Cyber-Physical Specification Mismatches

LUAN V. NGUYEN, University of Texas at Arlington
KHAZA ANUARUL HOQUE, University of Oxford
STANLEY BAK, Air Force Research Laboratory
STEVEN DRAGER, Air Force Research Laboratory
TAYLOR T. JOHNSON, Vanderbilt University

Embedded systems use increasingly complex software and are evolving into cyber-physical systems (CPS) with sophisticated interaction and coupling between physical and computational processes. Many CPS operate in safety-critical environments and have stringent certification, reliability, and correctness requirements. These systems undergo changes throughout their lifetimes, where either the software or physical hardware is updated in subsequent design iterations. One source of failure in safety-critical CPS is when there are unstated assumptions in either the physical or cyber parts of the system, and new components do not match those assumptions. In this work, we present 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 Simulink/Stateflow (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 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 a powertrain fuel control system is also introduced to illustrate the power of Hynger and Daikon in automatically identifying cyber-physical specification mismatches.

CCS Concepts: •**Theory of computation** →**Program specifications;** •**Software and its engineering** →Dynamic analysis;

Additional Key Words and Phrases: Cyber-physical systems, dynamic analysis, specifications

ACM Reference format:
Luan V. Nguyen, Khaza Anuarul Hoque, Stanley Bak, Steven Drager, and Taylor T. Johnson. 201X. Cyber-Physical Specification Mismatches. 1, 1, Article 1 (January 201X), 25 pages.
DOI: 0000001.0000001

Cyber-Physical Specification Mismatches

1 INTRODUCTION

Systems interacting with their physical environments are becoming increasingly dependent upon computers and software, such as in emerging cyber-physical systems (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

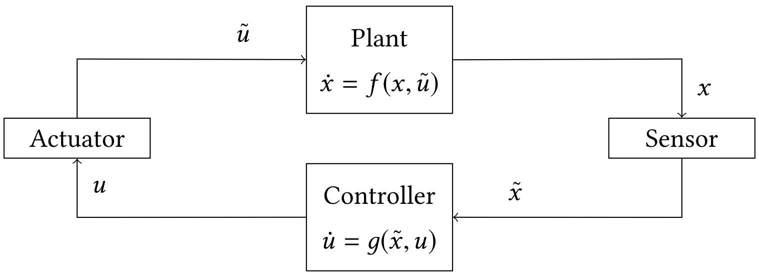


Fig. 1. High-level diagram of a closed-loop control system.

most engineered systems, the vast majority of resources are devoted to ensuring systems meet their specifications [7]. 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 disasters 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 Model-Based Design (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 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. (*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 paper, we a develop 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

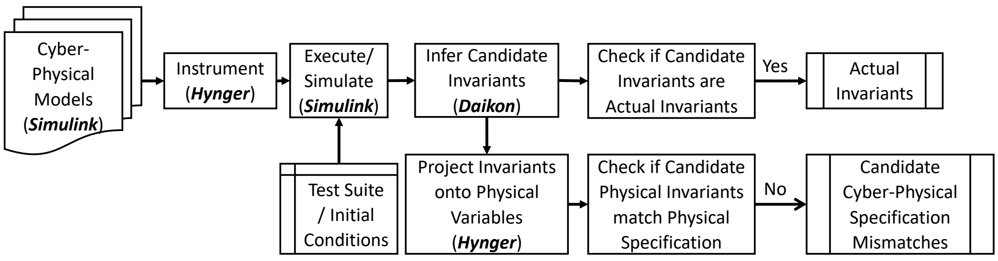

Cyber-Physical Models (Simulink)
Instrument (Hynger)
Execute/ Simulate (Simulink)
Infer Candidate Invariants (Daikon)
Check if Candidate Invariants are Actual Invariants
Yes
Actual Invariants
Test Suite / Initial Conditions
Project Invariants onto Physical Variables (Hynger)
Check if Candidate Physical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

Fig. 2. Preliminary overview of the methodology using Hynger and Daikon to infer candidate invariants and detect specification mismatches.

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 [31]. Candidate invariants inferred with our Hynger (for HYbrid iNvariant GEneratoR) software tool in conjunction with Daikon [17, 18] may be formally checked as actual invariants using a hybrid system model checker [20]. Figure 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 5.

Contributions. The primary contributions of this paper 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 [20], 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.

Overall, this journal has been substantially extended from our previous work [25]. In fact, we added the formal definitions of cyber-physical specification mismatches, cyber-physical input-output automata, and invariant checking problem to identify whether the inferred invariants are actual invariants. Moreover, two proof-of-concept CPS case studies including a buck converter and an abstract fuel control system are presented to show the capability of Hynger tool in automatically identifying potential cyber-physical specification mismatches of CPSs. The experimental results illustrate the feasibility of using dynamic invariant inference for analysis of embedded and cyber-physical systems. Before presenting the details of our approach, we first illustrate the pitfalls of CPS design reuse by citing examples of critical mistakes in existing, certified systems.

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 (including one of the author's) 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 [38]. This problem was widespread in part 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, 29]. 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 [32] 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 [28, 50]. 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 [50].

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 [45]. Cyber-physical specification mismatch is closely related to model inconsistency [47], architectural mismatch [21], and requirements consistency [52]. There are many benefits of dynamic analysis such as using implementations instead of models [17, 18, 45] to find dynamic program specifications [45], 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 [17, 18]. 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 [33]. 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 [37] and EyePhy [36] 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 [27]. 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 [10, 11, 17, 18, 45]. Daikon, which is used by Hynger, processes program traces to generate invariants [17, 18]. For several languages (C, C++, etc.), this process is performed without access to the source code by instrumenting the compiled program using Valgrind [39]. 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 [9], 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 [17, 18], 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 [11] and commercial tools like Agitagor [10] can be used for generating candidate invariants for other languages.

3 CYBER-PHYSICAL SYSTEM MODELS

The approach presented in this paper 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 [31]). 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 to formalize the specification mismatch problem. We will not define formal semantics of this structure or SLSF diagrams in this paper. 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) [30] may be used to specify the semantics, such as done in the HyLink tool [31]. 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 [34]. Interested readers can see some graphical examples of the translation in [31, 34]. Other formalisms like actors and hierarchical state machines are commonly used for formal modeling of other diagrammatic frameworks similar to SLSF [2, 8, 51, 53]. Given a formal model $\mathcal{A}$ and candidate specification Σ (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 [20]. In some instances, we know when an SLSF diagram corresponds precisely to a hybrid automaton model [31], 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 3.1 (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
- 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 3.1, 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 $root(M)$. For a block $v \in M$, the *children* of v are denoted as $children(v)$ 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 $parent(v)$ and is defined as the singleton set $\{w \in M \mid v \in children(w)\}$.

Clearly, $parent(root(M)) = \emptyset$. For a block $v \in M$, the *ancestors* of v are denoted as $ancestors(v)$ and defined inductively as the set of blocks $\{w \in M \mid v \cup w \in children(v) \cup children(w)\}$ (or equivalently, as the transitive closure of $children(v)$).

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 $type(x)$. 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 $val(x)$. The state-space of $\mathcal{A}$ is the set of valuations of all the variables 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 3.2 (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 [3]. 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 3.3 (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 3.3 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 [48]. 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 $\{w \in M \mid \mathit{parent}(w) = \mathit{parent}(v)\}$. Output variables of a block $v \in M$ may be connected to input variables of a block $w \in M$. Let $G_{\mathsf{V}} \stackrel{\Delta}{=} (V_{\mathsf{V}}, E_{\mathsf{V}})$ be a directed graph where the vertices V_{V} are variables of blocks $v \in M$ and the edges specify the interconnection between output variables to input variables for some model $w \in \mathit{siblings}(v)$, 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 V_{SP}. Typical examples of software physical variables include those used for sensed and sampled measurements, variables used in feedback control calculations, etc.

Example 3.4. Here, we describe a CPS case study used throughout the remainder of the paper for illustrating concepts. The case study is a DC-to-DC power converter (like buck, boost, and buck-boost converters) [40], 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 $m \in \{\mathit{Open}, \mathit{Close}, \mathit{DCM}\}$ 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 5V and "bucks" or drops the voltage to some lower DC voltage, say 2.5V. These circuits

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

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 [41].

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 [22]. 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 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 [22, 26, 40]).

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 3.5. 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 $Q \triangleq Loc \times \mathbb{R}^n$ 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 Δ_0, Δ_t and Δ_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 $\tau : [0, T] \longrightarrow val(Var)$ 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 ℓ to destination location ℓ' 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 $Init \subseteq Q$.

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 $\rho \triangleq s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \ldots$, where $s_0 \in Init$, and $s_i \longrightarrow s_{i+1}$ 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 $\rho \triangleq s_0 \longrightarrow s_1 \longrightarrow \ldots \longrightarrow s_k$.

Invariant Property. An *invariant property* φ 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 3.6 (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:
- $Loc = Loc_1 \times Loc_2$,
- $Var = Var_1 \cup Var_2$,
- $Q = Q_1 \times Q_2$,
- $O = O_1 \cup O_2$,
- $I = (I_1 \cup I_2) \setminus O$,
- $\forall \ell_1, \ell_2 \in Loc$, $Inv(\ell_1, \ell_2) = Inv_1(\ell_1) \wedge Inv_2(\ell_2)$
- $\forall \ell_1, \ell_2 \in Loc$, $\forall x \in Var$, $((\ell_1, \ell_2), val(x) \in Init)$ 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 τ onto the sets of variables and locations of component i.
- Given $\gamma_1 \in Trans_1$, $\gamma_1 \triangleq \left\langle \ell_1, \ell_1', g_1, u_1 \right\rangle$ and $\gamma_2 \in Trans_2$, $\gamma_2 \triangleq \left\langle \ell_2, \ell_2', g_2, u_2 \right\rangle$, 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 paper is a closed-loop model, e.g., the closed-loop buck converter. Such a model can be formally represented as a closed-loop

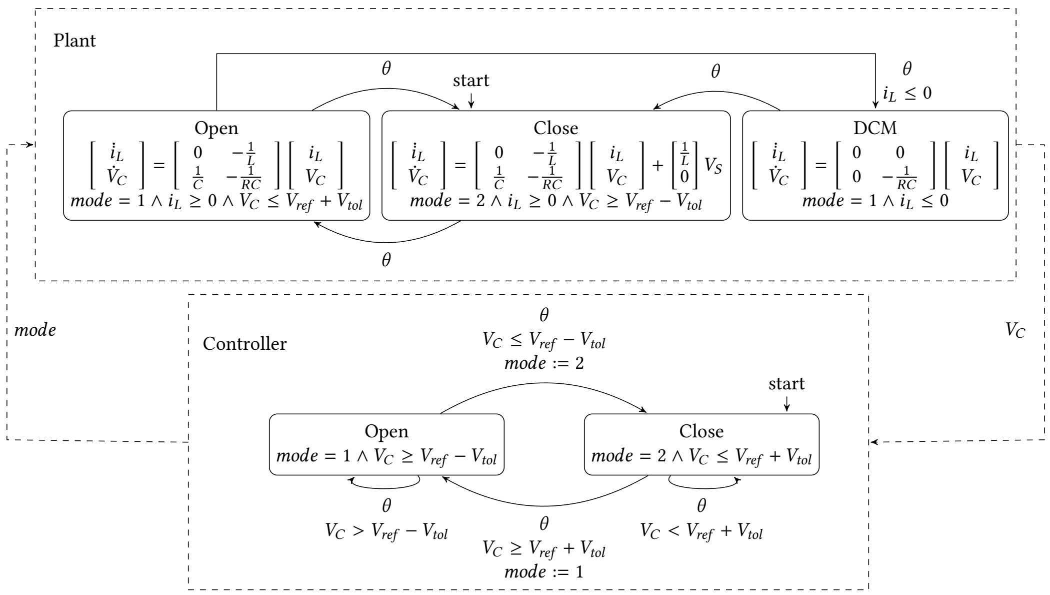

Plant
Open
$\begin{bmatrix} i_L \\ \dot{V}_C \end{bmatrix} = \begin{bmatrix} 0 & -\frac{1}{L} \\ \frac{1}{C} & -\frac{1}{RC} \end{bmatrix} \begin{bmatrix} i_L \\ V_C \end{bmatrix}$
$mode = 1 \wedge i_L \geq 0 \wedge V_C \leq V_{ref} + V_{tol}$
Close
$\begin{bmatrix} i_L \\ \dot{V}_C \end{bmatrix} = \begin{bmatrix}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

Fig. 3. A hybrid automaton models the buck converter plant with hysteresis controller.

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 controller CPIOA can be purely discrete. For instance, the hybrid automaton of the closed-loop buck converter (without sensor and actuator) shown in Figure 3 can be considered as one closed-loop CPIOA, where θ 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 3) using some hybrid systems model checkers such as SpaceEx [20].

3.2 Candidate Invariant Checking Problem

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

Definition 3.7 (Candidate Invariant Checking). Given a CPS model $\mathcal{A}$ with a set of candidate 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 3.7, 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 $\mathcal{A}$.

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.

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 [6, 35]. 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 ϕ, let $vars(\phi)$ be the set of variables appearing in ϕ. For a formula ϕ: (a) if $vars(\phi)$ are all physical, then ϕ is a *physical specification*, (b) if $vars(\phi)$ are all cyber, then ϕ is a *cyber specification*, and (c) if $vars(\phi)$ consists of both cyber and physical variables, then ϕ is a *cyber-physical specification.*

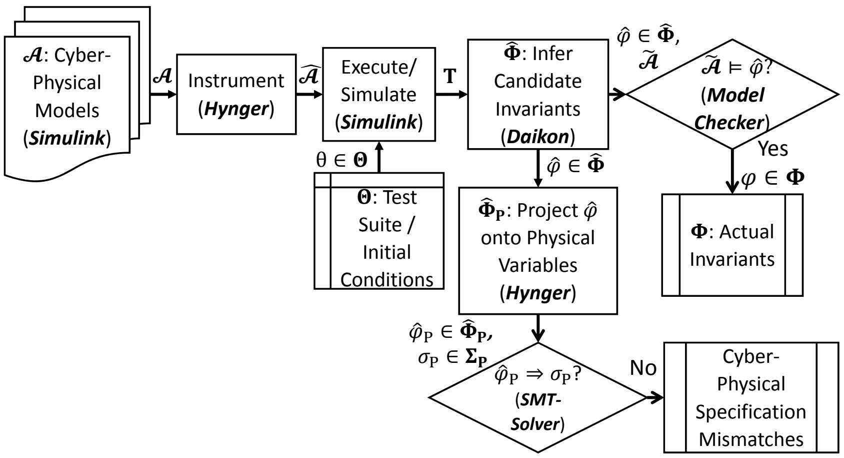
Next, while we will try to infer interesting specifications ϕ using dynamic analysis later in the paper, we first highlight examples of specifications made a priori in system design, as these are necessary to define specification mismatches. Let Σ be a set of specifications for $\mathcal{A}$, which is a set of formulas over the variables of $\mathcal{A}$. Referring to Figure 4, we separate the specification Σ into sets of cyber and physical specifications, written respectively as Σ_C and Σ_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 Σ_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, σ_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 σ_P^2 and σ_P^3 specify assumptions about the buck converter's environment, namely 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 4 and described next.

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 θ in a set of initial conditions Θ, 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 φ, and the set of actual invariants Φ is collected.



$\mathcal{A}$: Cyber-Physical Models (*Simulink*)
$\mathcal{A}$
Instrument (*Hynger*)
$\widetilde{\mathcal{A}}$
Execute/ Simulate (*Simulink*)
T
$\hat{\Phi}$: Infer Candidate Invariants (*Daikon*)
$\hat{\varphi} \in \hat{\Phi}$, $\widetilde{\mathcal{A}}$
$\widetilde{\mathcal{A}} \models \hat{\varphi}$? (*Model Checker*)
Yes
$\varphi \in \Phi$
Φ: Actual Invariants
$\theta \in \Theta$
Θ: Test Suite / Initial Conditions
$\hat{\varphi} \in \hat{\Phi}$
$\hat{\Phi}_P$: Project $\hat{\varphi}$ onto Physical Variables (*Hynger*)
$\hat{\varphi}_P \in \hat{\Phi}_P$, $\sigma_P \in \Sigma_P$
$\hat{\varphi}_P \Rightarrow \sigma_P$? (*SMT-Solver*)
No
Cyber-Physical Specification Mismatches

Fig. 4. Hynger overview, inference of physical specifications assumed by software, and cyber-physical specification mismatch identification.

Definition 4.1 (Cyber-Physical Specification Mismatch). Given an SLSF diagram $\mathcal{A}$ with a set of actual physical specifications Σ_P, let $\hat{\Phi}_P \triangleq \hat{\Phi} \downarrow \mathrm{V}_{SP}$ 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 4.1, $\hat{\Phi} \downarrow \mathrm{V}_{SP}$ denotes the projection or the restriction of $\hat{\Phi}$ to the set of software physical variable V_{SP}. In all cases, each candidate invariant $\hat{\varphi} \in \hat{\Phi}$ is projected (restricted) onto the software physical variables 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 [13][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 σ_P of a set of actual physical specifications Σ_P. Since $\hat{\varphi}_P$ and σ_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 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]Z3 may be downloaded: http://z3.codeplex.com/.

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 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 $\mathrm{V}_I(v)$ and the output variables $\mathrm{V}_O(v)$ 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 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 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, $\theta \in \Theta$). The output of the Hynger tool is a set of candidate invariants, which, when projected onto all the software physical variables 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.

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 7) in a different manner. The loop in the controller SLSF model of Figure 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 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 6 found for the C function in Figure 5.

Example C Program, Formal Specification, and Candidate Invariants Inferred. Figure 5 shows an example C function to illustrate the use of dynamic analysis with Daikon to find candidate

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

[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;
  }
```

Fig. 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.

```
============== 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
```

Fig. 6. Daikon candidate invariant output (with some additional markup in C-style comments for readability) for the `sum_array` example from Figure 5.

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 [17]. Additionally, a formalized correctness specification is given in the modern ANSI/ISO C Specification Language (ACSL), used by tools such as Frama-C [12]. 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 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 9 using Hynger and Daikon.

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 6.1 and detailed further in [40], *(b)* a solar array case study that uses a buck-boost converter [41], *(c)* benchmarks from S-TaLiRo [4], *(d)* benchmarks from Breach [14, 24],

Model	Solver	Tmax	Sim	SimInst	Inv	Overhead	BDAll	BDInst	BDPct
buck (Section 6.1)	ode45	0.0083	6.2985	38.4518	5.7335	7.0152	14	3	21.4286
buck (Section 6.1)	ode45	0.0083	6.4567	44.698	7.0913	8.021	14	4	28.5714
buck (Section 6.1)	ode45	0.0083	6.5301	78.3176	7.2224	13.0993	14	14	100
heat25830 [4]	ode45	50	4.6913	254.5776	14.09	57.2692	28	1	3.5714
heat25830 [4]	ode45	50	4.7328	2882.7808	15.6488	612.4233	28	10	35.7143
fuel1 [23]	ode15s	15	5.3747	976.6274	7.923	183.182	208	17	8.1731
fuel1 [23]	ode15s	15	4.2131	2824.2804	11.604	673.1137	208	63	30.2885
fuel2 [23]	ode15s	20	3.3838	36.8312	2.9881	11.7674	25	6	24
fuel2 [23]	ode15s	20	2.7353	42.4074	3.2771	16.7018	25	13	52
fuel3 [19]	ode15s	20	3.7425	292.9976	4.1131	79.3892	90	11	12.2222
fuel3 [19]	ode15s	20	3.6083	945.3992	4.3904	263.2236	90	46	51.1111

Table 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., $BDInst/BDAll$).

(e) benchmarks created as a part of the ARCH 2014 CPSWeek workshop (particularly [23, 40]) 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 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 [18, 39]. 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 paper, and the second model is derived from a collection of the automotive powertrain control benchmarks proposed by Toyota [24].

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,

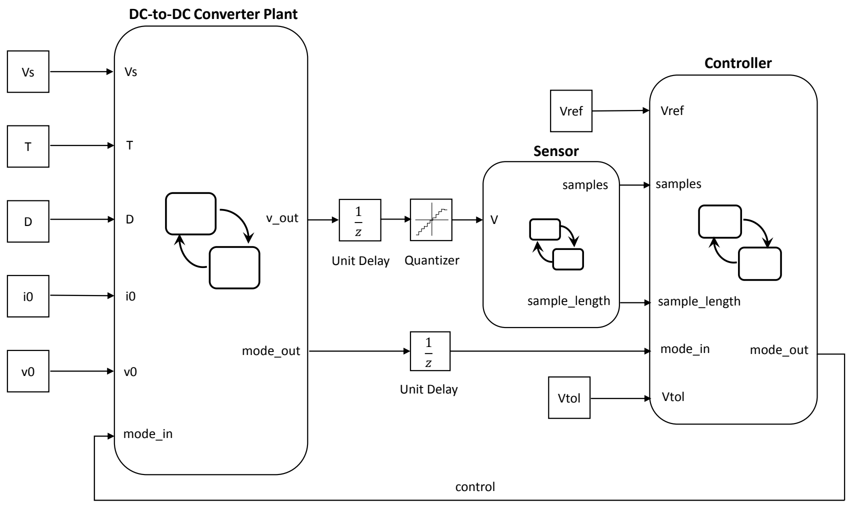


Fig. 7. General CPS case study architecture overview of the buck converter in SLSF. The system is composed of a plant (physical system) model, a controller (software/cyber), and potentially sensor and actuator models. The cyber model uses some of the physical model output states to determine a control action or input. The controller in SLSF appears in Figure 9, and the sensor model appears in Figure 8. An example of this closed-loop buck converter including only plant and controller can be formally represented as the hybrid automaton in Figure 3.

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 Σ_P in Section 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 is detected using Hynger and the methodology described in this paper. 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 [49]. Overall, these conversion errors may cause a significant impact to result in system failures.

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 7 with $V_S = 100$, $V_{ref} = 48V$, $V_{rip} = 5\%V_{ref} = 2.4V$, and assume that δ_S, δ_{ref} are equal to zero. The physical

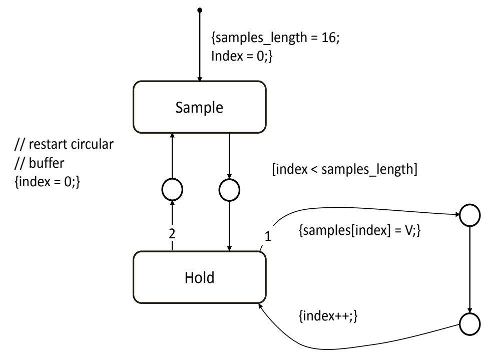


Fig. 8. Stateflow model of sensor with a sample and hold for the buck converter case study.

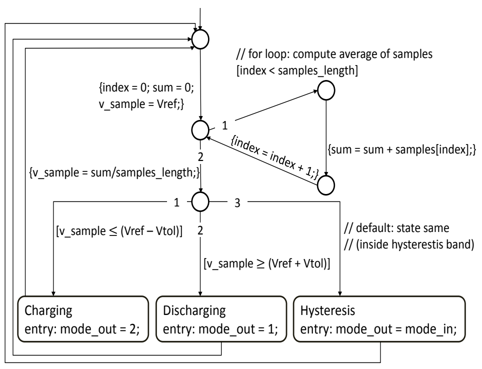

// for loop: compute average of samples
[index < samples_length]
{index = 0; sum = 0;
v_sample = Vref;}
1
2
{index = index + 1;}
{sum = sum + samples[index];}
{v_sample = sum/samples_length;}
1
3
[v_sample ≤ (Vref – Vtol)]
2
// default: state same
// (inside hysterestis band)
[v_sample ≥ (Vref + Vtol)]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

Fig. 9. Stateflow model of the buck-converter voltage hysteresis controller.

specification of the output voltage is $\sigma_P \triangleq t \geq t_s \implies 45.6V \leq V_{out}(t) \leq 50.4V$. 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 $\hat{\varphi}_P \triangleq t \geq t_s \implies 46.559V \leq V_{out}(t) \leq 50.203V$. Then, $\hat{\varphi}_P$ is considered as the candidate invariant of the system since the formula $\hat{\varphi}_P \implies \sigma_P$ 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 $\hat{\varphi}_P \triangleq t \geq t_s \implies 46.804V \leq V_{out}(t) \leq 51.118V$. Then, the formula $\hat{\varphi}_P \implies \sigma_P$ 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 $\sigma'_P \triangleq t \geq t_s \implies 34.2V \leq V_{out}(t) \leq 37.8V$. In this case, the inferred physical specification of the output voltage from Hynger and Daikon becomes $\hat{\varphi}'_P \triangleq t \geq t_s \implies 35.068V \leq V_{out}(t) \leq 39.053V$, so that the formula $\hat{\varphi}'_P \implies \sigma'_P$ is 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 $\hat{\varphi}_P \triangleq t \geq t_s \implies 45.853V \leq V_{out}(t) \leq 51.091V$. The check of the formula $\hat{\varphi}_P \implies \sigma_P$ 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 for the output voltage. The comparison between the actual physical specification σ_P and the physical specification $\hat{\varphi}_P$ inferred from Hynger and Daikon is shown in Table 2, and also illustrated in Figure 10. Note that in Table 2, $\hat{\varphi}_P$ describes the magnitude bound of the output voltage when $t \geq t_s$. The checks of the formula $\hat{\varphi}_P \implies \sigma_P$ occasionally return $False$, that are depicted in Figure 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

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

Table 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.

and our proposed methodology in 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 [16],

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

where $D = \frac{V_{ref}}{\eta \mathcal{V}_S}$ is a duty cycle, and η 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 1. It definitely shows that the system may have specification mismatches since the controller is encoded depending on wrong information about the physical plant.

Furthermore, changing the length of voltage measurement array (samples_length) in the sensor of the buck converter (shown in Figure 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 $\hat{\varphi}_P \triangleq t \geq t_s \Rightarrow 46.095V \leq V_{out}(t) \leq 50.788V$, which no longer implies the actual physical specification of the output voltage $\sigma_P \triangleq t \geq t_s \Rightarrow 45.6V \leq V_{out}(t) \leq 50.4V$.

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 system benchmarks provided by Toyota [23, 24], and further studied in [19]. The ultimate 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 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 [19]. 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*) λ: an air-fuel ratio, and (*d*) i: an integrator state, PI control signal. The evolution of the continuous variables in each mode

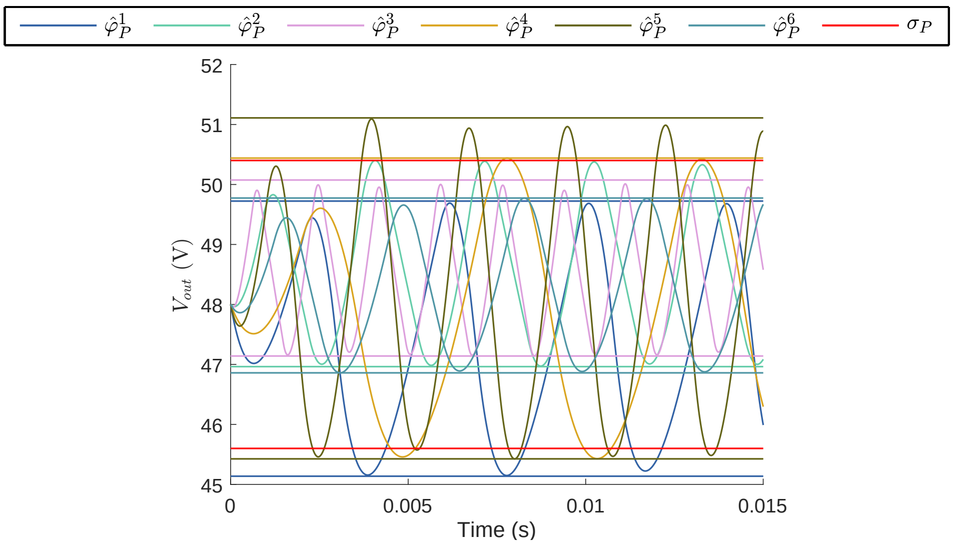


Fig. 10. A plot represents simulation traces and magnitude bounds of V_{out} of the buck converter with different values of R, L, and C. Here, σ_P denotes the actual bound of V_{out}, and $\hat{\varphi}_P^k$, $k \in [1, 6]$ denotes the inferred bound of V_{out} listed orderly in Table 2.

is governed by nonlinear polynomial differential equations as follows,

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

$$\dot{p_e} = c_1(2c_{23}\theta(c_{20}p^2 + c_{21}p + c_{22}) - (c_2 + c_3\omega p_e + c_4\omega p_e^2 + c_5\omega^2 p_e)) \tag{3}$$

$$\dot{\lambda} = c_{26}(c_{15} + (c_{16}c_{25}F_c + c_{17}c_{25}^2F_c^2 + c_{18}\dot{m}_c + c_{19}\dot{m}_c c_{25}F_c - \lambda) \tag{4}$$

$$\dot{i} = c_{14}(c_{24}\lambda - c_{11}), \tag{5}$$

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 $\dot{m}_c = c_{12}(c_2 + c_3\omega p + c_4\omega p^2 + c_5\omega^2 p)$. θ and ω are throttle angle (in degrees) and engine speed inputs (in *rpm*), respectively. The values of all constant parameters $c_j, j \in [1, 25]$, θ and ω are specified in [24]. 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 θ, ω, and sensor failure event *fail_event* [19].

Fuel Control System Plant Model. The plant can be modeled as a CPIOA with a single mode and two output physical variables p, λ whose continuous evolutions over time are described in Equation 2 and Equation 4, respectively. This model has an input cyber variable F_c, that is a fuel command.

Fuel Control System 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 3 and Equation 5, respectively. Here, p and λ are considered as two input cyber variables of the controller.

Reachability analysis of a sophisticated system like the fuel control 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 fuel control system has several actual physical specifications that can be found in [15]. 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 ±2% of its reference value λ_{ref}. The second physical specification requires the air-fuel ratio should be maintained within ±2% of λ_{ref} in the *normal* mode when $t \geq t_s$. These properties can be formally expressed as:

$$\sigma_P^1 \triangleq mode = startup \land t \leq t_s \Rightarrow 0.98\lambda_{ref} \leq \lambda(t) \leq 1.02\lambda_{ref} \tag{6}$$

$$\sigma_P^2 \triangleq mode = normal \land t \geq t_s \Rightarrow 0.98\lambda_{ref} \leq \lambda(t) \leq 1.02\lambda_{ref}. \tag{7}$$

Initially, we set $\lambda_{ref} = 14.7$, $\theta \in [8.8°, 90°]$, $w = 1800rpm$ $t_s = 9.5s$, 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 σ_P^1, the fuel control system may have specification mismatches when changing the engine speed and throttle inputs. For the second physical specification σ_P^2, the system may contain specification mismatches when changing controller and plant parameters.

Cyber-physical specification mismatches according to σ_P^1. With the initial setup mentioned earlier, the physical specification in Equation 6 becomes $\sigma_P \triangleq mode = startup \land t \leq 9.5 \Rightarrow 14.406 \leq \lambda(t) \leq 14.994$. Here, the magnitude bound of the air-fuel ratio at the *startup* mode of the system inferred from Hynger and Daikon is $\hat{\varphi}_P^1 \triangleq mode = startup \land t \leq 9.5 \Rightarrow 14.505 \leq \lambda(t) \leq 14.97$. 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 fuel control 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 $\hat{\varphi}_P^1 \triangleq mode = startup \land t \leq 9.5 \Rightarrow 14.129 \leq V_{out}(t) \leq 15.033$. Hence, the formula $\hat{\varphi}_P^1 \Rightarrow \sigma_P^1$ is false indicating that the fuel control 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 $\hat{\varphi}_P^1 \triangleq mode = startup \land t \leq 9.5 \Rightarrow 14.396 \leq V_{out}(t) \leq 14.849$. Hence, $\hat{\varphi}_P^1$ no longer implies σ_P^1. Therefore, there exists a cyber-physical specification mismatch when changing the throttle input as well.

Cyber-physical specification mismatches according to σ_P^2. Initially, the physical specification in Equation 7 is $\sigma_P^2 \triangleq mode = normal \land t \geq 9.5 \Rightarrow 14.406 \leq \lambda(t) \leq 14.994$. Here, the magnitude bound of the air-fuel ratio at the *normal* mode of the system inferred from Hynger and Daikon is $\hat{\varphi}_P^2 \triangleq mode = normal \land t \geq 9.5 \Rightarrow 14.645 \leq \lambda(t) \leq 14.84$. 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 fuel control system as we change the proportional and integral gains of its PI controller. Table 3 describes the comparison between the actual physical specification σ_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 λ when $t \geq t_s$ and $mode = normal$. In Table 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 fuel control system.

Controller Gain	$\hat{\varphi}_P^2 \downarrow \lambda$	$\hat{\varphi}_P^2 \Rightarrow \sigma_P^2$	$\sigma_P^2 \Rightarrow \hat{\varphi}_P^2$
$c_{13} = 0.01, c_{14} = 0.14$	$14.567 \leq \lambda(t) \leq 15.058$	*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 3. Experiment results illustrate the comparison between actual physical specifications and inferred physical invariants from Hynger and Daikon of the fuel control system when changing the proportional gain and the integral gain of its PI controller.

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 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, etc. 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 in Daikon would leverage the methodology presented in this paper.

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 [20] 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 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 [46]. 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 [5], a formal and dynamic runtime assurance environment may be feasible.

8 CONCLUSION & FUTURE WORKS

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.

Overall, there are several directions for future research, including: (*a*) extending the classes of invariants that may be inferred, particularly to nonlinear (polynomial) [42] and disjunctive/max-plus forms [44], potentially by integrating Daikon with techniques from Dig [43], (*b*) runtime assurance and verification with real-time reachability of inferred invariants [5], (*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.

ACKNOWLEDGMENTS

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS 1464311, EPCN 1509804, and SHF 1527398, the Air Force Research Laboratory (AFRL) through 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) under contract numbers FA9550-15-1-0258 and FA9550-16-1-0246. The U.S. government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of AFRL, AFOSR, or NSF.

REFERENCES

[1] 1996. *ARIANE 5 Flight 501 Failure, Report by the Inquiry Board.* Technical Report. ESA Inquiry Board, Paris, France. https://www.ima.umn.edu/~arnold/disasters/ariane5rep.html

[2] R. Alur, Thao Dang, J. Esposito, Yerang Hur, F. Ivancic, V. Kumar, P. Mishra, G.J. Pappas, and O. Sokolsky. 2003. Hierarchical modeling and analysis of embedded systems. *Proc. IEEE* 91, 1 (Jan. 2003), 11–28. DOI:http://dx.doi.org/10.1109/JPROC.2002.805817

[3] Rajeev Alur, Aditya Kanade, S Ramesh, and KC Shashidhar. 2008. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *Proceedings of the 8th ACM international conference on Embedded software.* ACM, 89–98.

[4] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. 2011. S-taliro: A tool for temporal logic falsification for hybrid systems. In *Tools and Algorithms for the Construction and Analysis of Systems.* Springer.

[5] Stanley Bak, Taylor T. Johnson, Marco Caccamo, and Lui Sha. 2014. Real-Time Reachability for Verified Simplex Design. In *IEEE Real-Time Systems Symposium (RTSS).* IEEE Computer Society, Rome, Italy.

[6] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. (2010). http://smt-lib.org/

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

[8] Saddek Bensalem, Marius Bozga, Axel Legay, Thanh-Hung Nguyen, Joseph Sifakis, and Rongjie Yan. 2014. Component-based verification using incremental design and invariants. *Software & Systems Modeling* (2014), 1–25. DOI:http://dx.doi.org/10.1007/s10270-014-0410-8

[9] Francesco Bernardini, Marian Gheorghe, Francisco Jose Romero-Campero, and Neil Walkinshaw. 2007. A Hybrid Approach to Modeling Biological Systems. In *Membrane Computing*, George Eleftherakis, Petros Kefalas, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa (Eds.). LNCS, Vol. 4860. Springer Berlin Heidelberg, 138–159. DOI:http://dx.doi.org/10.1007/978-3-540-77312-2_9

[10] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. 2006. 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). ACM, New York, NY, USA, 169–180. DOI: http://dx.doi.org/10.1145/1146238.1146258

[11] C. Csallner, N. Tillmann, and Y. Smaragdakis. 2008. DySy: Dynamic symbolic execution for invariant inference. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*. 281–290. DOI: http://dx.doi.org/10.1145/1368088.1368127

[12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C: A Software Analysis Perspective. In *Software Engineering and Formal Methods*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). LNCS, Vol. 7504. Springer Berlin Heidelberg, 233–247. DOI: http://dx.doi.org/10.1007/978-3-642-33826-7_16

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. 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)*. Springer-Verlag, 337–340.

[14] Alexandre Donzé. 2010. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Lecture Notes in Computer Science, Vol. 6174. Springer Berlin / Heidelberg, 167–170. DOI: http://dx.doi.org/10.1007/978-3-642-14295-6_17

[15] Parasara Sridhar Duggirala, Chuchu Fan, Sayan Mitra, and Mahesh Viswanathan. 2015. Meeting a Powertrain Verification Challenge. In *To appear in the Proceedings of International Conference on Computer Aided Verification (CAV 2015)*.

[16] Robert W. Erickson and Dragan Maksimović. 2004. *Fundamentals of Power Electronics* (2nd edition ed.). Springer. DOI: http://dx.doi.org/10.1007/b100747

[17] M.D. Ernst, J. Cockrell, William G. Griswold, and D. Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on* 27, 2 (2001), 99–123. DOI: http://dx.doi.org/10.1109/32.908957

[18] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (Dec. 2007), 35–45.

[19] Chuchu Fan, Parasara Sridhar Duggirala, Sayan Mitra, and Mahesh Viswanathan. 2015. Progress on Powertrain Verification Challenge with C2E2. In *ARCH '15: Proc. of the 2nd Workshop on Applied Verification for Continuous and Hybrid Systems*.

[20] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. 2011. SpaceEx: Scalable Verification of Hybrid Systems. In *Computer Aided Verification (CAV) (LNCS)*. Springer.

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

[22] Shamina Hossain, Sairaj Dhople, and Taylor T. Johnson. 2013. Reachability analysis of closed-loop switching power converters. In *Power and Energy Conference at Illinois (PECI)*. 130–134. DOI: http://dx.doi.org/10.1109/PECI.2013.6506047

[23] Xiaoqing Jin, Jyotirmoy V Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. 2014. Benchmarks for Model Transformations and Conformance Checking. In *1st International Workshop on Applied Verification for Continuous and Hybrid Systems (ARCH)*.

[24] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V. Deshmukh, and Sanjit A. Seshia. 2013. Mining requirements from closed-loop control models. In *Proceedings of the 16th international conference on Hybrid systems: computation and control (HSCC '13)*. ACM, New York, NY, USA, 43–52. DOI: http://dx.doi.org/10.1145/2461328.2461337

[25] Taylor T Johnson, Stanley Bak, and Steven Drager. 2015. Cyber-physical specification mismatch identification with dynamic analysis. In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*. ACM, 208–217.

[26] Taylor T. Johnson, Zhihao Hong, and A. Kapoor. 2012. Design verification methods for switching power converters. In *Power and Energy Conference at Illinois (PECI), 2012 IEEE*. 1–6. DOI: http://dx.doi.org/10.1109/PECI.2012.6184587

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

[28] Nancy G Leveson. 2002. System safety engineering: Back to the future. *Massachusetts Institute of Technology* (2002).

[29] J. L. Lions. 1996. *Ariane 5 Flight 501 Failure*. Technical Report. Paris, France. http://www.di.unito.it/~damiani/ariane5rep.html

[30] Nancy Lynch, Roberto Segala, and Frits Vaandrager. 2003. Hybrid I/O automata. *Information and Computation* 185, 1 (2003), 105–157. DOI: http://dx.doi.org/10.1016/S0890-5401(03)00067-1

[31] Karthik Manamcheri, Sayan Mitra, Stanley Bak, and Marco Caccamo. 2011. A step towards verification and synthesis from Simulink/Stateflow models. In *Proc. of the 14th Intl. Conf. on Hybrid Systems: Computation and Control (HSCC)*. ACM, 317–318. DOI: http://dx.doi.org/10.1145/1967701.1967749

[32] Kevin McCaney. 2014. Pentagon's rapid plan for maintaining air superiority. http://defensesystems.com/Articles/2014/05/01/DARPA-system-of-systems-SoSITE.aspx. (2014).

[33] Ramy Medhat, S. Ramesh, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2015. A Framework for Mining Hybrid Automata from Input/Output Traces. In *Proceedings of the 12th International Conference on Embedded Software (EMSOFT '15)*. IEEE Press, Piscataway, NJ, USA, 177–186. http://dl.acm.org/citation.cfm?id=2830865.2830885

[34] Stefano Minopoli and Goran Frehse. 2016. SL2SX translator: from Simulink to SpaceEx models. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*. ACM, 93–98.

[35] Leonardo Moura and Nikolaj Bjørner. 2009. Satisfiability Modulo Theories: An Appetizer. In *Formal Methods: Foundations and Applications*, Marcel Medeiros Oliveira and Jim Woodcock (Eds.). LNCS, Vol. 5902. Springer Berlin Heidelberg, 23–36. DOI: http://dx.doi.org/10.1007/978-3-642-10452-7_3

[36] Sirajum Munir, Mohsin Y Ahmed, and John A Stankovic. 2015. EyePhy: Detecting Dependencies in Cyber-Physical System Apps due to Human-in-the-Loop. (2015).

[37] Sirajum Munir, John Stankovic, and others. 2014. DepSys: Dependency aware integration of cyber-physical systems for smart homes. In *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*. IEEE, 127–138.

[38] National Highway Traffic Safety Administration (NHTSA). 2011. Honda Automatic Transmission Control Module Software (Recall #11V395000). (Aug. 2011).

[39] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. DOI: http://dx.doi.org/10.1145/1250734.1250746

[40] Luan Viet Nguyen and Taylor T. Johnson. 2014. 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.

[41] Luan Viet Nguyen, Hoang-Dung Tran, and T.T. Johnson. 2014. Virtual Prototyping for Distributed Control of a Fault-Tolerant Modular Multilevel Inverter for Photovoltaics. *Energy Conversion, IEEE Transactions on* 29, 4 (Dec. 2014), 841–850. DOI: http://dx.doi.org/10.1109/TEC.2014.2362716

[42] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using Dynamic Analysis to Discover Polynomial and Array Invariants. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 683–693.

[43] T Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. *ACM Transactions on Software Engineering and Methodology, to appear* (2014).

[44] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. Using Dynamic Analysis to Generate Disjunctive Invariants. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 608–619. DOI: http://dx.doi.org/10.1145/2568225.2568275

[45] Jeremy W. Nimmer and Michael D. Ernst. 2002. Automatic generation of program specifications. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '02)*. ACM, New York, NY, USA, 229–239. DOI: http://dx.doi.org/10.1145/566172.566213

[46] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically Patching Errors in Deployed Software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 87–102. DOI: http://dx.doi.org/10.1145/1629575.1629585

[47] A Reder and A Egyed. 2013. Determining the Cause of a Design Model Inconsistency. *Software Engineering, IEEE Transactions on* 39, 11 (Nov. 2013), 1531–1548. DOI: http://dx.doi.org/10.1109/TSE.2013.30

[48] E.J. Schwartz, T. Avgerinos, and D. Brumley. 2010. 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*. 317–331. DOI: http://dx.doi.org/10.1109/SP.2010.26

[49] Len Staller. 2005. Understanding analog to digital converter specifications. *Embedded Systems Design* (2005).

[50] Arthur G Stephenson, Daniel R Mulville, Frank H Bauer, Greg A Dukeman, Peter Norvig, LS LaPiana, PJ Rutledge, D Folta, and R Sackheim. 1999. Mars climate orbiter mishap investigation board Phase I report, 44 pp. *NASA, Washington, DC* (1999).

[51] Stavros Tripakis, Christos Stergiou, Chris Shaver, and Edward A. Lee. 2013. A modular formal semantics for Ptolemy. *Mathematical Structures in Computer Science* 23 (8 2013), 834–881. Issue Special Issue 04. DOI: http://dx.doi.org/10.1017/S0960129512000278

[52] M.W. Whalen, A. Gacek, D. Cofer, A. Murugesan, M.P.E. Heimdahl, and S. Rayadurgam. 2013. Your What Is My How: Iteration and Hierarchy in System Design. *Software, IEEE* 30, 2 (March 2013), 54–60. DOI: http://dx.doi.org/10.1109/MS.2012.173

[53] Changyan Zhou and Ratnesh Kumar. 2012. Semantic Translation of Simulink Diagrams to Input/Output Extended Finite Automata. *Discrete Event Dynamic Systems* 22, 2 (2012), 223–247. DOI: http://dx.doi.org/10.1007/s10626-010-0096-1