# CYBER-PHYSICAL SYSTEMS: FROM SPECIFICATION INFERENCE TO DESIGN ANALYSIS

by

LUAN VIET NGUYEN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Engineering
at The University of Texas at Arlington
May, 2018

Arlington, Texas

# ABSTRACT

# CYBER-PHYSICAL SYSTEMS: FROM SPECIFICATION INFERENCE TO DESIGN ANALYSIS

Luan Viet Nguyen, Ph.D.

The University of Texas at Arlington, 2018

Supervising Professor: Taylor T. Johnson

Due to the high degree of uncertainty and complexity, design and analysis of cyber-physical systems (CPS) are very challenging tasks. The challenge arises immediately in the early CPS development cycle, which is the stage of developing the requirements to capture the desirable behaviors of a system. Missing or incomplete requirements make the CPS design untestable or unverifiable, resulting in CPS failures. Hence, there is an urgent need for the development of specification language as well as specification inference techniques that can efficiently determine formal specifications, well-formulated requirements for CPS.

This dissertation presents three different methodologies to determine formal specifications and facilitate the design and analysis of CPS including:

1. the dynamic analysis prototype for automatically identifying cyber-physical specifica-

tion mismatches of unsafe CPS upgrades,

2. the first study of hyperproperties of real-valued signals, along with a new temporal logic to express them, and

3. the logic-driven classification mechanism to detect abnormal signal behaviors in both time and frequency domains.

The above approaches focus on developing formalisms and mining algorithms to formally express and infer many classes of CPS specifications. Based on that, we build a variety of prototypes that can automatically identify specification mismatches arising due to system upgrades, enable simulation-based verification, and detect abnormal model behaviors.

## ACKNOWLEDGMENTS

First and foremost, I would like to express my most profound gratitude and appreciation to my supervising professor, Dr. Taylor T. Johnson, for his continuous support, dedicated guidance and tremendous encouragement that helped me to achieve valuable milestones in my research career. His profound knowledge, great vision, and helpful advice were essential for me to overcome numerous obstacles and challenges during my doctoral study. Without his support, this dissertation could not be accomplished.

I would also like to thank Dr. Christoph Csallner, Dr. Jeff (Yu) Lei, and Dr. Gergely Zaruba for their interest in my research, taking time to serve on my dissertation committee and providing insightful comments and suggestions on my studies and this thesis. Especially, I greatly appreciate Dr. Christoph Csallner for the indispensable support he has given me during my last year at UTA.

My heartiest appreciation goes to Dr. James Kapinski, Dr. Xiaoqing Jin, Dr. Jyotirmoy V. Deshmukh, and Dr. Kent Butts for offering me an opportunity to work with them at Toyota Technical Center (TTC). Doing internships under their supervisions at TTC is one of the greatest moments in my doctoral study.

I would like to extend my thankfulness to my lab members and my colleagues, Hoang-Dung Tran, Dr. Omar Beg, and Shafiul Chowdhury for their valuable discussions, hands-on assistance, and insightful comments on my works.

I owe a debt of gratitude to all of my friends for their help and advice during my studies at UTA. In particular, I truly appreciate Dr. Minh Quang Nguyen for helping me when I first started my doctoral program and sharing with me his knowledge and experiences in research. My special thanks go to Dr. Cuong Manh Nguyen and Dr. Trang Thuy Thai for their unlimited inspiration and encouragement during my doctoral study. I also would like to thank Son Nguyen, Duong Le, and Cuong Ngo for their sweetest friendship and generous support during my time at UTA.

Most importantly and from the bottom of my heart, I would like to thank my parents Mr. Linh Viet Nguyen and Mrs. Tuyet Nga Thi Ly, and my parents-in-law Mr. Long Huu Bui and Mrs. Quynh Thuy Dao for their immeasurable love and tremendous support. I am greatly appreciative of my sister Tram Anh Thi Nguyen and my closest friend Anh Vu Vuong for taking good care of my parents. I would like to show my special thanks to my cousin, Mr. John Tuan Pham, my aunt, Mrs. Tro Truong and my aunt's husband, Thuan Le, for their invaluable support to me since my first day in the U.S.

The material presented in this dissertation is based upon our papers [114, 116, 117], which were supported by the National Science Foundation (NSF) under grant numbers CNS 1464311, CNS 1713253, EPCN 1509804, SHF 1527398, and SHF 1736323, the Air Force Research Laboratory (AFRL) through the AFRL's Visiting Faculty Research Program (VFRP) under contract number FA8750-13-2-0115, as well as contract numbers FA8750-15-1-0105, and FA8650-12-3-7255 via subcontract number WBSC 7255 SOI VU 0001, and the Air Force Office of Scientific Research (AFOSR) through AFOSR's Summer Faculty Fellowship Program (SFFP) under contract number FA9550-15-F-0001, as well as under contract numbers FA9550-15-1-0258 and FA9550-16-1-0246.

Lastly, I would take this opportunity to show my eternal appreciation to my wife, Dr. Loan Bui, for all her support, understanding and patience during my journey to the doctoral degree. She is my most treasured possession, who always enlightens my soul with her endless love. When I think about the months, years, and decades ahead, I always see her walking beside me.

*To my parents*

*To my loving wife*

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# Chapter 1

# INTRODUCTION

## 1.1 Motivation and Background

Cyber-physical systems (CPS) are networked computing devices that communicate with one another and tightly interact with physical environments. Today, CPS have extraordinary impacts on human life, appear everywhere as the foundations of smart homes, smart power/energy systems, intelligent transportation, and biotechnology. These systems are characterized by both continuous and discrete dynamics, with numerous subsystems interacting with each other in sophisticated manners. Due to the high degree of complexity, the design and analysis of CPS associated with modeling, verification, and validation activities are very challenging [9]. In the last two decades, many research efforts from both academic communities and industrial companies have been initiated in developing automated tools and methods to determine correct specifications and improve the modeling and verification of safety-critical CPS. However, safety-critical failures arising from the complex interaction of software and physical processes in CPS are still prevalent and rampant as exemplified by frequent product recalls across many CPS domains such as medical devices [7], aerospace [90, 92], and automotive [111]. These safety-critical failures in CPS are mounting evidence of a critical need for new scalable approaches in testing and verification to ensure design correctness and safe operation of CPS.

### 1.1.1 CPS Modeling and Verification

**Modeling.** CPS are characterized as mixed discrete-continuous systems, so they are often considered as hybrid systems that can be modeled through hybrid automata. Hybrid automata [10] are a modeling formalism which illustrate dynamical systems including both continuous states and dynamics as well as discrete states and transitions. In essence, hybrid automata augment finite state machines with a set of real-valued variables that evolve continuously over intervals of real time according to some ordinary differential equations or inclusions. Depending on the dynamics of systems, hybrid automata can be classified into different classes such as *time automata* [12], *rectangular hybrid automata* [127], *linear hybrid automata* [68], and *nonlinear hybrid automata* [69]. Several hybrid automata modeling frameworks have been developed by the academic community to model CPS such as Petri Nets [42], Shift [45], HyVisual [34], and PowerDEVS [26]. Other commercial environments including Simulink/Stateflow (SLSF), Modelica, and LabView are also widely used for modeling CPS across industrial domains.

**Verification.** Verification is an important process that aims to prove the correctness and reliability of a system design. Intuitively, verification procedure answers the question of whether we build a system in a correct manner. Formally speaking, given a system model with its desirable requirements, verification problem is to design an algorithm which can prove that the model always satisfies its requirements. If the model does not meet the requirements, it returns a *counterexample* (a specific behavior that violates the requirements) that allows us to determine a bug existing in the current system design.

Solving a verification problem, especially for CPS including continuous states and discrete transitions, is a very challenging task. For CPS modeled as hybrid automata, the verification problem is equivalent to computing all reachable states of the automata to ensure that there is no state of the model violating a given requirement. Unfortunately, such reachability computation is known to be undecidable [10], and hence there is no complete algorithm to determine whether a given system does or does not satisfy a formal requirement. To overcome this challenge, a standard approach is to compute an over-approximation for a set of reachable states. If an over-approximated reachable set does not contain any unsatisfied state, neither does the precise reachable set. Several automatic verification tools have been developed from academic communities to compute the over-approximated reachable sets of various types of hybrid automata such as PHAVer [59] and SpaceEx [60] for linear/affine hybrid automata, Flow* [36] and dReach [86] for nonlinear hybrid automata, C2E2 for hybrid systems modeled as continuous or hybrid SLSF models [53]. Alternatively, several interactive theorem prover tools such as PVS [125], SAL [44], and Keymaera [128] have been successfully applied to verify various classes of hybrid systems.

Although all of the existing hybrid analysis tools have similar formal semantics, their model description languages are quite different. Given a particular hybrid automaton model, it is difficult to compare the reachable sets of the model calculated by different tools to evaluate the verification results. To overcome this problem, Bak et al. introduced a conversion tool for hybrid automata called HyST [18], which allows the same model to be analyzed simultaneously in several verification tools such as HyCreate, Flow*, and dReach. HyST[1] takes a source input model in SpaceEx xml format, parses it to an intermediate representation, and then prints a resulting output to some desired formats. The automatic model generation by HyST saves a lot of time and effort for a developer involved in comparing the performance of newly developed algorithms by existing hybrid systems analysis tools. In addition, HyST can automatically translate hybrid automata to *trajectory-equivalent* SLSF models, which enables a *correct-by-construction* compositional design for CPS with embedding hybrid automata into large-scale SLSF models [17]. Furthermore, HyST can also be used as a *runtime monitor* with *randomized differential testing* for several components (from parsers to analysis algorithms) in hybrid analysis tools. Indeed, a test subject is the hybrid automaton randomly generated in the input format for SpaceEx using a prototype tool called HyRG[2] [118], which is then translated to equivalent automata in different formats by HyST. Next, the automata will be analyzed using different tools, such as SpaceEx, Flow*, dReach, and HyCreate, or simulated in SLSF. By comparing all analysis results, potential bugs existing in either HyST or those analysis tools can be determined [119].

Another difficulty of verifying hybrid systems is the *state-space explosion* or the curse of dimensionality. This problem can be interpreted that when the number of state variables of a system increases, the size of the system state-space will grow exponentially. This difficulty makes most of the existing formal verification techniques infeasible for verifying industrial large-scale system models. Instead, *simulation-based* verification is an alternative approach that can be applied to find a violating behavior of a large-scale model. Simulation-based ver-

[1]The tutorial and source code of HyST are available at: `https://github.com/verivital/hyst`

[2]The tool description and source code of HyRG are available at: `http://www.verivital.com/hyrg`

ification computes numerical approximations of system behaviors by extensively simulating a system and checking whether there is any executed trace violating a given specification. As a system may have infinitely many different executions, this approach does not provide formal guarantees for the system correctness, but presents only the probabilistic correctness assurances. However, since simulation is cheap and usually fast, it is often chosen by control engineers to perform testing or quality checking for large-scale design models in practice.

*Falsification* is a recently-developed simulation-based, automated best-effort approach which can be used to identify system behaviors that violate a given formal specification efficiently [113]. Formally speaking, given a hybrid system as a Simulink/Stateflow model and a set of properties, a falsification problem is to find inputs, parameters and initial conditions that drive the model toward violations of these properties. Although falsification cannot prove the system correctness, it can efficiently find bugs existing in the real-world models of CPS that are too complex to be formally verified [80]. Several falsification tools such as Breach [48], S-taliro [14], and S3CAM [153] have been introduced and successfully applied to falsify automotive control systems.

### 1.1.2 CPS Specifications

To successfully design and analyze safety-critical systems, the first important stage is to develop correct requirements (or specifications) that characterize the correctness and reliability of a system. Based on that, engineers can iteratively develop a precise model and then perform verification and validation activities to ensure that the model satisfies its requirements. Several important classes of CPS requirements include *safety* requirement (i.e., a system never reaches a bad state), *liveness* requirement (i.e., a system always eventually visits some good state), and *stability* requirement (i.e., a system eventually visits good state and stays there).

The requirement development stage is vital as any errors in this first stage of system development will result in errors in the later design stages and increase the development cost to fix them. Ambiguous and incomplete requirements make the design unverifiable and are often root-causes of system failures [16, 47, 67, 148, 149]. In practice, engineers often write specifications in terms of informal expressions such as natural language and then validate their design manually against those informal requirements. In many cases, they simply execute a system, acquire the simulation data and then use their domain expertise to justify the design quality subjectively. To avoid system failures and reduce development cost, it is essential to develop well-formulated requirements that accurately specify the design goals in a precise mathematical expression [47,76]. However, formal requirements are often difficult and costly to write, and determining them is not always an easy task that requires a significant investment of time and effort [16, 71].

In the last two decades, temporal logic has emerged as one of the most powerful formalisms that can be used to express specifications for many systems such as reactive systems, embedded control systems, and hybrid systems. Various classes of the temporal logic such as Linear Temporal logic (LTL) [129], Metric Temporal Logic (MTL) [88] and Signal Temporal Logic (STL) [95] have been developed and applied to formally specify different types of system requirements. Among these logics, STL is powerful in specifying many CPS requirements defined over continuous-real valued signals. However, there are two main limitations of STL that prevents its usage from expressing other class of CPS requirements. First, STL

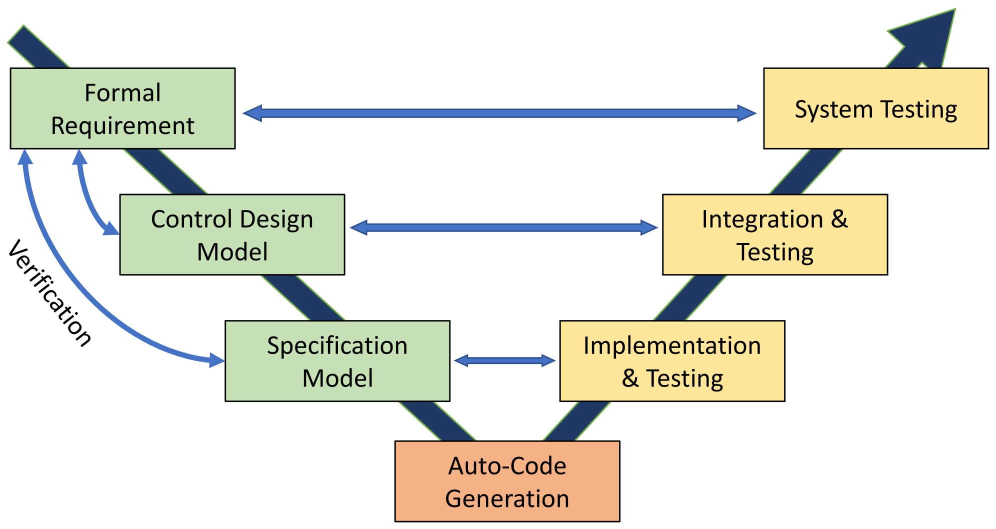


**Figure 1.1:** A typical V-process in model-based development.

only deals with a signal in a time domain; it cannot be used to specify a system requirement involving frequency information. In many cases, the most distinguished information is hidden in the frequency content of a signal, especially for signal processing applications. Second, STL only expresses properties of individual execution traces; we also could not use STL to specify and test requirements comparing multiple execution traces of a system. In this dissertation, we focus on the extensions of STL to overcome these limitations. In fact, we introduce a notion of time-frequency logic (TFL) using continuous wavelet transform to specify system requirements in a time-frequency domain, and a HyperSTL logic to express the hyperproperties of real-time, real-valued signals.

### 1.1.3 CPS Model-based Development

To determine the correct specifications and facilitate the verification and validation challenges of CPS, engineers can utilize a model-based development (MBD) paradigm as shown in Figure 1.1. This typical V-process allows us to check whether the design goals are met at every phase of the development. In this process, the phases on the left-hand side are with regards to specification and model development, while those on the right-hand side correspond to implementation, integration and operational testing activities. Here, system requirements are provided beforehand to ensure that the subsequent models, as well as the final system implementation, always produce behaviors that satisfy the requirements.

Instead of verifying that a final hardware/plant implementation meets system requirements, engineers can perform verification and validation activities on a system model to check whether the design satisfies system requirements early in the design process, before implementing the model on hardware. As a result, this will allow necessary improvements of the specifications and the design to be made early in the design process and reduce the cost of rework at later phases. The focus of this dissertation is to develop specification inference or requirement mining techniques to specify different classes of CPS requirements and facilitate verification and validation activities to be performed early on the specification model shown in Figure 1.1.

## 1.2 Dissertation Contributions and Organization

In summary, the contributions of this dissertation include the development of specification formalisms as well as sufficient mining algorithms to formally specify and verify many classes of CPS specifications.

Chapter 2 presents an automated method towards identifying unstated assumptions in CPS. Dynamic specifications in the form of candidate invariants of both the software and physical components are identified using dynamic analysis (executing and/or simulating the system implementation or model thereof). A prototype tool called Hynger (for HYbrid iNvariant GEneratoR) was developed that instruments SLSF model diagrams to generate traces in the input format compatible with the Daikon invariant inference tool, which has been extensively applied to software systems. Hynger, in conjunction with Daikon, is able to detect the candidate invariants of several CPS case studies. We use the running example of a DC-to-DC power converter, and demonstrate that Hynger can detect a specification mismatch where a tolerance assumed by the software is violated due to a plant change. Another case study of an automotive control system is also introduced to illustrate the power of Hynger and Daikon in automatically identifying cyber-physical specification mismatches.

In Chapter 3, we introduce the first study of *hyperproperties* of CPS. A hyperproperty is a property that requires two or more execution traces to check. This is in contrast to properties expressed using temporal logics such as LTL, MTL, and STL, which can be checked over individual traces. Hyperproperties are important as they are used to specify critical system performance objectives, such as those related to security, stochastic (or average) performance, and relationships between behaviors. We introduce a new formalism for specifying a class of hyperproperties defined over real-valued signals, called HyperSTL. The proposed logic extends signal temporal logic (STL) by adding existential and universal trace quantifiers into STL's syntax to relate multiple execution traces. Several instances of hyperproperties of CPSs including stability, security, and safety are studied and expressed in terms of HyperSTL formulae. Furthermore, we propose a testing technique that allows us to check or falsify the hyperproperties of CPS models. We present a discussion on the feasibility of falsifying or verifying various classes of hyperproperties for CPSs. We extend the quantitative semantics of STL to HyperSTL and show its utility in formulating algorithms for falsification of HyperSTL specifications. We demonstrate how we can specify and falsify HyperSTL properties for two case studies involving automotive control systems.

Chapter 4 proposes a technique to investigate abnormal behaviors of signals in both time and frequency domains using an extension of *time-frequency logic* that uses the *continuous wavelet transform*. Abnormal signal behaviors such as unexpected oscillations, called *hunting* behavior, can be challenging to capture in the time domain; however, these behaviors can be naturally captured in the time-frequency domain. We introduce the concept of parametric time-frequency logic and propose a parameter synthesis approach that can be used to classify hunting behavior. We perform a comparative analysis between the proposed algorithm, an approach based on support vector machines using linear classification, and a method that infers a signal temporal logic formula as a data classifier. We present experimental results based on data from a hydrogen fuel cell vehicle application and electrocardiogram data extracted from the MIT-BIH Arrhythmia Database.

Chapter 5 summarizes the dissertation and introduce potential directions for future research.

## 1.3 Copyright Permissions

Herein we include the required copyright permissions to reproduce parts of [114,116,117] in this dissertation.

- For parts of [114], which was accepted for publication in ACM Transactions on Cyber-Physical Systems, we acknowledge the upcoming ACM copyright: ©Reprinted, with permission from Luan V. Nguyen, Khaza Anuarul Hoque, Stanley Bak, Steven Drager, and Taylor T. Johnson, “Cyber-Physical Specification Mismatches,” *ACM Transactions on Cyber-Physical Systems, ACM, 2018* (to Appear).

- For parts of [117], we acknowledge the ACM copyright: ©Reprinted, with permission from Luan V. Nguyen, James Kapinski, Xiaoqing Jin, Jyotirmoy V. Deshmukh, and Taylor T. Johnson, “Hyperproperties of Real-valued Signals,” *in Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE ’17), ACM, New York, NY, USA, 104-113. DOI: https://doi.org/10.1145/3127041.3127058.*

- For parts of [116], we acknowledge the ACM copyright: ©Reprinted, with permission from Luan V. Nguyen, James Kapinski, Xiaoqing Jin, Jyotirmoy V. Deshmukh, Ken Butts, and Taylor T. Johnson, “Abnormal data classification using time-frequency temporal logic,” *in Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control (HSCC ’17). ACM, New York, NY, USA, 237-242. DOI: https://doi.org/10.1145/3049797.3049809.*

# Chapter 2

# CYBER-PHYSICAL SPECIFICATION MISMATCH AND SAFE CPS UPGRADES

## 2.1 Introduction

Systems interacting with their physical environments are becoming increasingly dependent upon computers and software, such as in emerging CPS. For instance, typical modern cars utilize hundreds of microprocessors, many communications buses, and a complex interconnection between sensors, actuators, and processors. In the design and development process for most engineered systems, the vast majority of resources are devoted to ensuring systems meet their specifications [24]. However, in spite of significant technical advances for designing verification and validation such as model checking, Software/Hardware-In-The-Loop (SIL/HIL) testing, automatic test case generation for software, and sophisticated simulators, there still remain products recalled across industries for safety concerns due to software problems and system integration between the cyber and physical subcomponents. The verification community typically focuses on the *developmental verification*, where a model of a system is developed and properties (specifications) are (manually, semi-automatically, or automatically) checked for that system. However, many product recalls and safety disas-

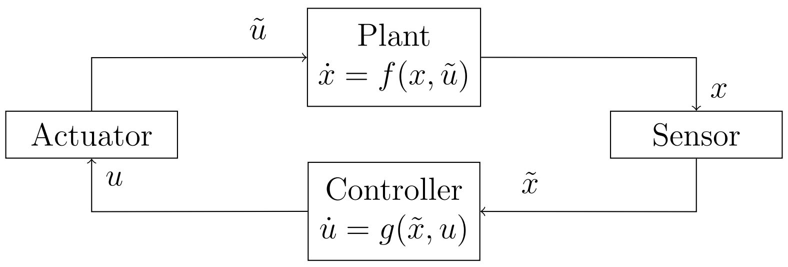


**Figure 2.1:** High-level diagram of a closed-loop control system.

ters induced by software bugs are not a result of design errors, but are the result of either: ($a$) implementation errors, or ($b$) reuse, upgrade, and maintenance errors. Initiatives like a priori MBD are important research efforts and may someday lead to synthesizing provably correct implementations from specifications. However, most systems being designed today still utilize a development process that has engineers writing the software, and systems are integrated with numerous components in a potentially error-prone process. For instance, a typical CPS that has been used widely in control systems is a closed-loop feedback controller shown in Figure 2.1, where a plant describes physical changes of the environment and a controller represents cyber/software computations corresponding to these changes. The physical evolution of the plant can be sensed and sampled by a sensor, and then fed into the controller. Based on the measurement of the plant provided by the sensor, the controller provides a corresponding control signal to regulate the physical changes in the plant. This control signal is converted by an actuator before sending it to the plant. Such a system may contain different possibilities of failure due to the following main reasons: ($a$) the controller may make wrong assumptions about the plant, sensor or actuator. For example, changing parameters of the plant, sensor, or actuator without updating the controller may produce potential specification mismatches. This controller-reuse issue can lead to safety failures such as the Honda vehicles recalls or the Ariane 5 flight 501 disaster described in

Section 2.2. ($b$) The plant may be influenced by uncontrolled factors (disturbances) from the environment, ($c$) the controller is initially encoded based on wrong information about the physical plant, ($d$) the sensor and actuator may have conversion errors, and ($e$) the control conflicts may arise when using a shared sensor and actuator network.

In this chapter, we propose a method to address such challenges that arise in the product evolution and upgrade process in CPS. Our proposed method enables dynamic analysis using well-established software engineering tools for large classes of Simulink/Stateflow (SLSF) models that are frequently used in CPS engineering. In particular, the method infers candidate invariants of SLSF models. Invariants are properties of a system that should always hold, while conditional invariants may hold at certain program points, for example, at the beginning or end of a function call (pre/post conditions). This is important because such models are amenable to formal verification using existing research tools and hybrid system model checkers. Finding invariants can aid this process as they represent potential abstractions with a possibly less complex representation than the set of reachable states. The SLSF block diagrams may be black box components, white box components, or even system implementations (such as when SLSF is used in SIL/HIL simulation). In the case when the underlying SLSF models are known, they may be formalized using hybrid automata [97]. Candidate invariants inferred with our Hynger (for HYbrid iNvariant GEneratoR) software tool in conjunction with Daikon [55,56] may be formally checked as actual invariants using a hybrid system model checker [60]. Figure 2.2 shows a preliminary overview of our proposed methodology. As a prelude, we just intuitively demonstrate how Hynger and Daikon can be used to detect specification mismatches. The proposed framework will be fully presented in Section 2.5.

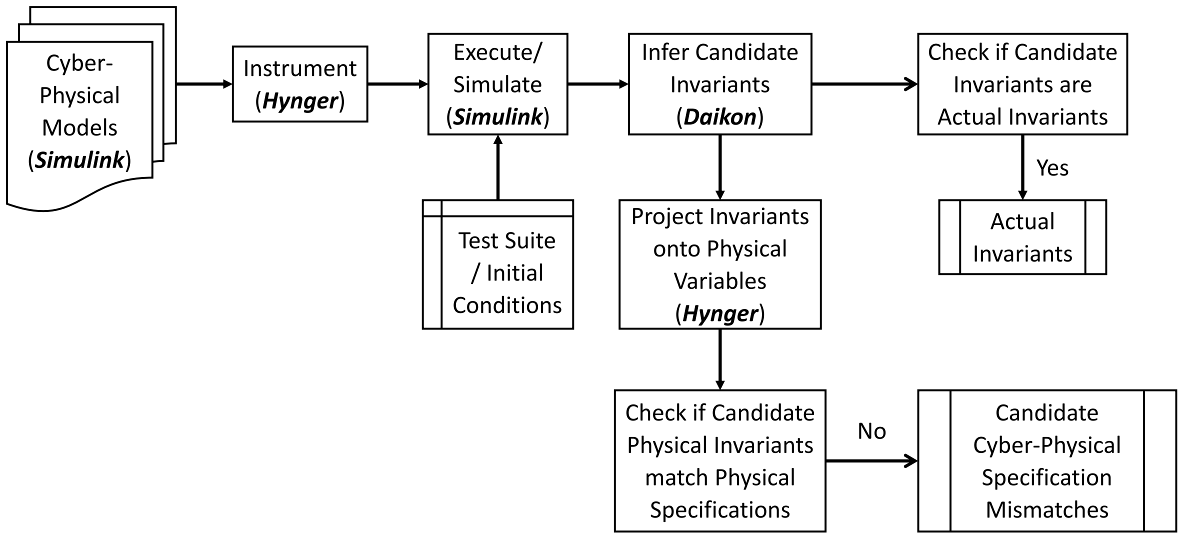

Cyber-Physical Models (***Simulink***)
Instrument (***Hynger***)
Execute/ Simulate (***Simulink***)
Test Suite / Initial Conditions
Infer Candidate Invariants (***Daikon***)
Check if Candidate Invariants are Actual Invariants
Yes
Actual Invariants
Project Invariants onto Physical Variables (***Hynger***)
Check if Candidate Physical Invariants match Physical Specifications
No
Candidate Cyber-Physical Specification Mismatches

**Figure 2.2:** Preliminary overview of the proposed methodology using Hynger and Daikon to infer candidate invariants and detect specification mismatches.

**Contributions.** The primary contributions of this chapter are: *(a)* the formalization of the cyber-physical specification mismatch problem, *(b)* a methodology for performing template-based automated invariant inference of white box (known) and black box (unknown) CPS models using dynamic analysis, *(c)* the Hynger software tool, which supports instrumenting large classes of SLSF diagrams for dynamic analysis using tools like Daikon, *(d)* a methodology for checking if the inferred invariants are actual invariants by using formal models of the underlying SLSF model diagrams and hybrid systems model checkers such as SpaceEx [60], etc., *(e)* two proof-of-concept CPS case studies using Hynger to automatically identify cyber-physical specification mismatches. These results can be used to help bridging the worlds of actual embedded systems software (e.g., detailed SLSF diagrams and generated C code) with hybrid system models.

## 2.2 Cyber-Physical Design Reuse and Upgrade

In this section, we review cases where CPS design reuse and upgrade have led to failures in existing systems. This motivates the need for our proposed method and our Hynger tool, which can be used to find and formalize unstated assumptions in CPS.

A recent example of a design-reuse problem is the National Highway Transportation and Safety Administration (NHTSA) recall of 1.5 million Honda vehicles due to Electronic Control Module (ECM) software problems that could damage the car's transmission, resulting in possible stalls. The root cause of the safety defect was the result of a physical component (a bearing in the transmission) being upgraded to an improved design between different model-year vehicles without appropriate ECM software updates [111]. This problem was widespread because there was a five year delay before the problem was identified, and it was used across model makes and years (e.g., from $2005-2010$ model year Accords, $2007-2010$ CR-Vs, and $2005-2008$ Elements). This difficulty in root-cause analysis emphasizes the point that such problems are probably underreported, and the reuse of components in CPS can lead to widespread serious problems.

Similar design-reuse problems have famously occurred in aviation—the Ariane 5 flight 501 disaster was a result of reusing Ariane 4's software without appropriate updates for the increased thrust of the new rocket [1, 92]. Here, software developers made an assumption about the physical dynamics of the rocket, but the software was reused from Ariane 4, while Ariane 5 had greater thrust, so this assumption was invalid. Finally, when considering the future of CPS, the Defense Advanced Research Projects Agency's System of Systems Integration Technology and Experimentation (DARPA SoSITE) program [99] describes modularized military aviation systems which are capable of rapid component swapping and upgrade. Left unaddressed, issues related to unstated assumptions in components are likely to get worse in future CPS, where changes can occur in the software and hardware.

Besides design-reuse problems, software upgrades without being thoroughly tested and validated may result in an epic system failure. One famous example of this type of problem is the disaster of Mars Climate Orbiter (MCO), developed by NASA's Jet Propulsion Laboratory (JPL). The root-cause of this disaster was that different parts of the software developer team were using different units of measurements. In fact, one part of the ground software upgraded by Lockheed Martin Astronautics (LMA) measured the thrusters in English units of pounds (force)-seconds instead of metric units of Newton-seconds as defined in its original Software Interface Specification (SIS) used by JPL [90,142]. Therefore, the trajectory of the MCO was erroneously calculated by ground support system computers using the incorrect thruster performance data. This type of software failure occurred due to the lack of adequate communication between different parts of the software team and the uncovered issues of verification and validation processes [142].

### 2.2.1 Related Work

The idea evaluated in this work, that of inferring physical system specifications from embedded software in conjunction with physical system models and evaluating them for mismatches, was inspired by previous work finding program specifications for pure software systems [124]. Cyber-physical specification mismatch is closely related to model inconsistency [133], architectural mismatch [63], and requirements consistency [145]. There are many benefits of dynamic analysis such as using implementations instead of models [55, 56, 124] to find dynamic program specifications [124], providing documentation over program evolution and checking if specifications change drastically over program evolution, etc. For one, models are not actually required for analysis, and implementations may be used [55, 56]. The benefit of executing a system implementation is that there are no mismatches between a model (potentially documentation-based) and implementation, since it is not necessary to have a model at all. The candidate specification generated may be viewed as a form of input-output abstraction of the actual implementation. The limitation includes results that are unsound without additional reasoning.

Recently, Medhat and his collaborators introduced a new framework for inferring hybrid automata from black-box implementations of embedded control systems by mining their input/output traces [103]. In their work, the input/output training traces collected from executing a system are clustered and then translated to event sequences. Under several assumptions, hybrid automata representing the behaviors of the system can be inferred using the input/output correlation. Although the work suffers some limitations, their proposed approach is a proof-of-concept of using dynamic analysis to infer the specifications of black-box systems. This work is highly relevant to our proposed method as there is an analogy between inferring hybrid automata and finding a candidate invariant for a black-box system. In fact, both of them can be considered as doing specification inference using dynamic analysis.

There are also several tools such as DepSys [110] and EyePhy [109] that used both static and dynamic analysis to detect and address the control conflict due to dependencies when using multiple CPS applications. Particularly, DepSys is a utility sensing and actuation infrastructure for a smart home that can simultaneously operate multiple CPS applications. The main novelty of DepSys is that it provides a comprehensive strategy to specify, detect and automatically address the control conflicts between sensors and actuators used in a home setting. Similarly, EyePhy is an integrated system that can detect dependencies and then perform a dependency comprehensive analysis across HIL CPS medical applications. A built-in simulator, HumMod, in EyePhy is able to model the complex interactions of the human body using more than 7,800 physiological variables. HumMod demonstrates the model parameters and the quantitative relationship among them in XML files that makes it easier to update the physiological models without the recompilation of the whole system. EyePhy can be considered as the first tool that performs the dependency analysis across applications' control actions on the human body. Additionally, the sensor networks with devices used in smart homes or medical devices can be built using the family of Smart Transducer Interface Standards (IEEE 1451). IEEE 1451 has been developed in order to provide the common communication interfaces for connecting transducers (sensors or actuators) to their instrumentation systems or control networks [89]. The Transducer Electronic Data Sheets (TEDS) embedded in smart transducers are memory devices, which store the manufacture-related information of the transducer such as manufacture ID, measurement ranges, serial number, etc. Thus, TEDS allows transducers to be self-identified and self-descriptive to the device networks. It also provides a standardized mechanism to facilitate the plug and play of transducers with different control networks. Hence, IEEE 1451 enables the access of transducer data through a common set of interfaces, allowing users to select transducers and networks for their applications. This advantage is crucial in facilitating the device and data interoperability, detecting and resolving conflicts due to dependencies when concurrently using multiple transducers in the device networks.

Finding specifications is a maturing field within software engineering [32, 40, 55, 56, 124]. Daikon, which is used by Hynger, processes program traces to generate invariants [55, 56]. For several languages (C, C++, etc.), this process is performed without access to the source code by instrumenting the compiled program using Valgrind [112]. This makes it difficult to use on non-x86/x86-64 platforms (although Valgrind is gaining access to other architectures), which is a serious limitation, as most embedded platforms utilize other architectures (e.g., ARM, AVR, PIC, 8051, MSP430, etc.). Due in part to these limitations, Hynger instruments architecture-independent SLSF diagrams directly. In the long run, the Hynger tool is envisioned to take an arbitrary SLSF model, instrument it, then analyze the resulting traces with dynamic analysis to identify broad classes of cyber-physical specification mismatches.

The most closely related work using Daikon is to find candidate invariants of hybrid models of biological system [27], and this also illustrates a proof-of-concept of using Daikon as a trace analyzer for non-purely software systems. Daikon can generate invariants of many forms for variables and data structures, such as: ranges ($a \leq x \leq b$), linear ($y = ax + b$), variable ordering ($x \leq y$), sortedness of lists, etc. Daikon works by instrumenting source code and/or compiled binaries with changes that allow for looking at variable values, then Daikon essentially checks if variables satisfy some template invariants. For instance, if an integer variable $x$ is observed to always be smaller than some number, say 50, Daikon may generate a candidate invariant of $x \leq 50$. Based on many advantages of using Daikon as a trace analyzer [55, 56], we prefer to use Hynger with Daikon to infer candidate invariants in our proposed framework. However, we note that Hynger can generate a trace file in many input formats that are compatible with other invariant-inference tools using dynamic analysis not just Daikon. Other research tools like DySy [40] and commercial tools like Agitagor [32] can be used for generating candidate invariants for other languages.

## 2.3 Cyber-Physical System Models

The approach presented in this chapter applies to the systems with formal models, informal models, and unknown models/implementations. The primary assumption is that interfaces to the models or systems are available as SLSF blocks. There are two main categories of blocks appearing in an SLSF diagram that are supported by our method, white box and black box systems. The white box systems may contain: ($a$) known, informal models, ($b$) known, informal implementations, or ($c$) known, formal models (e.g., hybrid automata, or more precisely, classes of SLSF diagrams that may be converted to hybrid automata [97]). The black box systems may be completely unknown, and may contain: ($a$) unknown implementations (e.g., compiled executable binaries with no source available, such as commercial off-the-shelf [COTS] components and other third-party systems), ($b$) unknown models, and ($c$) actual cyber-physical systems (e.g., embedded controllers, networked computers, and physical plants, all that may show up in HIL/SIL simulations interfaced with SLSF).

Next, we define a structure of CPS models used in SLSF. We will not define formal semantics of this structure or SLSF diagrams in this chapter. However, in the case where an SLSF diagram is a white box and formal semantics may be defined, a formal framework like hybrid input/output automata (HIOA) [94] may be used to specify the semantics, such as done in the HyLink tool [97]. Additionally, if an SLSF diagram is a white box and linear, we may also be able to use SL2SX Translator for transforming it into a corresponding formal model [104]. Interested readers can find some graphical examples of the translation in [97,104]. Other formalisms like actors and hierarchical state machines are commonly used for formal modeling of other diagrammatic frameworks similar to SLSF [11, 25, 143, 151]. Given a formal model $\mathcal{A}$ and candidate specification $\Sigma$ (e.g., found using Hynger), we can check if $\mathcal{A}$ meets the specification, i.e., $\mathcal{A} \models \Sigma$ by using a hybrid system model checker like SpaceEx [60]. In some instances, we know when an SLSF diagram corresponds precisely to a hybrid automaton model [97], and in these cases, we can check if candidate invariants found with Hynger are actual invariants.

First, we define the hierarchy represented by SLSF diagrams.

**Definition 1 (SLSF diagram)** *An SLSF diagram is a tuple $\mathcal{A} \triangleq \langle M, E, \mathsf{V} \rangle$, where:*

- *$M$ is a set of blocks (vertices) that represent block diagrams (and sub-blocks/models),*
- *$E \subseteq M \times M$ is a set of edges between blocks representing a parent-child hierarchy, and*
- *$\mathsf{V}$ is a set of variables, written as $\mathsf{V} \triangleq \bigcup_{v \in M} \mathsf{V}(v)$, where $\mathsf{V}(v)$ is a set of variables for each block $v \in M$.*

According to Definition 1, the graph $G \triangleq (M, E)$ defined by the vertices (blocks) $M$ and edges $E$ is a rooted tree, where $M$ are block diagrams and $E$ represents a parent-child hierarchical relationship (e.g., sub-modules and sub-blocks). Here, the root (i.e., top-level) block diagram of the model is the unique root of the tree, which we denote as $\mathit{root}(M)$. For a block $v \in M$, the *children* of $v$ are denoted as $\mathit{children}(v)$ and defined as the set of blocks $\{w \in M \mid w \in E(v)\}$. For a block $v \in M$, the *parent* of $v$ is denoted as $\mathit{parent}(v)$ and is defined as the singleton set $\{w \in M \mid v \in \mathit{children}(w)\}$. Clearly, $\mathit{parent}(\mathit{root}(M)) = \emptyset$. For a block $v \in M$, the *ancestors* of $v$ are denoted as $\mathit{ancestors}(v)$ and defined inductively as the set of blocks $\{w \in M \mid v \ \cup \ w \in \mathit{children}(v) \ \cup \ \mathit{children}(w)\}$ (or equivalently, as the transitive closure of $\mathit{children}(v)$).

For a block $v \in M$, the set of variables of $v$ is $\mathsf{V}(v)$ and is partitioned into sets of input and output variables, written respectively as $\mathsf{V}_I(v)$ and $\mathsf{V}_O(v)$, and we have $\mathsf{V}(v) = \mathsf{V}_I(v) \cup \mathsf{V}_O(v)$. A *variable* $x \in \mathsf{V}(v)$ is a name for referring to some state of $\mathcal{A}$, and is associated with a data type denoted $\mathit{type}(x)$. Typical data types are reals, floating points, arrays, lists, etc. The valuation of a variable $x \in \mathsf{V}(v)$ is the set of all values it may take and is denoted $\mathit{val}(x)$. The state-space of $\mathcal{A}$ is the set of valuations of all the variables $\mathsf{V}$. An element $s$ of the state-space is called a state, and a trace is a sequence of states. The SLSF diagram may also have internal (local) variables, although they are not externally visible, so we do not include them, as only input/output interfaces are visible for external observation and instrumentation.

Next, we define CPS models that appear in SLSF diagrams.

**Definition 2 (CPS model)** *A CPS model is an SLSF diagram with a set of $n$ typed variables, $\mathsf{V} = \{x_1, x_2, \ldots, x_n\}$, which is classified into two subsets as follows.*

- $\mathsf{V}_P = \{\alpha_1, \alpha_2, \ldots, \alpha_{n_p}\}$ *is a set of $n_p \leq n$ physical variables such that their values are continuously updated, and*

- $\mathsf{V}_C = \{\beta_1, \beta_2, \ldots, \beta_{n_c}\}$ *is a set of $n_c$ cyber variables that are discretely updated, where $n = n_p + n_c$.*

Here, the set of variables for each block of a CPS model is also partitioned into sets of physical and cyber variables, $\mathsf{V}(v) = \mathsf{V}_P(v) \cup \mathsf{V}_C(v)$. In practice, this may be accomplished with subtyping using, for example, an overloaded type for floats or fixed points used for approximations of real variables (e.g., in C, `typedef double physical; typedef physical temperature;`). The dynamic changes of the variables of the CPS model may be described using different SLSF blocks such as S-Function block, look-up table, etc. In case the CPS model is a white-box and simple enough, we may translate it to a formal framework like HIOA (e.g using Hylink). In fact, we can specify a set of real-valued variables and their dynamic changes for the converted formal model based on capturing the output variables from unit delay, integrator, state-space blocks in the corresponding SLSF diagram [13]. Moreover, we note that the input and output variables are disjoint, and the cyber and physical variables are disjoint, although these are not all mutually disjoint. Hence, we further classify the set of variables $\mathsf{V}(v)$ into different types as follows.

**Definition 3 (Variable Classification)** *For a block $v \in M$, a variable $x \in \mathsf{V}(v)$ is considered as:*

- *an* input cyber variable *if $x \in \mathsf{V}_C(v)$ and $x \in \mathsf{V}_I(v)$,*
- *an* output cyber variable *if $x \in \mathsf{V}_C(v)$ and $x \in \mathsf{V}_O(v)$,*
- *an* input physical variable *if $x \in \mathsf{V}_P(v)$ and $x \in \mathsf{V}_I(v)$, or*
- *an* output physical variable *if $x \in \mathsf{V}_P(v)$ and $x \in \mathsf{V}_O(v)$.*

We extend these notations in Definition 3 naturally to sets of variables if *all* variables in a set of variables fall into these classes, and will reference them as such. An arbitrary set of variables may not be mutually disjoint from each of the input, output, cyber, and physical variables. Thus, for a set of variables $X \subseteq \mathsf{V}$, we say: $(a)$ $X$ is *cyber-physical* if there exist both cyber and physical variables in $X$, $(b)$ $X$ is *input-output* if there exist both input and output variables in $X$, and $(c)$ $X$ is *cyber input-output*, *physical input-output*, *cyber-physical input*, or *cyber-physical output* for the other natural permutations.

Next, using these variable classes, we define classes of SLSF blocks appearing in SLSF diagrams. For a block $v \in M$, we say: $(a)$ $v$ is a *cyber-physical* block if there exist both cyber and physical variables in $\mathsf{V}(v)$, $(b)$ $v$ is a *cyber* block if there exist *only* cyber variables in $\mathsf{V}(v)$, and $(c)$ $v$ is a *physical* block if there exist *only* physical variables in $\mathsf{V}(v)$.

*Cyber-Physical Variable Interactions.* Next, we will formalize a notion of influence between cyber and physical models and their variables. For example, consider a typical closed-loop plant-controller architecture, where outputs of a plant are sensed, used as inputs to a controller, and outputs of the controller are converted by actuators as inputs to the plant (and potentially disturbances affect everything). Generally, we would say the plant is a physical model, the controller is a cyber model, and the sensors and actuators are cyber-physical models. However, it is clear that the physical variables of the plant affect the cyber variables of the controller, and vice-versa, albeit not directly, but through the transitive closure of input-output connections over all blocks in the SLSF diagram. We note that this is related to the notion of tainted variables in program analysis that is popular in security [138]. To formalize this notion, we specify interconnections between input and output variables between blocks $v \in M$ at the same hierarchical level in the diagram.

Input-output connections may only exist between models with the same parent (i.e., those in the same hierarchical structure). For a block $v \in M$, we denote all blocks with the same parent as $siblings(v)$, which is defined as the set $\{w \in M \mid parent(w) = parent(v)\}$. Output variables of a block $v \in M$ may be connected to input variables of a block $w \in M$. Let

$G_{\mathsf{V}} \triangleq (V_{\mathsf{V}}, E_{\mathsf{V}})$ be a directed graph where the vertices $V_{\mathsf{V}}$ are variables of blocks $v \in M$ and the edges specify the interconnection between output variables to input variables for some model $w \in$ *siblings*$(v)$, and we have $E_{\mathsf{V}} \subseteq \mathsf{V}(v) \times \mathsf{V}(w)$. In general, for a fixed block $v \in M$ and variable $x \in \mathsf{V}(v)$, this interconnection relation is a tree, rooted at the output variable $x$ and connected to possibly many input variables of other blocks $w \in M$ for $w \neq v$. For two blocks $v, w \in M$, we say $v$ *connects to* $w$ if there exists an output variable $y \in \mathsf{V}_O(v)$ and an input variable $u \in \mathsf{V}_I(w)$ with $E_{\mathsf{V}}(u) = y$, denoted $v \hookrightarrow w$. For two blocks $v, w \in M$, we say $v$ *has a path to* $w$ if $w$ is in the transitive closure of blocks that $v$ connects to (i.e., $v \hookrightarrow^* w$), denoted $v \rightsquigarrow w$. We note that the $\rightsquigarrow$ relation may have cycles, and such cases arise in feedback control loops. For a block $v \in M$, for an input variable $u \in \mathsf{V}_I(v)$ and output variable $y \in \mathsf{V}_O(v)$, we say $u$ *directly influences* $y$ if the value of $y$ changes as a function of $u$.[1] Finally, for two blocks $v, w \in M$ such that $v \rightsquigarrow w$, for an output variable $y \in \mathsf{V}_O(v)$ and an input variable $u \in \mathsf{V}_I(w)$, we say $y$ *influences* $u$ if there exists a sequence of directly influenced variables between $y$ and $u$. Thus, we can see that a cyber variable in one model may influence a physical variable in another model (or even its own model if there is a cycle), and vice-versa. The *software physical variables* are all cyber variables that are influenced by physical variables, and are denoted $\mathsf{V}_{SP}$. Typical examples of software physical variables include those used for sensed and sampled measurements, variables used in feedback control calculations, etc.

**Example 4** *Here, we describe a CPS case study used throughout the remainder of the chapter for illustrating concepts. The case study is a DC-to-DC power converter (like buck,*

[1]Internally the blocks may be very sophisticated, could represent complex physical systems, could be Turing complete, etc., so we use this abstract notion.

*boost, and buck-boost converters) [115], all of which have similar modeling, but we focus particularly on a buck converter. The buck converter has two real-valued state variables modeling the inductor current $i_L$ and the capacitor voltage $V_C$. These state variables are written in vector form as: $x = [i_L; V_C]$. The dynamics of the continuous variables in each mode $m \in \{Open, Close, DCM\}$ are specified as linear (affine) differential equations: $\dot{x} = A_m x + B_m u$, where $u = V_S$ is a source voltage. The $A_m$ matrices consist of $L > 0$, $R > 0$, $C > 0$ real-valued constants, respectively representing inductance (in Henries), resistance (in Ohms), and capacitance (in Farads). A buck converter takes an input voltage of say $5\,V$ and "bucks" or drops the voltage to some lower DC voltage, say $2.5\,V$. These circuits are used in many electronic devices (e.g., personal computers, cellphones, embedded systems, aircraft, satellites, cars). These circuits are also used as modular components in a variety of novel power electronics architectures, such as AC/DC microgrids and distributed DC-to-AC multilevel inverters [120].*

*The general architecture of the buck converter that we focus on consists of a plant (physical system) model and a controller (cyber model/software), along with models of actuators and sensors interfacing the plant and controller. A controller for the buck converter may be constructed as a hysteresis controller, which changes the mode of the buck converter plant based on the measured output voltage [72]. In fact, the converter is meant to transform a given source voltage $V_S$ to create an output voltage $V_{out}$ approximately equal to a desired reference voltage (or set-point) $V_{ref}$. To accomplish this, the switch controlling whether $V_S$ is connected to the output or not is toggled depending on whether $V_{out} > V_{ref}$ or $V_{out} < V_{ref}$. In practice, to avoid switching too often, a hysteresis band is used and switches occur when $V_{out} > V_{ref} + V_{tol}$ or $V_{out} < V_{ref} - V_{tol}$. The choice of $V_{tol}$, along with the system dynamics,* *will determine the voltage ripple $V_{rip}$ about the set-point $V_{ref}$. Typical specifications require the voltage ripple to be small, so that the output voltage $V_{out}$ is approximately $V_{ref}$, that is, $V_{rip}$ is chosen so that for $V_{out} = V_{ref} \pm V_{rip}$, we have $V_{out} \approx V_{ref}$. The sensor model performs quantization and sampling, as would occur in typical analog to digital conversion (ADC) used to digitize analog signal measurements. The actuator models likewise perform the inverse process of digital to analog conversion (DAC) to convert the digital (cyber) signals to analog signals.*

*Generally, we can model the plant as a physical block, the hysteresis controller as a cyber block, and the sensors and actuators as cyber-physical blocks in SLSF. The plant voltage is an output physical variable that affects the output cyber variable—a switching mode signal that enables the transition between each mode in the plant—of the controller, and vice-versa. This interaction between the plant and the controller is accomplished through the transitive closure of input-output connections with the sensor and the actuator in the SLSF model. We will formalize specifications and mismatches of the buck converter in Section 2.4. As a prelude, we highlight that Hynger finds its candidate invariant (that can be shown to be an actual invariant when modeled as a hybrid automaton [72, 79, 115]).*

### 2.3.1 Cyber-Physical Input-Output Automata

To further investigate cyber-physical specification mismatches of CPS models, we consider ones that may be formally represented as cyber-physical input-output automata.

**Definition 5** *A cyber-physical input-output automaton (CPIOA) $\tilde{\mathcal{A}}$ is a tuple, $\tilde{\mathcal{A}} \triangleq \langle$Loc, Var, Flow, Inv, Traj, Lab, Trans, Init$\rangle$, consisting of the following components:*

- *Loc: a finite set of discrete locations.*
- *Var: a finite set of $n$ continuous, real-valued variables, where $\forall x \in Var$, $val(x) \in \mathbb{R}$ and $val(x)$ is a valuation—a function mapping $x$ to a point in its type—here, $\mathbb{R}$; and $\mathcal{Q} \triangleq Loc \times \mathbb{R}^n$ is the state space. Var is the disjoint of a set of input variables $I$ and a set of output variables $O$. Furthermore, $C$ and $P$ classify Var into sets of cyber and physical variables, respectively.*
- *Inv: a finite set of invariants for each discrete location, $\forall \ell \in Loc$, $Inv(\ell) \subseteq \mathbb{R}^n$.*
- *Flow: a finite set of derivatives for each continuous variable $x \in Var$, and $Flow(\ell, x) \subseteq \mathbb{R}^n$ describes the continuous dynamics of each location $\ell \in Loc$. if $x$ is a physical variable, $Flow(\ell, x)$ is a non-zero Lipschitz continuous differential equation over time. Otherwise, if $x$ is a cyber variable, $Flow(\ell, x) = 0$.*
- *Traj: a finite set of continuous trajectory models the valuations of variables over an interval of real time $[0, T]$. Let $\Delta_0$, $\Delta_t$ and $\Delta_T$ be the valuations of variable $x$ at time points $0$, $t$, and $T$ respectively, $\forall t \in [0, T]$, $\forall x \in Var$, $\exists \ell \in Loc$, a trajectory $\tau \in Traj$ is a mapping function $\tau : [0, T] \rightarrow val(Var)$ such that:*
    - ◇ $\Delta_t = \Delta_0 + \int_{\delta=0}^{t} Flow(\ell, x) d\delta$, *and*
    - ◇ $\Delta_0 \models Inv(\ell)$, $\Delta_t \models Inv(\ell)$, *and* $\Delta_T \models Inv(\ell)$.
- *Lab: a finite set of synchronization labels.*
- *Trans: a finite set of transitions between locations; each transition is a tuple $\gamma \triangleq \langle \ell, \ell', g, u \rangle$, which can be taken from source location $\ell$ to destination location $\ell'$ when a*

*guard condition $g$ is satisfied, and the post-state is updated by an update map $u$.*

- *$Init$ is an initial condition, which consists of a set of locations in $Loc$ and a formula over $Var$, so that $Init \subseteq \mathcal{Q}$.*

Next, we define the semantics of a CPIOA $\tilde{\mathcal{A}}$ in term of executions. An execution of $\tilde{\mathcal{A}}$ is a sequence of states, written as $\rho \triangleq s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots$, where $s_0 \in Init$, and $s_i \rightarrow s_{i+1}$ is the update from the current-state $s_i$ to the post-state $s_{i+1}$, that is specified by the transition relations of the CPIOA $\tilde{\mathcal{A}}$ including: (a) a discrete transition that demonstrates the instantaneous state update, or (b) a continuous trajectory that represents the state update over a real time interval. We say a state $s_k$ is reachable from an initial state $s_0$ if there exists an execution $\rho \triangleq s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$.

**Invariant Property.** An *invariant property* $\varphi$ of a CPIOA $\tilde{\mathcal{A}}$ is a formula over $Var$ and $Loc$ that is always true for every reachable state of $\tilde{\mathcal{A}}$. Formally, we say $\tilde{\mathcal{A}} \models \varphi$ iff $\forall s \in Reach(\tilde{\mathcal{A}})$, $s \models \varphi$, where $Reach(\tilde{\mathcal{A}})$ denotes the set of reachable states of $\tilde{\mathcal{A}}$.

**Parallel Composition.** Consider two CPIOAs $\tilde{\mathcal{A}}_1 \triangleq \langle Loc_1, Var_1, Inv_1, Flow_1, Traj_1, Lab_1, Trans_1, Init_1\rangle$, and $\tilde{\mathcal{A}}_2 \triangleq \langle Loc_2, Var_2, Inv_2, Flow_2, Traj_2, Lab_2, Trans_2, Init_2\rangle$, we consider that $\tilde{\mathcal{A}}_1$ and $\tilde{\mathcal{A}}_2$ is *compatible* if $(a)$ $I_1 \subseteq O_2$, $(b)$ $I_2 \subseteq O_1$, and $(c)$ $O_1 \cap O_2 = \emptyset$. The parallel composition operation combines two compatible CPIOAs into a single CPIOA that represents the synchronous interaction between these two CPIOA when running simultaneously.

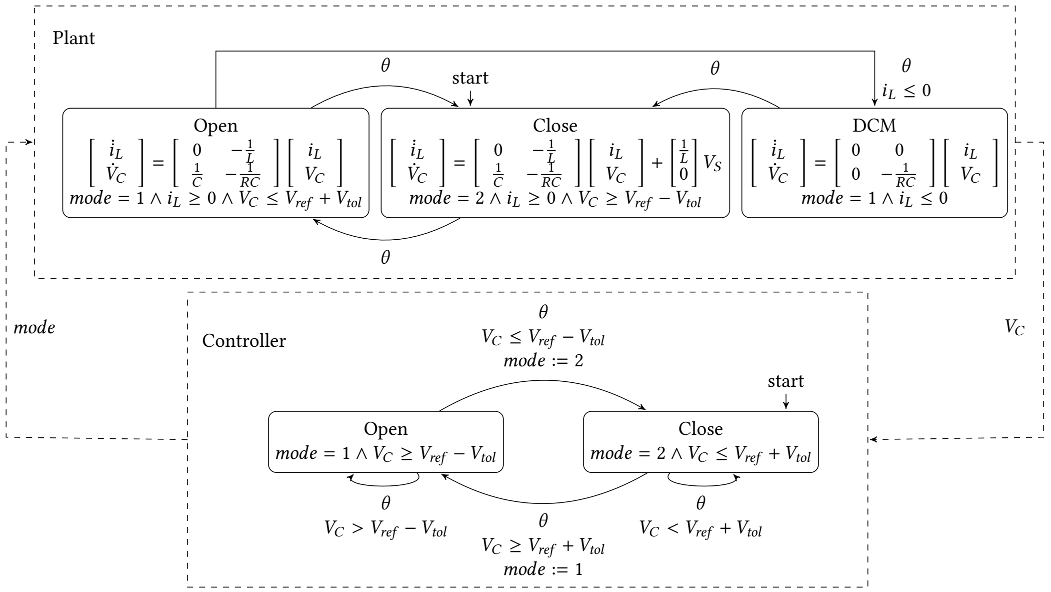
**Definition 6 (Parallel Composition)** *Given two compatible CPIOAs $\tilde{\mathcal{A}}_1$ and $\tilde{\mathcal{A}}_2$, the parallel composition of $\tilde{\mathcal{A}}_1$ and $\tilde{\mathcal{A}}_2$ is a CPIOA $\tilde{\mathcal{A}}$ , written as $\tilde{\mathcal{A}} \triangleq \tilde{\mathcal{A}}_1 \| \tilde{\mathcal{A}}_2$, where:*

- *$Loc = Loc_1 \times Loc_2$,*

- *$Var = Var_1 \cup Var_2$,*

- $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2$,
- $O = O_1 \cup O_2$,
- $I = (I_1 \cup I_2) \setminus O$,
- $\forall \ell_1, \ell_2 \in Loc$, $Inv(\ell_1, \ell_2) = Inv_1(\ell_1) \wedge Inv_2(\ell_2)$
- $\forall \ell_1, \ell_2 \in Loc$, $\forall x \in Var$, $((\ell_1, \ell_2), val(x) \in Init)$ *iff* $(\ell_1, val(x)) \in Init_1 \wedge (\ell_2, val(x)) \in Init_2$,
- $Lab = Lab_1 \cup Lab_2$,
- $\forall i \in \{1, 2\}$, *there is a trajectory* $\tau \in Traj$ *iff* $\tau \downarrow (Loc_i \cup Var_i) \in Traj_i$, *where* $\tau \downarrow (Loc_i \cup Var_i)$ *denotes the projection of* $\tau$ *onto the sets of variables and locations of component* $i$.
- *Given* $\gamma_1 \in Trans_1$, $\gamma_1 \triangleq \langle \ell_1, \ell_1', g_1, u_1 \rangle$ *and* $\gamma_2 \in Trans_2$, $\gamma_2 \triangleq \langle \ell_2, \ell_2', g_2, u_2 \rangle$, *there exists a transition* $\gamma \in Trans$, $\gamma \triangleq \langle \ell, \ell', g, u \rangle$ *iff:*
  - ◇ $\ell = (\ell_1, \ell_2)$, $\ell' = (\ell_1', \ell_2)$, $g = g_1$, *and* $u = u_1$, *or*
  - ◇ $\ell = (\ell_1, \ell_2)$, $\ell' = (\ell_1, \ell_2')$, $g = g_2$, *and* $u = u_2$, *or*
  - ◇ $\ell = (\ell_1, \ell_2)$, $\ell' = (\ell_1', \ell_2')$, $g = g_1 \wedge g_2$, *and* $u = u_1 \cup u_2$.

**Closed-loop CPIOA.** One type of CPS model that we focus on in this chapter is a closed-loop model, e.g., the closed-loop buck converter. Such a model can be formally represented as a closed-loop CPIOA, which is a parallel composition of a plant and controller CPIOA. The plant CPIOA has continuous dynamics modeled by ordinary differential equations, and the



Plant
$\theta$
start
$\theta$
$\theta$
$i_L \leq 0$
Open
$\begin{bmatrix} \dot{i}_L \\ \dot{V}_C \end{bmatrix} = \begin{bmatrix} 0 & -\frac{1}{L} \\ \frac{1}{C} & -\frac{1}{RC} \end{bmatrix} \begin{bmatrix} i_L \\ V_C \end{bmatrix}$
$mode = 1 \wedge i_L \geq 0 \wedge V_C \leq V_{ref} + V_{tol}$
Close
$\begin{bmatrix} \dot{i}_L \\ \dot{V}_C \end{bmatrix} = \begin{bmatrix} 0 & -\frac{1}{L} \\ \frac{1}{C} & -\frac{1}{RC} \end{bmatrix} \begin{bmatrix} i_L \\ V_C \end{bmatrix} + \begin{bmatrix} \frac{1}{L} \\ 0 \end{bmatrix} V_S$
$mode = 2 \wedge i_L \geq 0 \wedge V_C \geq V_{ref} - V_{tol}$
DCM
$\begin{bmatrix} \dot{i}_L \\ \dot{V}_C \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & -\frac{1}{RC} \end{bmatrix} \begin{bmatrix} i_L \\ V_C \end{bmatrix}$
$mode = 1 \wedge i_L \leq 0$
$\theta$
$mode$
Controller
$\theta$
$V_C \leq V_{ref} - V_{tol}$
$mode := 2$
$V_C$
start
Open
$mode = 1 \wedge V_C \geq V_{ref} - V_{tol}$
Close
$mode = 2 \wedge V_C \leq V_{ref} + V_{tol}$
$\theta$
$V_C > V_{ref} - V_{tol}$
$\theta$
$V_C \geq V_{ref} + V_{tol}$
$mode := 1$
$\theta$
$V_C < V_{ref} + V_{tol}$

**Figure 2.3:** A hybrid automaton models the buck converter plant with hysteresis controller.

controller CPIOA can be purely discrete. For instance, the hybrid automaton of the closed-loop buck converter (without sensor and actuator) shown in Figure 2.3 can be considered as one closed-loop CPIOA, where $\theta$ is a synchronization label and *mode* is a discrete control signal. The capacitor voltage variable $V_C$ is not only an output physical variable for the plant CPIOA, but is also an input cyber variable of the controller CPIOA. In this case, we can check whether the candidate invariants of the closed-loop buck converter found with Hynger and Daikon are actual invariants by investigating its formal model (e.g., a closed-loop CPIOA shown in Figure 2.3) using some hybrid systems model checkers such as SpaceEx [60].

## 2.3.2 Candidate Invariant Checking Problem

The formal definition of the candidate invariant checking problem for CPS is described as follows.

**Definition 7 (Candidate Invariant Checking)** *Given a CPS model $\mathcal{A}$ with a set of can-*

*didate invariants $\hat{\Phi}$, $\tilde{\mathcal{A}}$ is a formal model converted from $\mathcal{A}$, a candidate invariant $\hat{\varphi} \in \hat{\Phi}$ is considered as an actually invariant property of $\tilde{\mathcal{A}}$ iff Reach$(\tilde{\mathcal{A}}) \models \hat{\varphi}$.*

According to Definition 7, if a CPS model $\mathcal{A}$ is a white box system that can be represented in terms of a formal model such as a CPIOA $\tilde{\mathcal{A}}$, a hybrid system model checker may be used to check whether $\hat{\varphi}$ is an invariant property of $\tilde{\mathcal{A}}$. If there exists any reachable state of $\tilde{\mathcal{A}}$ that does not satisfy $\hat{\varphi}$, we can conclude that $\hat{\varphi}$ is not an actual invariant of the CPS model $\mathcal{A}$.

## 2.4 Cyber-Physical Specifications and Mismatches

In this section, we will formalize the concept of candidate cyber-physical specification mismatches of CPS, and introduce a potential method to detect such specification mismatches.

### 2.4.1 Cyber-Physical Specifications

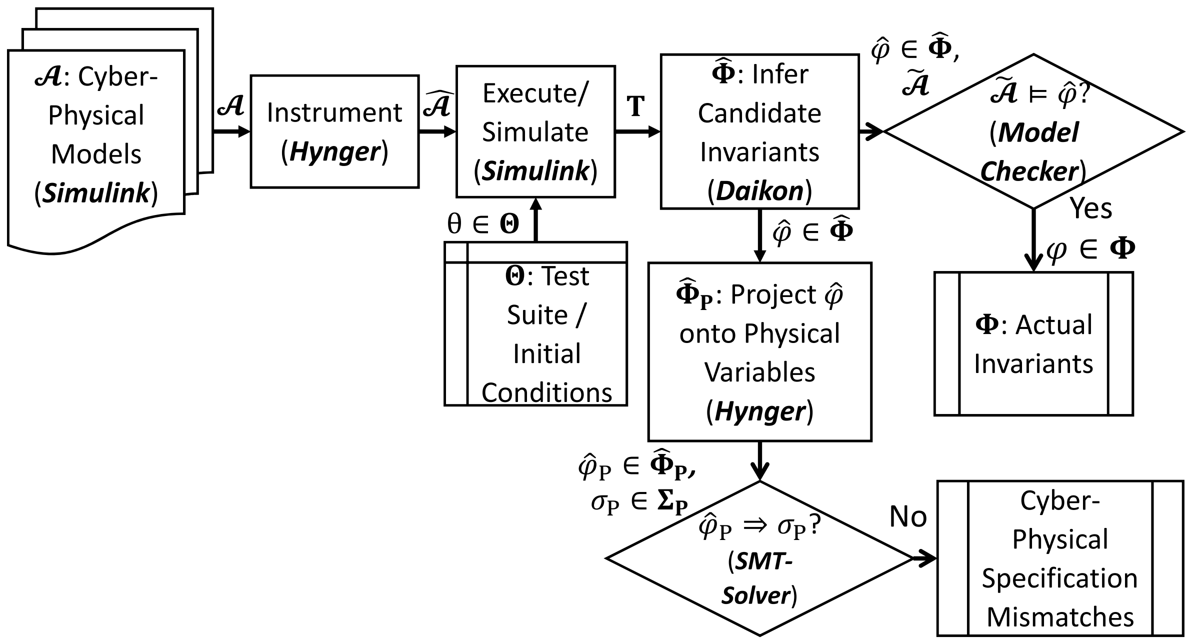
Our goal is to find specifications that are invariants or conditional invariants, so we do not consider more general temporal logic formulas. Under this assumption, a *specification* is equivalent to a predicate over the state-space of the system. Equivalently, a specification is a multi-sorted first-order logic (FOL) sentence (of a restricted class), so we assume the specification may be represented in the Satisfiability Modulo Theories (SMT) library standard language [20, 108]. Under these assumptions, candidate invariants may be specified as quantifier-free SMT formulas over the variables of the SLSF model, where the type of a variable corresponds to the SMT sort. For a formula $\phi$, let *vars*$(\phi)$ be the set of variables appearing in $\phi$. For a formula $\phi$: $(a)$ if *vars*$(\phi)$ are all physical, then $\phi$ is a *physical specifi-*

*cation*, $(b)$ if $vars(\phi)$ are all cyber, then $\phi$ is a *cyber specification*, and $(c)$ if $vars(\phi)$ consists of both cyber and physical variables, then $\phi$ is a *cyber-physical specification.*

Next, while we will try to infer interesting specifications $\phi$ using dynamic analysis later in the chapter, we first highlight examples of specifications made a priori in system design, as these are necessary to define specification mismatches. Let $\Sigma$ be a set of specifications for $\mathcal{A}$, which is a set of formulas over the variables of $\mathcal{A}$. Referring to Figure 2.4, we separate the specification $\Sigma$ into sets of cyber and physical specifications, written respectively as $\Sigma_C$ and $\Sigma_P$. These specifications include assumptions about the physical environment, such as the value of gravitational force, temperature bounds, time constants, etc. The physical specification also includes assumptions about the physical system's behavior and subcomponents, such as motor torque limits, temperature bounds of components, sampling rates, velocity limits, etc. Here $\Sigma_C$ denotes the set of cyber specifications. The cyber specifications include assumptions about software-physical interfaces, such as ADC resolution, DAC resolution, sampling rates, etc. It also includes assumptions about the software system, subcomponents, and software-software interfaces, such as data formats, control flow, event orderings, etc. For example, the buck converter has the following physical specifications:

$$\begin{aligned}
\sigma_P^1 &\triangleq t \geq t_s \Rightarrow V_{out}(t) = V_{ref}(t) \pm V_{rip}, \\
\sigma_P^2 &\triangleq V_S(t) = V_S(0) \pm \delta_S, \\
\sigma_P^3 &\triangleq V_{ref}(t) = V_{ref}(0) \pm \delta_{ref},
\end{aligned}$$

and $\Sigma_P \triangleq \{\sigma_P^1, \sigma_P^2, \sigma_P^3\}$. Here, $\sigma_P^1$ states that after some amount of constant startup time $t_s$, the output of the buck converter $V_{out}(t)$ remains near a reference (desired) output voltage $V_{ref}(t)$. Both $\sigma_P^2$ and $\sigma_P^3$ specify assumptions about the buck converter's environment, namely



$\mathcal{A}$: Cyber-Physical Models (*Simulink*)
$\mathcal{A}$
Instrument (*Hynger*)
$\widehat{\mathcal{A}}$
Execute/ Simulate (*Simulink*)
$\mathbf{T}$
$\widehat{\boldsymbol{\Phi}}$: Infer Candidate Invariants (*Daikon*)
$\hat{\varphi} \in \widehat{\boldsymbol{\Phi}}$, $\widetilde{\mathcal{A}}$
$\widetilde{\mathcal{A}} \vDash \hat{\varphi}$? (*Model Checker*)
Yes
$\varphi \in \boldsymbol{\Phi}$
$\boldsymbol{\Phi}$: Actual Invariants
$\theta \in \boldsymbol{\Theta}$
$\boldsymbol{\Theta}$: Test Suite / Initial Conditions
$\hat{\varphi} \in \widehat{\boldsymbol{\Phi}}$
$\widehat{\boldsymbol{\Phi}}_{\mathbf{P}}$: Project $\hat{\varphi}$ onto Physical Variables (*Hynger*)
$\hat{\varphi}_{\mathrm{P}} \in \widehat{\boldsymbol{\Phi}}_{\mathbf{P}}$, $\sigma_{\mathrm{P}} \in \boldsymbol{\Sigma}_{\mathbf{P}}$
$\hat{\varphi}_{\mathrm{P}} \Rightarrow \sigma_{\mathrm{P}}$? (*SMT-Solver*)
No
Cyber-Physical Specification Mismatches

**Figure 2.4:** Hynger overview, inference of physical specifications assumed by software, and cyber-physical specification mismatch identification.

that its source voltage $V_S$ and reference voltage $V_{ref}$ always remain near their initial values. We note that while time may not typically be thought of as a state of the system, it can be encoded in this way easily, for example, by including a state variable $t$ with $\dot{t} = 1$. To evaluate whether $\mathcal{A}$ has cyber-physical specification mismatches, we hypothesize that the cyber specification contains (at least a subset) of the physical specification. This process is made more explicit in Figure 2.4 and described next.

### 2.4.2 Cyber-Physical Specification Mismatches

A CPS model or implementation will be provided as an SLSF diagram, denoted $\mathcal{A}$ as formalized above. Next, $\mathcal{A}$ is instrumented using the Hynger yielding a modified SLSF diagram $\hat{\mathcal{A}}$. Now, $\hat{\mathcal{A}}$ is executed to generate a set of sampled, finite-precision traces T for each initial condition $\theta$ in a set of initial conditions $\Theta$, which effectively corresponds to a test suite. The traces T are analyzed using dynamic analysis methods, such as Daikon, to generate a set of candidate invariants $\hat{\Phi}$, each element $\hat{\varphi}$ of which may be checked as actual invariants if $\mathcal{A}$ corresponds to a formal model (e.g., a CPIOA) or may be converted to one, $\tilde{\mathcal{A}}$. If that is the case, then a hybrid system model checker may be employed to see if $\hat{\varphi}$ is an actual invariant $\varphi$, and the set of actual invariants $\Phi$ is collected.

**Definition 8 (Cyber-Physical Specification Mismatch)** *Given an SLSF diagram $\mathcal{A}$ with a set of actual physical specifications $\Sigma_P$, let $\hat{\Phi}_P \triangleq \hat{\Phi} \downarrow \mathsf{V}_{SP}$ be a set of candidate physical invariant, $\mathcal{A}$ has a cyber-physical specification mismatch iff: $\exists \sigma_P \in \Sigma_P, \forall \hat{\varphi}_P \in \hat{\Phi}_P, \sigma_P \not\models \hat{\varphi}_P$.*

In Definition 8, $\hat{\Phi} \downarrow \mathsf{V}_{SP}$ denotes the projection or the restriction of $\hat{\Phi}$ to the set of software physical variable $\mathsf{V}_{SP}$. In all cases, each candidate invariant $\hat{\varphi} \in \hat{\Phi}$ is projected (restricted) onto the software physical variables $\mathsf{V}_{SP}$ to yield a candidate physical invariant $\hat{\varphi}_P$ and corresponding set $\hat{\Phi}_P$. Such a projection may be computed using quantifier elimination methods available in many modern SMT solvers, such as Z3 [43][2]. Now, $\hat{\Phi}_P$ corresponds to the candidate, inferred physical invariants from the perspective of the cyber-physical system, each element of which may be compared to each element $\sigma_P$ of a set of actual physical specifications $\Sigma_P$. Since $\hat{\varphi}_P$ and $\sigma_P$ are both formulas, we construct new formulas $\hat{\varphi}_P \Rightarrow \sigma_P$ and $\sigma_P \Rightarrow \hat{\varphi}_P$, each of which may be discharged with an SMT solver. If these checks are not valid, then these specifications are candidate *cyber-physical mismatches*. These checks basically compare whether the inferred specification and actual specification are more

[2]Z3 may be downloaded: http://z3.codeplex.com/.

or less restrictive than one another, in terms of the sizes of corresponding sets of states satisfying the predicates. We hypothesize that it is generally the case that the inferred physical specification should always be stronger than the actual physical specification, and only the check $\hat{\varphi}_P \Rightarrow \sigma_P$ would be needed. This would correspond to the case where the software's assumptions about the physical world are *at least as* restrictive as those made in the actual physical specification. For instance, suppose that the physical specification of the output voltage of the buck converter is $\sigma_P \triangleq\ t \geq t_s \Rightarrow 4.8V \leq V_{out}(t) \leq 5.2V$, and the candidate physical invariant is $\hat{\varphi}_P \triangleq\ t \geq t_s \Rightarrow 4.9V \leq V_{out}(t) \leq 5.1V$, then the check of the formula $\hat{\varphi}_P \Rightarrow \sigma_P$ using an SMT solver like Z3 will indicate that the system does not have a specification mismatch. Otherwise, if the candidate physical invariant is $\hat{\varphi}_P \triangleq\ t \geq t_s \Rightarrow 4.7V \leq V_{out}(t) \leq 5.0V$, then the check of the formula $\hat{\varphi}_P \Rightarrow \sigma_P$ will indicate that the system has a specification mismatch. On the other hand, it may also be useful to check $\hat{\varphi}_P \Leftarrow \sigma_P$, which would correspond to cases where the inferred physical specification is weaker than the actual physical specification. In this case, there may be a trace that violates the actual specification, and this may be useful in analysis like falsification to drive simulations towards a violating behavior.

## 2.5 Hynger: Generating Invariants for SLSF Models

Hynger—HYbrid iNvariant GEneratoR—is a software tool developed for invariant inference of CPS models represented as SLSF block diagrams[3]. Hynger is written primarily in

[3]A prototype of Hynger with examples is available online at: https://bitbucket.org/verivital/hynger. The repository also includes Daikon input (`*.dtrace`) trace files generated from the examples, as well as the Daikon output candidate invariant (`*.inv`) files.

Matlab and uses the Matlab APIs to interact with SLSF diagrams. Hynger also uses some Java code (natively inside Matlab) to interface with Daikon, which is written in Java. Daikon versions 5.0.0 to 5.1.8 were tested with Hynger[4].

Given an SLSF model $\mathcal{A}$, Hynger automatically inserts callback functions into the model to print model variables at block inputs and outputs at certain events in the SLSF simulation loop. Consequently, a trace file generated by Hynger will then be formatted in the trace input format required by Daikon. While configurable, the default behavior of Hynger is to add instrumentation (observation) points for every input and output signal for every block (recursively) in the SLSF diagram. That is, Hynger walks the tree of blocks starting from the root, and for each $v \in M$, adds instrumentation points for the input variables $\mathsf{V}_I(v)$ and the output variables $\mathsf{V}_O(v)$ of $v$. Of course, this may incur a drastic performance overhead, so if this is not desired, the user may select only a subset of the blocks to instrument and our performance results (see Section 2.6) illustrate this distinction. When an SLSF model is simulated with these instrumentation callback functions added by Hynger, it will generate a trace file in the input trace format for Daikon. Hynger also provides the capability to automatically call Daikon from Matlab (by using an appropriate Java call to Daikon), which will then return the set of candidate invariants from each program point to the user.

The Hynger flow is summarized in Figure 2.4. The inputs are: $(a)$ SLSF diagrams (containing embedded software code and a set of physical variables along with their physical dynamics models [e.g., ODEs]), and $(b)$ a set of physical variables along with their dynamics models (specified as SLSF children diagrams), and $(c)$ a test suite for the embedded software and initial conditions for the physical simulation (such as noisy initial conditions, $\theta \in \Theta$).

[4]Daikon may be downloaded: http://plse.cs.washington.edu/daikon/.

```
/*@ requires n >= 0; // at least 0 elements
  @ requires \valid(b+ (0..n-1)); // all elements exist
  @ assigns \nothing; // no side effects
  @ ensures \result == \sum(0,n-1,\lambda integer j; b[j]);
  @ ensures \result >= 0; // false , array may be negative
*/
int sum_array(int b[], unsigned int n) {
    int i;
    int s = 0;
    /*@ loop invariant
        \forall integer j; (0 <= i <= n) ==> s == \sum(0,i-1,\lambda integer j; b[j]); */
    for (i = 0; i < n; i++) {
            s += b[i];
    }
    return s;
  }
```

**Figure 2.5:** Example C function that sums an array `b` of `n` integers. Requirements on the function inputs (i.e., preconditions on `b` and `n` for the function to be called) are specified as `requires` assertions in the ACSL language. Correctness specifications (i.e., postconditions following the function call) are specified as `ensures` assertions in the ACSL language.

The output of the Hynger tool is a set of candidate invariants, which, when projected onto all the software physical variables $\mathsf{V}_{SP}$, represent a candidate specification the software assumes for the physical parts of the system. Finally, candidate specifications can be checked for conformance with the actual physical requirements by comparing the two specifications: the actual physical specification and the candidate physical specification from the software perspective.

```
============== Precondition
..sum_array():::ENTER
b has only one value // it's a pointer to only one location of memory
b[] elements >= 0 // all elements were non-negative for this set of traces
n == 100 // all tests were 100 element arrays for this set of traces
size(b[]) == 100 // all tests were 100 element arrays
============== Postcondition
..sum_array():::EXIT
b[] == orig(b[]) // no side effects
return == sum(b[]) // does return the sum
sum(b[]) == sum(orig(b[]))
b[] elements >= 0
```

**Figure 2.6:** Daikon candidate invariant output (with some additional markup in C-style comments for readability) for the `sum_array` example from Figure 2.5.

### 2.5.1 Dynamic Invariant Inference with Daikon

Next, we illustrate the dynamic invariant inference methodology used by Daikon on a pure software example. However, this pure software example (a C function) is actually specified for the controller in the buck converter case study (shown in Figure 2.7) in a different manner. The loop in the controller SLSF model of Figure 2.9 also computes a sum of an array, and Daikon can find this specification for both the SLSF controller model using Hynger, and the C-frontend for the following example. Note that, in Figure 2.9 the digitized output voltage from the buck-converter plant is used to determine the mode of the switch. Here, $V_{tol}$ is denoted by the variable `Vtol`, $V_{ref}$ is `Vref`. We highlight that the controller computes a moving average by summing an array. With Hynger and Daikon, we automatically infer that the result of this is the sum of the samples, similar to the sum return specification shown in Figure 2.6 found for the C function in Figure 2.5.

*Example C Program, Formal Specification, and Candidate Invariants Inferred.* Figure 2.5 shows an example C function to illustrate the use of dynamic analysis with Daikon to find candidate invariants. The function computes and returns the sum of an array of integers. This example was recreated from an example in the original Daikon paper [55]. Additionally, a formalized correctness specification is given in the modern ANSI/ISO C Specification Language (ACSL), used by tools such as Frama-C [41]. Using Daikon and a small suite of unit tests, we were able to successfully find the invariant that returns from the function `sum_array`, the returned value is the sum of the elements in the array `b`. The suite of tests included arrays with: $(a)$ all the same length and same elements, $(b)$ all the same length and uniformly randomly chosen elements, $(c)$ different lengths and all the same elements, and $(d)$ different lengths and uniformly randomly chosen elements. Daikon successfully found the sum postcondition in all these cases with only a few test conditions. The candidate invariant outputs of Daikon appear in Figure 2.6, where we can see Daikon has inferred a candidate invariant that the function returns the sum of an array. We highlight that we find the sum return result of the moving average filter from Figure 2.9 using Hynger and Daikon.

## 2.6 Experimental Results

Hynger was tested on Windows 10 64-bit using Matlab 2016b, and 2017a, executed on a x86-64 laptop with a 2.3 GHz dual-core Intel i5-6200U processor and 12 GB RAM. All performance metrics reported were recorded on this system using Matlab 2017a. We tested and evaluated Hynger using a number of SLSF examples, including: *(a)* the closed-loop buck converter with sensor and hysteresis controller described in Section 2.6.1 and detailed further in [115], *(b)* a solar array case study that uses a buck-boost converter [120], *(c)* benchmarks from S-TaLiRo [14], *(d)* benchmarks from Breach [48,75], *(e)* benchmarks created as a part of the ARCH 2014 CPSWeek workshop (particularly [74,115]) and *(f)* example models provided by Mathworks. Overall, these examples vary from fairly simple with tens of blocks (such as the buck converter case study we detail), to complex (with hundreds of blocks). *Runtime Overhead from Instrumentation with Hynger and Invariant Inference with Daikon.* First, we present an aggregate performance evaluation for some of these examples in Table 2.1, with column descriptions appearing in the caption. Overall, the performance overhead of instrumenting diagrams and performing invariant inference is around an order of magnitude increase in the best cases, and two-to-three orders of magnitude increase in the worst cases, which we note is comparable with typical Daikon instrumentation frontends like Valgrind's overhead [56, 112]. We conducted performance profiling of Hynger and identified the main source of overhead (about 75 to 90 percent) as file I/O operations. Additionally, as Hynger has several different usage scenarios and operating modes (where it may be used to instrument few blocks [subsystem and function blocks by default], many blocks [all blocks except ones such as constants, scopes, etc.], every single block, or user-selected blocks), the table illustrates these differences to give some comparison of how the methods scale on a given model. Next, we will describe two CPS case studies in details to evaluate the capability of Hynger in detecting cyber-physical specification mismatches. The first model is the closed-loop buck converter that has been used to illustrate the concepts of this chapter, and the second model is derived from a collection of the automotive powertrain control benchmarks proposed by Toyota [75].

| Model | Solver | Tmax | Sim | SimInst | Inv | Overhead | BDAll | BDInst | BDPct |
|---|---|---|---|---|---|---|---|---|---|
| buck (Section 2.6.1) | ode45 | 0.0083 | 6.2985 | 38.4518 | 5.7335 | 7.0152 | 14 | 3 | 21.4286 |
| buck (Section 2.6.1) | ode45 | 0.0083 | 6.4567 | 44.698 | 7.0913 | 8.021 | 14 | 4 | 28.5714 |
| buck (Section 2.6.1) | ode45 | 0.0083 | 6.5301 | 78.3176 | 7.2224 | 13.0993 | 14 | 14 | 100 |
| heat25830 [14] | ode45 | 50 | 4.6913 | 254.5776 | 14.09 | 57.2692 | 28 | 1 | 3.5714 |
| heat25830 [14] | ode45 | 50 | 4.7328 | 2882.7808 | 15.6488 | 612.4233 | 28 | 10 | 35.7143 |
| fuel1 [74] | ode15s | 15 | 5.3747 | 976.6274 | 7.923 | 183.182 | 208 | 17 | 8.1731 |
| fuel1 [74] | ode15s | 15 | 4.2131 | 2824.2804 | 11.604 | 673.1137 | 208 | 63 | 30.2885 |
| fuel2 [74] | ode15s | 20 | 3.3838 | 36.8312 | 2.9881 | 11.7674 | 25 | 6 | 24 |
| fuel2 [74] | ode15s | 20 | 2.7353 | 42.4074 | 3.2771 | 16.7018 | 25 | 13 | 52 |
| fuel3 [58] | ode15s | 20 | 3.7425 | 292.9976 | 4.1131 | 79.3892 | 90 | 11 | 12.2222 |
| fuel3 [58] | ode15s | 20 | 3.6083 | 945.3992 | 4.3904 | 263.2236 | 90 | 46 | 51.1111 |

**Table 2.1:** Hynger performance results for several of the examples evaluated. Solver is the ODE solver used by SLSF. Tmax is the virtual simulation time in seconds (i.e., time from the perspective of the model). All runtime results are in seconds and are the mean of 20 runs. Sim is the simulation runtime ($s$). Inv is the invariant generation runtime (Daikon) ($s$). Overhead is the overall relative performance overhead (extra runtime) ($\times$) using Hynger and Daikon versus only SLSF simulation (i.e., $((SimInst + Inv)/Sim)$). BDInst and BDAll are the numbers of block diagrams instrumented and the overall number of block diagrams, respectively. BDPct is the percentage (%) of block diagrams instrumented using different Hynger modes of operation (i.e., $BDInst/BDAll$).

### 2.6.1 Closed-Loop Buck Converter Cyber-Physical Specification Mismatch

A basic cyber-physical specification mismatch is easy to encode in the buck converter, since the software controller inherently uses a tolerance to encode the desired output voltage ripple. This hysteresis tolerance band is typically chosen based on the system dynamics and desired output voltage ripple to ensure the output voltage meets the ripple specification. As a concrete example, the physical specification may contain a fixed constraint that $V_{out} = V_{ref} \pm V_{rip}$, e.g., $V_{ref} = 5V$ and $V_{rip} = 0.1V$. The hysteresis band $V_{tol}$ is then selected based on the system dynamics to ensure $4.9V \leq V_{out} \leq 5.1V$ so that it meets the requirements of the physical specifications defined by $\Sigma_P$ in Section 2.4.1.

*Sources of Cyber-Physical Specification Mismatches of the Closed-Loop Buck Converter.* There are different possibilities of specification mismatch that may occur to the closed-loop buck converter. We present three scenarios that result in specification mismatches. First, if the plant parameters change (i.e., different circuit elements are used), and the software is not updated with a new hysteresis band $V_{tol}$ to accommodate the changes in the plant dynamics, then a specification mismatch manifests. This mismatch can be detected using Hynger and the methodology described in this chapter. Of course, this is a somewhat obvious mismatch, as the controller relies on variables computed as functions of the plant parameters (here, the $R$, $L$, and $C$ values, as well as the source and desired/reference output voltage values). So if these plant components are changed, clearly the software must be updated. Second, the hysteresis controller is initially constructed using wrong information about the physical evolution of the plant. In fact, the hysteresis band $V_{tol}$ is far different from the actual output voltage ripples $V_{rip}$ of the plant. Third, the analog sensor of the buck converter may have ADC conversion errors that reduce the accuracy of the voltage measurement. These errors can be an offset error, a full-scale error, differential and integral non-linearity errors, etc. Moreover, a typical error that cannot be avoided in ADC sensor is the quantization error [141]. Overall, these conversion errors may cause a significant impact to result in system failures.

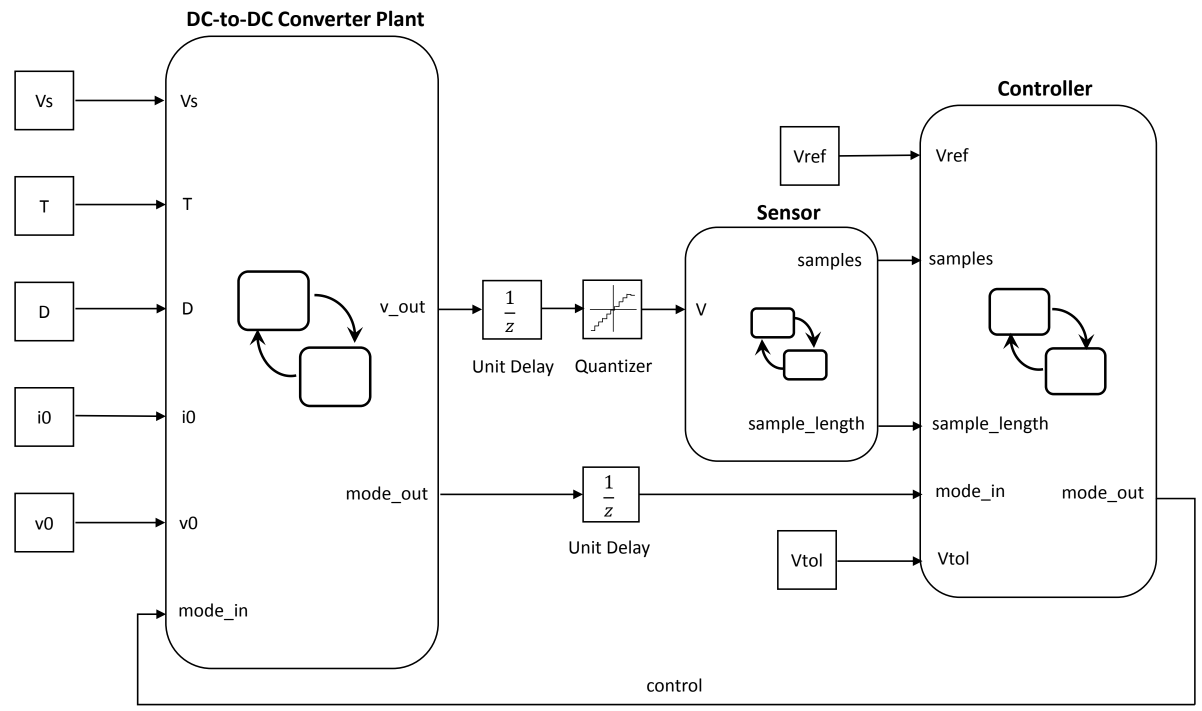


**Figure 2.7:** General CPS case study architecture overview of the buck converter in SLSF. The system is composed of a plant (physical system) model, a controller (software/cyber), and potentially sensor and actuator models. The cyber model uses some of the physical model output states to determine a control action or input. The controller in SLSF appears in Figure 2.9, and the sensor model appears in Figure 2.8. An example of this closed-loop buck converter including only plant and controller can be formally represented as the hybrid automaton in Figure 2.3.

*Experimental Results in Identifying Cyber-Physical Specification Mismatches of the Closed-Loop Buck Converter.* We consider the closed-loop buck converter $\mathcal{A}$ shown in Figure 2.7 with $V_S = 100$, $V_{ref} = 48V$, $V_{rip} = 5\%V_{ref} = 2.4V$, and assume that $\delta_S$, $\delta_{ref}$ are equal to zero. The physical specification of the output voltage is $\sigma_P \triangleq \; t \geq t_s \Rightarrow 45.6V \leq V_{out}(t) \leq 50.4V$. For the initial setup, with $R = 6\Omega$, $L = 2.65mH$, $C = 2.2mF$, and a sampling frequency $f_s = 60kHz$, the magnitude bound of the output voltage inferred from Hynger and Daikon is $\hat{\varphi}_P \triangleq \; t \geq t_s \Rightarrow 46.559V \leq V_{out}(t) \leq 50.203V$. Then, $\hat{\varphi}_P$ is considered as the candidate

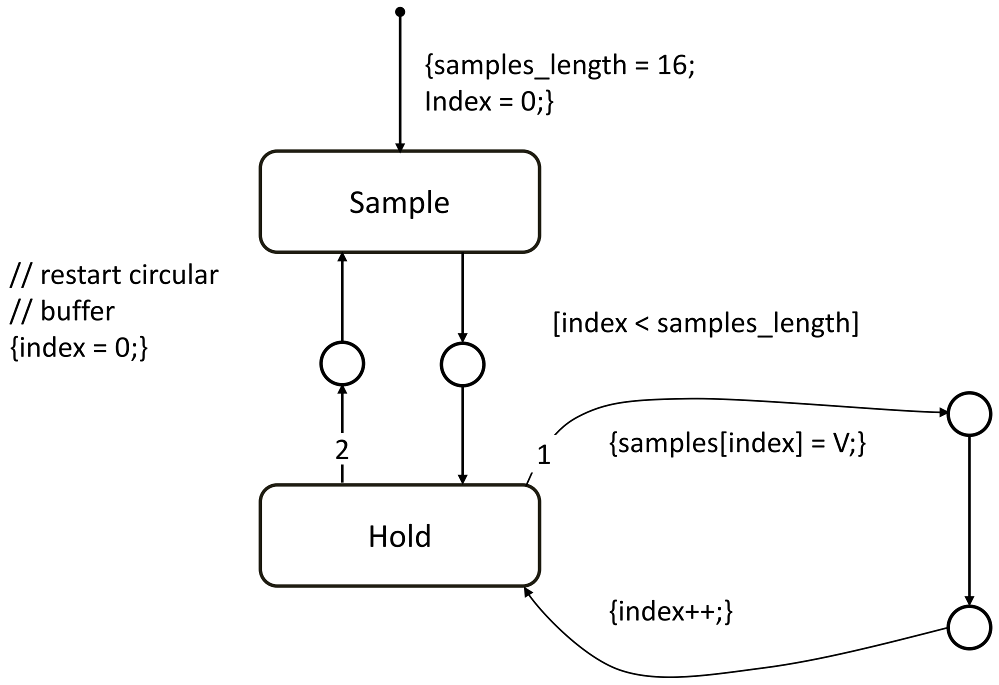


**Figure 2.8:** Stateflow model of sensor with a sample and hold for the buck converter case study.

invariant of the system since the formula $\hat{\varphi}_P \Rightarrow \sigma_P$ is true. Next, we investigate different possibilities of cyber-physical specification mismatches that may occur when changing the source voltage, the desired/reference output voltage, the sampling frequency, and the plant parameters of the buck converter.

First, we increase the source voltage $V_S$ from $100V$ to $120V$, the new magnitude bound of the output voltage inferred from Hynger and Daikon is $\hat{\varphi}_P \triangleq\ t \geq t_s \Rightarrow 46.804V \leq V_{out}(t) \leq 51.118V$. Then, the formula $\hat{\varphi}_P \Rightarrow \sigma_P$ is false, that indicates the system may have a cyber-physical specification mismatch.

Second, we drop the desired/reference output voltage $V_{ref}$ to $36V$. Thus, the physical specification of the output voltage becomes $\sigma'_P \triangleq\ t \geq t_s \Rightarrow 34.2V \leq V_{out}(t) \leq 37.8V$. In this case, the inferred physical specification of the output voltage from Hynger and Daikon becomes $\hat{\varphi}'_P \triangleq\ t \geq t_s \Rightarrow 35.068V \leq V_{out}(t) \leq 39.053V$, so that the formula $\hat{\varphi}'_P \Rightarrow \sigma'_P$ is

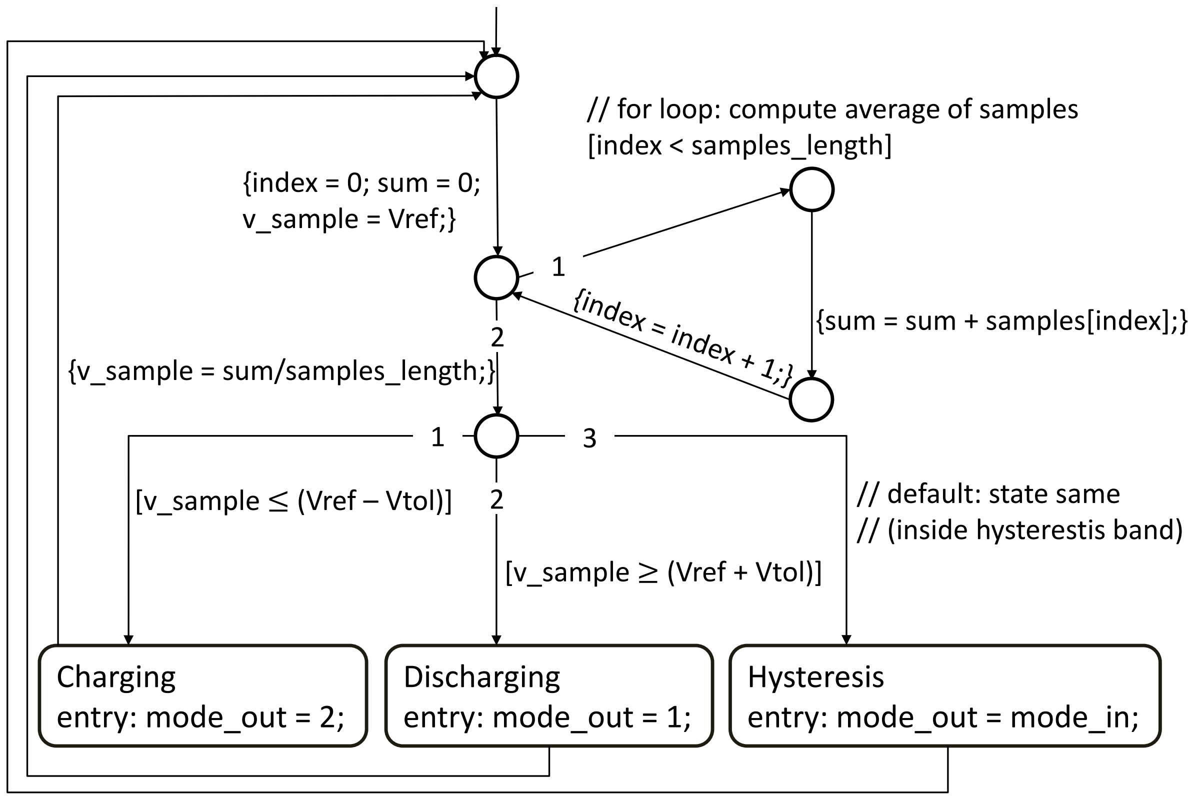

// for loop: compute average of samples
[index < samples_length]
{index = 0; sum = 0;
v_sample = Vref;}
1
{index = index + 1;}
{sum = sum + samples[index];}
2
{v_sample = sum/samples_length;}
1
3
2
[v_sample ≤ (Vref – Vtol)]
// default: state same
// (inside hysterestis band)
[v_sample ≥ (Vref + Vtol)]
Charging
entry: mode_out = 2;
Discharging
entry: mode_out = 1;
Hysteresis
entry: mode_out = mode_in;

**Figure 2.9:** Stateflow model of the buck-converter voltage hysteresis controller.

false. Therefore, changing the reference output voltage may also produce a cyber-physical specification mismatch for the buck converter.

Third, we decrease the sampling frequency $f_s$ from $60kHz$ to $30kHz$. As a result, the new inferred physical specification of the output voltage from Hynger and Daikon is $\hat{\varphi}_P \triangleq t \geq t_s \Rightarrow 45.853V \leq V_{out}(t) \leq 51.091V$. The check of the formula $\hat{\varphi}_P \Rightarrow \sigma_P$ will return false to indicate that the system may contain a cyber-physical specification mismatch.

Next, we keep the controller unchanged and vary the values of $R$, $L$, and $C$ to change the plant parameters. We then run the buck converter with Hynger in conjunction with Daikon, and collect candidate physical specifications associated with the output voltage. The comparison between the actual physical specification $\sigma_P$ and the physical specification $\hat{\varphi}_P$

inferred from Hynger and Daikon is shown in Table 2.2, and also illustrated in Figure 2.10. Note that in Table 2.2, $\hat{\varphi}_P$ describes the magnitude bound of the output voltage when $t \geq t_s$. The checks of the formula $\hat{\varphi}_P \Rightarrow \sigma_P$ occasionally return $False$, that are depicted in Figure 2.10 when the bound of the inferred output voltage overlaps its actual bound. This indicates that changing the plant parameters without updating the controller may produce cyber-physical specification mismatches. That also proves the capability of Hynger and our proposed methodology in automatically detecting a candidate cyber-physical specification mismatch of CPS.

Another possibility of the specification mismatch may occur when the controller is encoded based on wrong information about the plant. For the buck converter, the hysteresis controller is built with an assumption that the output voltage ripple $V_{rip}$ is equal to 5% of the reference voltage $V_{ref}$. However, the actual value of $V_{rip}$ may be much smaller than this assumption percentage. The percentage of the output voltage ripple of the buck converter is calculated as follows [54],

$$\frac{V_{rip}}{V_{ref}} = \frac{1-D}{8LCf_s^2}, \tag{2.1}$$

where $D = \frac{V_{ref}}{\eta \mathcal{V}_S}$ is a duty cycle, and $\eta$ is an efficiency coefficient of the converter. Here, with $L = 2.65mH$, $C = 2.2mF$, $f_s = 60kHz$, $\eta = 0.79$, $V_{ref} = 48V$, and $\mathcal{V}_S = 100V$, the percentage of the output voltage ripple is approximately equal to 0.0002%. Thus, the hypothesized output voltage ripple used to build the controller is far larger than the actual output voltage ripple calculated by Equation 2.1. It definitely shows that the system may have specification mismatches since the controller is encoded depending on wrong information about the physical plant.

| Parameter Values | $\hat{\varphi}_P$ | $\hat{\varphi}_P \Rightarrow \sigma_P$ | $\sigma_P \Rightarrow \hat{\varphi}_P$ |
|---|---|---|---|
| $R = 4\Omega$, $L = 2.65mH$, $C = 2.2mF$ | $45.137V \le V_{out}(t) \le 49.723V$ | *False* | *False* |
| $R = 8\Omega$, $L = 2.65mH$, $C = 2.2mF$ | $46.964V \le V_{out}(t) \le 50.405V$ | *False* | *False* |
| $R = 6\Omega$, $L = 0.65mH$, $C = 2.2mF$ | $47.141V \le V_{out}(t) \le 50.074V$ | *True* | *False* |
| $R = 6\Omega$, $L = 6.65mH$, $C = 2.2mF$ | $45.429V \le V_{out}(t) \le 50.439V$ | *False* | *True* |
| $R = 6\Omega$, $L = 2.65mH$, $C = 1.2mF$ | $45.426V \le V_{out}(t) \le 51.109V$ | *False* | *True* |
| $R = 6\Omega$, $L = 2.65mH$, $C = 3.2mF$ | $46.859V \le V_{out}(t) \le 49.774V$ | *True* | *False* |

**Table 2.2:** Experimental data showing the comparison between actual physical specifications and inferred physical invariants from Hynger and Daikon of the buck converter system. Here, the plant component is changed due to the changes of $R$, $L$, and $C$ values.

Furthermore, changing the length of voltage measurement array (samples_length) in the sensor of the buck converter (shown in Figure 2.8) may also cause a specification mismatch. For example, if we increase it from 16 to 32, the inferred physical specification using Hynger and Daikon becomes $\hat{\varphi}_P \triangleq\ t \ge t_s \Rightarrow 46.095V \le V_{out}(t) \le 50.788V$, which no longer implies the actual physical specification of the output voltage $\sigma_P \triangleq\ t \ge t_s \Rightarrow 45.6V \le V_{out}(t) \le 50.4V$.

## 2.6.2 Abstract Fuel Control System Benchmarks

In the second case study, we present the potential cyber-physical specification mismatches of the abstract fuel control (AFC) system benchmarks provided by Toyota [74,75], and further studied in [58]. The goal of these benchmarks is to determine the fuel rate that should be injected into the manifold to maintain the air-fuel ratio within a desirable range using the

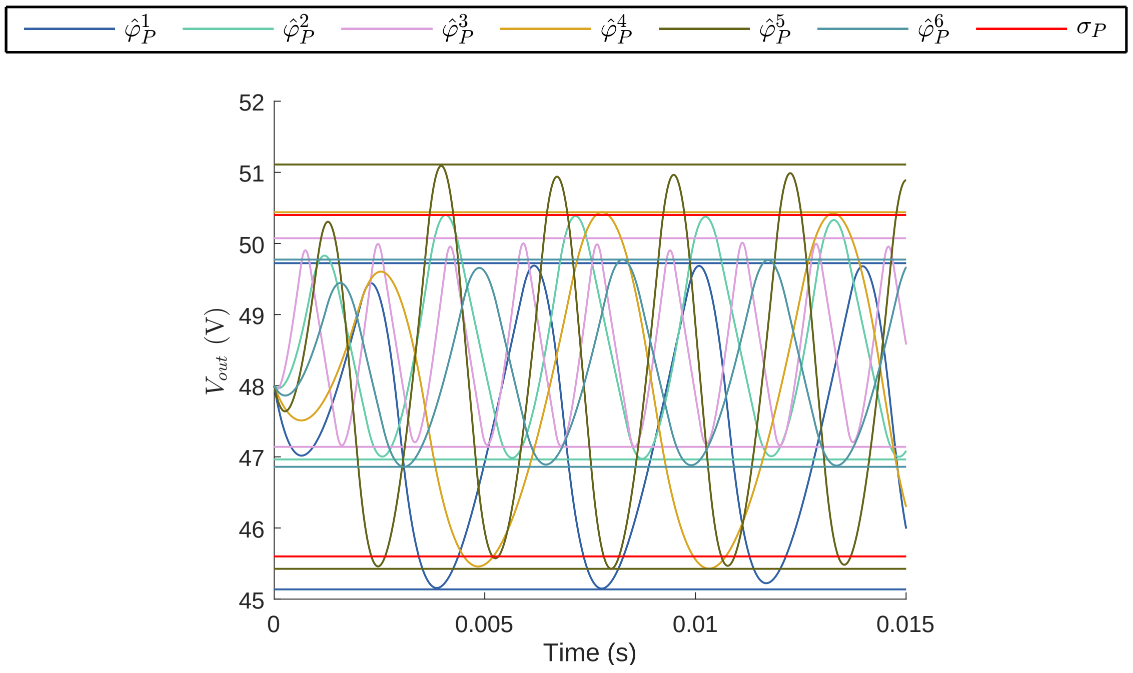


**Figure 2.10:** A plot represents simulation traces and magnitude bounds of $V_{out}$ of the buck converter with different values of $R$, $L$, and $C$. Here, $\sigma_P$ denotes the actual bound of $V_{out}$, and $\hat{\varphi}_P^k$, $k \in [1, 6]$ denotes the inferred bound of $V_{out}$ listed orderly in Table 2.2.

feedforward and Proportional-Integral (PI) controllers. Particularly, we focus on the third model of the benchmarks including a sequence of Simulink blocks and Stateflow chart that increase levels of sophistication and fidelity of the system [58]. The model consists of four operation modes and four continuous variables. The modes include *startup*, *normal*, *power*, and *failure*; and the variables are $(a)$ $p$: an intake manifold pressure, $(b)$ $p_e$: an intake manifold pressure estimate, $(c)$ $\lambda$: an air-fuel ratio, and $(d)$ $i$: an integrator state, PI control signal. The evolution of the continuous variables in each mode is governed by nonlinear polynomial differential equations as follows,

$$\dot{p} = c_1(2\theta(c_{20}p^2 + c_{21}p + c_{22}) - \dot{m}_c) \tag{2.2}$$

$$\dot{p_e} = c_1(2c_{23}\theta(c_{20}p^2 + c_{21}p + c_{22}) - (c_2 + c_3\omega p_e + c_4\omega p_e^2 + c_5\omega^2 p_e)) \tag{2.3}$$

$$\dot{\lambda} = c_{26}(c_{15} + (c_{16}c_{25}F_c + c_{17}c_{25}^2F_c^2 + c_{18}\dot{m}_c + c_{19}\dot{m}_c c_{25}F_c - \lambda) \tag{2.4}$$

$$\dot{i} = c_{14}(c_{24}\lambda - c_{11}), \tag{2.5}$$

where $F_c = \frac{1}{c_{11}}(1 + i + c_{13}(c_{24}\lambda - c_{11}))(c_2 + c_3\omega p_e + c_4\omega p_e^2 + c_5\omega^2 p_e)$, and $\dot{m}_c = c_{12}(c_2 + c_3\omega p + c_4\omega p^2 + c_5\omega^2 p)$. $\theta$ and $\omega$ are throttle angle (in degrees) and engine speed inputs (in *rpm*), respectively. The values of all constant parameters $c_j, j \in [1, 25]$, $\theta$ and $\omega$ are specified in [75]. We note that this system can be formally represented as a closed-loop CPIOA, which is the parallel composition of a plant and controller model, and both of them have three exogenous inputs including $\theta$, $\omega$, and sensor failure event *fail_event* [58].

*AFC Plant Model.* The plant can be modeled as a CPIOA with a single mode and two output physical variables $p$, $\lambda$ whose continuous evolutions over time are described in Equation 2.2 and Equation 2.4, respectively. This model has an input cyber variable $F_c$, that is a fuel command.

*AFC Controller Model.* The controller model is a CPIOA with four operation modes including *startup*, *normal*, *power*, and *failure*. The controller has two output physical variables $p_e$, and $i$ whose continuous evolutions over time are described in Equation 2.3 and Equation 2.5, respectively. Here, $p$ and $\lambda$ are considered as two input cyber variables of the controller.

Reachability analysis of a sophisticated system like the AFC system is a major contribution to both industrial and research community. However, it is a challenge to design and verify such a system using existing hybrid system verification tools. Instead, we can attempt to verify some safety requirements of the system. The AFC system has several actual physical specifications that can be found in [52]. In this section, we select two main physical specifications to evaluate the capability of Hynger and the proposed methodology. The first physical specification requires the undershoot and overshoot of the air-fuel ratio of the system should be in the settling region of $\pm 2\%$ of its reference value $\lambda_{ref}$. The second physical specification requires the air-fuel ratio should be maintained within $\pm 2\%$ of $\lambda_{ref}$ in the *normal* mode when $t \geq t_s$. These properties can be formally expressed as:

$$\sigma_P^1 \triangleq \ mode = startup \wedge t \leq t_s \Rightarrow 0.98\lambda_{ref} \leq \lambda(t) \leq 1.02\lambda_{ref} \tag{2.6}$$

$$\sigma_P^2 \triangleq \ mode = normal \wedge t \geq t_s \Rightarrow 0.98\lambda_{ref} \leq \lambda(t) \leq 1.02\lambda_{ref}. \tag{2.7}$$

Initially, we set $\lambda_{ref} = 14.7$, $\theta \in [8.8°, 90°]$, $w = 1800 rpm$ $t_s = 9.5s$, and the maximum simulation time $T_{max} = 20s$, the proportional and integral gains of the PI controller are $c_{13} = 0.04$ and $c_{14} = 0.14$, respectively. Next, we investigate different possibilities of cyber-physical specification mismatches for each physical specification. For the first physical specification $\sigma_P^1$, the AFC system may have specification mismatches when changing the engine speed and throttle inputs. For the second physical specification $\sigma_P^2$, the system may contain specification mismatches when changing controller and plant parameters.

*Cyber-physical specification mismatches according to $\sigma_P^1$.* With the initial setup mentioned earlier, the physical specification in Equation 2.6 becomes $\sigma_P \triangleq \ mode = startup \wedge t \leq 9.5 \Rightarrow 14.406 \leq \lambda(t) \leq 14.994$. Here, the magnitude bound of the air-fuel ratio at the *startup* mode of the system inferred from Hynger and Daikon is $\hat{\varphi}_P^1 \triangleq \ mode = startup \wedge t \leq 9.5 \Rightarrow 14.505 \leq \lambda(t) \leq 14.97$. Thus, the check of the formula $\hat{\varphi}_P^1 \Rightarrow \sigma_P^1$ is valid, that indicates $\hat{\varphi}_P^1$ is a candidate invariant of the AFC system. Next, we vary the input values and observe the consequent behaviors of the system.

First, we vary the value of the engine speed and keep other parameters unchanged. Assuming $w = 2200rpm$, the inferred physical specification of the air-fuel ratio from Hynger and Daikon becomes $\hat{\varphi}_P^1 \triangleq \ mode = startup \wedge t \leq 9.5 \Rightarrow 14.129 \leq V_{out}(t) \leq 15.033$. Hence, the formula $\hat{\varphi}_P^1 \Rightarrow \sigma_P^1$ is false indicating that the AFC system may contain a cyber-physical specification mismatch as we change the engine speed input.

Second, we change the range of the throttle input to $[40°, 70°]$. Then, the inferred physical specification of the air-fuel ratio from Hynger and Daikon becomes $\hat{\varphi}_P^1 \triangleq \ mode = startup \wedge t \leq 9.5 \Rightarrow 14.396 \leq V_{out}(t) \leq 14.849$. Hence, $\hat{\varphi}_P^1$ no longer implies $\sigma_P^1$. Therefore, there exists a cyber-physical specification mismatch when changing the throttle input as well.

*Cyber-physical specification mismatches according to* $\sigma_P^2$. Initially, the physical specification in Equation 2.7 is $\sigma_P^2 \triangleq \ mode = normal \wedge t \geq 9.5 \Rightarrow 14.406 \leq \lambda(t) \leq 14.994$. Here, the magnitude bound of the air-fuel ratio at the *normal* mode of the system inferred from Hynger and Daikon is $\hat{\varphi}_P^2 \triangleq \ mode = normal \wedge t \geq 9.5 \Rightarrow 14.645 \leq \lambda(t) \leq 14.84$. Then, we can consider $\hat{\varphi}_P^2$ as a candidate invariant of the system because the formula $\hat{\varphi}_P \Rightarrow \sigma_P$ is true.

Next, we investigate whether there is a specification mismatch for the AFC system as we change the proportional and integral gains of its PI controller. Table 2.3 describes the comparison between the actual physical specification $\sigma_P^2$ and the physical specification $\hat{\varphi}_P^2$ inferred from Hynger and Daikon, where $\hat{\varphi}_P^2 \downarrow \lambda$ denotes the inferred bound for $\lambda$ when $t \geq t_s$ and $mode = normal$. In Table 2.3, the check of the formula $\hat{\varphi}_P^2 \Rightarrow \sigma_P^2$ returns false in some cases (e.g., when $c_{13} = 0.04$, $c_{14} = 0.04$) indicating that the changes in the controller gains may produce cyber-physical specification mismatches for the AFC system.

| Controller Gain | $\hat{\varphi}^2_P \downarrow \lambda$ | $\hat{\varphi}^2_P \Rightarrow \sigma^2_P$ | $\sigma^2_P \Rightarrow \hat{\varphi}^2_P$ |
|---|---|---|---|
| $c_{13} = 0.01$, $c_{14} = 0.14$ | $14.567 \leq \lambda(t) \leq 15.058$ | *False* | *False* |
| $c_{13} = 0.02$, $c_{14} = 0.14$ | $14.592 \leq \lambda(t) \leq 15.033$ | *False* | *False* |
| $c_{13} = 0.06$, $c_{14} = 0.14$ | $14.634 \leq \lambda(t) \leq 14.955$ | *True* | *False* |
| $c_{13} = 0.8$, $c_{14} = 0.14$ | $14.642 \leq \lambda(t) \leq 14.929$ | *True* | *False* |
| $c_{13} = 0.04$, $c_{14} = 0.04$ | $14.649 \leq \lambda(t) \leq 15.007$ | *False* | *False* |
| $c_{13} = 0.04$, $c_{14} = 0.34$ | $14.581 \leq \lambda(t) \leq 14.937$ | *True* | *False* |
| $c_{13} = 0.04$, $c_{14} = 0.64$ | $14.577 \leq \lambda(t) \leq 14.888$ | *True* | *False* |
| $c_{13} = 0.04$, $c_{14} = 0.94$ | $14.589 \leq \lambda(t) \leq 14.855$ | *True* | *False* |

**Table 2.3:** Experiment results illustrate the comparison between actual physical specifications and inferred physical invariants from Hynger and Daikon of the AFC system when changing the proportional gain and the integral gain of its PI controller.

## 2.7 Discussion

Identifying a cyber-physical specification mismatch of CPS with dynamic analysis is a challenging problem. Although the Hynger prototype in conjunction with Daikon can detect potential cyber-physical specification mismatches of CPS, such as those in the case studies described in Section 2.6, however, it has some limitations. First, the Daikon tool used by Hynger may only infer extremely limited classes of nonlinear invariants by default (e.g., squares like $x^2$), and not general polynomials (e.g., $x^2 + y^2 + z^3$). So we plan to extend the invariant templates to be able to capture more interesting relations, particularly for physical variables. Second, although Daikon can infer candidate invariants in terms of logical predicates over variables, it has limitation for checking complex specifications related to real-time requirements such as STL, MTL and HyperSTL [117]. Industrial-scale CPS usually have safety and liveness requirements depending on precise real-time relations of signals, so strengthening the capability of checking temporal logic like STL, MTL and HyperSTL in Daikon would leverage the methodology presented in this chapter.

Additionally, while the Hynger tool is a prototype, it can be envisioned to take an arbitrary SLSF model, instrument it, feed the resulting traces to Daikon to generate candidate invariants, then check if these candidate invariants are actual invariants or not (using, e.g., SpaceEx [60] or other hybrid system model checkers), as well as identify specification mismatches. For example, the candidate invariants inferred from Hynger and Daikon of the buck converter including only plant and controller represented in term of hybrid automata in Figure 2.3 would easily be checked to see whether they are actually invariants using SpaceEx. In long term, Hynger could be extended for runtime assurance tasks like detecting and thwarting security violations and attacks, similar to the ClearView tool that also uses Daikon [126]. ClearView's success for software systems illustrates that finding sets of candidate invariants and monitoring their evolution over time may be useful for runtime assurance and resiliency methods in CPS. If the candidate invariants are checked at runtime using a real-time reachability method [19,78], a formal and dynamic runtime assurance environment may be feasible.

## 2.8 Conclusion

The results illustrate the feasibility of using dynamic invariant inference for analysis of embedded and cyber-physical systems. The Hynger prototype enables a powerful extension of dynamic invariant inference to CPS for two main reasons. First, it enables potentially model-free and black box invariant inference, since the internals of the SLSF blocks may remain unknown. If no model is available (in the black box case), the candidate invariants represent what may be the most formal model available, otherwise (in the white box case), then candidate invariants represent a candidate abstraction of that model. If the candidate invariants are actual invariants, this is powerful, as they represent what is likely a less complex representation of the set of reachable states of the system. Second, if we view the SLSF models as hybrid automata in a formal context, it represents the first use of dynamic execution analysis for hybrid systems with sophisticated software state and discrete complexity. Two proof-of-concept CPS case studies including the DC-to-DC power converter and the powertrain fuel control system are presented to illustrate the capability of Hynger in detecting potential cyber-physical specification mismatches.

# Chapter 3

# HYPERPROPERTIES OF REAL-VALUED SIGNALS

## 3.1 Introduction

Hyperproperties were first proposed by Clarkson and Schneider to characterize properties of security policies that cannot be defined over individual traces, such as service level agreements and information-flow properties [38]. In this work, we extend the notion of hyperproperties to cover a broad range of requirements for CPS, and we present a taxonomy of hyperproperties used to address security and control design concerns for CPS. Also, we provide practical techniques for automating the process of testing hyperproperties of CPS.

In contrast to trace properties expressed over individual execution traces, hyperproperties are defined over multiple execution traces. For example, one execution of a system cannot be checked against a service level agreement property such as *"the average time elapsed between a user's request and response over all executions should be less than 1 second"*; the property can only be evaluated over all system execution traces. Moreover, we can consider an information-flow policy of *noninterference* specified as *"for all pairs of traces of a system that have the same low-level security inputs, they will also have the same low-level security output"* [65, 131]. This noninterference property is a hyperproperty as it is expressed over all pairs of traces of a system.

Hyperproperties generalize more traditional formal properties by specifying relationships between disparate execution traces, instead of behaviors of individual execution traces. Traditional logics that consider traces individually, such as LTL, cannot be used to specify hyperproperties, and thus, hyperproperties are more expressive. Logics such as CTL and CTL* allow properties over multiple paths of a computation tree, but they do not permit comparisons between the paths themselves. Instead, to express and efficiently check hyperproperties, Clarkson et al., introduced notions of HyperLTL and HyperCTL* [37]. Both logics directly extend LTL and allow us to reason about more than one execution trace at a time. The main difference between HyperLTL and HyperCTL* is that the former requires trace quantifiers appearing at the beginning of a formula, but the latter allows us to specify them within a formula.

Although hyperproperties are well studied in the context of security policies for software systems, hyperproperties have not been explored for CPS. For CPS that include stochastic factors such as noise, environment disturbance, or transducer inaccuracies, it is realistic for design engineers to expect that the system has some acceptable performance in a probabilistic sense rather than requiring an absolute performance limit be met for all individual behaviors. Acceptable performances defined over the averages of settling time, overshoot, undershoot, or error bounds cannot be specified and checked using individual execution traces; they must be quantified over all execution traces.

Recently, security-aware function modeling of CPS has emerged as an important research topic in computer science and system verification. CPS, that are an integration between cyber and physical subcomponents, can be vulnerable to both cyber-based and physical-based attacks [6, 61, 106, 144]. For instance, consider a modern automobile, which is a complex system composed of many computer units such as an Engine Control Unit (ECU), the Transmission Control Module (TCM), and an Electronic Brake Control Module (EBCM), all interacting with the physical world via sensors and actuators. Cyber-based attackers can gain access to the communication channels of a modern automobile through wireless or in-vehicle networks. As a result, attackers can infiltrate an ECU or EBCM to stall the engine or disable the brake system [87, 136]. An alternative method of attack involves gaining physical access to the system, for example by manipulating the signals processed by the sensors (known as sensor *spoofing*), to compromise secure information or to alter system behaviors [6, 139]. Instances of active physical-based attacks include vehicle braking system attacks, where faulty data is injected into the wheel speed sensor of a vehicle to disrupt the braking function [144], and insulin delivery device attacks, where glucose level sensor data is corrupted to compromise the function of the insulin delivery service [91]. A passive physical-based attack, also called a side-channel attack, is based on physically observing the system behavior and using leaked information to gain insights into the system implementation [82, 84, 132]. Some well-known side channel attacks are power analysis attacks [83], timing attacks [85], electromagnetic attacks [134] and differential fault analysis attacks [29].

Designing safety-critical CPS that are entirely secure from both cyber-based and physical-based attacks is challenging or impossible. A reasonable approach is to iteratively improve CPS control designs using a *falsification* technique. Falsification is an automated best-effort approach to identify system behaviors that violate a given formal specification [113]. The design approach would be to first formally specify safety properties of CPS that protect the systems against possible cyber-based and physical-based attacks using formalisms such as temporal logic and to then iteratively improve the design using falsification, which would automatically identify vulnerabilities in the designs. Despite the attractiveness of falsification techniques, attacks for CPS often need to be defined over multiple execution traces of the system, which is something that cannot be expressed or falsified using existing temporal logics such as LTL, MTL, and STL. Thus we propose an extension to these logics that would be compatible with the appropriate specifications. In this chapter, we present a study of hyperproperties including stability, security and safety, as applied to CPS. We introduce several instances of hyperproperties capturing relationships (e.g input-output relationships) between multiple traces of CPS. We extend the syntax and semantics of STL [50] to specify hyperproperties over dense-time real-valued signals, which results in a new logic called HyperSTL. Basically, we add quantifiers at the beginning of an STL formula to express relationships between multiple traces. We also introduce a testing algorithm based on a fragment of HyperSTL and apply it to find falsifying traces for hyperproperties of industrial Simulink models. Moreover, we provide a discussion on the feasibility of falsifying or verifying various classes of hyperproperties for CPS.

**Related work.** The study of hyperproperties for CPS evaluated in this chapter was inspired by the previous work of Clarkson and Schneider, who introduced hyperproperties to express security policies such as secure information flows and service level agreements [38]. In [35], Bryans et. al. presented a general formalization of opacity policies that prevent observers from deducing the truth value of a predicate; those opacity policies require behaviors to be specified over multiple paths of a system. In earlier work [102], McLean showed that some “possibilistic” security properties like restrictiveness [100], noninterference [65] and nondeducibility [146] are closure properties that cannot be expressed by individual execution traces. In [102], those properties are specified with respect to different sets of trace contractors called selective interleaving functions.

Following the introduction of hyperproperties [38], Clarkson et al. introduced HyperLTL and HyperCTL*, which are extensions to existing temporal logics, to express and check classes of information-flow hyperproperties [37]. These logics extended LTL and CTL* by adding the path quantifiers that permit specifications involving multiple paths in the system. Model checking algorithms and complexity of fragments of HyperLTL and HyperCTL* were also given in [37], which were then further exploited and applied to check some classes of information-flow hyperproperties in [131].

Prototype implementations of model checkers for HyperLTL and HyperCTL*, which assume the system is modeled as a Kripke structure, can verify some information-flow hyperproperties of a discrete-time system, but extending that work to check hyperproperties defined over continuous traces is a challenging endeavor. For complex CPS models or for models built in frameworks with proprietary or otherwise obfuscated semantics, such as Simulink®, formal verification of hyperproperties is effectively impossible, as no corresponding Kripke structure may be obtained from those models[1]. Alternatively, an easier but still difficult task is to develop an efficient testing framework, which could be used to check hy-

[1]Some have created their own translation of Simulink models to modeling languages with well-defined formal semantics (for example, see [4, 152]), but these translations necessarily only handle a subset of the SLSF modeling language. This is due to the fact that some Simulink constructs correspond to behaviors that cannot be modeled using standard frameworks for hybrid systems. One such construct is the Variable Transport Delay block, which, roughly speaking, corresponds to a delay differential equation, a construct that is not handled by standard modeling frameworks for hybrid systems.

perproperties for finite collections of traces or could be used to falsify hyperproperties of a CPS model; this is the contribution of the work presented herein.

In [147], Xu et al. introduced a notion of CensusSTL that utilizes STL by adding an *outer* logic to quantify the number of individual agents of a multiagent system whose behaviors satisfy an *inner* STL formula. CensusSTL is similar to the HyperSTL proposed in this chapter; however, the former is only able to specify group behaviors from different components of an individual trace while the latter allows us to express relationships between multiple traces.

## 3.2 Preliminaries

In this section, we review the concepts of signal, system, trace property and STL.

**Signal.** We define a signal $w$ as a function $w : \mathbb{T} \to \mathbb{D}$, where $\mathbb{T} \subseteq \mathbb{R}_{\geq 0}$ is the time domain. If $\mathbb{D} = \mathbb{B}$, $w$ is a Boolean signal whose value is either true or false, and if $\mathbb{D} = \mathbb{R}$, then we say that the signal is real-valued. A *trace*, $\mathbf{w} : \mathbb{T} \to \mathbb{D}_1 \times \ldots \times \mathbb{D}_n$ , is a collection of $n$ signals, where $\forall t \in \mathbb{T}, \mathbf{w}(t) \triangleq (w_1(t), w_2(t), ..., w_n(t))$. Intuitively, we can consider $\mathbf{w}$ as one execution trace of a continuous-time system with $n$ variables that describes an evolution of the system. In what follows, we reserve the use of bold letters like $\mathbf{w}$, $\mathbf{w}'$ for traces (i.e., tuples of signals), while we use lowercase italicized letters such as $w_i$ to represent signals.

**System.** We define a deterministic or nonstochastic[2] cyber-physical system $\Sigma$ as a function

[2]Note the contrast with *stochastic* systems. In stochastic systems, one or more parts of the system have randomness associated with them; for instance, the value of a particular system parameter may be drawn from a probability distribution. The key difference is that the stochastic system may not produce the same output for a given input. Unless otherwise specified, all the systems that we consider in this chapter are

mapping a given input trace in $(\mathbb{T} \to \mathbb{D}^m)$ to an output trace in $(\mathbb{T} \to \mathbb{D}^n)$. We denote by $[\![\Sigma]\!]$ the set of traces $\mathbf{w}$ such that the first $m$ components of $\mathbf{w}$ correspond to the $m$ input signals for $[\![\Sigma]\!]$, and the next $n$ components correspond to the $n$ output signals.

**Trace properties.** A trace property $\varphi$ is a finite or infinite set of individual traces. A trace property is either satisfied or violated by any given set of traces [8, 131]. A set of traces $W$ satisfies the trace property $\varphi$ if $W \subseteq \varphi$. As noted above, an individual trace can have several components, for example, a trace could contain $m$ input signals and $n$ output signals of a given system $\Sigma$. We say that the trace property $\varphi$ holds for a system $\Sigma$ (denoted as $\Sigma \models \varphi$) if the set of input-output traces compatible with the system description is contained in the trace property, i.e., $[\![\Sigma]\!] \subseteq \varphi$.

**Signal Temporal Logic.** Next, we recall the concept of STL which can be used to specify the trace properties of continuous real-valued signals.

*Syntax.* The syntax of STL is defined as follows:

$$\varphi := \mathit{true} \mid \phi \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathbf{U}_I \varphi_2 \; ,$$

A signal operator $\phi$ is a formula of the form $y(\mathbf{w}(t)) \geq 0$, where $y$ is an arbitrary real-value function. A notion $I$ is an interval over $\mathbb{R}_{\geq 0}$ such as $[a, b)$, $(a, b)$, $(a, b]$, $[a, b]$, $(a, +\infty)$, or $[a, +\infty)$, and $a$, $b$ are real numbers and $0 \leq a < b$. If $I$ is not specified, we assume that $I = [0, \infty)$. We also allow Boolean operators $\vee$ and $\implies$ with their standard meaning. Temporal operators used in STL formulas include *always* ($\Box$), *eventually* ($\Diamond$), and *until* ($\mathbf{U}$), respectively, where $\Diamond_I \varphi = \mathit{true}\mathbf{U}_I \varphi$, and $\Box_I \varphi = \neg\Diamond_I \neg\varphi$. For example, a trace $\mathbf{w} \triangleq \{w_1, w_2\}$ satisfies the formula $\varphi \triangleq \Diamond_{[1,2)}(w_1(t) > w_2(t))$ if there exists a time instance $t$, $1 \leq t < 2$ such

deterministic.

that $w_1$ is greater than $w_2$. Next, we will describe the Boolean and quantitative semantics of STL.

*Boolean Semantics.* The Boolean semantics of STL are specified by the following conditions:

$$\begin{aligned}
(\mathbf{w}, t) \models \phi &\quad \text{iff} \quad \phi(\mathbf{w}, t) = \mathit{true} \\
(\mathbf{w}, t) \models \neg\varphi &\quad \text{iff} \quad \mathbf{w} \not\models \phi \\
(\mathbf{w}, t) \models \varphi_1 \wedge \varphi_2 &\quad \text{iff} \quad \mathbf{w} \models \varphi_1 \text{ and } \mathbf{w} \models \varphi_2 \\
(\mathbf{w}, t) \models \varphi_1 \mathrm{U}_I \varphi_2 &\quad \text{iff} \quad \exists t_1 \in t + I \text{ s.t. } (\mathbf{w}, t_1) \models \varphi_2 \text{ and } \forall t_2 \in [t, t_1], (\mathbf{w}, t_2) \models \varphi_1
\end{aligned}$$

*Quantitative Semantics.* The quantitative semantics of STL reflect the *robustness satisfactions* of STL formulas [50, 95]. Given $\gamma$ is a real-valued function of a formula $\varphi$, a trace $\mathbf{w}$, and a time $t$, the quantitative semantics $\gamma(\varphi, \mathbf{w}, t)$ is defined as follows:

$$\begin{aligned}
\gamma(\mathbf{w}(t) \geq 0, \mathbf{w}, t) &= y(\mathbf{w}(t)) \\
\gamma(\neg\varphi, \mathbf{w}, t) &= -\gamma(\varphi, \mathbf{w}, t) \\
\gamma(\varphi_1 \wedge \varphi_2, \mathbf{w}, t) &= \min(\gamma(\varphi_1, \mathbf{w}, t), \gamma(\varphi_2, \mathbf{w}, t)) \\
\gamma(\varphi_1 \mathrm{U}_I \varphi_2, \mathbf{w}, t) &= \sup_{t_1 \in t+I} \min(\chi(\varphi_2, \mathbf{w}, t_1), \inf_{t_2 \in [t, t_1]} \gamma(\varphi_1, \mathbf{w}, t_2))
\end{aligned}$$

Instead of returning the Boolean values, the quantitative semantics provide a real value representing the distance to the robustness satisfaction value of a formula $\varphi$ at each time instance. Intuitively, this distance quantifies *how much* a trace satisfying a given trace property.

## 3.3 Hyperproperties of Real-Valued Signals

Hyperproperties generalize formal properties of a system by considering sets of sets of execution traces, instead of only sets of execution traces.

**Definition 9 (Hyperproperty)** *Let $S$ denote the set of all traces. Let the power set of $S$ be written as $P \triangleq \mathcal{P}(S)$. A* hyperproperty *is any subset of $\mathcal{P}(S)$.*

We say a set of traces $W$ satisfies a hyperproperty $\phi \subseteq P$ if $W \in \phi$. Given a hyperproperty $\phi$ and a system $\Sigma$, the falsification task is to find a non-empty set $W \subseteq [\![\Sigma]\!]$ such that $W \notin \phi$. Similarly, given a hyperproperty $\phi$ and a system $\Sigma$, the verification task is to show that $[\![\Sigma]\!] \in \phi$.

In this section, we introduce hyperproperties for deterministic systems to characterize properties such as security, safety, and stability. We focus on a class of hyperproperties capturing relationships ( e.g., the input-output relationship) between multiple traces of a system. We will show several examples of hyperproperties related to safety, stability and security for CPS. In rest of this section, we use $d_{sup}(\mathbf{w}, \mathbf{w}')$ to denote the sup-norm distance between traces $\mathbf{w}$ and $\mathbf{w}'$, where $d_{sup}(\mathbf{w}, \mathbf{w}') = \sup_{t \in \mathbb{R}_{\geq 0}} ||\mathbf{w}(t) - \mathbf{w}'(t)||$.

- *Robust behavior* is a requirement that guarantees that small differences in system inputs result in small differences in system outputs. Consider the following property: "*For all pairs of traces of a system with an input difference less than $\epsilon_1$, the output difference should be bounded by $\epsilon_2$*". Such a property is a hyperproperty as it requires at least two execution traces to check. This hyperproperty can be formally written as:

$$\phi_1 \triangleq \{W \in P \mid \forall \mathbf{w}, \mathbf{w}' \in W : d_{sup}(\mathbf{w}_{in}, \mathbf{w}'_{in}) \leq \epsilon_1 \implies d_{sup}(\mathbf{w}_{out}, \mathbf{w}'_{out}) \leq \epsilon_2\}. \quad (3.1)$$

This type of property is related to certain stability notions, such as bounded input, bounded output (BIBO) stability and the $\mathcal{L}_2$ gain, as these notions also bound the variation in the output, based on bounded variation in the input. We note, however, that the robust behavior hyperproperty differs from BIBO stability and the $\mathcal{L}_2$ gain, as the robust behavior hyperproperty is specified over all pairs of execution traces while the BIBO and $\mathcal{L}_2$ properties are defined based on individual traces. The robust behavior hyperproperty is also related to *bisimulation relations* [57] and *conformance-closeness* [3] for a dynamical system, as all three of these properties are based on some constraints on the distances between multiple traces. In fact, we may specify bisimulation or conformance-closeness functions in terms of hyperproperties. Lastly, we note that the robust behavior hyperproperty is perhaps most closely related to Lipschitz Robustness of systems [70], which bounds differences in output behaviors based on bounded differences in input behaviors, though Lipschitz Robustness was originally developed for timed input/output systems as opposed to general CPS models.

- *Side-channel attacks* are attacks against cryptographic devices based on studying leaking information about the operations they process, such as power consumption, heat generation, and execution time. The side channel attack is an instance of an inactive physical-based attack that can be used against a CPS in which some physical behaviors are observable. Attackers can deduce the working principle of a system without either access to the system itself or an understanding of the internal operation of the system. For example, attackers can analyze an abnormal change in the power consumption of an integrated circuit while an encryption process is being executed and then reconstruct the encryption key to access secret data [83, 84]. The following property permits side-channel attacks:

$$\phi_2 \triangleq \{W \in P \mid \exists \mathbf{w} \in W : \forall \mathbf{w}' \in W : (d_{sup}(\mathbf{w}, \mathbf{w}') > 0 \\ \wedge \mathit{Power}(\mathbf{w}(t)) > c_1) \implies \mathit{Power}(\mathbf{w}'(t)) < c_2\}, \tag{3.2}$$

where $\mathit{Power}(\mathbf{w}(t))$ represents the power consumption corresponding to $\mathbf{w}$ over time, and $c_1, c_2$ are arbitrary constants such that $c_1 > c_2$. A system that satisfies this property allows an attacker to detect that a particular behavior has occurred ($\mathbf{w}$ in Formula 3.2) by monitoring the power associated with the behavior. The property is a hyperproperty as it is expressed in terms of multiple traces. To ensure the safety of a system from the power-monitoring attack, the system should satisfy $\neg\phi_2$. We note that other classes of side-channel attacks such as timing attacks, electromagnetic attacks, and differential fault analysis attacks can be specified using properties similar to Formula 3.2.

- *Robust control invariance* is a property that can be used to synthesize safe controllers, or more to the point, can be utilized to determine whether a safe controller exists for systems with disturbances [30]. Informally, the property states that, for a given set of behaviors that is deemed safe, a control action exists, such that the system remains within the safe set for any allowable disturbance input. This can be stated formally as follows:

$$\phi_3 \triangleq \{W \in P \mid \exists \mathbf{w} \in W : \forall \mathbf{w}' \in W : (\mathbf{w}, \mathbf{w}') \models \phi\}, \tag{3.3}$$

where $(\mathbf{w}, \mathbf{w}') \models \phi$ means that the pair $(\mathbf{w}, \mathbf{w}')$ satisfies some property $\phi$. In this formulation, $w_u(t)$ is the component of $\mathbf{w}$ that represents the controller action, $w_d(t)$ is a disturbance input, $w_y(t)$ is a system output, and $(\mathbf{w}, \mathbf{w}') \models \phi$ enforces both that $w_u = w'_u$

and $w'_y(t) \in \Omega$, where $\Omega$ is the set of safe behaviors. The robust control invariance property is related to fault data injection (FDI) attacks, which are active physical-based attacks where attackers try to input faulty data into a system to corrupt the behavior of the controller. For example, attackers can spoof the sensors of DC microgrids by injecting false data such as the past outputs of the sensors at previous time instants. This instance of FDI attack is also well known as a *replay* attack [23,91,144]. FDI attacks have been studied widely for CPS, and many techniques have been proposed to efficiently detect those attacks in the early stages [23,93,98]. However, the optimal solution is to design a system that can defend itself against FDI attacks [105]. To guarantee that a system can defend against a sensor attack, given a specification $\phi$, it must be possible to choose a controller that ensures that the output of the system always satisfies $\phi$, i.e. $\phi_3$ must hold.

## 3.3.1 Beyond Hyperproperties?

A hyperproperty is more expressive than a trace property as it is defined over a set of sets of traces and requires multiple traces to check. If a system is modeled as trace sets, one interesting question to ask is whether there are system properties inexpressible as hyperproperties. For security policies, all properties of trace sets can be considered as hyperproperties, so the answer may be *negative* [8, 38]. For CPS, there may exist some properties that are challenging to classify.

Consider the following property specifying the *Lyapunov stability* of a dynamical control system:

$$\phi_{Ly} \triangleq \{\forall \epsilon \in [0, \infty), \exists \delta \in [0, \epsilon), \forall \mathbf{w} \in W : ||\mathbf{w}(0)|| < \delta \implies (t > 0 \land ||\mathbf{w}(t)|| < \epsilon)\}. \quad (3.4)$$

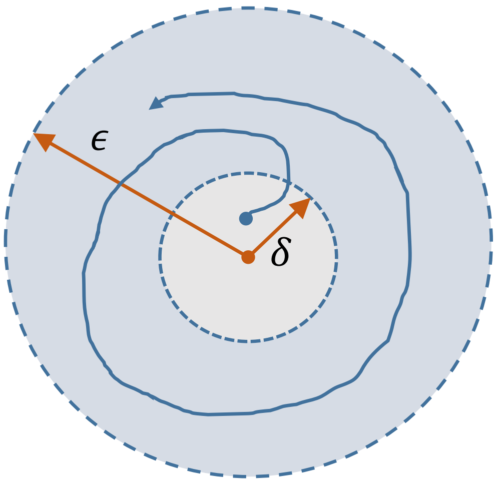


**Figure 3.1:** Illustration of a Lyapunov stable system.

Intuitively, this property indicates that a system is Lyapunov stable if for any $\epsilon$-*ball* around the origin, there exists a $\delta$-*ball* around the origin ($\delta < \epsilon$) such that if the system starts within the $\delta$-*ball*, then it will never leave the $\epsilon$-*ball* [28]. The illustration of a Lyapunov stable system is shown in Figure 3.1.

Lyapunov stability is specified over the space of parameters and execution traces, and involves two alternations between universal and existential quantifiers. As we cannot check the Lyapunov stability with individual traces, it is not a trace property; so is it a hyperproperty? Consider the parameters $\delta$ and $\epsilon$ as constant signals, and then rewrite Lyapunov stability as follows:

$$\begin{aligned} \phi'_{Ly} \triangleq \{W \in P \mid \forall \mathbf{w} \in W \ : \exists \mathbf{w}' \in W \ : \forall \mathbf{w}'' \in W : \\ ||w''_{out}(0)|| < w'_{\delta}(0) \implies (t > 0 \wedge ||w''_{out}(t)|| < w_{\epsilon}(t))\}, \end{aligned} \tag{3.5}$$

where a trace $\mathbf{w}$ is composed of two constant input signals $w_\delta$, $w_\epsilon$ and an output signal $w_{out}$. By mapping parameters into constant signals, we can express interesting properties of the system as hyperproperties. Then Lyapunov stability is a hyperproperty that requires multiple traces to check; and it can be formally specified using the HyperSTL introduced in the next section. As to the original question of whether all system properties of interest can be specified as hyperproperties, we leave this open.

**Remark 10** *Although we focus on describing hyperproperties defined over real-valued signals, we note that there are other hyperproperties that can be specified in the context of CPS as well. For instance, the* nondeducibility property *is an important information-flow security policy that prevents a low-level observer with sufficient knowledge of a target CPS from deducing high-level (confidential) information. The nondeducibility property is defined such that for each low-level input trace, there are more than one possible high-level input traces that produce the same output. Intuitively, an attacker should not be able to distinguish between permissible high-level behaviors based on low-level behaviors [62, 101]. On the other hand, the* noninterference property *is another important information-flow security policy that requires that high-level security users should not interfere with low-level security users. Intuitively, the outputs observed by the low-level security users remain unchanged despite the actions of the high-level security users [65]. Other variants of the noninterference property such as* noninference *[102],* observational determinism *[150],* declassification *[135], and* quantitative nonterinference *[140] are also hyperproperties that need to be specified over multiple traces. Though the nondeducibility and noninterference properties are relevant for CPS, in many cases their impact on and from real-valued signals is tenuous, and so we do not treat them further herein.*

## 3.4 HyperSTL

In this section, we introduce HyperSTL, a temporal logic that can be used to specify a class of hyperproperties of real-valued signals. The syntax and semantics of HyperSTL are naturally extended from those of STL by adding existential and universal trace quantifiers into STL's syntax to relate multiple execution traces [50].

*Syntax.* Let $\mathbf{v}$ be a *trace variable* from an infinite set of trace variables $\mathcal{V}$. The syntax of HyperSTL is then defined as follows:

$$\phi := \exists \mathbf{v}.\phi \mid \forall \mathbf{v}.\phi \mid \varphi$$

$$\varphi := \mathit{true} \mid \mu_{\mathcal{V}} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \mathbf{U}_I \varphi$$

Here, we add a universal quantifier $\forall$ and an existential quantifier $\exists$ to the syntax to indicate whether we want to specify that a formula holds over all traces or over at least one trace, respectively. For instance, $\forall \mathbf{v}.\exists \mathbf{v}'.\phi$ means that for any trace $\mathbf{w}$ assigned to trace variable $\mathbf{v}$ , there exists a trace $\mathbf{w}'$ that can be assigned to trace variable $\mathbf{v}'$ such that $\phi$ holds on these two traces. We define $\Pi : \mathcal{V} \rightarrow S$ as a *trace assignment* (i.e., a valuation), which is a partial function mapping trace variables to traces, and $S$ is a set of all infinite traces. Let $v_i$ be the projection of a trace variable $\mathbf{v}$ along its $i^{\text{th}}$ component, the projection of a trace assignment $\Pi(v_i)$ maps $v_i$ to the $i^{\text{th}}$ component of a trace $\mathbf{w}$ (i.e., $w_i$). Also, we abuse the subscript notation of a trace's component to write its corresponding trace variable's component in a HyperSTL formula, e.g., $w_{out}$ is represented by $v_{out}$. A relationship between multiple traces can be Booleanized through atomic predicates of the form $\mu_{\mathbf{v}} = f(\Pi(\mathbf{v}_1)(t), \Pi(\mathbf{v}_2)(t), ..., \Pi(\mathbf{v}_k)(t)) > 0$, where $f$ is an arbitrary real-valued function over trace variables $\mathbf{v}_1, ..., \mathbf{v}_k \in \mathcal{V}$. Note that we use trace variables such as $\mathbf{v}$, $\mathbf{v}'$ to express HyperSTL formula and the corresponding traces represented by these trace variables like $\mathbf{w}$ $\mathbf{w}'$ to interpret the formula. Consider the HyperSTL formula $\phi := \exists \mathbf{v}.\forall \mathbf{v}'.\Box_{[0,1]}(||\mathbf{v}-\mathbf{v}'|| < 1)$. This property says that there is always a trace $\mathbf{w}$, such that for all times in the interval $[0, 1]$, every other trace $\mathbf{w}'$ is at a bounded distance of 1 from $\mathbf{w}$.

*Boolean Semantics.* A HyperSTL formula satisfied by a set of traces $W$ at a time $t$ is written as $\Pi, t \models_W \phi$, the validity judgment of a HyperSTL formula at a given time $t$ is specified according to the following recursive semantics:

$$
\begin{array}{rcl}
\Pi, t \models_W \exists \mathbf{v}.\phi & \text{iff} & \text{exists}\, \mathbf{w} \in W \;:\; \Pi(\mathbf{v}) = \mathbf{w} \,\text{and}\, \Pi, t \models_W \phi \\
\Pi, t \models_W \forall \mathbf{v}.\phi & \text{iff} & \text{forall}\, \mathbf{w} \in W \;:\; \Pi(\mathbf{v}) = \mathbf{w} \,\text{and}\, \Pi, t \models_W \phi \\
\Pi, t \models_W \mu_{\mathcal{V}} & \text{iff} & f(\Pi(\mathbf{v}_1)(t), \Pi(\mathbf{v}_2)(t), ..., \Pi(\mathbf{v}_k)(t)) > 0 \\
\Pi, t \models_W \neg\varphi & \text{iff} & \Pi, t \not\models_W \varphi \\
\Pi, t \models_W \varphi_1 \wedge \varphi_2 & \text{iff} & \Pi, t \models_W \varphi_1 \,\text{and}\, \Pi, t \models_W \varphi_2 \\
\Pi, t \models_W \varphi_1 \mathrm{U}_I \varphi_2 & \text{iff} & \exists t_1 \in t + I \,\text{s.t}\, \Pi, t_1 \models_W \varphi_2 \,\text{and}\, \forall t_2 \in [t, t_1] \,\text{s.t}\, \Pi, t_2 \models_W \varphi_1
\end{array}
$$

Using HyperSTL, we can express the hyperproperties described in Section 3.3 over some time interval $[t_1, t_2]$ as follows[3].

- The robust behavior in Formula 3.1 can be specified as:

$$\phi_1' \triangleq \forall \mathbf{v}.\forall \mathbf{v}'.\ \Box_{[t_1,t_2]}(d_{sup}(\mathbf{v}_{in}, \mathbf{v}_{in}') \leq \epsilon_1 \implies d_{sup}(\mathbf{v}_{out}, \mathbf{v}_{out}') \leq \epsilon_2). \tag{3.6}$$

[3] For a robust control invariance hyperproperty, an instance of the corresponding HyperSTL formula will be shown in Section 3.7.2.

- The power-monitoring attack in Formula 3.2 can be written as:

$$\phi'_2 \triangleq \exists \mathbf{v}.\forall \mathbf{v}'.\ \Box_{[t_1,t_2]}((d_{sup}(\mathbf{v},\mathbf{v}') > 0 \wedge \mathit{Power}(\mathbf{v}) > c_1) \implies \mathit{Power}(\mathbf{v}') < c_2). \quad (3.7)$$

Furthermore, we can rewrite the Lyapunov stability specified in Formula 3.5 as the following HyperSTL formula,

$$\phi''_{Ly} \triangleq \forall \mathbf{v}.\exists \mathbf{v}'.\forall \mathbf{v}''.\ (v''_{out} < v'_{\delta} \implies \Box_{(0,\infty)} v''_{out} < v_{\epsilon}). \quad (3.8)$$

According to the possible alternation of quantifiers in a HyperSTL's syntax, we classify the above HyperSTL formulae into two fragments:

(a) *alternation-free* HyperSTL formulae including one type of quantifier, and

(b) *k-alternation* HyperSTL formulae that have $k$ number of alternations between existential and universal quantifiers.

Thus, the robust behavior property can be expressed using alternation-free HyperSTL while the power-monitoring attack property can be specified using 1-alternation HyperSTL. The Lyapunov stability property is more complex as it must be expressed using 2-alternation HyperSTL.

**Falsification or Verification of Hyperproperties?** We have introduced several classes of hyperproperties for CPS and a temporal logic approach to express them. Next, we investigate whether we can falsify or verify those hyperproperties using existing methods. Hyperproperties are more complex and expressive than traditional properties, and performing falsification and verification for hyperproperties is harder, in many cases. Despite this, we observe that certain classes of hyperproperties can be falsified or verified. For instance, we can falsify an alternation-free HyperSTL formula that contains a universal quantifier (e.g., the robust behavior hyperproperty), and we can verify an alternation-free HyperSTL formula that contains an existential quantifier. For the class of hyperproperties that includes alternating quantifiers, falsification or verification are often undecidable unless we impose some assumption about the sets of execution traces (e.g., quantified over some finite set of traces with bounded time).

### 3.4.1 t-HyperSTL

We introduce t-HyperSTL as a fragment of HyperSTL in which a nesting structure of temporal logic formulas involving different traces is not allowed. For example, a formula $\forall \mathbf{v}.\exists \mathbf{v}'.\Box_{[0,2]}\mathbf{v} > 1 \implies \Diamond_{[1,2]}\mathbf{v}' > 2$ is allowed but a formula $\forall \mathbf{v}.\exists \mathbf{v}'.\Box_{[0,2]}(\mathbf{v} > 1 \implies \Diamond_{[1,2]}\mathbf{v}' > 2)$ is not allowed. Also, t-HyperSTL restricts the *until* operator to be specified over an individual trace, e.g., t-HyperSTL does not allow the formula $\forall \mathbf{v}.\exists \mathbf{v}'.(\mathbf{v} > 1)\mathbf{U}_{[0,1]}(\mathbf{v}' > 2)$.

Inherited from the syntax of HyperSTL, t-HyperSTL formulae are also classified into alternation-free and k-alternation types. t-HyperSTL suffices to express the class of hyperproperties formulated in Section 3.3, and its corresponding semantics, which is more restrictive than that of HyperSTL, allow us to perform falsification for these hyperproperties.

*Quantitative Semantics.* The quantitative semantics of t-HyperSTL reflects the *robustness satisfaction* of a t-HyperSTL formula. It is a natural extension of those for STL [50,95]. Given $\chi$ is a real-valued function of a formula $\varphi$, a trace assignment $\Pi$, a trace variable $\mathbf{v}$, and a time $t$, the quantitative semantics of t-HyperSTL is defined inductively as follows:

$$\exists \mathbf{v}.\chi(\varphi, \Pi, t) = \sup_{\mathbf{w} \in W} \chi(\varphi, \Pi(\mathbf{v}) = \mathbf{w}, t)$$

$$\forall \mathbf{v}.\chi(\varphi, \Pi, t) = \inf_{\mathbf{w} \in W} \chi(\varphi, \Pi(\mathbf{v}) = \mathbf{w}, t)$$

$$\chi(\mu_{\mathcal{V}} > 0, \Pi, t) = \mu_{\mathcal{V}}$$

$$\chi(\neg\varphi, \Pi, t) = -\chi(\varphi, \Pi, t)$$

$$\chi(\varphi_1 \wedge \varphi_2, \Pi, t) = \min\left(\chi(\varphi_1, \Pi, t), \chi(\varphi_2, \Pi, t)\right)$$

$$\chi(\varphi_1 \mathrm{U}_I \varphi_2, \Pi, t) = \sup_{t_1 \in t+I} \min\left(\chi(\varphi_2, \Pi, t_1), \inf_{t_2 \in [t, t_1]} \chi(\varphi_1, \Pi, t_2)\right)$$

## 3.5 Falsifying alternation-free t-HyperSTL

We first consider the falsification of alternation-free t-HyperSTL formulae. This fragment of HyperSTL is expressive enough to capture a broad range of hyperproperties specifying input-output relationships over all pairs of execution traces. We use a translation scheme called self-composition [21], which allows us to falsify an alternation-free t-HyperSTL formula that includes only universal quantifiers using a robust testing method for a normal STL formula. Then, given an alternation-free t-HyperSTL that includes universal quantifiers, we attempt to find a set of falsifying traces for CPS corresponding to this formula.

**Falsification algorithm.** The procedure that addresses the falsification problem of a system $\Sigma$ with respect to a given hyperproperty $\varphi_h$ over a time duration $T$ is shown in Algorithm 1, and further interpreted as follows.

- □ We first transform the alternation-free t-HyperSTL formula $\varphi_h$ into the equivalent STL formula $\varphi_{STL}$.

**Algorithm 1** Falsification of alternation-free t-HyperSTL

```
Require: a system $\Sigma$, a parameter space $\Theta$,
a t-HyperSTL formula $\varphi_h$, a time duration $T$,
a maximum number of simulations N
begin
  $\varphi_{STL} \leftarrow \mathsf{HyperSTL2STL}(\varphi_h)$ // transform specification
  $\Sigma' \leftarrow \mathsf{NewSystemGen}(\Sigma, \varphi_h)$ // transform model
  $\chi_{min}, \Theta_f \leftarrow \mathsf{FalsifySTL}(\Sigma', \varphi_{STL}, \Theta, T, N)$
  if $\chi_{min} < 0$ then
    return $\Theta_f$
  end
end
```

- ☐ We then call a function NewSystemGen to generate a new model that contains copies of the original system. The number of copies is equal to the number of quantifiers of the formula $\varphi_h$.

- ☐ Then, we apply existing falsification mechanisms for an STL formula such as Breach[4] [49] to compute the minimum robustness value $\chi_{min}$ of the system $\Sigma'$ according to $\varphi_{STL}$. Breach allows us to parametrically generate different input signals over a parameter space. For example, parameters can represent control points, and an input signal can be created using interpolation between these points. If $\chi_{min}$ is negative we return the optimal set of parameters $\Theta_f \in \Theta$ that produces a falsifying behavior.

[4]Breach [49] is a tool that applies a best-effort approach to automatically check whether a system satisfies a given STL formula.

We note that, unlike formal verification, performing falsification cannot ensure a system is always safe; even if falsification fails to identify a falsifying behavior, a counterexample may still exist.

**Example 11** *Consider a mechanical mass-spring damper system whose dynamics are defined by the second-order ordinary differential equation:*

$$\ddot{x}(t) + 2\dot{x}(t) + 5x(t) = 3F(t), \tag{3.9}$$

*where $x$ is the vertical position of the mass, and $F$ is the random external force. The robust behavior hyperproperty of the system is specified as follows: for all pairs of traces of the system with the external force difference less than $\epsilon_1$ , the output difference should be bounded by $\epsilon_2$; here $\epsilon_1 = 0.2$ and $\epsilon_2 = 0.3$. We apply the Algorithm 1 to falsify the robust behavior hyperproperty for the system with a duration $T = 10$ seconds. Formula 3.6 can be reduced to the normal STL formula as follows:*

$$\phi_M \triangleq \square_{[0,10]}(\rho_{in} \leq \epsilon_1 \implies \rho_{out} \leq \epsilon_2), \tag{3.10}$$

where a trace $\mathbf{p} \triangleq (\rho_{in}, \rho_{out})$ of the system $\Sigma'$ captures the input-output difference between two traces $\mathbf{w}, \mathbf{w}'$ of the original system $\Sigma'$, e.g., $\rho_{in}(t) = ||w_{in}(t) - w'_{in}(t)||$. Here, the system $\Sigma'$ contains two copies of the mechanical mass-spring damper system $\Sigma$. The falsification result shown in Figure 3.2 illustrates the inductive checking procedure for the satisfaction of Formula 3.10 using Breach, where $alw_{[0,10]}$ is equivalent to $\square_{[0,10]}$, and the left y-axis denotes robustness degree. Here, we observe that the violation of the robust behavior hyperproperty of the mechanical mass-spring damper system occurs during the overshoot period of the outputs of the system.

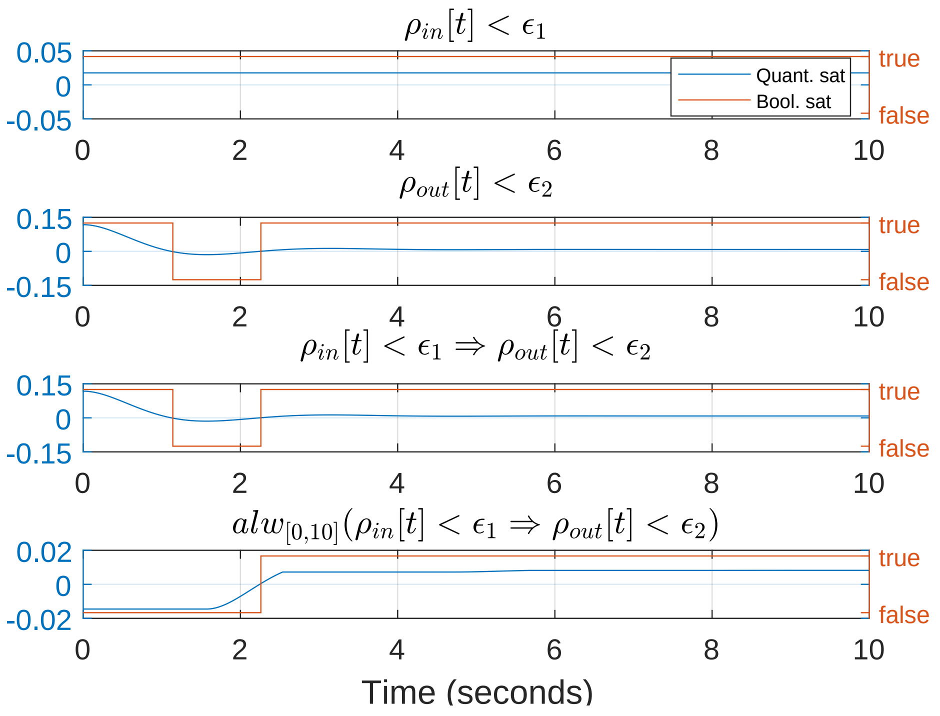


**Figure 3.2:** Falsification result of the mass-spring damper system. The counterexample pair of traces found by Breach for the robust behavior hyperproperty.

**Remark 12** *There is a duality between addressing the falsification problem of an alternation-free t-HyperSTL that only contains universal quantifiers and solving the verification problem of an alternation-free t-HyperSTL that only contains existential quantifiers. Given an alternation-free t-HyperSTL such as* $\exists \mathbf{v}.\exists \mathbf{v}'.\phi_e$*, our purpose is to extensively simulate a system and find a single pair of execution traces of the system that satisfies* $\phi_e$*. Here, we do not attempt to falsify the system, but verify the system. Thus, this process is dual to finding the falsifying traces of the system corresponding to the formula* $\forall \mathbf{v}.\forall \mathbf{v}'.\neg\phi_e$*. Also, we note that we can leverage Algorithm 1 such that it includes a parameter synthesis approach to mine hyperproperties for CPS, as in [73, 76]. For instance, we could use a requirement mining approach to automatically infer appropriate values for the* $\epsilon_1$ *and* $\epsilon_2$ *variables in Formula 3.10.*

## 3.6 Falsifying k-alternation t-HyperSTL

Falsifying k-alternation t-HyperSTL formulas is a challenging task, as it requires us to examine all execution traces of a system. Consider a 1-alternation t-HyperSTL formula such as $\exists \mathbf{v}.\forall \mathbf{v}'.\phi$; falsifying a system for this property is as hard as verifying the system, since we need to show that for all traces $\mathbf{w} \in S$, there exists a trace $\mathbf{w}'$ that the formula $\phi$ is violated, where $S$ is an infinite set of traces. It is even more difficult to perform falsification for CPS whose dynamics evolve continuously over time. Furthermore, if a hyperproperty contains more than one alternation of quantifiers (e.g. the Lyapunov stability property), the falsifying algorithm may suffer an exponential growth in complexity. Despite this, if we assume a CPS can be modeled by a finite set of traces, we can develop a falsifying algorithm for the system that can prove or disprove $\phi$.

In general, there may not exist a unique answer to the question of whether we can verify or falsify a system with respect to the formula $\exists \mathbf{v}.\forall \mathbf{v}'.\phi$ using finite simulations. We can consider several possible answers for that question as follows.

- ☐ **Case 1**: if both $\mathbf{w}, \mathbf{w}'$ belong to some infinite set of traces, then we can neither verify nor falsify $\phi$.

- ☐ **Case 2**: if $\mathbf{w}$ belongs to an infinite set of traces and $\mathbf{w}'$ belongs to a finite set of traces, then we cannot falsify but we can verify $\phi$.

- ☐ **Case 3**: if $\mathbf{w}$ belongs to a finite set of traces and $\mathbf{w}'$ belongs to an infinite set of traces, then we cannot verify but we can falsify $\phi$.

- ☐ **Case 4**: If both $\mathbf{w}$ and $\mathbf{w}'$ belong to a finite set of $n$ traces, we are able to verify the system with $n$ simulations as well as falsify the system with $\frac{n(n-1)}{2}$ simulations.

We note that in all of the cases that we are able to falsify the system corresponding to the formula $\exists \mathbf{v}.\forall \mathbf{v}'.\phi$ with finite simulations, we can apply Algorithm 1 to transform the falsification problem to another equivalent problem that uses a traditional STL specification. The falsification procedure is similar to solving the falsification problem of alternation-free t-HyperSTL.

For the case that both execution traces of a system, $\mathbf{w}$ and $\mathbf{w}'$, belong to some infinite sets, and if we have a verification oracle to address the last quantifier (e.g., by conservatively estimating the set of possible system behaviors, under certain conditions), we can either falsify or verify the system. Given a set of initial states, a verification oracle can be a method that mathematically overapproximates the reachable set of the system or a simulation-based technique [2, 64] that may verify the system with finite simulations.

Alternatively, for a hyperproperty that requires two or more alternations of quantifiers to express, even if we have a verification oracle corresponding to the last quantifier, we can neither falsify nor verify a system. Using a verification oracle, the feasibility of addressing the falsification and verification problems associated with a $k$-alternation t-HyperSTL formula is equivalent to that of a $(k-1)$-alternation t-HyperSTL formula; this is shown in Table 3.1. We emphasize that any hyperproperties for general CPS that are as complex as, or more complicated than Lyapunov stability, are not verifiable or falsifiable without reasonable restrictions on sets of execution traces.

| Type | A1: Finite Simulation | | A2 : Verification Oracle on the Last Quantifier |
|---|---|---|---|
| | Falsification | Verification | |
| $\forall$ | Yes | No | - |
| $\exists$ | No | Yes | - |
| $\forall\exists$ | No | No | $\forall$ |
| $\exists\forall$ | No | No | $\exists$ |
| $\forall\exists\forall$ | No | No | $\forall\exists$ |
| $\exists\forall\exists$ | No | No | $\exists\forall$ |

**Table 3.1:** Feasibility of solving the falsification and verification problems for properties and hyperproperties expressed using STL and k-alternation t-HyperSTL under two assumptions: A1) using finite simulation and A2) applying a verification oracle that can do reachability analysis with respect to the last quantifier.

## 3.7 Case study

In this section, we introduce two proof-of-concept case studies in the domain of automotive control systems: a) an industrial-scale Simulink model of a closed-loop airpath control (APC) system and b) a Simulink model of a fault-tolerant fuel (FTF) control system. We will demonstrate how to apply the testing framework of HyperSTL built on top of Breach to falsify the robust behavior hyperproperty of the APC system, and the robust control invariance hyperproperty of the FTF system under FDI attacks.

### 3.7.1 Airpath Control Model

We use a prototype APC system to evaluate the capability of our proposed method on an industrial control system. The APC is a key subsystem for a hydrogen Fuel-Cell (FC) vehicle powertrain. The purpose of the APC is to regulate the air flow rate into the FC stack using multiple actuators. The FC stack generates electrical power for the vehicle using a mixture of air and hydrogen. The FC stack only operates under restricted conditions, such as temperature, pressure and moisture level within the stack. An excess of moisture in the stack will impede the performance while moisture deficiency could permanently damage the FC stack. Thus, to achieve high performance while still operating the system in a safe regime, the controller is required to accurately regulate the air flow rate.

The closed-loop Simulink model of the APC system is complex; it contains more than 7,000 Simulink blocks such as integrators, saturations, S-Function blocks, lookup tables, and data store memory blocks. The model has two input signals including i) the ambient temperature and ii) the fuel cell current request (FCI). Details of the system, such as units and expected signal ranges, are suppressed due to proprietary concerns. Intuitively, an FCI value is proportional to the desired torque requested by the driver, which is ultimately based on the accelerator pedal angle. The output of the APC system is an air flow rate (AFR). The purpose of the controller model is to regulate the AFR to some desirable reference value. To ensure the APC system works properly, for some small perturbations of the ambient temperature and FCI values, the differences in AFR values should be bounded within a desirable range. In other words, to avoid unexpected changes in the air flow rate at the inlet of an FC stack, which may cause undesirable behavior, the system should satisfy the

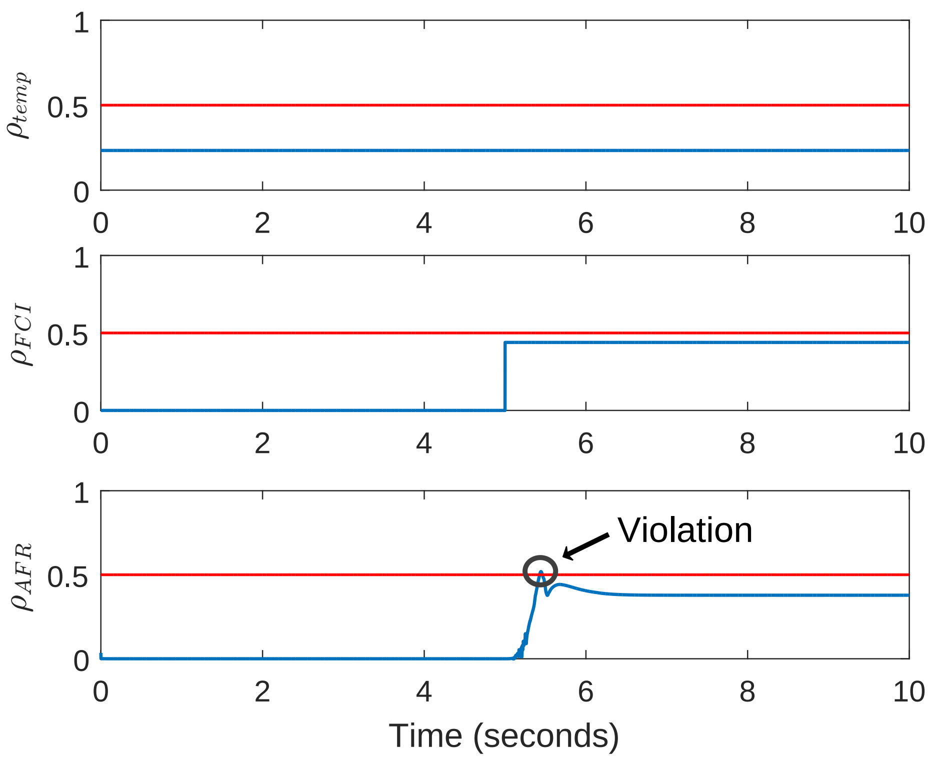


**Figure 3.3:** Falsification result of the APC system. The counterexample pair of traces found by Breach for the robust behavior hyperpropperty.

robust behavior hyperproperty. The robust behavior hyperproperty of the APC system can be formalized as follows,

$$\begin{aligned}\phi_{APC} \triangleq \{W \in P \mid \forall \mathbf{w}, \mathbf{w}' \in W : (d_{sup}(w_{temp}, w'_{temp}) \leq \epsilon_1 \wedge d_{sup}(w_{FCI}, w'_{FCI}) \leq \epsilon_2)\\ \implies d_{sup}(w_{AFR}, w'_{AFR}) \leq \epsilon_3)\},\end{aligned} \tag{3.11}$$

which can be translated to the following STL formula using Algorithm 1 to perform the falsification task,

$$\phi'_{APC} \triangleq \square_{[0,T]}((\rho_{temp} \leq \epsilon_1 \wedge \rho_{FCI} \leq \epsilon_2) \implies \rho_{AFR} \leq \epsilon_3), \tag{3.12}$$

where a trace $\mathbf{w}$ is composed of the temperature and FCI input signals $w_{temp}$ and $w_{FCI}$ respectively, and the AFR output signal $w_{AFR}$. Here, we create a new model including two copies of the original APC system; and a trace $\mathbf{p} \triangleq (\rho_{temp}, \rho_{FCI}, \rho_{AFR})$ of the new model captures the input-output difference between two traces $\mathbf{w}, \mathbf{w}'$ of the original model, for instance, $\rho_{temp}(t) = ||w_{temp}(t) - w'_{temp}(t)||$.

The result of falsification of the robust behavior hyperpropety of the APC system is shown in Figure 3.3, where the blue lines present the distance signals $\rho_{temp}$, $\rho_{FCI}$, $\rho_{AFR}$ respectively, and the red lines demonstrate their corresponding bounds. Here, the parameter values selected by a design engineer are normalized to 0.5. That is, $\epsilon_1 = 0.5$, $\epsilon_2 = 0.5$, and $\epsilon_3 = 0.5$. The sampling time is 0.001024 seconds and the simulation time $T$ is 10 seconds. For proprietary reasons, we normalize the quantities and suppress the units for the data shown in the figure. The counterexample pairs of traces reported by Breach demonstrate a behavior where the output difference exceeds its allowed bounds when the input differences are still less than their given thresholds, which is a violation of Formula 3.12. Finding this counterexample is significant, as it can help automotive control engineers to improve the controller design to eliminate such an undesirable behavior of the APC system.

### 3.7.2 Fault-tolerant Fuel Model

We consider a fault-tolerant fuel (FTF) model that includes both Simulink blocks and Stateflow charts[5]. The model has two external input signals, engine speed and throttle command, and one output signal, which is the effective air-fuel ratio inside the combustion chamber. The model also contains four sensors measuring throttle angle, engine speed, the amount of residual oxygen in the exhaust gas (EGO), and the manifold absolute pressure

[5]We use a modified version of the FTF model available at https://www.mathworks.com/help/simulink/examples/modeling-a-fault-tolerant-fuel-control-system.html

(MAP). The controller has three different control strategies: a normal operation mode, which is used when no sensor faults are present, a fault mitigation mode, which is used when one sensor fault has occurred, and a mode that disables fuel control, which is used when two or more sensor faults are detected. We only consider the normal and fault mitigation modes for this example. The goal of the controller is to regulate the air-fuel ratio output, denoted as $\lambda$, so that it remains within a desirable range, despite a failure in at most one sensor.

In this case study, we evaluate the ability of the FTF controller to tolerate an engine speed sensor fault. In the original version of the model, a speed sensor fault consists of the speed sensor output being set to 0.0 rad/sec; the controller detects the fault when the sensor reading equals 0.0. In the modified version that we use, we do not fix the controller mode based on the sensor reading, but instead we evaluate the controller performance when either the normal or fault mitigation modes are selected. In the modified version of the model that we use, a speed sensor fault consists of a sensor output producing a fixed but randomly selected value in the sensor range $[0, 620]$ rad/sec. This kind of sensor fault could occur when an attacker uses a sensor spoofing approach to inject incorrect measurements into the sensor readings or when a real fault occurs in the speed sensor. We use the robust control invariance property to specify desired controller performance in the presence of the indicated class of sensor faults:

$$\phi_{FTF} \triangleq \exists \mathbf{v}.\forall \mathbf{v}'.\Box_{[\tau,\infty]}(d_{sup}(v_u, v'_u) = 0 \implies 0.8\lambda_{ref} \leq v'_\lambda \leq 1.2\lambda_{ref}), \tag{3.13}$$

where $\lambda_{ref}$ is the reference value of the air-fuel ratio $\lambda$, and $\tau$ is the settling time. Here, a trace variable $\mathbf{v}$ can be mapped to a trace $\mathbf{w}$ composed of the controller input $w_u$ corresponding to a controller mode decision, a disturbance $w_d$ representing the fixed random sensor input injected into the speed sensor, and an output $w_\lambda$. In general, we cannot falsify Formula 3.13 according to the discussion shown in Table 3.1; however, for systems like the FTF model that have a finite set of control strategies, we can effectively perform falsification by creating a new model that contains copies of the original system, one copy for each control mode (two copies, in this case). The external input (the speed sensor reading) is connected to each of the copies of the model. The specification $\phi_{FTF}$ is converted to the following equivalent formula in standard STL:

$$\hat{\phi}_{FTF} \triangleq \forall w_d . \Box_{[\tau,\infty]}(0.8\lambda_{ref} \leq w_{\lambda_1} \leq 1.2\lambda_{ref}) \vee \Box_{[\tau,\infty]}(0.8\lambda_{ref} \leq w_{\lambda_2} \leq 1.2\lambda_{ref}), \qquad (3.14)$$

where $w_{\lambda_1}$ and $w_{\lambda_2}$ are the air-fuel ratios of the first and second copies of the model. We note that Formula 3.14 is arrived at by applying the quantitative semantics provided in Sec. 3.4; the disjunction in Formula 3.14 appears due to the $\exists$ quantifier in Formula 3.13, which effectively applies a max operator over the two available control modes. The formula $\hat{\phi}_{FTF}$ can be tested using the falsification methods for traditional STL available in Breach.

Figure 3.4 illustrates the falsification result of the FTF model. The blue lines correspond to a simulation trace representing the falsifying behavior, the green line illustrates an instance of the correct speed, and the red lines represent the error bound of $\lambda$, where $\tau = 10$ seconds, $T = 50$ seconds, and $\lambda_{ref} = 14.6$. Based on the results, we can conclude that there exists a trace, which includes outputs $w_{\lambda_1}$ and $w_{\lambda_2}$ that both evolve beyond the tolerance bound regardless of whether the controller operates in the normal mode or the fault mitigation mode (i.e., the performance requirement is violated despite which control mode is used). This experiment demonstrates the capability of using a falsification approach to automatically test hyperproperties for CPS.

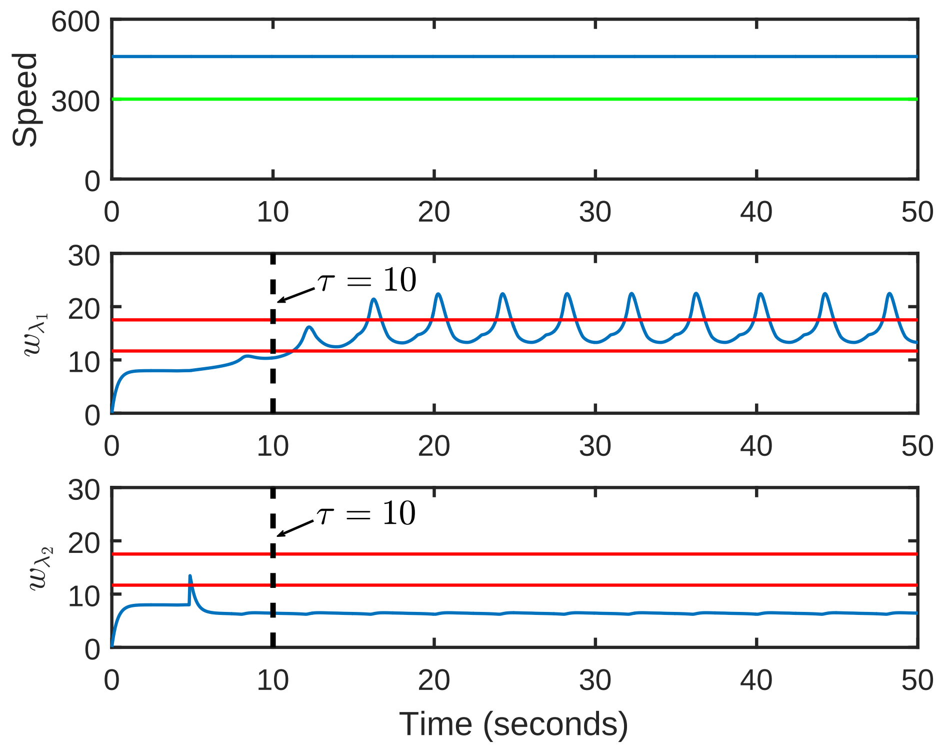


**Figure 3.4:** A pair of falsifying traces found by Breach illustrating the FTF model cannot tolerate the fault under a speed sensor fault.

## 3.8 Conclusion

In this chapter, we represented the first study of the hyperproperties of CPS. We defined a new temporal logic, called HyperSTL, to express several hyperproperties including stability, security, and safety for CPS. HyperSTL allows us to effectively specify more general requirements of CPS rather than STL as it can express the relationships between multiple execution traces. The testing framework of t-HyperSTL, a fragment of HyperSTL, was also given and applied to falsify the robust behavior hyperproperty of a hydrogen fuel-cell power-train model, and the robust control invariance hyperproperty of the fuel control model under a fault data injection attack. We also discuss the feasibility of performing the falsification and verification for various classes of hyperproperties for CPS.

# Chapter 4

# ABNORMAL DATA CLASSIFICATION USING TIME-FREQUENCY LOGIC

## 4.1 Introduction

For the last decade, signal temporal logic (STL) [95] has been successfully extended and applied in many domains such as exploring requirements for closed-loop control systems [76], identifying oscillatory behaviors of biology systems [46], and formalizing and recognizing music melodies [51]. Recently, Kapinski et al. introduced a new signal library template for constructing formal requirements of automotive control applications using STL [81]. These requirements involve various control signal behaviors such as settling time, overshoot, and steady state errors. Although most of such control signal behaviors can be characterized in the time domain, some abnormal signal behaviors such as *hunting* (undesirable oscillations) or *spikes* (abrupt, momentary jumps in signal values) are challenging to capture without frequency information. In most practical control systems, hunting behaviors are considered undesirable, or at least not ideal, and care is taken to minimize or eliminate the behavior. In signal processing, hunting behavior can manifest around sharp transitions, as a result of compression artifacts; this occurs, for example, in image processing, resulting in ghostly bands near edges, or in audio compression, resulting in forward echo problems. In circuit design, a hunting behavior can be the unwanted oscillation of an output current or voltage, which may cause a significant rise in power consumption, temperature, electromagnetic radiation, or settling time [77]. Although some hunting behaviors can be defined loosely as an oscillation around a given average and can be well captured using STL, some modulated hunting signals are challenging to detect using only time domain information [81]. Because hunting signals relate to oscillatory properties, it is appropriate to investigate them using time-frequency analysis.

The first attempt to introduce a specification formalism for both time and frequency properties of a signal, called time-frequency logic (TFL), was proposed by Donzé and his collaborators [51]. There, a signal is preprocessed using a Short-Time Fourier Transform (STFT) [39] to generate a spectral signal that represents the evolution of the STFT coefficients at some particular frequency over time. The time-frequency predicates and arithmetic expressions constructed from this spectral signal are added into an STL formula to yield a TFL formula. TFL was originally applied to music, though it can be easily extended to other application domains. A key limitation of the approach using the STFT is the inherent trade-off required between resolution in the time domain and resolution in the frequency domain; it is difficult or impossible to obtain satisfactory resolution in both time and frequency using the STFT for the analysis. Such limitations can be overcome using the continuous wavelet transform (CWT).

In the following, we extend the notion of TFL by using the CWT to specify and check time-frequency properties of signals. We introduce the concept of parametric time-frequency logic (PTFL) and use it to perform parameter synthesis for the purpose of classifying hunting behavior. Previous efforts have focused on data classification of time-series signals using STL [22, 31, 76], but identifying some abnormal behaviors such as hunting requires both time and frequency information [81]. Moreover, existing classification methods require an extensive amount of data, and the inferred classifier is often difficult for engineers to interpret. In contrast, our proposed method using PTFL can efficiently classify abnormal behaviors with an interpretable data classifier and requires less data than existing techniques. We note that although the below presentation is focused on one behavior type, it is straightforward to extend the work to detect other abnormal behaviors such as noise, spikes, or other anomalous behavior, in the time-frequency domain. We evaluate the proposed algorithm by comparing the performance against two existing classification techniques: a traditional machine learning technique using a support vector machine with a linear kernel, and a method that infers STL formulae as data classifiers [31]. To perform the evaluation, we use data sets from two different domains, the automotive and medical domains.

## 4.2 Time-Frequency Logic Using CWT

Although many control system behaviors can be naturally characterized in the time domain, there are some signal behaviors, such as hunting and spikes, that are challenging to capture without frequency information. This is especially true for non-stationary signals whose frequency components vary over time; for this class of signals, it is essential to analyze the signal properties in the time-frequency domain. Fourier transform (FT) breaks signals into series of sinusoidal components with different frequencies and phases, so it reveals what frequency components existing in signals. However, the shortcoming of the FT is that it is unable to associate features in the frequency domain with their locations in the time domain. Short-Time Fourier Transform (STFT) is a popular transformation that has been widely used in time-frequency analysis [39]. STFT partitions a signal into small segments (each segment is assumed to be stationary) whose lengths are equal to the width of a chosen window function. This function will modulate the signal to emphasize the time instant associated with each segment. Unfortunately, the STSF provides a fixed time-frequency resolution so that it is not effective for signals that need to be analyzed with different time-frequency resolutions [130]. Moreover, it is difficult to choose a proper window function with an appropriate size that not only provides both desirable time and frequency resolutions but also does not violate a stationary condition [130]. To overcome the limitation of the STFT, we use the CWT to analyze a signal in the time-frequency domain.

### 4.2.1 Continuous Wavelet Transform

The CWT of a signal $x(t)$ is formally defined as follows:

$$Wf(\zeta,\tau) = \int_{-\infty}^{+\infty} x(t)\psi^*_{\zeta,\tau}(t), \tag{4.1}$$

where $\psi^*_{\zeta,\tau}(t)$ is the complex conjugation of a basic wavelet function $\psi_{\zeta,\tau}(t)$ which is derived from a mother-wavelet function $\psi(t)$. This function has zero average in the time domain, i.e. $\int_{-\infty}^{+\infty}\psi(t)dt = 0$. Furthermore, a basic wavelet function $\psi_{\zeta,\tau}(t)$ can be written as:

$$\psi_{\zeta,\tau}(t) = \frac{1}{\sqrt{\zeta}}\psi\left(\frac{t-\tau}{\zeta}\right), \tag{4.2}$$

where $\zeta \in \mathbb{R}_{>0}$ is a scale parameter representing the width of the basic wavelet function, $\tau \in \mathbb{R}$ is a translation factor representing the location of the basic wavelet function, and $\frac{1}{\sqrt{\zeta}}$ is the energy normalization across different scales. Thus, the CWT maps an original signal to a function of $\zeta$ and $\tau$ that provides both time and frequency information. Note that the scale factor is inversely proportional to the frequency of a signal [130]. The CWT in Equation 4.1 measures the similarity between a basic wavelet function and a signal. Indeed, if a signal $x(t)$ has a frequency component $f$ corresponding to a particular scale $\zeta$ of a wavelet function $\psi_{\zeta,\tau}(t)$, then the portion of $x(t)$ at some particular time interval where $f$ exists will be similar to $\psi_{\zeta,\tau}(t)$. As a result, the CWT coefficients of $x(t)$ corresponding to $f$ will be relatively large over this time interval. Moreover, the time-frequency energy density of the CWT is equivalent to the square norm of the CWT coefficients:

$$P_W f(\zeta,\tau) = ||Wf(\zeta,\tau)||^2. \tag{4.3}$$

**Time-frequency resolution.** In contrast to the STFT, the CWT can either dilate or compress the window size of the wavelet function, and translate it along the time axis. The Heisenberg box [96] is a range of times and frequencies that indicates the accuracy of a time-frequency transformation. Although the area of the Heisenberg box does not change, the time and frequency resolutions can be varied depending on the value of $\zeta$. As a result, the CWT can analyze all frequency components within a signal by considering appropriate scales of the mother-wavelet function. For instance, the CWT can use the wavelet function with a short duration and low scale for analyzing high frequency components, and vice versa. This advantage of the CWT allows us to efficiently analyze a signal that includes abnormal behaviors such as spikes and hunting.

**Example 13** *Suppose we have a signal $x(t)$ composed from different sinusoid components*

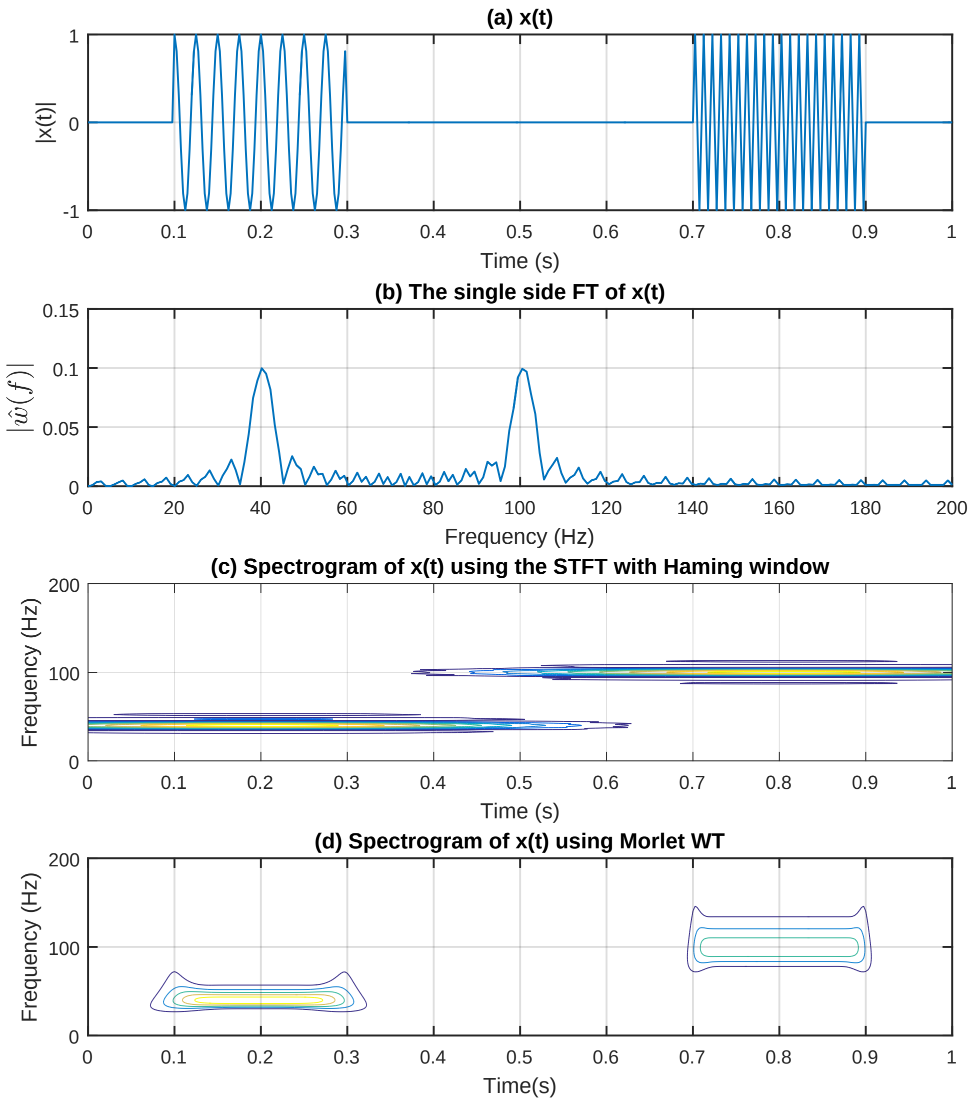


**Figure 4.1:** Plot showing the difference between the transformations of x(t) using the FT, STFT and CWT

*such that:*

$$x(t) = \begin{cases} cos(80\pi t), & if\ 0.1 \leq t < 0.3 \\ sin(200\pi t), & if\ 0.7 \leq t < 0.9 \\ 0, & otherwise \end{cases}$$

*Figure 4.1 illustrates the differences in representing a signal in the time-frequency domain* *between the FT, STFT, and CWT. Here, the FT provides us the exact dominant frequencies of the signal x(t) that are respectively equal to* $40Hz$ *and* $100Hz$*, but it does not reveal when these frequencies occur within x(t). The spectrogram (representing the square norm of the transform coefficients in the time-frequency domain) of x(t) using the STFT with Hanning window function [66] is shown in Figure 4.1(c). The STFT can localize the dominant frequency bandwidths of x(t) occurring over time. However, the time resolution is very poor as the time intervals including these dominant frequencies are overlapped each other. On the other hand, Figure 4.1(d) shows that using the Morlex WT [39] can give us a much better time resolution with a slightly lower frequency resolution compared to the STFT. Overall, x(t) has a better time-frequency presentation using the Morlex WT rather than the STFT. Especially, this advantage of the CWT is essential in specifying the signal properties over some precise time intervals using temporal logic.*

### 4.2.2 Time-Frequency Logic

TFL is an extension of STL that can be used to specify both time and frequency properties of a signal [51]. In TFL, a signal predicate is defined over the signal representing the evolution of the STFT coefficient at a particular frequency over time. Given a pair $(f, \tau)$ of frequency and time, the STFT of a signal $x(t)$ is obtained by:

$$S_{f,\tau} = \int_{-\infty}^{+\infty} x(t)\psi_L(t-\tau)e^{-2i\pi ft}dt\,, \tag{4.4}$$

where $\psi_L(t)$ is a window function. A *spectral signal* $y(t) = ||S_{f,t}||^2$ is the projection of the spectrogram of $x(t)$ on a particular frequency $f$. Such a signal can be incorporated in TFL formulae to form some interesting time-frequency specifications. We can see that a TFL formula is actually an STL formula in which the signal predicate is defined over $y(t)$ instead of $x(t)$. TFL has been used to formalize and recognize music melodies, where time-frequency requirements are simply specified as $\varphi \triangleq ||S_{f_p,t}||^2 > \theta$, where $f_p$ is the pitch frequency and $\theta$ is the STFT coefficient threshold [51]; however, the shortcomings of the STFT mentioned previously may reduce the ability of TFL to precisely specify and evaluate time-frequency properties of a signal. We extend TFL to use the CWT to obtain spectral signals from a given time-series signal. In effect, we construct a TFL formula based on the CWT coefficients of the spectral signals instead of the STFT coefficients. Because the CWT can appropriately use various scaling factors, $\zeta$, to analyze all frequency components at different time intervals, it gives us an ability to study signals at flexible time-frequency resolutions.

Although the following presentation focuses on the classification of hunting behaviors, we note that the proposed approach using TFL and CWT can be used to capture other time-frequency specifications as well. For instance, consider the property: "*For some time in the future, the dominant frequency of the signal is $\omega$ for 5 time units, and the dominant frequency subsequently rises to twice of this value within 10-time units.*" Here, the *dominant frequency*, $f(t)$, of a signal $x(t)$ is defined as the frequency corresponding to the maximum magnitude frequency component of the signal at time $t$, as provided by a CWT. Such a time-frequency property can be written as a TFL formula, $\varphi \triangleq \Diamond(\Box_{[0,5]}(f = \omega) \wedge \Diamond_{[5,15]}(f = 2\omega))$. Then, the TFL formula $\varphi$ can be evaluated as a normal STL formula using Breach[1] [49]. Consider another property such as "*At some time in the future the energy densities of the signal within a particular time interval and a particular frequency bandwidth are always greater than some threshold value $\theta$.*" This property can be specified as a TFL formula,

[1]Breach [49] is a tool that allows evaluation of STL and TFL formulae on signals.

$\phi \triangleq \Diamond\Box_{[t_1,t_2]}(z(f,t) > \theta)$, where $z(f,t)$ is a spectral signal that captures the minimum value of the CWT coefficients of a signal over some frequency bandwidth $[f_1, f_2]$.

**Parametric Time-Frequency Logic.** We introduce parametric time-frequency logic (PTFL), which is an extension of TFL where the parameters in TFL template formulae are symbolic parameters. Similar to the concept of parameter signal temporal logic (PSTL) introduced in [15], PTFL allows constants in intervals bounding the temporal operators and constant values in the predicates of PTFL formulae to be replaced with parameters.

The $p$ parameters in a PTFL formula are classified into two sets:

(a) $\Upsilon = \{\tau_1, ..., \tau_{p_t}\}$ is a set of $p_t$ *time* parameters occurring in the time intervals of the temporal operators, and

(b) $\Theta = \{\theta_1, ..., \theta_{p-p_t}\}$ is a set of $p - p_t$ *threshold* parameters occurring in the signal predicates.

For any fixed values of $\Upsilon$ and $\Theta$, a PTFL formula $\varphi(\tau_1, \dots, \tau_{p_t}, \theta_1, \dots, \theta_{p-p_t})$ yields a TFL formula corresponding to the fixed values of the parameters. For instance, consider a PTFL formula $\varphi(\tau, \theta) \triangleq \Box_{[0,\tau]}(y(t) > \theta)$, where $y(t)$ is a spectral signal, $\tau$ and $\theta$ are time and threshold parameters, respectively. The formula $\varphi(5, 10)$ is defined as the TFL formula $\Box_{[0,5]}(y(t) > 10)$.

## 4.3 Hunting Classification

In this section, we will describe three different approaches using PTFL and TFL to efficiently classify hunting behaviors in signals. Informally, a hunting behavior is an undesirable oscillation appearing within a signal over some time interval.

### 4.3.1 Parameter Synthesis Approach

We now propose a method to classify hunting behavior based on mining parameters of the following PTFL formula:

$$\varphi_h \triangleq \bigwedge_{i=1}^{m} \Diamond_{[0,\tau_i]}(Wf_i(t) > \theta_i). \tag{4.5}$$

Intuitively, this formula specifies that "*the energy densities of the given signal at particular frequencies are eventually greater than some threshold value*". Here, $Wf_i(t)$ is a spectral signal over time that captures the energy densities of the CWT of an original time-series signal $x(t)$ at a particular frequency $f_i \in F$. Note that $F$ is a set of frequencies based on the scales of the CWT. Each spectral signal, $Wf_i(t)$, is the row vector of the matrix representing the energy densities of the CWT of $x(t)$; such a matrix is obtained using Equation 4.1 and Equation 4.3. Also, $\tau_i \in \Upsilon$ and $\theta_i \in \Theta$ denote a time and threshold parameter corresponding to each spectral signal $Wf_i(t)$. We note that the satisfaction value of the property $\varphi_h$ monotonically increases in $\tau_i$ and decreases in $\theta_i$. Because of monotonicity, we can exponentially reduce the search over the parameter space so that the synthesis procedure is efficient [76]. Figure 4.2 conceptually illustrates a spectral signal $Wf_i(t)$, and an instance of a hunting behavior that may occur within a signal. We say that a signal $x(t)$ contains hunting behavior if the property $\varphi_h$ holds. Overall, the hunting classification problem can be written as follows.

- **Given** the following inputs:
  - a set of labeled traces $\Psi \triangleq \{\Psi_\alpha, \Psi_\beta\}$, where $\Psi_\alpha$ and $\Psi_\beta$ denote a set of training and testing traces, respectively. Moreover, we the notation $\Psi.B$ and $\Psi.G$ to respectively denote the set of traces with and without hunting behavior. Note that all traces in

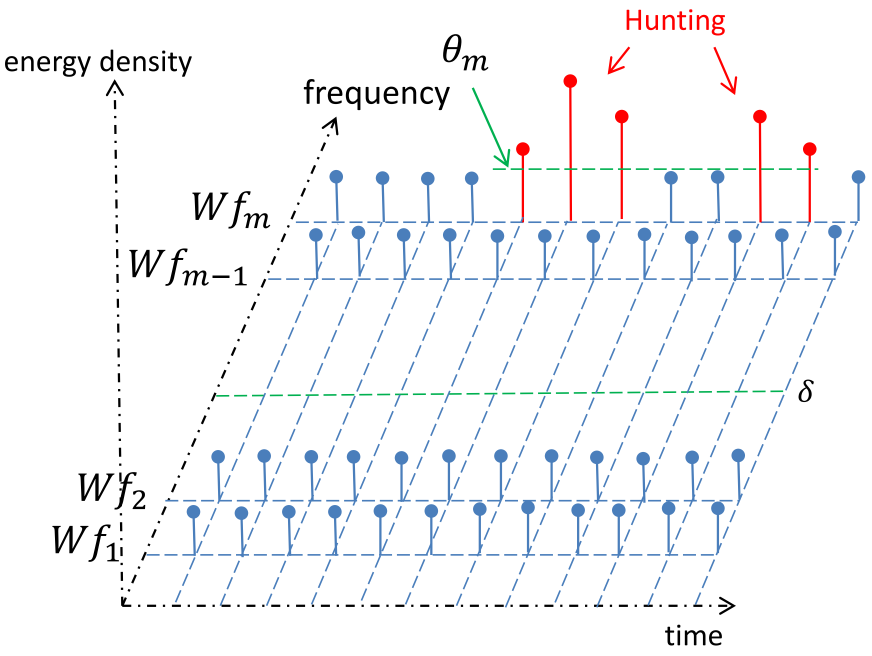


**Figure 4.2:** A sketch illustrates the hunting classification problem using time-frequency parameter synthesis. The set of spectral signals $Wf_i$ is acquired from the CWT of an original time-series signal.

the training set exhibit hunting behavior, so that $\Psi_\alpha = \Psi_\alpha.B$

- ◦ a cut-off frequency $\delta$.
- ◦ sets of parameters $\Upsilon$, and $\Theta$.

- **Find** values for $\Upsilon$ and $\Theta$, such that:
  - ◦ $x_j(t) \models \varphi_h(\Upsilon, \Theta)$ for all $x_j(t) \in \Psi_\beta.B$.
  - ◦ $x_j(t) \not\models \varphi_h(\Upsilon, \Theta)$ for all $x_j(t) \in \Psi_\beta.G$.

We introduce the cut-off frequency $\delta$ to reduce the effort to exhaustively mine parameters over the entire time-frequency domain. It is essential for the control engineers to indicate that hunting behavior only occurs at some high-frequency region above $\delta$.

**Classification Algorithm.** Next, we propose a heuristic to automatically obtain values for

$\Upsilon$ and $\Theta$ that can be used to separate the hunting and non-hunting signals. An overview of the heuristic is described in Algorithm 2. The heuristic can be interpreted as follows.

Line 2 initializes a matrix $\Sigma$ that represents the $k$ m-dimensional spectral signals transformed from $k$ original time-series signals in the training set using the CWT. We iterate over each trace in $\Psi_\alpha$ to construct sets of spectral signals $\{Wf_1(t), ..., Wf_m(t)\}$ using the CWT, and assign them to $\Sigma$. Next, we call the function TruncateParam to reduce the effort of exhaustively mining all parameters over the entire time-frequency domain. Here, $\Sigma'$ represents the $k$ n-dimensional $(n < m)$ matrix of $\Sigma$ corresponding to the frequency range above $\delta$. Next, we call the function HuntingParamSyn incorporated inside Breach to mine values for $\Upsilon$ and $\Theta$. Then, we test the classifier with a given set of testing traces $\Psi_\beta$. The function Classifier checks the satisfaction of $\varphi_h$ for each trace in $\Psi_\beta$, and returns the misclassification rate (MCR) value and the set of misclassified traces $\Psi_m$. The values of $\Upsilon$, $\Theta$ and the set $\Psi_m$ are then returned for further analysis. Furthermore, we can call EnhancedParam function to strengthen the values $\Upsilon$ and $\Theta$ and reduce the MCR value for the purpose of optimizing the classifier formula. Note that in the case studies, we do not use this function to evaluate the performance of the classifier to avoid the bias in our comparative analysis.

### 4.3.2 Decision Tree Approach

An approach based on decision trees to classify time series data using STL formulae was implemented in the tool DT4STL [31]. That method uses a parameterized procedure to infer STL formulae from labeled data. Given a two-class training data and a set of PSTL templates, a decision tree for classification is recursively built such that each node of a tree is

**Algorithm 2** Hunting Classification Using Parameter Synthesis

```
function HuntingClassification($\Psi_\alpha, \Psi_\beta, \delta$)
  $\Sigma \leftarrow 0$
  for each trace $x_j(t) \in \Psi_\alpha$, $j \leq k$
    $\Sigma(j,:,:) \leftarrow Wf_1(t), ..., Wf_m(t) \leftarrow CWT(x_j(t))$
  end for
  $\Sigma' \leftarrow$ TruncateParam($\delta, \Sigma$)
  $\Upsilon, \Theta \leftarrow$ HuntingParamSyn($\Sigma'$)
  $MCR, \Psi_m \leftarrow$ Classifier($\Upsilon, \Theta, \Psi_\beta$)
  return $\Upsilon$, $\Theta$, $\Psi_m$
end function
function EnhancedParam($\Psi_m, \Psi_\alpha, \Psi_\beta, \delta$)
  if $\Psi_m.B \neq \emptyset$ then
    $\Psi'_\alpha \leftarrow \Psi_\alpha \cup \Psi_m.B$
    HuntingClassification($\Psi'_\alpha, \Psi_\beta, \delta$)
  end if
end function
```

associated with a simple formula, selected from the given PSTL templates. The parameter synthesis is then conducted to find the STL formula that yields the best split for the data at each node. This technique can be used to automatically construct classifiers based on STL formula, but to achieve a low MCR value, the inferred STL formulae may be long and not easily interpretable by engineers. In this section, we apply this approach to classify hunting versus non-hunting signals. Instead of inferring an STL formula, we intend to infer a TFL formula as a data classifier. Thus, we need to transform original time series data into a collection of time-frequency data (spectral signals).

We assume that control engineers initially designate the frequency threshold separating hunting versus non-hunting behavior. A hunting behavior is specified as any oscillatory behavior occurring at frequencies above some specified cut-off frequency $\delta$. Thus, the time-frequency profile of a hunting signal at some frequency component $f > \delta$ contains larger values for the CWT coefficients compared to those of non-hunting signals. So we define the spectral signal WThcoef based on the CWT coefficients of the signal in a high-frequency region such that:

$$\mathsf{WThcoef}(t) = \max_{\zeta \in [\frac{f_c}{T_s F_{max}}, \frac{f_c}{T_s \delta})} P_W f(\zeta, t), \tag{4.6}$$

where $f_c$ is a center frequency associated with the mother-wavelet function, $F_{max}$ is the maximum frequency that appears in the CWT, and $T_s$ is the sampling period. We use such a spectral signal as an input for the DT4STL to infer a simple TFL formula. Note that in this scenario, the inferred TFL formula captures the non-hunting behavior of a signal.

### 4.3.3 Support Vector Machine Approach

Next, we present another approach that can solve the problem of hunting classification: linear classification using support vector machines (SVM) [137]. A linear SVM is a set of hyperplanes or decision boundaries that can correctly separate data into two classes. The general form of hyperplanes is $\langle w \cdot x \rangle + b = 0$, where $w$ is a normal to the hyperplane, and $\frac{b}{||w||}$ is the perpendicular distance from the hyperplane to the origin. The sign of the linear discriminant function $f(x) \triangleq \langle w \cdot x \rangle + b$ determines on which side of the decision boundary the test data point is located. The distance from the decision boundary to the closest data point determines the *margin* of the linear classifier. Suppose that we have a set of $n$ labeled training data $(x_i, c_i), ..., (x_n, c_n)$ where $x_i \in \mathbb{R}^d$ and $c_i \in \{1, -1\}$, the constrained optimization problem of linear classification using SVM is written as:

$$\begin{aligned} &\underset{w,b}{\text{minimize}} && \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\zeta_i \\ &\text{subject to} && c_i(\langle w \cdot x_i \rangle + b) \geq 1 - \zeta_i,\ i = 1, \dots, n \\ & && \zeta_i \geq 0. \end{aligned} \tag{4.7}$$

Here, $\zeta$ is a slack variable. If $0 < \zeta \leq 1$, the data point lies somewhere between the margin and the correct side of hyperplane, and the data point is misclassified if $\zeta > 1$. $C$ is a regularization parameter that defines the trade-off between errors of the SVM on training data and margin maximization. A large value of $C$ results in the low possibility of misclassified training data points, because the optimization in Equation 4.7 will choose a narrow margin hyperplane that correctly separates training data points as much as possible. In contrast, a small value of $C$ will result in a large margin hyperplane, but it may yield a better result in terms of correctly separating testing data points. Due to space limitation, we will not discuss the formal optimization problem solved to obtain the SVM, but refer interested readers to [137]. In this work, instead of applying the linear SVM directly to original time series signals, we need to preprocess them to yield a corresponding set of time-frequency features. For each time-series signal $x(t)$, we collect a real-valued vector $W^{max} \triangleq [Wf_1^{max}, ..., Wf_m^{max}]$ such that each element $Wf_i^{max} \in W^{max}$ is the maximum value of a spectral signal $Wf_i(t)$. Such a vector will be used as a time-frequency feature to design the SVM.

## 4.4 Case Studies

In this section, we evaluate the capabilities of three different methods to classify hunting behavior for two case studies. The first case study is based on data from an air compressor motor speed (ACMS) system in a fuel cell (FC) vehicle application. The second case study is based on electrocardiogram (ECG) data. In both examples, we apply the Morlet CWT [96] to perform the time-frequency analysis on the time-series signals.

### 4.4.1 ACMS Data

The ACMS system uses a compressor to regulate the air intake of a hydrogen FC vehicle. An FC stack uses a mixture of air and hydrogen to generate electrical power for the vehicle. Accurate control of the compressor which translates to control of the quantities of hydrogen and oxygen (air) is required to achieve good performance and proper operation from the FC stack. Also, the water balance (moisture level) within the stack needs to be carefully regulated, which requires regulation of the air pressure at the inlet of the stack. The task of the ACMS system is to regulate air flow and air pressure delivered to the inlet of the FC stack.

We consider ACMS data from an FC vehicle application. Specifics of the data, such as units and descriptions of the measured quantities are omitted here for proprietary reasons. The ACMS data are partitioned into a collection of traces that are 100 seconds in length and are labeled as either good (the trace does not exhibit hunting behavior) or bad (the trace does exhibit hunting behavior). The ACMS data has a sampling period of 0.02 seconds. We note that the same training data is used for all of the evaluations, though the parameter

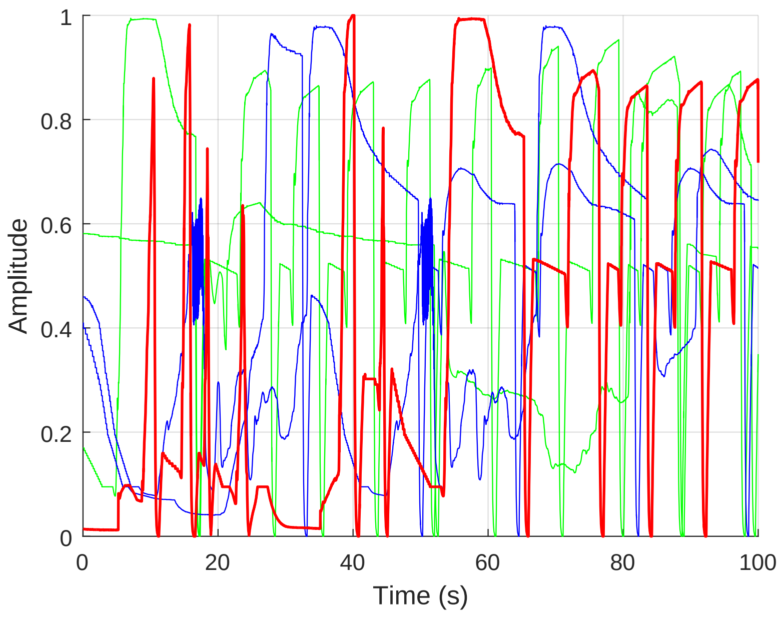


**Figure 4.3:** The classified testing data of the ACMS signals using parameter synthesis approach.

synthesis approach only uses the bad traces. In this experiment, we use the training data including 50 total traces, in which 30 traces are labeled as good and the others are labeled as bad. We also use the same testing data including 10 good traces and 10 bad traces for all of the evaluations.

**Parameter Synthesis.** We now illustrate the performance of the classification heuristic shown in Algorithm 2 to classify hunting behavior for the ACMS signals. Because we do not know the frequency range where a hunting behavior may occur, we exhaustively mine all parameters $\tau_i \in \Upsilon$ and $\theta_i \in \Theta$. We choose the maximum frequency of the CWT as $F_{max} = 25$Hz. Here, the Algorithm 2 will search for the best $\theta_i \in [0, 1]$ and $\tau_i \in [0, 100]$ such that all spectral signals transformed from original time-series traces in the training data satisfy $\varphi_h$. We then use Breach with the optimized parameters of $\varphi_h$ to classify good versus bad traces in the testing set.

Figure 4.3 shows the experimental results of classifying abnormal ACMS signals, using the function HuntingClassification. In the figure, we only show five representative signals in which good traces correctly classified are shown in green, and bad traces correctly classified are shown in blue. The one good trace that is misclassified is shown in red. The total running time of the classification process is approximately 3 minutes.

**Decision Tree Approach.** Next, we utilize the DT4STL toolbox to infer TFL formulae that can be used to classify hunting behavior for the ACMS data.

We preprocess the training data to yield the corresponding set of spectral signals WThcoef with $\delta = 15$Hz and $F_{max} = 25$Hz. We then run the DT4STL toolbox with this set of spectral signals using 2-fold cross-validation. As a result, we obtain the two following TFL formulae:

$$\varphi_{h1} \triangleq \Box_{[37.4,98.2)}(\mathsf{WThcoef} < 0.0435) \tag{4.8}$$

$$\varphi_{h2} \triangleq \Box_{[1.29,91.3)}(\mathsf{WThcoef} < 0.0394). \tag{4.9}$$

The procedure takes approximately 75 seconds to infer each formula. Using Breach, we then evaluate those formulae with the set of testing data. The formula $\varphi_{h1}$ gives us all misclassified traces that are bad traces with the MCR value being equal to 25%. On the other hand, the formula $\varphi_{h2}$ results in one misclassified trace, which is a bad trace.

**SVM Approach.** We apply the SVM method to classify normal versus abnormal ACMS data. We first transform all of the traces in the training data into sets of time-frequency features. Next, we run the linear SVM to learn the decision boundaries that separate data as either good or bad. Finally, we predict the testing data from the learned decision boundaries with different values of the SVM classifier margin $C$.

The MCR of the hunting classification for the ACMS data using SVM is 10% with $C = 10$ and reduces to 5% with $C = 100$. In this case, a larger value of $C$ gives a better result for

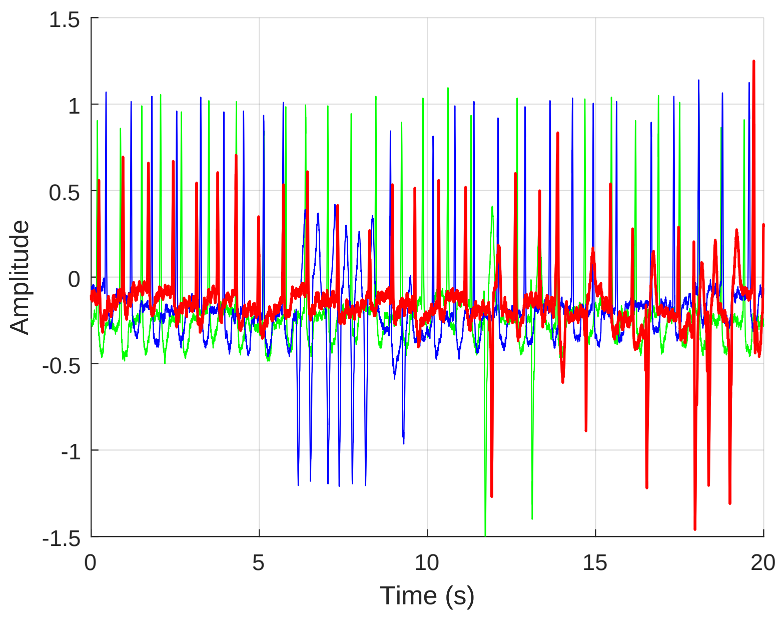


**Figure 4.4:** The classified testing data of the ECG signals using parameter synthesis approach.

the classification. Moreover, the classification process takes only 0.393 seconds.

### 4.4.2 ECG Data

An electrocardiogram (ECG) test is a noninvasive procedure used to monitor the electrical activities of a heart via a collection of electrodes attached to the patient's skin. A doctor can read an ECG output signal to diagnose abnormal structure or function of the patient's heart. A normal ECG signal includes three signals: (a) the P wave representing the depolarization or contraction of the atrium (b) the QRS complex (the R wave) indicating the ventricular depolarization and (c) the T wave describing the ventricular repolarization. The distance between two consecutive R peaks is considered as a heartbeat. A healthy patient has a resting normal heartbeat (frequency) from 60 to 100 beats per minute (bpm).

In this chapter, we focus on classifying the ECG signal that may contain a ventricular tachycardia (VT), a very fast heart rhythm arising in the ventricles that may cause a sudden heart failure. VT is defined as a sequence of three or more ventricular beats with the frequency varying from 110 to 250 bpm. Thus, a VT can be considered as a hunting behavior in an ECG signal. We conduct our classification approaches on the MIT-BIH Arrhythmia ECG Database. These data contain a variety of ECG signals collected from patients 23 to 89 years of age, including patients who experience ventricular arrhythmia [107]. We transform ECG signals 20 seconds in duration (provided at a sampling period of 0.0028 secs.) to spectral signals using the Morlet CWT. Here, the maximum frequency of the CWT is $F_{max} = 4.5$Hz ($\sim$ 270 bpm). For all of the evaluations, we use the same training data including 20 bad traces (the traces do contain a VT) and 40 good traces (the traces do not contain a VT), and the same testing data including 10 good traces and 10 bad traces.

**Parameter Synthesis.** In this scenario, we only mine the parameters for 20 bad traces in the training dataset. Here, we will search for the best $\theta_i \in [0, 5]$ and $\tau_i \in [0, 20]$. Figure 4.4 shows the experimental results of using the function HuntingClassification to classify abnormal ECG signals that contain VT. Here, we only show three signals for illustration. The approach results in one (5%) misclassified (red) trace, which is a bad trace. The total running time of the classification process is approximately 1 minute.

**Decision Tree Approach.** Next, we utilize the DT4STL toolbox to classify hunting behavior for the ECG data. We first preprocess the training data to yield the corresponding set of spectral signals WThcoef with $\delta = 1.5$Hz. Then, we run the DT4STL toolbox with this set of spectral signals using 2-fold cross-validation. As a result, we obtain two following

TFL formulae:

$$\phi_{h1} \triangleq \Box_{[1.73,17.3)}(\mathsf{WThcoef} < 3.16) \tag{4.10}$$

$$\phi_{h2} \triangleq \Box_{[2.36,20)}(\mathsf{WThcoef} < 3.21). \tag{4.11}$$

The procedure takes approximately 105 seconds to infer each formula. We then use Breach to evaluate these formulae with a set of spectral data acquired from the CWT of 10 good traces and 10 bad traces in the testing data. The MCR values of using $\phi_{h1}$ and $\phi_{h2}$ to classify these data are both equal to 5% (but misclassified traces are different).

**SVM Approach.** Finally, we apply the SVM approach to classify hunting in the ECG data. Note that we use the same training and testing data used for the other methods. The hunting classification of the ECG data using an SVM results in 5% MCR for all values of $C$ (the one misclassified trace is a bad trace), and the classification procedure takes 0.3 seconds.

## 4.5 Discussion

In this section, we discuss the trade-offs related to the three classification approaches presented above to classify normal versus abnormal signals. Table 4.1 shows an aggregate performance evaluation between the approaches in four different categories, including (a) the ability to interpret the structure and parameters used to define the classifier, (b) the computation time, (c) the capacity to localize where bad behavior occurs in a signal, and (d) the ability to correctly classify normal versus abnormal signals. Although the linear SVM can classify abnormal signals much faster and more accurately than the parameter synthesis and the decision tree approaches, the main drawback of this method is that it cannot reveal

|  | PS | DT4STL | SVM |
|---|---|---|---|
| Interpretation of data classifier | ○ | △ | × |
| Computation time | × | × | ○ |
| Bad behavior localization | ○ | ○ | × |
| Low misclassification rate | △ | △ | ○ |

**Table 4.1:** The comparison between parameter synthesis (PS) using PTFL, `DT4STL` toolbox using TFL, and linear SVM in classifying abnormal signals, where ○, △, × respectively denote good, ok, bad.

where the bad behavior occurs within a signal. We found that the decision tree approach can infer specifications that accurately classify data as either good or bad; however, it is not easy to interpret the inferred formula unless the user has some expertise about the input data. If a dataset is not homogeneous (i.e., both normal and abnormal signals are very different from each other), the `DT4STL` toolbox may infer a complicated formula that cannot be easily interpreted. The parameter synthesis using PTFL and the decision tree approach using TFL have similar performance except the former provides a clearer intuition about the classifier, as the temporal logic formula that results is usually simpler for the PTFL case. Overall, we conclude that a traditional machine learning technique such as the linear SVM is the best choice if the only goal is to classify data as either good or bad, and the most important thing is to select a proper feature on which to base the classification algorithm. Otherwise, if the designer additionally wishes to both understand the meaning of a data classifier and automatically localize where abnormal behaviors occur within a signal, we conclude that the parameter synthesis approach is the best option, as a simple temporal logic formula that defines the classifier results from the analysis.

## 4.6 Conclusion

In this chapter, we present the extension of TFL using CWT that can be used to specify and capture signal properties in the time-frequency domain. We propose the parameter synthesis algorithm using PTFL to classify hunting behavior appearing within a signal. Furthermore, we perform the comparison analysis between the proposed algorithm, the traditional machine learning techniques using SVM with linear kernel, and the work of inferring STL formula as data classifier. In fact, we apply these techniques to classify hunting behavior for the ACMS data provided by the Fuel-cell group at Toyota, and the ECG data extracted from the MIT-BIH Arrhythmia Database.

# Chapter 5

# DISSERTATION SUMMARY AND FUTURE WORKS

On the whole, this dissertation presented different approaches to determine formal specifications and facilitate the design and analysis of CPS. This chapter summarizes the contributions of those approaches and proposes a specific future research direction for each of them.

Chapter 2 formalizes a problem of cyber-physical specification mismatches arisen in the product evolution and upgrade process in CPS, and presents a dynamic analysis prototype tool Hynger to capture such mismatches. Our experimental results prove that Hynger, in conjunction with Daikon, can efficiently detect candidate invariants of two CPS case studies including a DC-to-DC power converter and an automotive control system. Overall, there are several directions for future research, including: $(a)$ extending the classes of invariants that may be inferred, particularly to nonlinear (polynomial) [121] and disjunctive/max-plus forms [123], potentially by integrating Daikon with techniques from Dig [122], $(b)$ runtime assurance and verification with real-time reachability of inferred invariants [19], $(c)$ improving and refining Hynger, particularly with regard to performance (such as using Daikon in the online mode with direct pipes between Hynger and Daikon, so that file I/O is minimized), and $(d)$ analyzing more industrial-scale CPS using Hynger.

In Chapter 3, we presented the first study of hyperproperties of real-valued signals that characterizes many CPS requirements including safety, security, and stability. We also introduced HyperSTL, which is a new extension of STL to specify a class of hyperproperties defined over multiple continuous-time traces of CPS. Furthermore, we provided a technique that automates the process of testing hyperproperties for CPS using HyperSTL. Our proposed methodology has been applied to specify and falsify several HyperSTL properties of industrial-scale automotive control systems. There are several directions for the future work of HyperSTL. We first plan to introduce a library of HyperSTL formulae that encapsulates different general classes of hyperproperties of CPS including those presented in this dissertation. Second, the falsification algorithm of HyperSTL proposed in this dissertation is incomplete as it relies on self-composition (i.e., making copies of a system) and only falsifies a restricted class of hyperproperties. Thus, extending the falsification algorithm to bypass self-composition to falsify more interesting hyperproperties is planned. Also, the monitoring algorithms of HyperLTL recently proposed in [5, 33] could be applied to HyperSTL.

Chapter 4 introduces another extension of STL using the continuous wavelet transform to specify time-frequency properties of signals in a time-frequency domain. We proposed a method using parametric time-frequency logic as an interpretable data classifier that can efficiently classify abnormal behaviors of continuous signals. We have successfully applied our method to perform anomaly detection on the data sets extracted from automotive and medical domains. For a future direction, we will conduct the parametrized procedure using PTFL to detect and classify other abnormal behaviors such as spike. We intend to generalize the technique using PTFL to classify and capture requirement dependencies between different sets of signal traces. For instance, a possible requirement we want to capture is that "*if there* *exist anomaly sudden changes in a signal $x_1$, a signal $x_2$ will then eventually exhibit some hunting behaviors in the near future*".

# BIBLIOGRAPHY

[1] Ariane 5 flight 501 failure, report by the inquiry board. Technical report, ESA Inquiry Board, Paris, France, July 1996.

[2] H. Abbas, B. Hoxha, G. Fainekos, and K. Ueda. Robustness-guided temporal logic testing and verification for stochastic cyber-physical systems. In *Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), 2014 IEEE 4th Annual International Conference on*, pages 1–6. IEEE, 2014.

[3] H. Abbas, H. Mittelmann, and G. Fainekos. Formal property verification in a conformance testing framework. In *Formal methods and models for codesign (memocode), 2014 twelfth acm/ieee international conference on*, pages 155–164. IEEE, 2014.

[4] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science*, 109:43 – 56, 2004.

[5] S. Agrawal and B. Bonakdarpour. Runtime verification of k-safety hyperproperties in hyperltl. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*, pages 239–252. IEEE, 2016.

[6] M. Al Faruque, F. Regazzoni, and M. Pajic. Design methodologies for securing cyber-physical systems. In *Proceedings of the 10th International Conference on Hardware/-Software Codesign and System Synthesis*, pages 30–36. IEEE Press, 2015.

[7] H. Alemzadeh, R. Iyer, Z. Kalbarczyk, and J. Raman. Analysis of safety-critical computer failures in medical devices. *Security Privacy, IEEE*, 11(4):14–26, 2013.

[8] M. W. Alford, J.-P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, and F. B. Schneider. *Distributed systems: methods and tools for specification. An advanced course.* Springer-Verlag New York, Inc., 1985.

[9] R. Alur. *Principles of cyber-physical systems.* MIT Press, 2015.

[10] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical computer science*, 138(1):3–34, 1995.

[11] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, Jan. 2003.

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

[13] R. Alur, A. Kanade, S. Ramesh, and K. Shashidhar. Symbolic analysis for improving simulation coverage of simulink/stateflow models. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 89–98. ACM, 2008.

[14] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011.

[15] E. Asarin, A. Donzé, O. Maler, and D. Nickovic. Parametric identification of temporal properties. In *Runtime Verification*, pages 147–160. Springer, 2011.

[16] M. I. Babar, M. Ramzan, and S. A. Ghayyur. Challenges and future trends in software requirements prioritization. In *Computer Networks and Information Technology (ICCNIT), 2011 International Conference on*, pages 319–324. IEEE, 2011.

[17] S. Bak, O. A. Beg, S. Bogomolov, T. T. Johnson, L. V. Nguyen, and C. Schilling. Hybrid automata: from verification to implementation. *International Journal on Software Tools for Technology Transfer*, pages 1–18, 2017.

[18] S. Bak, S. Bogomolov, and T. T. Johnson. Hyst: a source transformation and translation tool for hybrid automaton models. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 128–133. ACM, 2015.

[19] S. Bak, T. T. Johnson, M. Caccamo, and L. Sha. Real-time reachability for verified simplex design. In *IEEE Real-Time Systems Symposium (RTSS)*, Rome, Italy, Dec. 2014. IEEE Computer Society.

[20] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0, 2010.

[21] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 100–114. IEEE, 2004.

[22] E. Bartocci, L. Bortolussi, and G. Sanguinetti. Data-driven statistical learning of temporal logic properties. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 23–37. Springer, 2014.

[23] O. Beg, T. Johnson, and A. Davoudi. Detection of false-data injection attacks in cyber-physical dc microgrids. *IEEE Transactions on Industrial Informatics*, 2017.

[24] B. Beizer. *Software testing techniques (2nd ed.).* Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[25] S. Bensalem, M. Bozga, A. Legay, T.-H. Nguyen, J. Sifakis, and R. Yan. Component-based verification using incremental design and invariants. *Software & Systems Modeling*, pages 1–25, 2014.

[26] F. Bergero and E. Kofman. Powerdevs: a tool for hybrid system modeling and real-time simulation. *Simulation*, 87(1-2):113–132, 2011.

[27] F. Bernardini, M. Gheorghe, F. J. Romero-Campero, and N. Walkinshaw. A hybrid approach to modeling biological systems. In G. Eleftherakis, P. Kefalas, G. Paun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 4860 of *LNCS*, pages 138–159. Springer Berlin Heidelberg, 2007.

[28] N. P. Bhatia and G. P. Szegö. *Stability theory of dynamical systems.* Springer Science & Business Media, 2002.

[29] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In *Annual International Cryptology Conference*, pages 513–525. Springer, 1997.

[30] F. Blanchini. Survey paper: Set invariance in control. *Automatica*, 35(11):1747–1767, Nov. 1999.

[31] G. Bombara, C.-I. Vasile, F. Penedo, H. Yasuoka, and C. Belta. A decision tree approach to data classification using signal temporal logic. In *Proceedings of the 19th international conference on Hybrid systems: computation and control.* ACM, 2016.

[32] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, ISSTA '06, pages 169–180, New York, NY, USA, 2006. ACM.

[33] N. Brett, U. Siddique, and B. Bonakdarpour. Rewriting-based runtime verification for alternation-free hyperltl. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 77–93. Springer, 2017.

[34] C. Brooks, A. Cataldo, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, and H. Zheng. Hyvisual: A hybrid system visual modeler. *University of California, Berkeley, Technical Memorandum UCB/ERL M*, 5, 2005.

[35] J. W. Bryans, M. Koutny, L. Mazaré, and P. Y. Ryan. Opacity generalised to transition systems. *International Journal of Information Security*, 7(6):421–435, 2008.

[36] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *International Conference on Computer Aided Verification*, pages 258–263. Springer, 2013.

[37] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. Temporal logics for hyperproperties. In *International Conference on Principles of Security and Trust*, pages 265–284. Springer, 2014.

[38] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[39] L. Cohen. *Time-frequency analysis*, volume 299. Prentice hall, 1995.

[40] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 281–290, 2008.

[41] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Software Engineering and Formal Methods*, volume 7504 of *LNCS*, pages 233–247. Springer Berlin Heidelberg, 2012.

[42] R. David and H. Alla. Petri nets for modeling of dynamic systems: A survey. *Automatica*, 30(2):175–202, 1994.

[43] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '08/ETAPS '08, pages 337–340. Springer-Verlag, 2008.

[44] L. De Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. Sal 2. In *International Conference on Computer Aided Verification*, pages 496–500. Springer, 2004.

[45] A. Deshpande, A. Göllü, and P. Varaiya. Shift: A formalism and a programming language for dynamic networks of hybrid automata. In *International Hybrid Systems Workshop*, pages 113–133. Springer, 1996.

[46] P. Dluhoš, L. Brim, and D. Šafránek. On expressing and monitoring oscillatory dynamics. *arXiv preprint arXiv:1208.3853*, 2012.

[47] A. Dokhanchi, B. Hoxha, and G. Fainekos. Metric interval temporal logic specification elicitation and debugging. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, pages 70–79. IEEE, 2015.

[48] A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 167–170. Springer Berlin / Heidelberg, 2010.

[49] A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Computer Aided Verification*, pages 167–170. Springer, 2010.

[50] A. Donzé, T. Ferrere, and O. Maler. Efficient robust monitoring for stl. In *Computer Aided Verification*, pages 264–279. Springer, 2013.

[51] A. Donzé, O. Maler, E. Bartocci, D. Nickovic, R. Grosu, and S. Smolka. On temporal logic and signal processing. In *Automated Technology for Verification and Analysis*, pages 92–106. Springer, 2012.

[52] P. S. Duggirala, C. Fan, S. Mitra, and M. Viswanathan. Meeting a powertrain verification challenge. In *To appear in the Proceedings of International Conference on Computer Aided Verification (CAV 2015)*, 2015.

[53] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok. C2e2: a verification tool for stateflow models. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 68–82. Springer, 2015.

[54] R. W. Erickson and D. Maksimović. *Fundamentals of Power Electronics*. Springer, 2nd edition edition, 2004.

[55] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, 2001.

[56] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[57] G. E. Fainekos, A. Girard, and G. J. Pappas. Temporal logic verification using simulation. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 171–186. Springer, 2006.

[58] C. Fan, P. S. Duggirala, S. Mitra, and M. Viswanathan. Progress on powertrain verification challenge with C2E2. In *ARCH '15: Proc. of the 2nd Workshop on Applied Verification for Continuous and Hybrid Systems*, April 2015.

[59] G. Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *International workshop on hybrid systems: computation and control*, pages 258–273. Springer, 2005.

[60] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *Computer Aided Verification (CAV)*, LNCS. Springer, 2011.

[61] T. T. Gamage, B. M. McMillin, and T. P. Roth. Enforcing information flow security properties in cyber-physical systems: A generalized framework based on compensation. In *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*, pages 158–163. IEEE, 2010.

[62] T. T. Gamage, B. M. McMillin, and T. P. Roth. Enforcing information flow security properties in cyber-physical systems: A generalized framework based on compensation. In *COMPSAC Workshops*, pages 158–163. IEEE Computer Society, 2010.

[63] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pages 179–179, Apr. 1995.

[64] A. Girard and G. J. Pappas. Verification using simulation. In *International Workshop on Hybrid Systems: Computation and Control*, pages 272–286. Springer, 2006.

[65] J. A. Goguen and J. Meseguer. Security policies and security models. In *Security and Privacy, 1982 IEEE Symposium on*, pages 11–11. IEEE, 1982.

[66] F. J. Harris. On the use of windows for harmonic analysis with the discrete fourier transform. *Proceedings of the IEEE*, 66(1):51–83, 1978.

[67] G. B. Health and S. Executive. *Out of Control: Why Control Systems Go Wrong and how to Prevent Failure*. Topic reports. HSE Books, 1995.

[68] T. A. Henzinger. The theory of hybrid automata. In *Verification of Digital and Hybrid Systems*, pages 265–292. Springer, 2000.

[69] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE transactions on automatic control*, 43(4):540–554, 1998.

[70] T. A. Henzinger, J. Otop, and R. Samanta. *Lipschitz Robustness of Timed I/O Systems*, pages 250–267. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[71] G. J. Holzmann. The logic of bugs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 81–87. ACM, 2002.

[72] S. Hossain, S. Dhople, and T. T. Johnson. Reachability analysis of closed-loop switching power converters. In *Power and Energy Conference at Illinois (PECI)*, pages 130–134, 2013.

[73] B. Hoxha, A. Dokhanchi, and G. Fainekos. Mining parametric temporal logic properties in model-based design for cyber-physical systems. *International Journal on Software Tools for Technology Transfer*, pages 1–15.

[74] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Benchmarks for model transformations and conformance checking. In *1st International Workshop on Applied Verification for Continuous and Hybrid Systems (ARCH)*, 2014.

[75] X. Jin, A. Donzé, J. V. Deshmukh, and S. A. Seshia. Mining requirements from closed-loop control models. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*, HSCC '13, pages 43–52, New York, NY, USA, 2013. ACM.

[76] X. Jin, A. Donzé, J. V. Deshmukh, and S. A. Seshia. Mining requirements from closed-loop control models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1704–1717, 2015.

[77] H. W. Johnson, M. Graham, et al. *High-speed digital design: a handbook of black magic*, volume 1. Prentice Hall Upper Saddle River, NJ, 1993.

[78] T. T. Johnson, S. Bak, M. Caccamo, and L. Sha. Real-time reachability for verified simplex design. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(2):26, 2016.

[79] T. T. Johnson, Z. Hong, and A. Kapoor. Design verification methods for switching power converters. In *Power and Energy Conference at Illinois (PECI), 2012 IEEE*, pages 1–6, Feb. 2012.

[80] J. Kapinski, J. V. Deshmukh, X. Jin, H. Ito, and K. Butts. Simulation-based approaches for verification of embedded control systems: an overview of traditional and advanced modeling, testing, and verification techniques. *IEEE Control Systems*, 36(6):45–64, 2016.

[81] J. Kapinski, X. Jin, J. Deshmukh, A. Donze, T. Yamaguchi, H. Ito, T. Kaga, S. Kobuna, and S. Seshia. ST-Lib: A library for specifying and classifying model behaviors. 2016.

[82] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. In *European Symposium on Research in Computer Security*, pages 97–110. Springer, 1998.

[83] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Annual International Cryptology Conference*, pages 388–397. Springer, 1999.

[84] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Moderator-Ravi. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*, pages 753–760. ACM, 2004.

[85] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[86] S. Kong, S. Gao, W. Chen, and E. Clarke. dreach: $\delta$-reachability analysis for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 200–205. Springer, 2015.

[87] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, et al. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447–462. IEEE, 2010.

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

[89] K. Lee. Ieee 1451: A standard in support of smart transducer networking. In *Instrumentation and Measurement Technology Conference, 2000. IMTC 2000. Proceedings of the 17th IEEE*, volume 2, pages 525–528. IEEE, 2000.

[90] N. G. Leveson. System safety engineering: Back to the future. *Massachusetts Institute of Technology*, 2002.

[91] C. Li, A. Raghunathan, and N. K. Jha. Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system. In *e-Health Networking Applications and Services (Healthcom), 2011 13th IEEE International Conference on*, pages 150–156. IEEE, 2011.

[92] J. L. Lions. Ariane 5 flight 501 failure. Technical report, Paris, France, July 1996.

[93] L. Liu, M. Esmalifalak, Q. Ding, V. A. Emesih, and Z. Han. Detecting false data injection attacks on power grid by sparse optimization. *IEEE Transactions on Smart Grid*, 5(2):612–621, 2014.

[94] N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

[95] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.

[96] S. Mallat. *A wavelet tour of signal processing.* Academic press, 1999.

[97] K. Manamcheri, S. Mitra, S. Bak, and M. Caccamo. A step towards verification and synthesis from Simulink/Stateflow models. In *Proc. of the 14th Intl. Conf. on Hybrid Systems: Computation and Control (HSCC)*, pages 317–318. ACM, 2011.

[98] K. Manandhar, X. Cao, F. Hu, and Y. Liu. Detection of faults and attacks including false data injection attack in smart grid using kalman filter. *IEEE transactions on control of network systems*, 1(4):370–379, 2014.

[99] K. McCaney. Pentagon's rapid plan for maintaining air superiority. `http://defensesystems.com/Articles/2014/05/01/DARPA-system-of-systems-SoSITE.aspx`, 2014.

[100] D. McCullough. Specifications for multi-level security and a hook-up. In *Security and Privacy, 1987 IEEE Symposium on*, pages 161–161. IEEE, 1987.

[101] J. McLean. Security models and information flow. In *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pages 180–187. IEEE, 1990.

[102] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Research in Security and Privacy, 1994. Proceedings., 1994 IEEE Computer Society Symposium on*, pages 79–93. IEEE, 1994.

[103] R. Medhat, S. Ramesh, B. Bonakdarpour, and S. Fischmeister. A framework for mining hybrid automata from input/output traces. In *Proceedings of the 12th International Conference on Embedded Software*, EMSOFT '15, pages 177–186, Piscataway, NJ, USA, 2015. IEEE Press.

[104] S. Minopoli and G. Frehse. Sl2sx translator: from simulink to spaceex models. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, pages 93–98. ACM, 2016.

[105] S. Mishra, Y. Shoukry, N. Karamchandani, S. Diggavi, and P. Tabuada. Secure state estimation: optimal guarantees against sensor attacks in the presence of noise. In *Information Theory (ISIT), 2015 IEEE International Symposium on*, pages 2929–2933. IEEE, 2015.

[106] Y. Mo, T. H.-J. Kim, K. Brancik, D. Dickinson, H. Lee, A. Perrig, and B. Sinopoli. Cyber–physical security of a smart grid infrastructure. *Proceedings of the IEEE*, 100(1):195–209, 2012.

[107] G. B. Moody and R. G. Mark. The impact of the mit-bih arrhythmia database. *IEEE Engineering in Medicine and Biology Magazine*, 20(3):45–50, 2001.

[108] L. Moura and N. Bjørner. Satisfiability modulo theories: An appetizer. In M. M. Oliveira and J. Woodcock, editors, *Formal Methods: Foundations and Applications*, volume 5902 of *LNCS*, pages 23–36. Springer Berlin Heidelberg, 2009.

[109] S. Munir, M. Y. Ahmed, and J. A. Stankovic. Eyephy: Detecting dependencies in cyber-physical system apps due to human-in-the-loop. 2015.

[110] S. Munir, J. Stankovic, et al. Depsys: Dependency aware integration of cyber-physical systems for smart homes. In *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*, pages 127–138. IEEE, 2014.

[111] National Highway Traffic Safety Administration (NHTSA). Honda automatic transmission control module software (recall #11v395000), Aug. 2011.

[112] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[113] T. Nghiem, S. Sankaranarayanan, G. Fainekos, F. Ivancić, A. Gupta, and G. J. Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '10, pages 211–220, New York, NY, USA, 2010. ACM.

[114] L. V. Nguyen, K. Hoque, S. Bak, S. Drager, and T. T. Johnson. Cyber-physical specification mismatches. *ACM Transactions on Cyber-Physical Systems*, 2018.

[115] L. V. Nguyen and T. T. Johnson. Benchmark: Dc-to-dc switched-mode power converters (buck converters, boost converters, and buck-boost converters). In *Applied Verification for Continuous and Hybrid Systems Workshop (ARCH 2014)*, Berlin, Germany, Apr. 2014.

[116] L. V. Nguyen, J. Kapinski, X. Jin, J. V. Deshmukh, K. Butts, and T. T. Johnson. Abnormal data classification using time-frequency temporal logic. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*, HSCC '17, pages 237–242, New York, NY, USA, 2017. ACM.

[117] L. V. Nguyen, J. Kapinski, X. Jin, J. V. Deshmukh, and T. T. Johnson. Hyperproperties of real-valued signals. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE '17, pages 104–113, New York, NY, USA, 2017. ACM.

[118] L. V. Nguyen, C. Schilling, S. Bogomolov, and T. T. Johnson. Hyrg: a random generation tool for affine hybrid automata. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 289–290. ACM, 2015.

[119] L. V. Nguyen, C. Schilling, S. Bogomolov, and T. T. Johnson. Runtime verification for hybrid analysis tools. In *Runtime Verification*, pages 281–286. Springer, 2015.

[120] L. V. Nguyen, H.-D. Tran, and T. Johnson. Virtual prototyping for distributed control of a fault-tolerant modular multilevel inverter for photovoltaics. *Energy Conversion, IEEE Transactions on*, 29(4):841–850, Dec. 2014.

[121] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 683–693, Piscataway, NJ, USA, 2012. IEEE Press.

[122] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. DIG: A dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology, to appear*, 2014.

[123] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to generate disjunctive invariants. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 608–619, New York, NY, USA, 2014. ACM.

[124] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 229–239, New York, NY, USA, 2002. ACM.

[125] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[126] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*, pages 87–102, New York, NY, USA, 2009. ACM.

[127] W. K. Peter. *The Theory of Rectangular Hybrid Automata.* PhD thesis, PhD thesis, Cornell University, 1996.

[128] A. Platzer and J.-D. Quesel. Keymaera: A hybrid theorem prover for hybrid systems (system description). In *International Joint Conference on Automated Reasoning*, pages 171–178. Springer, 2008.

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

[130] R. Polikar. The wavelet tutorial. 1996.

[131] M. N. Rabe. A temporal logic approach to information-flow control. 2016.

[132] S. Ravi, A. Raghunathan, and S. Chakradhar. Tamper resistance mechanisms for secure embedded systems. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 605–611. IEEE, 2004.

[133] A. Reder and A. Egyed. Determining the cause of a design model inconsistency. *Software Engineering, IEEE Transactions on*, 39(11):1531–1548, Nov. 2013.

[134] P. Rohatgi. Electromagnetic attacks and countermeasures. In *Cryptographic Engineering*, pages 407–430. Springer, 2009.

[135] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 255–269. IEEE, 2005.

[136] F. Sagstetter, M. Lukasiewycz, S. Steinhorst, M. Wolf, A. Bouard, W. R. Harris, S. Jha, T. Peyrin, A. Poschmann, and S. Chakraborty. Security challenges in automotive hardware/software architecture design. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 458–463. EDA Consortium, 2013.

[137] B. Scholkopf and A. J. Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2001.

[138] E. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331, May 2010.

[139] Y. Shoukry, P. Martin, P. Tabuada, and M. Srivastava. Non-invasive spoofing attacks for anti-lock braking systems. In *Proceedings of the 15th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'13, pages 55–72, Berlin, Heidelberg, 2013. Springer-Verlag.

[140] G. Smith. On the foundations of quantitative information flow. In *International Conference on Foundations of Software Science and Computational Structures*, pages 288–302. Springer, 2009.

[141] L. Staller. Understanding analog to digital converter specifications. *Embedded Systems Design*, 2005.

[142] A. G. Stephenson, D. R. Mulville, F. H. Bauer, G. A. Dukeman, P. Norvig, L. LaPiana, P. Rutledge, D. Folta, and R. Sackheim. Mars climate orbiter mishap investigation board phase i report, 44 pp. *NASA, Washington, DC*, 1999.

[143] S. Tripakis, C. Stergiou, C. Shaver, and E. A. Lee. A modular formal semantics for ptolemy. *Mathematical Structures in Computer Science*, 23:834–881, 8 2013.

[144] J. Wan, A. Canedo, and M. A. Al Faruque. Security-aware functional modeling of cyber-physical systems. In *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*, pages 1–4. IEEE, 2015.

[145] M. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. Heimdahl, and S. Rayadurgam. Your what is my how: Iteration and hierarchy in system design. *Software, IEEE*, 30(2):54–60, Mar. 2013.

[146] J. T. Wittbold and D. M. Johnson. Information flow in nondeterministic systems. In *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, pages 144–161. IEEE, 1990.

[147] Z. Xu and A. A. Julius. Census signal temporal logic inference for multiagent group behavior analysis. *IEEE Transactions on Automation Science and Engineering*, 2016.

[148] R. R. Young. *The requirements engineering handbook*. Artech House, 2004.

[149] P. Zave. An operational approach to requirements specification for embedded systems. *IEEE transactions on Software Engineering*, (3):250–269, 1982.

[150] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 29–43. IEEE, 2003.

[151] C. Zhou and R. Kumar. Semantic translation of simulink diagrams to input/output extended finite automata. *Discrete Event Dynamic Systems*, 22(2):223–247, 2012.

[152] L. Zou, N. Zhan, S. Wang, and M. Fränzle. Formal verification of simulink/stateflow diagrams. In *Automated Technology for Verification and Analysis - 13th International Symposium, Shanghai, China*, pages 464–481, 2015.

[153] A. Zutshi, S. Sankaranarayanan, J. V. Deshmukh, and X. Jin. Symbolic-numeric reachability analysis of closed-loop control software. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, pages 135–144. ACM, 2016.