Cyber-Physical Specification Mismatch Identification with Dynamic Analysis

Taylor T. Johnson
University of Texas at Arlington, USA

Stanley Bak
Air Force Research Laboratory, USA

Steven Drager
Air Force Research Laboratory, USA

ABSTRACT

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.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Verification

Keywords

Cyber-physical systems, dynamic analysis, specifications

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 most engineered systems, the vast majority of resources are devoted to ensuring systems meet their specifications [1]. However, in spite of significant technical advances for design 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 are frequent product recalls across industries for safety concerns due to software problems and systems integration between the cyber and physical subcomponents.

The verification community typically focuses on *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 write software and systems are integrated from numerous components in a potentially error-prone process.

In this paper, we develop methods to address such challenges that arise in the product evolution and upgrade process in CPS. We develop a method to enable 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, our method infers candidate invariants of SLSF models. Invariants are properties of a system that 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 systems model checkers, and finding invariants can aid this process, as they represent potential abstractions with a possibly less complex representation than the set of reachable

DISTRIBUTION A. Approved for public release; Distribution unlimited. (Approval AFRL PA #88ABW-2014-4829, 17 OCT 2014).

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 [2]. Candidate invariants inferred with our Hynger (for HYbrid iNvariant GEneratoR) software tool in conjunction with Daikon [3, 4] may be formally checked as actual invariants using a hybrid systems model checker [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 proof-of-concept power electronics CPS case study using Hynger to automatically identify cyber-physical specification mismatches. Overall, these results can be used to help bridge the worlds of actual embedded systems software (e.g., detailed SLSF diagrams and generated C code) with hybrid systems models. 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

In this section, we review cases where CPS design reuse has led to mistakes in existing systems. This motivates the need for our 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 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 [6]. Specifically [6]: "Beginning with model year 2005 4-cylinder Accord and Element vehicles, specifications for the secondary shaft bearing outer race material and shape were modified in order to accommodate increased engine torque. These modifications, which improved the long-term durability of the component but reduced its resistance to shock, are not appropriately addressed in the automatic transmission control module software of the affected vehicles." 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 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 [7, 8]. The following quotations from the inquiry into the cause of Ariane 5 flight 501's failure [8] highlight the issues with cyber-physical reuse and specification mismatches (emphasis added):

- "The design of the **Ariane 5** SRI [Inertial Reference System] is practically the same as that of an SRI which is presently used on **Ariane 4**, *particularly as regards the software*."
- "The value of BH [Horizontal Bias] was much higher than expected because the early part of *the trajectory of **Ariane 5** differs from that of **Ariane 4*** and results in *considerably higher horizontal velocity values*."
- "**Ariane 5** has a *high initial acceleration* and a trajectory which leads to a *build-up of horizontal velocity which is five times more rapid* than for **Ariane 4**. The *higher horizontal velocity* of **Ariane 5** generated, within the 40-second timeframe, leads to the excessive value which caused the inertial system computers to cease operation."
- "In **Ariane 4** flights using the same type of inertial reference system there has been no such failure because the trajectory during the first 40 seconds of flight is such that *the particular variable related to horizontal velocity cannot reach*, with an adequate operational margin, a value beyond the limit present in the software."
- "The reason for the three remaining variables, including the one denoting horizontal bias, being unprotected was that further reasoning indicated that they *were either physically limited or that there was a large margin of safety*, a reasoning which in the case of the variable BH turned out to be faulty."

Here, software 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 DARPA SoSITE program [9] describes modularized military aviation systems which are capable of rapid component swapping and upgrade. Left unaddressed, issues related unstated assumptions in components are likely to get worse in such future CPS, where changes can occur in the software and hardware.

3. BUCK CONVERTER WITH HYSTERESIS CONTROL

Next, 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) [10], all of which have similar modeling, but we focus particularly on a buck converter. 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 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 [11]. For these converters, one of the physical specifications is that, at steady-state, the output voltage is always bounded within a tolerance of a desired (reference) voltage (e.g., $V_{out}(t) = V_{ref}(t) \pm \epsilon$). We will formalize specifications and mismatches in Section 4. As a prelude, we highlight that Hynger finds this candidate invariant (that can be shown to be an actual invariant when modeled as a hybrid automaton [10, 12, 13]).

Additionally, experimental results for additional examples

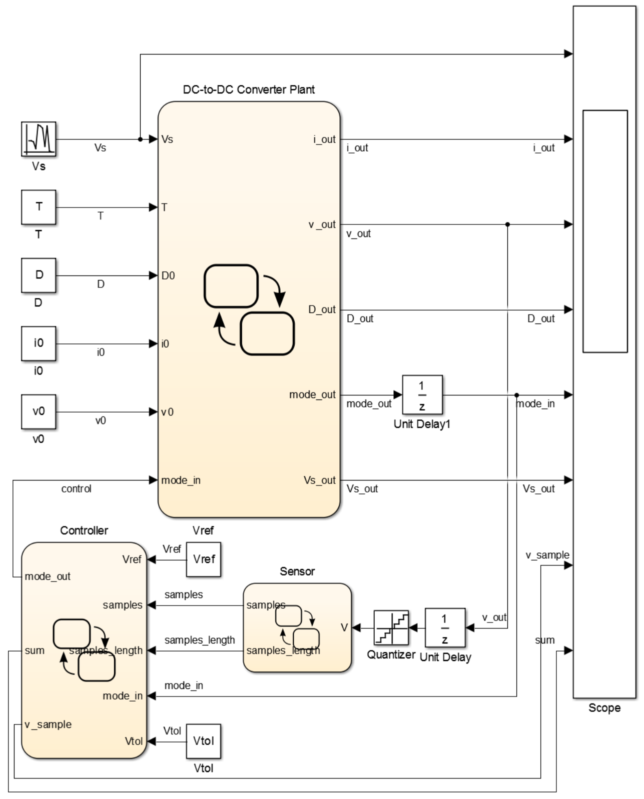


Figure 1: General CPS case study architecture overview in SLSF. The system is composed of a plant (physical system) model, a controller (software/cyber), and potentially sensor and actuator models. The cyber model uses some of the physical model output states to determine a control action or input. For the buck converter, the plant model is described as a hybrid automaton in Figure 3, the controller appears in Figure 4, and the sensor model appears in Figure 2.

using Hynger are presented later in Section 6, although we do not have space to describe them in detail in this paper. The general architecture of the case study 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. 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. An overview of the buck converter system architecture appears in the root SLSF diagram of Figure 1.

3.1 Buck Converter Plant Model

We briefly describe a switched-system plant model of a buck converter, but in-depth derivations of these models and non-idealized versions in power electronics textbooks [14, 15]. The buck converter plant may be modeled as a hybrid automaton as described in detail [10–13].

The buck converter has two real-valued state variables modeling the inductor current i_L and the capacitor voltage V_C. These state variables are written in vector form as: $x = [i_L; V_C]$. The dynamics of the continuous variables in each mode $m \in \{Open, Close\}$—in short, o and c—are specified as linear (affine) differential equations: $\dot{x} = A_m x + B_m u$, where $u = V_S$. Figure 3 shows a hybrid automaton model for the converter and is easily translated to a continuous-time Stateflow model (see, e.g., [10]). 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).

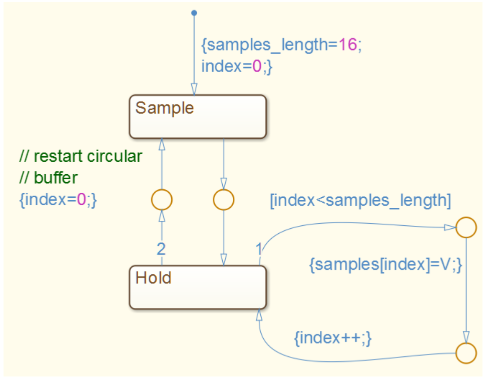


Figure 2: Stateflow model of sensor with a sample-and-hold for the buck converter case study.

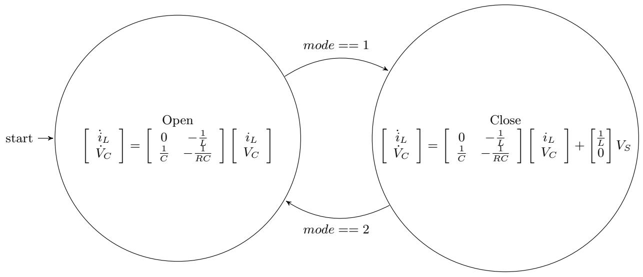

start
Open
$\begin{bmatrix} \dot{i}_L \\ \dot{V}_C \end{bmatrix} = \begin{bmatrix} 0 & -\frac{1}{L} \\ \frac{1}{C} & -\frac{1}{RC} \end{bmatrix} \begin{bmatrix} i_L \\ V_C \end{bmatrix}$
Close
$\begin{bmatrix} \dot{i}_L \\ \dot{V}_C \end{bmatrix} = \begin{bmatrix} 0 & -\frac{1}{L} \\ \frac{1}{C} & -\frac{1}{RC} \end{bmatrix} \begin{bmatrix} i_L \\ V_C \end{bmatrix} + \begin{bmatrix} \frac{1}{L} \\ 0 \end{bmatrix} V_S$
$mode == 1$
$mode == 2$

Figure 3: Hybrid automaton model of the buck converter plant. A corresponding continuous-time Stateflow model is used for the plant.

3.2 Closed-Loop Buck Converter

Many control strategies exist for closed-loop feedback control of buck converters. The control strategy we will consider is hysteresis control, which operates intuitively similar to typical heating, ventilation, and cooling (HVAC) control with a thermostat. 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 $V_{out} \approx V_{ref}$. We note that V_{tol} appears in the controller Figure 4, so an assumption is made about the relationship between V_{tol} and V_{rip} in software.

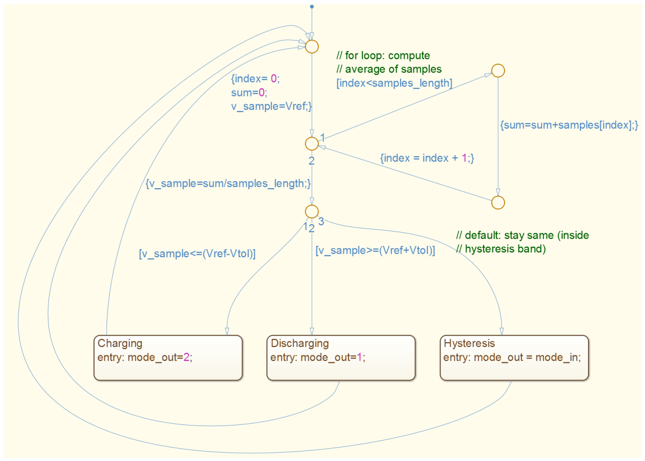


Figure 4: Stateflow model of buck-converter voltage hysteresis controller. The digitized output voltage from the buck-converter plant is used to determine the mode of the switch. Here, V_{tol} is denoted by the variable `Vtol`, V_{ref} is `Vref`. We highlight that the controller computes a moving average 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.

3.3 Dynamic Invariant Inference with Daikon

Next, we illustrate the dynamic invariant inference methodology used by Daikon on a purely software example. However, this purely software example (a C function) is actually specified for the controller in the buck converter case study in a different manner. The loop in the controller SLSF model of Figure 4 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.

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 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 [3]. Additionally, a formalized correctness specification is given in the modern ANSI/ISO C Specification Language (ACSL) specification language, used by tools such as Frama-C [16]. Using Daikon and a small suite of unit tests, we were able to successfully find the invariant that at all 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 4 using Hynger and Daikon.

4. CYBER-PHYSICAL SPECIFICATIONS AND MISMATCHES

The approach presented in this paper applies to 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 a 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 [2]). 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).

4.1 Cyber-Physical System Models

We next define a structure of CPS models used in SLSF to formalize the specification mismatch problem. We will not define a formal semantics of this structure or SLSF diagrams here. However, in the case where the SLSF diagram is white box and a formal semantics may be defined, a formal framework like hybrid input/output automata (HIOA) [17] may be used to specify the semantics, such as done in the HyLink tool [2]. Other formalisms like actors and hierarchical state machines are commonly used for formal modeling of other diagrammatic frameworks similar to SLSF [18–21]. 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 systems model checker like SpaceEx [5]. For instance, in some instances, we know when a SLSF diagram corresponds precisely hybrid automaton model [2], 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 and similar graphical modeling languages. A SLSF diagram is a tuple $\mathcal{A} \triangleq \langle M, E, \mathsf{V} \rangle$, where: ($a$) M is a set of blocks (vertices) that represent block diagrams (and sub-blocks/models), (b) $E \subseteq M \times M$ is a set of edges between blocks representing a parent-child hierarchy, and (c) for each block $v \in M$, $\mathsf{V}(v)$ is a set of variables, and $\mathsf{V} \triangleq \bigcup_{v \in M} \mathsf{V}(v)$. 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 will denote $\mathit{root}(M)$. For a block $v \in M$, the *children* of v are denoted $\mathit{children}(v)$ and defined as the set of blocks $\{w \in M \mid w \in E(v)\}$. For a block $v \in M$, the *parent* of v is denoted $\mathit{parent}(v)$ and is defined as the singleton set $\{w \in M \mid v \in \mathit{children}(w)\}$. Clearly, $\mathit{parent}(\mathit{root}(M)) = \emptyset$. For a block $v \in M$, the *ancestors* of v are denoted $\mathit{ancestors}(v)$ and defined inductively as the set of blocks $\{w \in M \mid v \cup w \in \mathit{children}(v) \cup \mathit{children}(w)\}$ (or equivalently, as the transitive closure of $\mathit{children}(v)$).

For a block $v \in M$, the set of variables of v is $\mathsf{V}(v)$ and

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

Figure 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
```

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

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 $\mathbf{x}$ of the state-space is called a state, and a trace is a sequence of states. The SLSF diagram may also be 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. Furthermore, the set of variables of v is also partitioned into sets of physical and cyber variables, denoted respectively as $\mathsf{V}_P(v)$ and $\mathsf{V}_C(v)$, and we have $\mathsf{V}(v) = \mathsf{V}_P(v) \cup \mathsf{V}_C(v)$. In practice, this may accomplished with subtyping using e.g. an overloaded type for floats or fixed points used for approximations of real variables (e.g., in C, `typedef double physical; typedef physical temperature;`).

Note that the input and output variables are disjoint, and the cyber and physical variables are disjoint, although these are not all mutually disjoint. For a block $v \in V$ and variable $x \in \mathsf{V}(v)$, we say: (a) x is an *input cyber variable* if $x \in \mathsf{V}_C(v)$ and $x \in \mathsf{V}_I(v)$, (b) x is an *output cyber variable* if $x \in \mathsf{V}_C(v)$ and $x \in \mathsf{V}_O(v)$, (c) x is an *input physical variable* if $x \in \mathsf{V}_P(v)$ and $x \in \mathsf{V}_I(v)$, and (d) x is an *output cyber variable* if $x \in \mathsf{V}_P(v)$ and $x \in \mathsf{V}_O(v)$. We extend these notations 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 cyber and physical 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 CPS models 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 model. We note that this is related to the notion of tainted variables in program analysis that is popular in security [22]. 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.

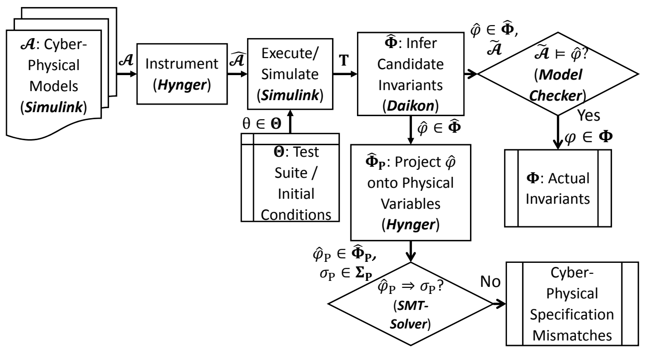
Input-output connections may only exist between models with the same parent (i.e., those in the same hierarchical structure). For a block $v \in M$, we denote all blocks with same parent as *siblings*(v), which is defined as the set $\{w \in M \mid parent(w) = parent(v)\}$. Output variables of a block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

4.2 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 represented in the Satisfiability Modulo Theories (SMT) library standard language [23, 24]. Under these assumptions, candidate invariants may be specified as quantifier-free SMT formulas over the variables of the SLSF model, V, where the type of a variable corresponds to the SMT sort. For a formula ϕ, let *vars*(ϕ) be the set of variables appearing in ϕ. For a formula ϕ: (a) if *vars*(ϕ) are all physical, then ϕ is a *physical specification*, (b) if *vars*(ϕ) are all cyber, then ϕ is a *cyber specification*, and (c) if *vars*(ϕ) 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 7, we separate the specification Σ into sets of cyber and physical specification, written respectively as Σ_C and Σ_P. Here Σ_P denotes the

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



𝓐: Cyber-Physical Models (Simulink)
𝓐
Instrument (Hynger)
𝓐̂
Execute/ Simulate (Simulink)
T
Φ̂: Infer Candidate Invariants (Daikon)
φ̂ ∈ Φ̂, 𝓐̂
𝓐̂ ⊨ φ̂? (Model Checker)
Yes
φ ∈ Φ
Φ: Actual Invariants
θ ∈ Θ
Θ: Test Suite / Initial Conditions
φ̂ ∈ Φ̂
Φ̂P: Project φ̂ onto Physical Variables (Hynger)
φ̂P ∈ Φ̂P, σP ∈ ΣP
φ̂P ⇒ σP? (SMT-Solver)
No
Cyber-Physical Specification Mismatches

Figure 7: Hynger overview, inference of physical specifications assumed by software, and cyber-physical specification mismatch identification.

set of physical specifications. This specification includes 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 specification. The cyber specification includes 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 7 and described next.

4.3 Cyber-Physical Specification Mismatches

A CPS model or implementation will be provided as a 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 hybrid automaton) or may be converted to one, $\tilde{\mathcal{A}}$. If that is the case, then a hybrid systems model checker may be employed to see if $\hat{\varphi}$ is an actual invariant φ, and the set of actual invariants Φ is collected.

In all cases, next 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 [25]. 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 correspond 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. 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.

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[2]. 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[3].

Given a 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, formatted in the trace input format required by Daikon. While configurable, the default behavior of Hynger is to add instrumentation (observation) points for every input and output signal for every block (recursively) in the SLSF diagram. That is, Hynger walks the tree of blocks starting from the root, and for each $v \in M$, adds instrumentation points for the input variables $\mathsf{V}_I(v)$ and the output variables $\mathsf{V}_O(v)$ of v. Of course, this may incur a drastic performance overhead, so if this is not desired, the user may select only a subset of the blocks to instrument and our performance results (see Section 6) illustrate this distinction. When a 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.

[2]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.

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

The Hynger flow is summarized in Figure 7. 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.

6. EXPERIMENTAL RESULTS

Hynger was tested on Windows 8.1 64-bit using Matlab 2014a and 2014b, executed on a modern x86-64 laptop with a 2.7 GHz quad-core Intel i7-4800MQ processor and 32 GB RAM. All performance metrics reported were recorded on this system using Matlab 2014b. 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 3 and detailed further in [10], *(b)* a solar array case study that uses a buck-boost converter [11], *(c)* benchmarks from S-TaLiRo [26], *(d)* benchmarks from Breach [28,29], *(e)* benchmarks created as a part of the ARCH 2014 CPSWeek workshop (particularly [10,27]) 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 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 [4,30]. 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 illustrate these differences to give some comparison of how the methods scale on a given model.

Model	Solver	Tmax	Sim	SimInst + Inv	SimInst	Inv	Overhead	BDAll	BDInst	BDPct
buck (Section 3)	ode45	0.0083333	4.0575	32.1449	28.7098	3.4351	7.2299	14	3	21.4286
buck (Section 3)	ode45	0.0083333	2.2973	30.5966	26.8555	3.7411	13.8531	14	4	28.5714
buck (Section 3)	ode45	0.0083333	2.1576	54.7123	51.1736	3.5387	25.8249	14	14	100
heat25830 [26]	ode45	50	2.1139	189.7082	188.235	1.4732	89.743	28	1	3.5714
heat25830 [26]	ode45	50	3.604	1675.9307	1674.8379	1.0928	465.0181	28	10	35.7143
fuel1 [27]	ode15s	15	3.3557	412.4223	409.2978	3.1244	122.9025	208	17	8.173
fuel1 [27]	ode15s	15	1.5293	1434.4582	1428.3545	6.1038	938.0088	208	63	30.2885
fuel2 [27]	ode15s	20	1.2206	254.8157	252.5842	2.2315	208.7635	135	13	9.630

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 5 runs. Sim is the simulation runtime (s). SimInst + Inv is the instrumented simulation time (Hynger) plus invariant generation runtime (Daikon) (s). SimInst is the instrumented simulation runtime (Hynger) (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., $BDInst/BDAll$).

Closed-Loop Buck Converter Cyber-Physical Specification Mismatch. A basic cyber-physical specification mismatch is easy to encode in the buck converter, since the software controller inherently uses a tolerance to encode the desired output voltage ripple. This hysteresis tolerance band is typically chosen based on the system dynamics and desired output voltage ripple to ensure the output voltage meets the ripple specification. As a concrete example, the physical specification may contain a constraint that $V_{out} = V_{ref} \pm V_{rip}$, and fix for example $V_{ref} = 5V$ and $V_{rip} = 0.1V$. The hysteresis band V_{tol} is then be selected based on the system dynamics to ensure $4.9V \leq V_{out} \leq 5.1V$, so as to meet the requirements of the physical specifications defined by Σ_P in Section 4.2. If the plant changes (i.e., different circuit elements are used), and the software is not updated with a new hysteresis band V_{tol} to accommodate the plant dynamics changes, 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.

7. 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 purely software systems [31]. Cyber-physical specification mismatch is closely related to model inconsistency [32], architectural mismatch [33], and requirements consistency [34]. There are many benefits of dynamic analysis such as using implementations instead of models [3,4,31] to find dynamic program specifications [31], such as providing documentation over program evolution, checking if specifications change drastically over program evolution, etc. For one, models are not actually required for analysis, and implementations may be used [3,4]. 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 is results are unsound without additional reasoning.

Finding specifications is a maturing field within software engineering [3,4,31,35–37]. Daikon, which is used by Hynger, processes program traces to generate invariants [3,4]. For several languages (C, C++, etc.), this process is performed without access to the source code by instrumenting the compiled program using Valgrind [30]. 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 [38], 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 $x \leq 50$. Other research tools like DySy [36] and commercial tools like Agitagor [35] can be used for generating candidate invariants for other languages.

There are also semi-formal and formal tools for analyzing SLSF diagrams akin to the Hynger tool developed in this work. For example, there are commercial tools such as Reactive Systems' Reactis and Esterel's SLSF-to-Lustre translation tool. For discrete or discretized systems, translations to formal models like extended input/output automata have been developed [19]. Some recent simulation-based verification, testing, and falsification approaches in hybrid systems and CPS like those in S-TaLiRo, Breach, and C2E2 can be viewed as forms of dynamic analysis (potentially with additional annotations and proof steps) [26, 28, 29, 39–41]. Additionally, there are alternative methods for finding specifications for SLSF models [29, 39]. While such approaches have been applied to infer parameters used in more general specifications than just invariants (e.g., temporal properties specified in variants of dense-time temporal logic like metric [42] and signal temporal logic [29]), our work is differentiated in several regards. Of course, to use tools like Daikon, template specifications are also required. However, the class of templates allowed by Daikon include significantly more complex software state (arrays, algebraic data types like lists, etc.) than that possible using, e.g., STL, so we illustrate this with the array example described earlier in this paper (Figures 4 and 5) that shows up frequently in CPS software (a moving-average filter). Additionally, the templates are encoded in Daikon itself, but the user can add new templates to Daikon. As we rely on Daikon for dynamic analysis, our method is template-based, albeit we have access to a large set of templates, specifically for the cyber (software) aspects, such as potentially complex data structures that can not be handled by existing template-based methods for SLSF specification inference [29].

8. CONCLUSION

The results illustrate the feasibility of using dynamic invariant inference for analysis of embedded and cyber-physical systems (CPS). 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. Supposing no model is available (in the black box case), the candidate invariants represent what may be the most formal model available. If a formal model is available (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 a first use of dynamic execution analysis for hybrid systems with sophisticated software state and discrete complexity.

Future Work. 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 extending the invariant templates to be able to capture more interesting relations, particularly for physical variables, is planned. 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 actually invariants or not (using, e.g., SpaceEx [5] or other hybrid systems model checkers), as well as identify specification mismatches. 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 [43]. 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 [44], a formal and dynamic runtime assurance environment may be feasible.

Overall, there are several directions for future research, including: (a) extending the classes of invariants that may be inferred, particularly to nonlinear (polynomial) [45] and disjunctive/max-plus forms [46], potentially by integrating Daikon with techniques from Dig [47], (b) checking if the inferred invariants are actual invariants by using formal models of the underlying SLSF model diagrams using hybrid systems model checkers such as SpaceEx [5], etc., (c) runtime assurance and verification with real-time reachability of inferred invariants [44], (d) 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 (e) analyzing industrial-scale CPS using Hynger.

Acknowledgments

The material presented in this paper is based upon work supported by the Air Force Research Laboratory's Information Directorate (AFRL/RI) through the Visiting Faculty Research Program (VFRP) under contract number FA8750-13-2-0115 and the Air Force Office of Scientific Research (AFOSR). Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the AFRL/RI or AFOSR.

9. REFERENCES

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

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

[3] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *Software Engineering, IEEE Transactions on*, vol. 27, no. 2, pp. 99–123, 2001.

[4] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, Dec. 2007.

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

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

[7] J. L. Lions, "Ariane 5 flight 501 failure," Paris, France, Tech. Rep., Jul. 1996. [Online]. Available: http://www.di.unito.it/~damiani/ariane5rep.html

[8] "Ariane 5 flight 501 failure, report by the inquiry board," ESA Inquiry Board, Paris, France, Tech. Rep., Jul. 1996. [Online]. Available: https://www.ima.umn.edu/~arnold/disasters/ariane5rep.html

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

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

[11] L. V. Nguyen, H.-D. Tran, and T. Johnson, "Virtual prototyping for distributed control of a fault-tolerant modular multilevel inverter for photovoltaics," *Energy Conversion, IEEE Transactions on*, vol. 29, no. 4, pp. 841–850, Dec. 2014.

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

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

[14] R. P. Severns and G. Bloom, *Modern DC-to-DC Switchmode Power Converter Circuits*. New York, New York: Van Nostrand Reinhold Company, 1985.

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

[16] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," in *Software Engineering and Formal Methods*, ser. LNCS, G. Eleftherakis, M. Hinchey, and M. Holcombe, Eds. Springer Berlin Heidelberg, 2012, vol. 7504, pp. 233–247.

[17] N. Lynch, R. Segala, and F. Vaandrager, "Hybrid I/O automata," *Information and Computation*, vol. 185, no. 1, pp. 105–157, 2003.

[18] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, P. Mishra, G. Pappas, and O. Sokolsky, "Hierarchical modeling and analysis of embedded systems," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 11–28, Jan. 2003.

[19] C. Zhou and R. Kumar, "Semantic translation of simulink diagrams to input/output extended finite automata," *Discrete Event Dynamic Systems*, vol. 22, no. 2, pp. 223–247, 2012.

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

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

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

[23] L. Moura and N. Bjørner, "Satisfiability modulo theories: An appetizer," in *Formal Methods: Foundations and Applications*, ser. LNCS, M. M. Oliveira and J. Woodcock, Eds. Springer Berlin Heidelberg, 2009, vol. 5902, pp. 23–36.

[24] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard: Version 2.0," 2010. [Online]. Available: http://smt-lib.org/

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

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

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

[28] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds. Springer Berlin / Heidelberg, 2010, vol. 6174, pp. 167–170.

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

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

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

[32] A. Reder and A. Egyed, "Determining the cause of a design model inconsistency," *Software Engineering, IEEE Transactions on*, vol. 39, no. 11, pp. 1531–1548, Nov. 2013.

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

[34] M. Whalen, A. Gacek, D. Cofer, A. Murugesan, M. Heimdahl, and S. Rayadurgam, "Your what is my how: Iteration and hierarchy in system design," *Software, IEEE*, vol. 30, no. 2, pp. 54–60, Mar. 2013.

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

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

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

[38] F. Bernardini, M. Gheorghe, F. J. Romero-Campero, and N. Walkinshaw, "A hybrid approach to modeling biological systems," in *Membrane Computing*, ser. LNCS, G. Eleftherakis, P. Kefalas, G. Paun, G. Rozenberg, and A. Salomaa, Eds. Springer Berlin Heidelberg, 2007, vol. 4860, pp. 138–159.

[39] H. Yang, B. Hoxha, and G. Fainekos, "Querying parametric temporal logic properties on embedded systems," in *International Conference on Testing Software and Systems*, ser. Lecture Notes in Computer Science, B. Nielsen and C. Weise, Eds. Springer Berlin Heidelberg, 2012, vol. 7641, pp. 136–151.

[40] P. S. Duggirala, S. Mitra, and M. Viswanathan, "Verification of annotated models from executions," in *Proceedings of the Eleventh ACM International Conference on Embedded Software (EMSOFT '13)*. Piscataway, NJ, USA: IEEE Press, 2013.

[41] Z. Huang and S. Mitra, "Proofs from simulations and modular annotations," in *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '14. New York, NY, USA: ACM, 2014, pp. 183–192.

[42] J. Ouaknine and J. Worrell, "Some recent results in metric temporal logic," in *Formal Modeling and Analysis of Timed Systems*, ser. LNCS, F. Cassez and C. Jard, Eds. Springer Berlin Heidelberg, 2008, vol. 5215, pp. 1–13.

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

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

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

[46] ——, "Using dynamic analysis to generate disjunctive invariants," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 608–619.

[47] ——, "DIG: A dynamic invariant generator for polynomial and array invariants," *ACM Transactions on Software Engineering and Methodology, to appear*, 2014.