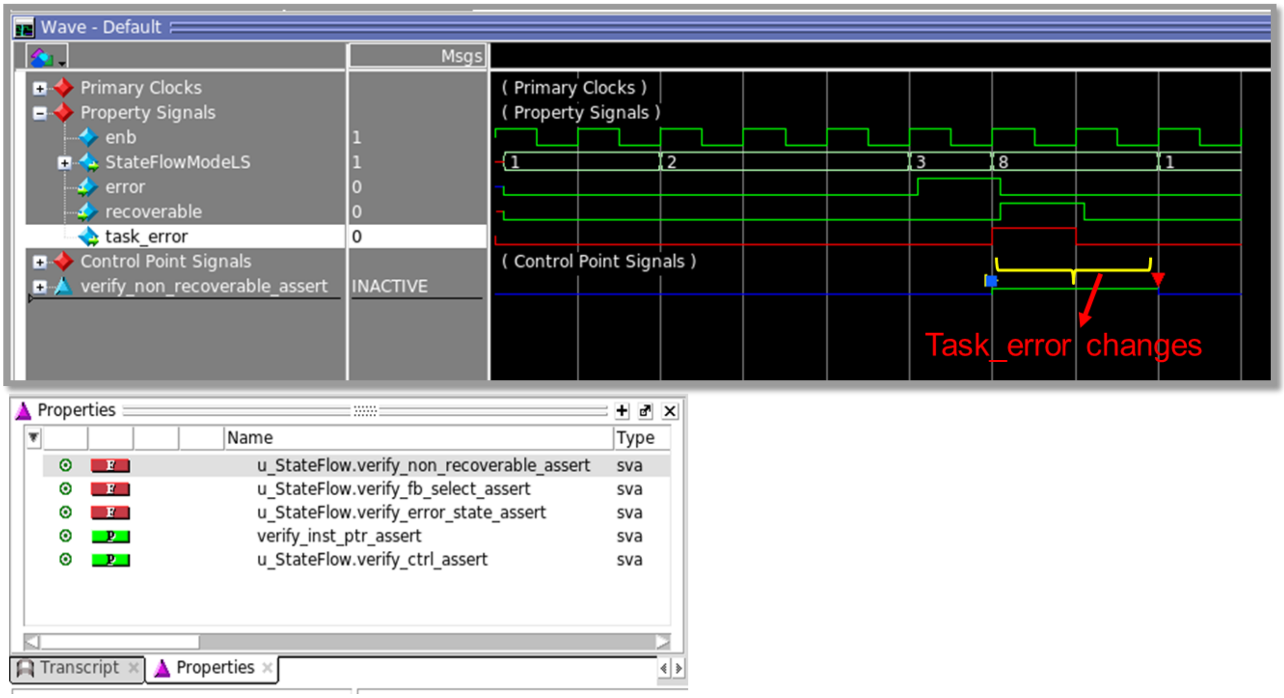
The error handling property described in Section 4.5 "c. *Formal Verification of the SymPLe architecture for a lower-level requirement*" was verified at the code level using ABV (Figure 91). The property failed due to the design error in error handling in the SymPLe Local sequencer component. Mentor Questa tools generates the counterexample showing the inputs for which the property failed in Figure 92.



```
property verify_non_recoverable;
@ (posedge enb)
(((StateFlowModeLS == StateFlowModeType_WRITE_STATE) ||
(StateFlowModeLS == StateFlowModeType_FB_EXECUTE)  || (StateFlowModeLS
== StateFlowModeType_FETCH)) &&
 (error ==1'b1) && (recoverable ==1'b0)) |-> ##2 (task_error);
endproperty

verify_non_recoverable_assert : assert property (verify_non_recoverable)
      else
  $display("@%0dns Assertion Failed for Non recoverable task_error", $time);
```

```
⊙  F          u_StateFlow.verify_non_recoverable_assert    sva        7      clk_enable      80ns
```

Figure 91: Assertion Based Verification of HDL code to verify error handling.



| Name | Msgs |
|---|---|
| Primary Clocks | |
| Property Signals | |
| enb | 1 |
| StateFlowModeLS | 1 |
| error | 0 |
| recoverable | 0 |
| task_error | 0 |
| Control Point Signals | |
| verify_non_recoverable_assert | INACTIVE |

Properties

| | | Name | Type |
|---|---|---|---|
| ⊙ | F | u_StateFlow.verify_non_recoverable_assert | sva |
| ⊙ | F | u_StateFlow.verify_fb_select_assert | sva |
| ⊙ | F | u_StateFlow.verify_error_state_assert | sva |
| ⊙ | P | verify_inst_ptr_assert | sva |
| ⊙ | P | u_StateFlow.verify_ctrl_assert | sva |

Figure 92: Mentor Questa Counter example when property fails.

### *A 5.3 Example of Monitor properties derived from low level HDL verification of system functionality*

In SymPLe architecture the co-ordination between the data and control signals are critical for correct functionality of the system. For example, synchronization of two signals DONE and CTRL was important to ensure that the data bits and the control bits in the model are synchronized in SymPLe Functional Blocks. The timing behavior of the model was formally verified with respect to these two signals. In the SymPLe Functional Blocks, the DONE signal should go high seven clock cycles after the CTRL signal going high. We verify this property formally on the HDL code corresponding to an AND functional block in SymPLe using the Questa Tools.

Figure 93 shows verification of a “logical AND” function block in SymPLe at the model level. Here, the “DONE” signal (which indicates that the results of the AND functional block is ready) should go high 7 clock cycles after the trigger to the functional block (CTRL==1) going high. Simulink DV analyses the property for the AND function block and gives a result indicating if the property was True, False or undecided. The same property was checked on the HDL code using ABV. Figure 94 shows the same property written in System Verilog. Failure of such timing properties may put the system in indeterministic state. Therefore, it is important to monitor such conditions at runtime.

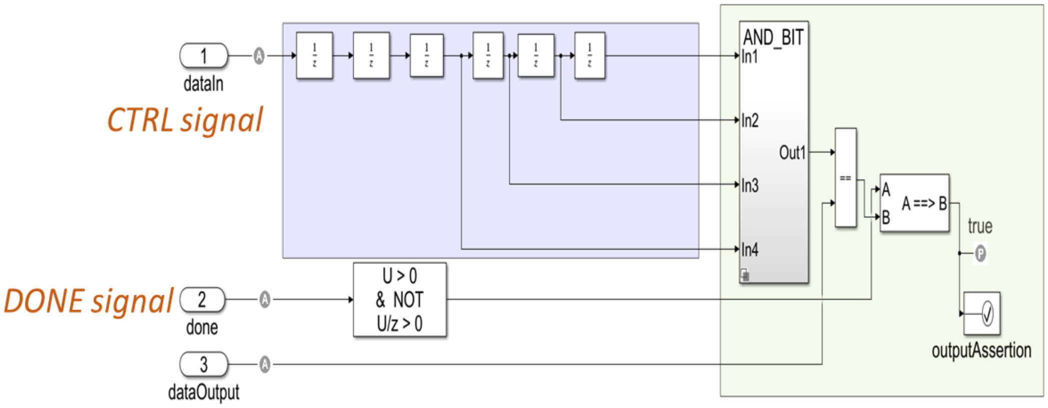


Figure 93: Simulink DV Property Proving for a SymPLe AND functional block.

```
     property verify_done;
   @(posedge clk) disable iff(!CTRL)
   CTRL |-> ##7 (DONE==1'b1);
endproperty
   verify_done_assert : assert property (verify_done);
else
   $display("@%0dns Assertion failed for DONE", $time);
```

Figure 94: Assertion Based Formal Verification performed Using Mentor Questa tool.

## A 6. FPGA in Loop Implementation of SymPLe

FIL is used to evaluate the correctness of the code and hardware implementation of the system design. SymPLe was implemented on Xilinx Kintex 7 FPGA. All the test cases run on the model, were run on the SymPLe hardware implementation and we checked for equivalence between them. FIL testing was performed at both unit level and the application level in SymPLe. Any difference between the model and FIL outputs could indicate a potential issue in the hardware implementation. Figure 95 shows the FIL implementation of the EDGSS application. Here equivalence testing was performed to ensure the outputs of the model and the FPGA match. Performing Processor in Loop implementation is explained in **Appendix A 9.**

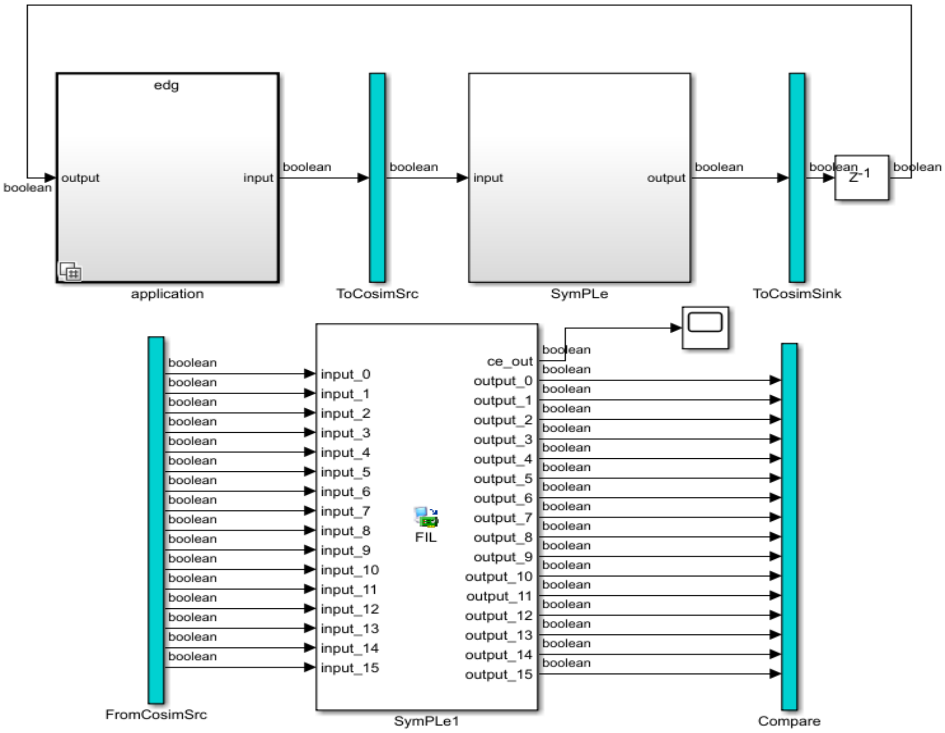


Figure 95: FPGA in Loop implementation of EDGSS application.

### *A 6.1 Example of Monitor properties refined after consideration of environmental factors and variations*

When we implement a design on a hardware platform, there can be a number of factors that we should consider that could affect the functionality of the system. For example, clock glitches, signal variations have to be considered while formulating runtime properties. We refine the runtime monitor properties to take these factors into consideration. For example, we consider properties that ensure that the system is not stuck in a “Read Input” state if there is no input from a sensor. We ensure that the “TimeOut” feature incorporated in our design is able to detect this issue in such scenarios.

## A 7. Hardware Fault Injection

Hardware Fault Injection is important to ensure that the runtime monitors are able to detect the faults effectively. HDL code can be generated for the fault saboteurs that were used to inject fault at the model [91]. This fault injection code can be implemented along with the design code on the FPGA and faults can be injected on the hardware implementation of the design. Fault injection using fault saboteurs is explained in Chapter 5.

### *A 7.1 Example of Monitor properties refined after fault injection on hardware to detect safety violations*

Sometimes the runtime monitor properties are refined if they are not able to detect faults occurring at the hardware level. In our research we did not perform hardware fault injection, but this step would further refine some of the runtime monitor properties.

## A 8. Software in Loop (SIL)

SIL is performed by executing the code on the host machine and ensuring that the results match the simulation results of the model in Simulink[167]. SIL is non-real time execution of the code and is synchronized with the simulation results. It helps access functional equivalence between the model and the automatically generated code and analyze code coverage. The test cases formulated and run on the model can be run on the code and the results can be compared in order to confirm equivalence. Code coverage can be analyzed by tools such as Bullseye or LDRA through the Simulink environment. Figure 96 shows the SIL model that runs the code simulator on the PC.

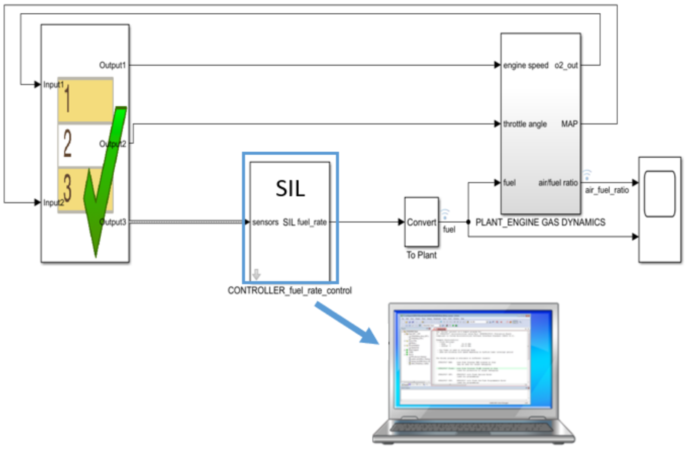


Figure 96: Software in Loop Simulation.

# A 9. Processor in Loop (PIL)

Object code is created for the source code and the executable is run on actual hardware while performing PIL. Like SIL, PIL is also a non-real time execution which is synchronized with simulation. Test vectors run on the model can be run on the hardware platform and the results can be compared and checked for equivalence. Figure 97 shows the PIL model that runs the code hardware in non-real time mode.

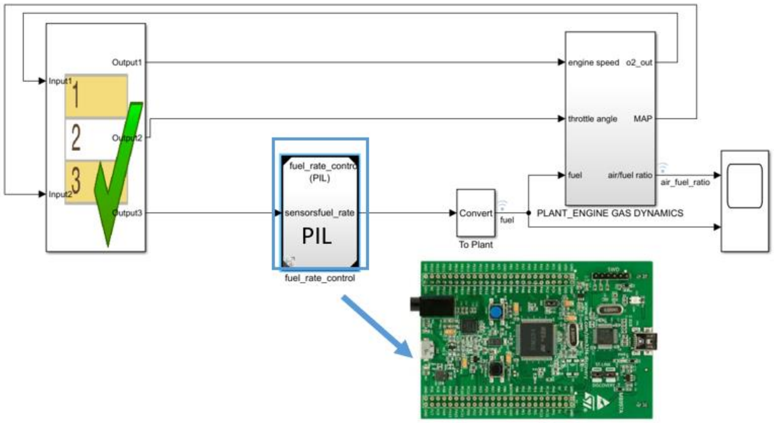


Figure 97: Processor in Loop Implementation.

### A 9.1 PIL External Mode

Real time execution of the code can be performed by running the object code on hardware with Simulink in External mode [168]. The hardware exchanges data with Simulink via a shared memory interface.

# References

[1] "Unlocking_the_potential_of_the_Internet_of_Things_Executive_summary.pdf." Accessed: Nov. 19, 2020. [Online]. Available: https://www.mckinsey.com/~/media/McKinsey/Industries/Technology.

[2] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: a cyber-physical systems approach*, Second edition. Cambridge, Massachuetts: MIT Press, 2017.

[3] matthew.lynberg.ctr@dot.gov, "Automated Vehicles for Safety," *NHTSA*, Sep. 07, 2017. https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety (accessed Oct. 30, 2019).

[4] L. S. Wheatcraft, "Thinking Ahead to Verification and Validation," p. 25, 2012.

[5] V. L. Foreman, F. M. Favaró, J. H. Saleh, and C. W. Johnson, "Software in military aviation and drone mishaps: Analysis and recommendations for the investigation process," *Reliability Engineering & System Safety*, vol. 137, pp. 101–111, May 2015, doi: 10.1016/j.ress.2015.01.006.

[6] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, "Byzantine Fault Tolerance, from Theory to Reality," in *Computer Safety, Reliability, and Security*, Berlin, Heidelberg, 2003, pp. 235–248, doi: 10.1007/978-3-540-39878-3_19.

[7] "A performance assessment of a byzantine resilient fault-tolerant computer | Computers in Aerospace Conference." https://arc.aiaa.org/doi/abs/10.2514/6.1989-3064 (accessed Dec. 02, 2019).

[8] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016, doi: 10.1109/TSE.2016.2521368.

[9] "Assumptions Used in the Safety Assessment Process and the Effects of Multiple Alerts and Indications on Pilot Performance," p. 13.

[10] C. R. Elks, *A theory of run-time verification for safety critical reactive systems*. University of Virginia, 2005.

[11] A. Goodloe, "Challenges in High-Assurance Runtime Verification," in *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, vol. 9952, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2016, pp. 446–460.

[12] M. Wu, H. Zeng, C. Wang, and H. Yu, "Safety Guard: Runtime Enforcement for Safety-Critical Cyber-Physical Systems: Invited," in *Proceedings of the 54th Annual Design Automation Conference 2017 on - DAC '17*, Austin, TX, USA, 2017, pp. 1–6, doi: 10.1145/3061639.3072957.

[13] J. Schumann, P. Moosbrugger, and K. Y. Rozier, "R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems?," p. 15.

[14] P. Daian, S. Shiraishi, A. Iwai, B. Manja, and G. Rosu, "RV-ECU: Maximum Assurance In-Vehicle Safety Monitoring," Apr. 2016, pp. 2016-01–0126, doi: 10.4271/2016-01-0126.

[15] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[16] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to Runtime Verification," in *Lectures on Runtime Verification*, vol. 10457, E. Bartocci and Y. Falcone, Eds. Cham: Springer International Publishing, 2018, pp. 1–33.

[17] A. E. Goodloe and L. Pike, "Monitoring Distributed Real-Time Systems: A Survey and Future Directions," *(NASA/CR-2010-216724)*, p. 49, Jul. 2010.

[18] W. Ahrendt *et al.*, "COST Action IC 1402 ArVI: Runtime Verification Beyond Monitoring -- Activity Report of Working Group 1," *arXiv:1902.03776 [cs]*, Feb. 2019, Accessed: Dec. 03, 2019. [Online]. Available: http://arxiv.org/abs/1902.03776.

[19] A. Desai, "A Tutorial on Runtime Verification and Assurance," p. 33.

[20] J. Ayerdi *et al.*, "Towards a Taxonomy for Eliciting Design-Operation Continuum Requirements of Cyber-Physical Systems," in *2020 IEEE 28th International Requirements Engineering Conference (RE)*, Zurich, Switzerland, Aug. 2020, pp. 280–290, doi: 10.1109/RE48521.2020.00038.

[21] B. Combemale and M. Wimmer, “Towards a Model-Based DevOps for Cyber-Physical Systems,” in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, Cham, 2020, pp. 84–94, doi: 10.1007/978-3-030-39306-9_6.

[22] A. Gusmão, M. Silva, T. Poleto, L. Silva, and A. Costa, “Cybersecurity Risk Analysis Model Using Fault Tree Analysis and Fuzzy Decision Theory,” *International Journal of Information Management*, vol. 43, p. 248, Aug. 2018, doi: 10.1016/j.ijinfomgt.2018.08.008.

[23] A. Humayed, J. Lin, F. Li, and B. Luo, “Cyber-Physical Systems Security -- A Survey,” *arXiv:1701.04525 [cs]*, Jan. 2017, Accessed: Jun. 06, 2019. [Online]. Available: http://arxiv.org/abs/1701.04525.

[24] A. W. Werth, “TOWARDS DISTINGUISHING BETWEEN CYBERATTACKS AND FAULTS IN CYBER-PHYSICAL SYSTEMS,” 2014.

[25] M. Gibson, “Achieving Verifiable and High Integrity Instrumentation and Control Systems through Complexity Awareness and Constrained Design,” 15–8044, 1547345, Jul. 2019. doi: 10.2172/1547345.

[26] G. Tamura *et al.*, “Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems,” in *Software Engineering for Self-Adaptive Systems II*, vol. 7475, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 108–132.

[27] C. Sánchez *et al.*, “A survey of challenges for runtime verification from advanced application domains (beyond software),” *Form Methods Syst Des*, vol. 54, no. 3, pp. 279–335, Nov. 2019, doi: 10.1007/s10703-019-00337-w.

[28] A. P. Fournaris, A. Komninos, A. S. Lalos, A. P. Kalogeras, C. Koulamas, and D. Serpanos, “Design and Run-Time Aspects of Secure Cyber-Physical Systems,” in *Security and Quality in Cyber-Physical Systems Engineering*, S. Biffl, M. Eckhart, A. Lüder, and E. Weippl, Eds. Cham: Springer International Publishing, 2019, pp. 357–382.

[29] Y. Zhao and F. Rammig, “Model-based Runtime Verification Framework,” *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 1, pp. 179–193, Oct. 2009, doi: 10.1016/j.entcs.2009.09.035.

[30] S. P. Miller, “Bridging the Gap Between Model-Based Development and Model Checking,” in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 5505, S. Kowalewski and A. Philippou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 443–453.

[31] K. Goseva-Popstojanova, “Report: Survey on Model-Based Software Engineering and Auto-Generated Code,” p. 56, 2016.

[32] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, “Model-based Engineering in the Embedded Systems Domain: An Industrial Survey on the State-of-practice,” *Softw. Syst. Model.*, vol. 17, no. 1, pp. 91–113, Feb. 2018, doi: 10.1007/s10270-016-0523-3.

[33] A. Murugesan *et al.*, “From requirements to code: model based development of a medical cyber physical system,” 2014, Accessed: Mar. 31, 2017. [Online]. Available: https://works.bepress.com/sokolsky/114/.

[34] A. Joshi, S. P. Miller, M. Whalen, and M. P. E. Heimdahl, “A proposal for model-based safety analysis,” in *24th Digital Avionics Systems Conference*, Oct. 2005, vol. 2, p. 13 pp. Vol. 2-, doi: 10.1109/DASC.2005.1563469.

[35] M. W. Whalen, A. Murugesan, and M. P. E. Heimdahl, “Your what is my how: Why requirements and architectural design should be iterative,” in *2012 First IEEE International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks)*, Sep. 2012, pp. 36–40, doi: 10.1109/TwinPeaks.2012.6344559.

[36] “Flight-Critical Systems Design Assurance,” U.S. Department of Transportation Federal Aviation Administration, Jul. 2012. Accessed: Nov. 19, 2019. [Online]. Available: http://www.tc.faa.gov/its/worldpac/techrpt/ar11-28.pdf.

[37] G. S. Tallant, J. M. Buffington, W. A. Storm, P. O. Stanfill, and B. H. Krogh, “Validation & Verification for Emerging Avionic Systems,” p. 4.

[38] C. Liu, “Design time and runtime collaborative defense to enhance embedded system security,” Thesis, University of Delaware, 2016.

[39] M. Mauritz, A. Rausch, and I. Schaefer, “Dependable ADAS by Combining Design Time Testing and Runtime Monitoring,” p. 9.

[40] M. Wolf and D. Serpanos, “Safety and Security in Cyber-Physical Systems and Internet-of-Things Systems,” *Proceedings of the IEEE*, vol. 106, no. 1, pp. 9–20, Jan. 2018, doi: 10.1109/JPROC.2017.2781198.

[41] C. Colombo, “Combining testing and runtime verification,” presented at the Computer Science Annual Workshop CSAW’12, Msida. 19-20, Nov. 2012, Accessed: Sep. 25, 2020. [Online]. Available: https://www.um.edu.mt/library/oar/handle/123456789/23043.

[42] J. M. Chimento, W. Ahrendt, and G. Schneider, “Testing meets static and runtime verification,” in *Proceedings of the 6th Conference on Formal Methods in Software Engineering - FormaliSE ’18*, Gothenburg, Sweden, 2018, pp. 30–39, doi: 10.1145/3193992.3194000.

[43] N. Leveson, “STPA Handbook,” p. 188.

[44] M. V. Stringfellow, N. G. Leveson, and B. D. Owens, “Safety-Driven Design for Software-Intensive Aerospace and Automotive Systems,” *Proceedings of the IEEE*, vol. 98, no. 4, pp. 515–525, Apr. 2010, doi: 10.1109/JPROC.2009.2039551.

[45] B. Ahmed, “Synthesis of a Context-Aware Safety Monitor for an Artificial Pancreas System,” p. 72.

[46] M. S. Yasar, D. Evans, and H. Alemzadeh, “Context-aware Monitoring in Robotic Surgery,” in *2019 International Symposium on Medical Robotics (ISMR)*, Atlanta, GA, USA, Apr. 2019, pp. 1–7, doi: 10.1109/ISMR.2019.8710192.

[47] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, “Introduction to Runtime Verification,” in *Lectures on Runtime Verification*, vol. 10457, E. Bartocci and Y. Falcone, Eds. Cham: Springer International Publishing, 2018, pp. 1–33.

[48] M. Leucker and C. Schallhart, “A brief account of runtime verification,” *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[49] Y. Falcone, S. Krstić, G. Reger, and D. Traytel, “A Taxonomy for Classifying Runtime Verification Tools,” in *Runtime Verification*, vol. 11237, C. Colombo and M. Leucker, Eds. Cham: Springer International Publishing, 2018, pp. 241–262.

[50] A. Kane, “Runtime Monitoring for Safety-Critical Embedded Systems,” Carnegie Mellon University, 2015.

[51] A. Kane, O. Chowdhury, A. Datta, and P. Koopman, “A Case Study on Runtime Monitoring of an Autonomous Research Vehicle (ARV) System,” in *Runtime Verification*, vol. 9333, E. Bartocci and R. Majumdar, Eds. Cham: Springer International Publishing, 2015, pp. 102–117.

[52] K. Watanabe, E. Kang, C.-W. Lin, and S. Shiraishi, “INVITED: Runtime Monitoring for Safety of Intelligent Vehicles,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, San Francisco, CA, Jun. 2018, pp. 1–6, doi: 10.1109/DAC.2018.8465912.

[53] M. Wu, H. Zeng, C. Wang, and H. Yu, “Safety Guard: Runtime Enforcement for Safety-Critical Cyber-Physical Systems: Invited,” in *Proceedings of the 54th Annual Design Automation Conference 2017 on - DAC ’17*, Austin, TX, USA, 2017, pp. 1–6, doi: 10.1145/3061639.3072957.

[54] L. Convent, S. Hungerecker, T. Scheffel, M. Schmitz, D. Thoma, and A. Weiss, “Hardware-Based Runtime Verification with Embedded Tracing Units and Stream Processing,” in *Runtime Verification*, vol. 11237, C. Colombo and M. Leucker, Eds. Cham: Springer International Publishing, 2018, pp. 43–63.

[55] C. Watterson and D. Heffernan, “Runtime verification and monitoring of embedded systems,” *IET Software*, vol. 1, no. 5, pp. 172–179, Oct. 2007, doi: 10.1049/iet-sen:20060076.

[56] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, “Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 3, pp. 1–25, Apr. 2017, doi: 10.1145/3035965.

[57] M. A. Wahab, “Hardware support for the security analysis of embedded software : applications on information flow control and malware analysis,” Theses, CentraleSupélec, 2018.

[58] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, “Integration of ROP/JOP Monitoring IPs in an ARM-based SoC,” p. 6, 2016.

[59] M. Peña-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas, Y. Morilla, and P. Martín-Holgado, “Online Error Detection Through Trace Infrastructure in ARM Microprocessors,” *IEEE Transactions on Nuclear Science*, vol. 66, no. 7, pp. 1457–1464, Jul. 2019, doi: 10.1109/TNS.2019.2921767.

[60] T. Lu, J. Lin, L. Zhao, Y. Li, and Y. Peng, “A Security Architecture in Cyber-Physical Systems: Security Theories, Analysis, Simulation and Application Fields,” *IJSIA*, vol. 9, no. 7, pp. 1–16, Jul. 2015, doi: 10.14257/ijsia.2015.9.7.01.

[61] P. Daian, S. Shiraishi, A. Iwai, B. Manja, and G. Rosu, “RV-ECU: Maximum Assurance In-Vehicle Safety Monitoring,” Apr. 2016, pp. 2016-01–0126, doi: 10.4271/2016-01-0126.

[62] M. Leccadito, “A Hierarchical Architectural Framework for Securing Unmanned Aerial Systems,” *Theses and Dissertations*, Jan. 2017, [Online]. Available: https://scholarscompass.vcu.edu/etd/5037.

[63] B. W. Johnson, *Design and analysis of fault tolerant digital systems*. USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[64] V. Gunes, S. Peter, T. Givargis, and F. Vahid, “A Survey on Concepts, Applications, and Challenges in Cyber-Physical Systems,” *KSII Transactions on Internet and Information Systems*, vol. 8, pp. 4242–4268, Dec. 2014, doi: 10.3837/tiis.2014.12.001.

[65] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan. 2004, doi: 10.1109/TDSC.2004.2.

[66] “Software Security: The Trinity of Trouble.” https://freedom-to-tinker.com/2006/02/15/software-security-trinity-trouble/ (accessed Sep. 27, 2019).

[67] B. W. Johnson, “An Introduction to the Design and Analysis of Fault-Tolerant Systems,” p. 108.

[68] J. J. P. C. Rodrigues and A. Gawanmeh, *Cyber-Physical Systems for Next-Generation Networks*. IGI Global, 1AD.

[69] A. Avizienis, J.- Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan. 2004, doi: 10.1109/TDSC.2004.2.

[70] “Taxonomy of the attacks.” https://journals.plos.org/plosone/article/file?type=supplementary&id=info:doi/10.1371/journal.pone.0188759.s002 (accessed Nov. 16, 2020).

[71] S. Hansman and R. Hunt, “A taxonomy of network and computer attacks,” *Computers & Security*, vol. 24, no. 1, pp. 31–43, Feb. 2005, doi: 10.1016/j.cose.2004.06.011.

[72] M. Yampolskiy, P. Horvath, X. Koutsoukos, Y. Xue, and J. Sztipanovits, *Taxonomy for description of cross-domain attacks on CPS*. 2013, p. 142.

[73] “Common Attack Pattern Enumeration and Classification (CAPEC).” https://samate.nist.gov/BF/Enlightenment/CAPEC.html (accessed Nov. 23, 2020).

[74] G. Bakirtzis, B. T. Carter, C. R. Elks, and C. H. Fleming, “A model-based approach to security analysis for cyber-physical systems,” in *2018 Annual IEEE International Systems Conference (SysCon)*, Apr. 2018, pp. 1–8, doi: 10.1109/SYSCON.2018.8369518.

[75] *Design Methods for Reactive Systems*. Elsevier, 2003.

[76] K. T. Cheng and A. S. Krishnakumar, “Automatic functional test generation using the extended finite state machine model,” in *Proceedings of the 30th international on Design automation conference - DAC ’93*, Dallas, Texas, United States, 1993, pp. 86–91, doi: 10.1145/157485.164585.

[77] C. Sánchez *et al.*, “A Survey of Challenges for Runtime Verification from Advanced Application Domains (Beyond Software),” *arXiv:1811.06740 [cs]*, Nov. 2018, Accessed: Jun. 06, 2019. [Online]. Available: http://arxiv.org/abs/1811.06740.

[78] B. Alpern and F. B. Schneider, “Recognizing safety and liveness,” *Distrib Comput*, vol. 2, no. 3, pp. 117–126, Sep. 1987, doi: 10.1007/BF01782772.

[79] E. Bartocci *et al.*, “First international Competition on Runtime Verification: rules, benchmarks, tools, and final results of CRV 2014,” *Int J Softw Tools Technol Transfer*, vol. 21, no. 1, pp. 31–70, Feb. 2019, doi: 10.1007/s10009-017-0454-5.

[80] M. Shanahan, “The Event Calculus Explained,” in *Artificial Intelligence Today*, vol. 1600, M. J. Wooldridge and M. Veloso, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 409–430.

[81] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma, “TeSSLa: Temporal Stream-based Specification Language,” *arXiv:1808.10717 [cs]*, Aug. 2018, Accessed: Nov. 08, 2019. [Online]. Available: http://arxiv.org/abs/1808.10717.

[82] R. Chevalier, M. Villatel, D. Plaquin, and G. Hiet, “Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode,” *Proceedings of the 33rd Annual Computer Security Applications Conference*, pp. 399–411, Dec. 2017, doi: 10.1145/3134600.3134622.

[83] A. Francalanza, J. A. Pérez, and C. Sánchez, “Runtime Verification for Decentralised and Distributed Systems,” in *Lectures on Runtime Verification*, vol. 10457, E. Bartocci and Y. Falcone, Eds. Cham: Springer International Publishing, 2018, pp. 176–210.

[84] L. Pike, A. Goodloe, R. Morisset, and S. Niller, “Copilot: A Hard Real-Time Runtime Monitor,” in *Runtime Verification*, vol. 6418, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 345–359.

[85] S. Pirmin, “Runtime Verification with TeSSLa on Enzian,” ETH Zurich, Switzerland.

[86] “TeSSLa: Temporal Stream-based Specification Language.” https://www.tessla.io/ (accessed Sep. 20, 2020).

[87] N. G. Leveson, “The Role of Software in Spacecraft Accidents,” p. 27.

[88] L. Zhu, L. Bass, and G. Champlin-Scharff, “DevOps and Its Practices,” *IEEE Softw.*, vol. 33, no. 3, pp. 32–34, May 2016, doi: 10.1109/MS.2016.81.

[89] “Preliminary Hazard Analysis,” in *Basic Guide to System Safety*, John Wiley & Sons, Ltd, 2014, pp. 71–90.

[90] N. Leveson, “A new accident model for engineering safer systems,” *Safety Science*, vol. 42, no. 4, pp. 237–270, Apr. 2004, doi: 10.1016/S0925-7535(03)00047-X.

[91] A. V. Jayakumar, “Systematic Model-based Design Assurance and Property-based Fault Injection for Safety Critical Digital Systems,” *Theses and Dissertations*, Jan. 2020, [Online]. Available: https://scholarscompass.vcu.edu/etd/6239.

[92] “Model-Based Systems Engineering: An Emerging Approach for Modern Systems - IEEE Journals & Magazine.” https://ieeexplore.ieee.org/abstract/document/5722047 (accessed Jun. 24, 2019).

[93] P. Braun *et al.*, “Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impact of a Technology Platform in the Area,” European Commission, Brussels, Belgium, Nov. 2005. Accessed: Jan. 22, 2020. [Online]. Available: http://www.pst.ifi.lmu.de/People/former-members/koch/publications/2005/helmerich-et-al-embeddedsystems-study-181105_en.pdf.

[94] L. Hart, “Introduction To Model-Based System Engineering (MBSE) and SysML,” p. 43.

[95] “Model-Based Design - Examining the technical and social aspects of Model-Based Design,” *Model-Based Design*. https://mburkeonmbd.com/ (accessed Nov. 19, 2020).

[96] A. Bergmann, “Benefits and Drawbacks of Model-based Design,” *KMUTNB: IJAST*, vol. 7, no. 3, pp. 15–19, Sep. 2014, doi: 10.14416/j.ijast.2014.04.004.

[97] EPRI, “Emergency Diesel Generator Digital Control System Upgrade Requirements.” http://www.epri.com/abstracts/Pages/ProductAbstract.aspx?ProductId=000000003002002098 (accessed Dec. 04, 2016).

[98] “Emergency Diesel Generator Digital Control System Upgrade Requirements.” https://www.epri.com/research/products/000000003002002098 (accessed Jun. 19, 2020).

[99] “IEC Certification Kit (for ISO 26262 and IEC 61508).” https://www.mathworks.com/products/iec-61508.html (accessed Nov. 03, 2020).

[100] “Questa® Property Checking.” https://www.mentor.com/products/fv/questa-property-checking (accessed Sep. 27, 2019).

[101] K. Ceesay-Seitz, “Automated verification of a System-on-Chip for radiation protection fulfilling Safety Integrity Level 2,” CERN, European Organization for Nuclear Research, 2019.

[102] H. G. Gurbuz and B. Tekinerdogan, “Model-based testing for software safety: a systematic mapping study,” *Software Qual J*, vol. 26, no. 4, pp. 1327–1372, Dec. 2018, doi: 10.1007/s11219-017-9386-2.

[103] “Simulink Coverage.” https://www.mathworks.com/products/simulink-coverage.html (accessed Sep. 27, 2019).

[104] “Digilent ZYBO,” *Xilinx*. https://www.xilinx.com/support/university/boards-portfolio/xup-boards/DigilentZYBO.html (accessed Oct. 26, 2020).

[105] “FPGA-in-the-Loop Simulation - MATLAB & Simulink.” https://www.mathworks.com/help/hdlverifier/ug/fpga-in-the-loop-fil-simulation.html (accessed Jun. 21, 2020).

[106] “V-Model - an overview | ScienceDirect Topics.” https://www.sciencedirect.com/topics/engineering/v-model (accessed Nov. 06, 2019).

[107] “Simulink® Design Verifier Reference,” MathWorks Simulink. Accessed: Nov. 04, 2020. [Online]. Available: https://www.mathworks.com/help/releases/R2018b/pdf_doc/sldv/sldv_ref.pdf.

[108] “Simulink — Blocks.” https://www.mathworks.com/help/simulink/referencelist.html?type=block&s_tid=CRUX_topnav (accessed Nov. 04, 2020).

[109] “Effects of Communication Delays on an ABS Control System - MATLAB & Simulink.” https://www.mathworks.com/help/simevents/examples/effects-of-communication-delays-on-an-abs-control-system.html (accessed May 18, 2020).

[110] S. Potluri, C. Diedrich, S. R. Roy Nanduru, and K. Vasamshetty, “Development of Injection Attacks Toolbox in MATLAB/Simulink for Attacks Simulation in Industrial Control System Applications,” in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, Jul. 2019, vol. 1, pp. 1192–1198, doi: 10.1109/INDIN41052.2019.8972171.

[111] S.-F. Lokman, A. T. Othman, and M.-H. Abu-Bakar, “Intrusion detection system for automotive Controller Area Network (CAN) bus system: a review,” *J Wireless Com Network*, vol. 2019, no. 1, p. 184, Dec. 2019, doi: 10.1186/s13638-019-1484-3.

[112] L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem,” *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, p. 20.

[113] A. Arora and S. S. Kulkarni, “Detectors and correctors: a theory of fault-tolerance components,” in *Proceedings. 18th International Conference on Distributed Computing Systems (Cat. No.98CB36183)*, Amsterdam, Netherlands, 1998, pp. 436–443, doi: 10.1109/ICDCS.1998.679772.

[114] “CortexFamily.” http://www.emcu.it/CortexFamily/CortexFamily.html (accessed Nov. 19, 2020).

[115] “ARM CoreSight Architecture Specification v2.0,” p. 182, 2004.

[116] “Intel® 64 and IA-32 Architectures Software Developer Manuals,” *Intel*. https://www.intel.com/content/www/us/en/develop/articles/intel-sdm.html (accessed Oct. 11, 2020).

[117] J. Lask, “Getting printf Output from Target to Debugger,” *SEGGER Blog*, Oct. 21, 2016. https://blog.segger.com/getting-printf-output-from-target-to-debugger/ (accessed Oct. 24, 2019).

[118] A. Ltd, “Microprocessor Cores and Technology – Arm,” *Arm | The Architecture for the Digital World*. https://www.arm.com/products/silicon-ip-cpu (accessed Sep. 23, 2020).

[119] “Which-ARM-Cortex-Core-Is-Right-for-Your-Application.pdf.” https://www.silabs.com/documents/public/white-papers/Which-ARM-Cortex-Core-Is-Right-for-Your-Application.pdf (accessed Mar. 14, 2020).

[120] A. Ltd, “Learn the Architecture | Introducing the Arm architecture,” *Arm Developer*. https://developer.arm.com/architectures/learn-the-architecture/introducing-the-arm-architecture/single-page (accessed Sep. 23, 2020).

[121] R. Boys, “Serial Wire Viewer (SWV) for ARM Cortex-M3 Processors ARM Real-Time Trace aids debugging.” ARM, Ltd.

[122] A. Ltd, “Learn the Architecture | Understanding trace,” *Arm Developer*. https://developer.arm.com/architectures/learn-the-architecture/understanding-trace/single-page (accessed Oct. 11, 2020).

[123] “Cortex-M3 Technical Reference Manual.” https://developer.arm.com/documentation/ddi0337/e/system-debug/itm (accessed Oct. 11, 2020).

[124] F. Baldassari, “Profiling Firmware on Cortex-M,” *Interrupt*, Jun. 02, 2020. https://interrupt.memfault.com/blog/profiling-firmware-on-cortex-m (accessed Oct. 12, 2020).

[125] A. Ltd, “Cortex-M4,” *ARM Developer*. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4 (accessed Nov. 07, 2019).

[126] “System Trace Macrocell Packs Major Benefits for High-Performance SoC System Debug,” p. 6, 2014.

[127] “IAR Finding bugs in Arm Cortex-M3 and -M4 applications.” https://www.iar.com/support/resources/articles/How-to-find-hard-to-find-bugs-in-ARM-Cortex-M3-and-M4-applications/ (accessed Nov. 08, 2019).

[128] “RealView Compilation Tools Developer Guide,” p. 204.

[129] “Embedded Trace Macrocell Architecture Specification,” p. 420, 2011.

[130] “CoreSight Program Flow Trace Architecture Specification,” p. 252, 2011.

[131] A. Ltd, “Resources | Using the CoreSight ELA-500 Embedded Logic Analyzer with Arm DS-5,” *Arm Developer*. https://developer.arm.com/tools-and-software/embedded/legacy-tools/ds-5-development-studio/resources/tutorials/using-the-coresight-ela-500-embedded-logic-analyzer-with-arm-ds-5 (accessed Oct. 14, 2020).

[132] A. Ltd, “DSTREAM family | High Speed Serial Trace Probe (HSSTP),” *Arm Developer*. https://developer.arm.com/tools-and-software/embedded/debug-probes/dstream-family/dstream/high-speed-serial-trace-probe (accessed Nov. 18, 2020).

[133] “Debug Interface - an overview | ScienceDirect Topics.” https://www.sciencedirect.com/topics/computer-science/debug-interface (accessed Nov. 08, 2019).

[134] M. Unemyr, “Cortex-M debugging: Introduction to Serial Wire Viewer (SWV) event- and data tracing.” http://blog.atollic.com/cortex-m-debugging-introduction-to-serial-wire-viewer-swv-event-and-data-tracing (accessed Nov. 17, 2020).

[135] Y. Bai, *Microcontroller Engineering with MSP432: Fundamentals and Applications*. CRC Press, 2016.

[136] “CoreSight Components Technical Reference Manual,” p. 376.

[137] N. Decker *et al.*, “Online analysis of debug trace data for embedded systems,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, Germany, Mar. 2018, pp. 851–856, doi: 10.23919/DATE.2018.8342124.

[138] “CEDAR Tools,” *Accemic*. / (accessed Oct. 16, 2020).

[139] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir, “A Survey of Runtime Monitoring Instrumentation Techniques,” *Electron. Proc. Theor. Comput. Sci.*, vol. 254, pp. 15–28, Aug. 2017, doi: 10.4204/EPTCS.254.2.

[140] M. Poirier, “Hardware Assisted Tracing on ARM with CoreSight and OpenCSD,” p. 35.

[141] “OpenCSD – Operation and Use of the Library,” *Linaro*. https://www.linaro.org/blog/opencsd-operation-use-library/ (accessed Nov. 14, 2020).

[142] M. A. Wahab *et al.*, “A novel lightweight hardware-assisted static instrumentation approach for ARM SoC using debug components,” *arXiv:1812.01667 [cs]*, Dec. 2018, Accessed: Oct. 16, 2020. [Online]. Available: http://arxiv.org/abs/1812.01667.

[143] S. M. A. Zeinolabedin, J. Partzsch, and C. Mayr, “Real-time Hardware Implementation of ARM CoreSight Trace Decoder,” *IEEE Des. Test*, pp. 1–1, 2020, doi: 10.1109/MDAT.2020.3002145.

[144] A. Weiss and A. Lange, “Trace-data processing and profiling device,” US9286186B2, Mar. 15, 2016.

[145] “Tracing Embedded Multi-Core Systems.” https://accemic.com/wp-content/uploads/2020/07/HANSER-automotive_03_2020-EN-light.pdf (accessed Nov. 18, 2020).

[146] “Keil MicroVision IDE.” http://www2.keil.com/mdk5/uvision/ (accessed Nov. 18, 2020).

[147] H. Hogl, D. Rath, and H. Hoegl, “Open On-Chip Debugger – OpenOCD –,” p. 10.

[148] “ARM Development Tools Comparison.” http://www2.keil.com/armtools/compare (accessed Nov. 18, 2020).

[149] J. Aparicio, *japaric/itm-tools*. 2020.

[150] “STM32F4 Projects,” *GitHub*. https://github.com/MaJerle/stm32f429 (accessed Nov. 14, 2020).

[151] “Autonomous Emergency Braking with Sensor Fusion - MATLAB & Simulink.” https://www.mathworks.com/help/driving/ug/autonomous-emergency-braking-with-sensor-fusion.html (accessed Oct. 26, 2020).

[152] K. Koscher *et al.*, “Experimental Security Analysis of a Modern Automobile,” p. 16.

[153] J. Petit, B. Stottelaar, and F. Kargl, “Remote Attacks on Automated Vehicles Sensors: Experiments on Camera and LiDAR,” p. 13.

[154] S. Sharma, A. Flores, C. Hobbs, J. Stafford, and S. Fischmeister, “Safety and Security Analysis of AEB for L4 Autonomous Vehicle Using STPA,” p. 13 pages, 2019, doi: 10.4230/OASICS.ASD.2019.5.

[155] A. V. Jayakumar, “Systematic Model-based Design Assurance and Property-based Fault Injection for Safety Critical Digital Systems,” p. 136.

[156] “Discovery kit with STM32F407VG MCU,” p. 4.

[157] “Arm Aurora High Speed Serial Trace Port (HSSTP).” https://www.pls-mc.com/products/arm-aurora-high-speed-serial-trace-port-hsstp/ (accessed Aug. 02, 2020).

[158] “STM32-MAT/TARGET,” *STMicroelectronics*. https://www.st.com/en/development-tools/stm32-mat-target.html (accessed Jul. 12, 2020).

[159] V. Delebarre and J.-F. Etienne, “Proving Global Properties with the Aid of the SIMULINK DESIGN VERIFIER Proof Tool,” *Formal Methods: Industrial Use from Model to the Code*, pp. 183–223, 2013.

[160] “Model Checks for Secure Coding (CERT C, CWE, and ISO/IEC TS 17961 Standards) - MATLAB & Simulink.” https://www.mathworks.com/help/slcheck/ug/model-checks-for-secure-coding-cert-c-cwe-and-isoiec-ts-17961-standards.html (accessed Oct. 31, 2019).

[161] “CLeaflet.pdf.” http://www.sigada.org/conf/sigada2003/SIGAda2003-CDROM/SIGAda2003-Logos/PolySpace/Marketing/Product_Leaflets/CLeaflet.pdf (accessed Nov. 05, 2019).

[162] “Polyspace.” https://www.mathworks.com/products/polyspace.html (accessed Nov. 05, 2019).

[163] “ModelSim SE User’s Manual,” p. 668.

[164] C. N. Coelho and H. D. Foster, “Assertion-Based Verification,” in *Advanced Formal Verification*, R. Drechsler, Ed. Boston, MA: Springer US, 2004, pp. 167–204.

[165] A. Dahan *et al.*, “Combining system level modeling with assertion based verification,” in *Sixth international symposium on quality electronic design (isqed’05)*, Mar. 2005, pp. 310–315, doi: 10.1109/ISQED.2005.32.

[166] J. Havlicek and Y. Wolfsthal, “PSL AND SVA : TWO STANDARD ASSERTION LANGUAGES ADDRESSING COMPLEMENTARY ENGINEERING NEEDS,” 2003.

[167] R. Anderson, “On-Target Testing in the Simulink Model-Based Design Environment,” p. 32.

[168] “Real-Time Execution in External Mode - MATLAB & Simulink.” https://www.mathworks.com/help/sldrt/ug/simulink-external-mode.html (accessed Nov. 07, 2019).