2013 IEEE 37th Annual Computer Software and Applications Conference

Cyber/Physical Co-Verification for Developing Reliable Cyber-Physical Systems

Yu Zhang[†‡*], Fei Xie[‡], Yunwei Dong[†], Xingshe Zhou[†] and Chunyan Ma[§]
[†]*School of Computer Science, Northwestern Polytechnical University, Xi'an, 710072, China*
[‡]*Department of Computer Science, Portland State University, Portland, OR 97207, USA*
[§]*School of Software & Microelectronics, Northwestern Polytechnical University, Xi'an, 710072, China*
[*]*yuzhang.nwpu@gmail.com*

***Abstract*—Cyber-Physical Systems (CPS) tightly integrate cyber and physical components and transcend discrete and continuous domains. It is greatly desired that the physical components being controlled and the software implementation of control algorithms can be verified together. We present an efficient approach to reachability analysis of Hybrid Automata Pushdown System (*HAPS*) models for cyber/physical co-verification of CPS. We have realized this approach and applied it to real-world control systems. The evaluation has shown that *HAPS* is an effective model for co-verification of CPS and our approach has major potential in verifying system-level properties of CPS, therefore improving the reliability of CPS.**

***Keywords*-Cyber-Physical Systems; Co-Verification; Model Checking; Symbolic Execution**

I. Introduction

Cyber-Physical Systems (CPS) [1] are physical and engineered systems whose operations are integrated, monitored, and controlled by embedded computational cores. CPS tightly integrate discrete cyber components and continuous physical components. These two types of components cooperatively deliver system functionalities and jointly contribute to system-level properties. CPS engineering must account for the interacting and interdependent behaviors of both types of components. As a consequence, formal verification of correctness properties is a key step in guaranteeing safety of many important CPS such as medical devices, automobiles, robotics, avionics and other critical infrastructures.

Recently, there has been extensive research on formal verification of CPS, including [2], [3], [4] and many others. The focus of these approaches has mainly been on correctness of high-level control algorithm designs. CPS are often modeled as hybrid automata [5], which combines automaton transitions for capturing discrete changes with differential equations for continuous changes. Well-known tools for verifying such systems include iSat [6], HyTech [7] and Uppaal [8]. However, existing tools have focused on high-level mathematical models and largely ignored the implementation aspects of controllers. In such a situation, a number of simplifying assumptions have been made when modeling controllers. There are two major issues:

Firstly, control engineers often overlook numerically unstable implementations. For example, Fig. 1 illustrates the C language implementation of an algorithm [9] that first computes the function $\mathtt{x} = (c_1b_2 - c_2b_1)/(a_1b_2 - a_2b_1)$ (a_1, a_2, b_1, b_2, c_1, $c_2 \in \mathbb{R}$), and returns 1 iff x is non-negative; otherwise return 0. For the input a1 = 37639840, a2 = 29180479, b1 = −46099201, b2 = −35738642, c1 = 0, and c2 = 1, the computation of x should be −46099201 by the algorithm. However, the program computes the value x = 0.343466 on a PC running Linux using the gcc compiler. Since x determines the output of the function Control_Test, which may lead to a downstream choice of a different controller being activated, such an unstable computation may result in compromised safety of systems.

```
1  /*****************************************
2  *Output: return 1 if x is non-negative;
3  *        otherwise 0.
4  *****************************************/
5  int Control_Test(float a1, float a2, float b1,
       float b2, float c1, float c2)
6  {
7    ...
8    float x = (c1*b2-c2*b1)/(a1*b2-a2*b1);
9    if (x > 0.0)  return 1;
10   else return 0;
11 }
```

Figure 1: An example of unstable software implementation.

Secondly, software engineers often ignore key features of control system design goals. When designing controllers, the control engineers explicitly consider key features of control systems, such as robustness. The field of feedback control is mature enough to answer many questions in controller design based on efficient algorithms and tools. Once a control system has been designed and verified, software engineers implement the controller. However, it is not clear if these design goals are maintained on the software implementation level. It is often unknown whether the software controller is safe or not with slight perturbations in the inputs and outputs to the system. Input values of the controller with slight perturbations when sampling can execute entirely different code paths and can potentially produce very different outputs.

In this paper, we present a cyber/physical co-verification approach to ensure high confidence of CPS. The foundation of this approach is a Hybrid Automata Pushdown System

0730-3157/13 $26.00 © 2013 IEEE
DOI 10.1109/COMPSAC.2013.88

as a unifying model for cyber/physical co-verification. Co-verification, verifying software and physical components (or plants) together, is essential to establishing the correctness of a complete system. It takes into account both the numerical implementations and key features of control system design. We construct co-verification model in a C program and verify the C program through symbolic execution. Our reachability analysis checks whether a system is safe or not in all the paths of the systems within a bounded step k.

One major challenge in co-verification is the integration of software and plant representations within the same formal model. Software and plant verification utilize different models. For verification of software implementations, one of the most popular models is pushdown systems whose semantics closely resemble the semantics of software programs. Plants in CPS operate in a time continuum, typically medelled as hybrid automata. However, for co-verification, it is not desired to model both software and plant as pushdown systems or hybrid automata (see section III). This paper makes the following contributions:

- *Co-verification model*. We designed a formal co-verification model, Hybrid Automata Pushdown System (*HAPS*), to capture software and plant designs, as well as their concurrent executions and interactions. The core contribution is our process for constructing a *HAPS* by synchronizing a pushdown system that abstracts software and a hybrid automata that abstracts plant.
- *Model checking of HAPS*. We developed a method for checking safety properties of a *HAPS*. We verify these properties of a *HAPS* through reachability analysis based on symbolic execution. Safety properties are desired correlations among software and plant events. Events in CPS are basically boolean propositions over cyber and physical variables.

We have implemented our approach and successfully applied it to co-verification of real-world control systems. Preliminary evaluation has shown that *HAPS* is an effective model for cyber/physical co-verification and our approach has major potential in improving reliability of CPS.

The rest of this paper is organized as follows. Section II discusses the related work. Section III introduces the background of this work. Section IV presents our co-verification model. Section V describe how to construct a *HAPS*. Section VI discuss how to conduct reachability analysis on a *HAPS*. Section VII elaborates on application of our approach to real-world systems. Section VIII concludes the paper and discusses future work.

II. Related Work

The CPS concept transcends embedded systems and hybrid systems. Embedded systems research has been largely focused on hardware and software and, particularly, their interactions. Hybrid systems research has been largely focused on the interactions between discrete and continuous domains on high level. CPS research takes a comprehensive view of the system, instead of focusing on a single semantic gap, focuses on multiple semantic gaps simultaneously.

In embedded systems research, various formal languages have been proposed for specifying embedded systems, e.g., Co-design Finite State Machines (CFSMs) [10], and petri-net based languages such as PRES [11]. However, they do not consider physical dynamics. In [12], a static analysis tool FLUCTUAT was presented, which applied affine arithmetic to reason about the precision of floating point C code. It has successfully been used on small, but numerically important, parts of embedded controllers and has been able to identify numerically unstable computations. However, this work only focused on discrete computing without considering physical dynamics in verification.

Hybrid systems are modeled as hybrid automata, which combines automaton transitions for capturing discrete changes with differential equations for continuous changes. Well-known tools for verifying hybrid systems include HyTech [7] and Uppaal [8]. For analysis of hybrid systems, it is often useful to abstract a system in a way that preserves the properties being analyzed while hiding the irrelevant details. There has been much research on abstracting hybrid systems, largely categorized into equivalent abstractions and sufficient abstractions. Equivalent abstractions (surveyed in [13]) such as language-preserving or bi-simulation abstractions of timed automata, rectangular automata, and o-minimal hybrid automata have enabled the basic algorithms of tools such as HyTech and Uppaal.

There is an emerging trend in verification of hybrid systems to utilize abstractions based on Satisfiability Modulo Theories (SMT) [14]. In [6], they used an interval-based solver for ordinary differential equations (ODEs) under an SMT framework. In [15], a framework for verifying hybrid systems is developed, under which it is proven that the decision problem for bounded logic formulas over the real numbers with general nonlinear functions are decidable. This approach has a key drawback: The focus has been on control logic design on high level with simplifying assumptions.

III. Background

This section introduces the background of our research. First, we analyze timing parameters of control tasks. Then, we review the fundamentals of Pushdown System and Hybrid Automata. Finally, we introduce our case studies.

A. Timing Parameters of Control Task

The basic timing parameters of a control task [16] are shown in Fig. 2. It is assumed that the control task is released periodically at times given by $t_k = T * k$, where T is the fixed sampling interval of the controller and k is the number of controller iterations. Let t_i and t_o represent the A/D and D/A conversion periodical instants, respectively. Due to preemption and blocking from other tasks in the OS,

the actual start of the task may be delayed for some time L_s. This is called the sampling latency of the controller. A dynamic scheduling policy will introduce variations in this interval. The sampling jitter is defined by the range between the minimum and maximum sampling latencies in all task instances, $J_s \in [\min L_s, \max L_s]$.

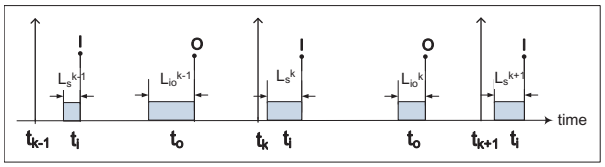

Figure 2: Timing analysis of control task.

After some computation time and possible further preemption from other tasks, the controller will actuate the control signal. The delay of the actuation is called the input-output latency, denoted L_{io}. Varying execution times or task scheduling will introduce variations in this interval. The input-output jitter is defined by $J_{io} \in [\min L_{io}, \max L_{io}]$.

We also assume that $t_i\prime$ and $t_o\prime$ represent the A/D and D/A the worst-case conversion instants respectively. The actual A/D and D/A conversion instant are $t_i = t_i\prime - J_s$ and $t_o = t_o\prime - J_{io}$, respectively.

B. Pushdown System as Software Models

A Pushdown System *PDS* [17] is a tuple $(G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$, where G is a finite set of global states, Γ is a finite stack alphabet, $\Delta \sqsubseteq (G \times \Gamma) \times (G \times \Gamma^*)$ is a finite set of transition rules, and $\langle g_0, \omega_0 \rangle$ is the initial configuration. A configuration of *PDS* is a pair $\langle g, \omega \rangle$ representing a software state, where $g \in G$ is a global state and $\omega \in \Gamma^*$ is a stack content. An *PDS* rule is written as $\langle g, \omega \rangle \hookrightarrow \langle g\prime, \omega \rangle$, where $((g, \gamma), (g\prime, \omega)) \in \Delta$, and (g, γ) is referred to as the head of this rule. The set of all configurations is denoted as $Conf(P)$.

C. Hybrid Automata as Plant Models

A Hybrid Automata *HA* [4] is syntactically a tuple $(\tilde{x}, \tilde{x_0}, V, v_0, inv, dif, E, act, lab, syn)$, consisting of the following components:

- $\tilde{x}$ is a vector of n real-valued variables $\tilde{x} = (x_1, x_2, ..., x_n) \in \mathbb{R}^n$. A specific evaluation of $\tilde{x}$, denoted as $\tilde{s} = (s_1, s_2, ..., s_n) \in \mathbb{R}^n$ is called a data state of *HA*;
- $\tilde{x_0}$ is the initial data state;
- V is a set of locations, where different control laws apply. A state of hybrid automaton *HA* is denoted as $(v, \tilde{s})$, where $v \in V$ and $\tilde{s} \in \mathbb{R}^n$ is a data state;
- v_0 is the initial location;
- inv is the location invariants, a labeling function that assigns each location $v \in V$ a set of inequalities over $\tilde{x}$;
- dif is the continuous activities, a function that assigns each location $v \in V$ a set of inequalities over $\tilde{\dot{x}}$ and $\tilde{x}$;
- E is the set of events: edges between locations. Formally, $E \sqsubseteq V \times V$;
- act is the discrete actions, a labeling function assigns to each event $e = (v, v') \in E$ a set of inequalities over $\tilde{x}$ and $\tilde{x}'$, where $\tilde{x}' = \left(x_1', x_2', ..., x_n'\right)$ refers to the new value of $\tilde{x}$ after event e. The event $e = (v, v')$ is enabled only when the value of $\tilde{x}$ in v satisfies $act(e)$;
- lab is a set of synchronization labels. The set lab is called the alphabet of *HA*;
- syn is the labeling function that assigns to each event $e \in E$ a set of synchronization labels from lab.

D. TableSat

TableSat, as shown in Fig. 3, is an interactive platform from the University of Michigan, which emulates in 1-degree-of-freedom the dynamics, sensing, and actuation capabilities required for satellite attitude control.

Figure 3: TableSat.

TableSat is driven by two computer fans and experiences extremely low friction on its central pivot point. It contains a high-precision rate gyro to measure angular velocity. An onboard Diamond Systems Prometheus PC/104 computer communicates to a ground station via a wireless 802.11b interface. The computer interfaces to sensors through 16-bit analog-to-digital converters and to actuators through amplified 16-bit digital-to-analog channels. The equations of TableSat motion are:

$$I\dot{\omega} = 2lK_{\omega f}v - f_{TS}(\omega) \quad (1)$$

$$\dot{v} = -\alpha v + K_{v\dot{\omega}}(V - f_{fan}(v)) \quad (2)$$

where I is the TableSat moment of inertia, ω is the TableSat angular velocity, v is the speed of the fan, l is the fan moment arm, f_{TS} is the TableSat friction and is a function of ω, $K_{\omega f}$ is the fan speed to force constant, V is the voltage applied to the fan, α is the fan time constant, $K_{v\dot{\omega}}$ is the fan voltage to change in speed constant, and f_{fan} is the friction in the fans and is function of v.

The controller manipulates the voltage applied to the fan to enforce a target angular velocity of TableSat. Here we set two voltage (0V and 12V) options for the controller to choose. The motion law of TableSat is denoted by the two

different voltages: one is 12V voltage applied to the fans (see Eq. 3), the other is 0V voltage applied to the fans (see Eq. 4). Fig. 4 shows the motion of TableSat by hybrid automata. The model consists of 6 discrete locations corresponding to each node, 3-dimensional continuous states $\tilde{x} = (\omega, v, t) \in \mathbb{R}^3$, and 8 discrete state transitions corresponding to the edges. Let `t` represents the internal timer. According to analysis in section III-A, each motion law is divided into three stages in a loop: sampling, input-output and rest (represent by $(t_k,\ t_i]$, $(t_i,\ t_o]$ and $(t_o,\ t_{k+1}]$, respectively). So TableSat has 6 discrete locations (12V dynamic: `On_AD`, `On_DA` and `On_Rest`; 0V dynamic: `Off_AD`, `Off_DA` and `Off_Rest`). Each discrete transition is enabled by its guard condition. For example, a discrete transition d from `On_DA` to `On_Rest` has a guard condition $t > t_o \cap$ `On` (e =(`On_DA`, `On_Rest`), $\tilde{x} \vdash act(e) = t > t_o,\ $ `On` $\in lab$). When the controller sends command to the fan (i.e., when boolean variable, `On` or `Off`, is set to true), the motion of TableSat switches to the corresponding motion law. An edge entering `On_AD` represents the initial constraint. The set of initial state is (`On_AD`, 0, 0, 0).

$$\dot{v} = -\alpha v + K_{v\dot{\omega}}(12 - f_{fan}(v)) \tag{3}$$

$$\dot{v} = -\alpha v + K_{v\dot{\omega}}(0 - f_{fan}(v)) \tag{4}$$

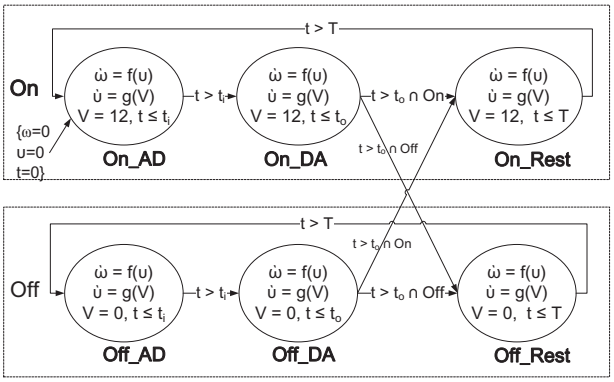

Figure 4: Plant model of TableSat.

IV. Verification Model for Cyber-Physical System

Since software and physical plant are designed in different approaches, they are commonly represented by different formal models. Having separate formal models for software and physical plant is cumbersome for co-verification. A unifying model that combines the merits of Pushdown System and Hybrid Automata is desired so that software, physical plant and their interactions can be analyzed together as an integrated system.

We synthesize a *HAPS* by synchronizing a Pushdown System and a Hybrid Automata. *HAPS* captures both the synchronous transitions and asynchronous transitions on the system level. While *PDS* is a suitable model for programs, it is not designed to accept any inputs. In co-verification, physical plant behaviors also affect software executions, e.g., through sampling; therefore, the *PDS* software model should be extended to accept inputs from the *HA* model.

Definition 1. A Labeled Pushdown System (*LPDS*) is a tuple $(I, G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$, where I is a finite input alphabet, G is a finite set of global states, Γ is a finite stack alphabet, $\Delta \sqsubseteq (G \times \Gamma) \times I \times (G \times \Gamma^*)$ is a finite set of transition rules, and $\langle g_0, \omega_0 \rangle$ is the initial configuration. An *LPDA* rule is written as $\langle g, \Upsilon \rangle \xrightarrow{\tau} \langle g\prime, \omega \rangle \in \Delta$.

For co-verification, we define: (1) the input alphabet of *LPDS*, I, as the power set of the set of atomic propositions that may hold on a state of *HA*; (2) the alphabet of *HA*, *lab*, as the power set of the set of atomic propositions that may hold on a state of *LPDS*; and (3) two synchronization labeling functions as follows:

- L_{P2H}: $(G \times \Gamma) \rightarrow lab$, which associates the head of an *LPDA* configuration with the set of propositions that hold on it;
- L_{H2P}: $V \rightarrow I$, which associates a state of *HA* with the set of propositions that hold on it.

Since the state transitions of both the *LPDS* and *HA* are labeled by the symbols, the two models are synchronized. It is further desired that a unifying model is built to combine them; thus, their behaviors can be analyzed together as an integrated system.

Enabledness. An *LPDS* rule $r = \langle g, \Upsilon \rangle \xrightarrow{\tau} \langle g\prime, \omega \rangle \in \Delta$ is enabled by a *HA* location $v \in V\ iff\ \tau \subseteq L_{H2P}(v)$; otherwise, r is disabled by v. On the other hand, the *HA* discrete transition $d = (v, \tilde{x}) \xrightarrow{\alpha} (v\prime, \tilde{x}\prime)$ is enabled by the *LPDS* configuration $\langle g, \Upsilon \rangle\ iff\ \alpha \subseteq L_{P2H}(\langle g, \Upsilon \rangle)$; otherwise, d is disabled by $\langle g, \Upsilon \rangle$.

Indistinguishability. Given a *HA* discrete transition $d = (v, \tilde{x}) \xrightarrow{\alpha} (v\prime, \tilde{x}\prime)$ $(e = (v, v'),\ \tilde{x} \vdash act(e))$, an *LPDS* rule $r = \langle g, \Upsilon \rangle \xrightarrow{\tau} \langle g\prime, \omega \rangle \in \Delta$ is indistinguishable to d $iff\ \alpha \subseteq L_{P2H}(\langle g, \Upsilon \rangle) \bigcap L_{P2H}(\langle g\prime, \omega \rangle)$. On the other hand, d is indistinguishable to $r\ \ iff\ \tau \subseteq L_{H2P}(v) \bigcap L_{H2P}(v\prime)$.

Independence. Given a *HA* discrete transition d and an *LPDS* rule r, if they are indistinguishable to each other, d and r are said to be independent; otherwise if either d or r is not indistinguishable to the other but they still enable each other, d and r are said to be dependent. The independence relation is symmetric.

Definition 2. A Hybrid Automata Pushdown System (*HAPS*) is a tuple $(G\prime, \Gamma, \Delta\prime, inv, dif, \langle (v_0, g_0), \tilde{x_0}, \omega_0 \rangle)$, where

- $G\prime$, as $V \times G \times \tilde{x}$, is the set of global states;
- Γ is the stack content from *LPDA*;
- $\Delta\prime$ is a set of transition rules which are constructed by Algorithm 1;
- inv is the location invariants from *HA*;
- dif is the continuous activities from *HA*;

- $\langle (v_0, g_0), \tilde{x_0}, \omega_0 \rangle$ is initial state of *HAPS*.

The input alphabets I and *lab* are used to synchronize the *LPDS* rules and *HA* transitions. Once *HAPS* is constructed, these input alphabets are no longer useful. We will construct the whole system discrete transition by Algorithm 1. E and *act* are used to capture discrete transition of *HA*. Therefore, they are not in the definition of *HAPS*.

The cartesian product between *LPDS* and *HA* are actually done through code instrumentation. Algorithm 1 creates a cartesian product of the transition rules from *LPDS* and *HA* respectively. Given a *LPDS* rule $r = \langle g, \Upsilon \rangle \xrightarrow{\tau} \langle g\prime, \omega \rangle \in \Delta$ and a *HA* discrete transition $d = (v, \tilde{x}) \xrightarrow{\alpha} (v\prime, \tilde{x}\prime)$ ($e = (v, v')$, $\tilde{x} \vdash act(e)$), the algorithm constructs *HAPS* rules from r and d only if they enable each other. When r and d are dependent, *LPDS* and *HA* must transition together, which is modeled by the *HAPS* rule $\langle (g, v), \Upsilon \rangle \rightarrow \langle (g\prime, v\prime), \omega \rangle$. In such situation, the *HAPS* rule represents synchronous transitions of *LPDS* and *HA*. When r and d are independent, *LPDS* and *HA* can transition asynchronously. One can run while the other one self-loops. There are two types of *HAPS* rules: *LPDS* transitions and *HA* self-loops as modeled by $\langle (g, q), \Upsilon \rangle \rightarrow \langle (g\prime, q), \omega \rangle$ while *HA* transitions and *LPDS* self-loops as modeled by $\langle (g, q), \Upsilon \rangle \rightarrow \langle (g, q\prime), \Upsilon \rangle$.

Algorithm 1: GENERATE_HAPS_RULES

```
1  Δ_sync ← φ, Δ_hori ← φ, Δ_vert ← φ;
2  forall the r = ⟨g, Υ⟩ --τ--> ⟨g′, ω⟩ ∈ Δ do
3      forall the d = (v, x̃) --α--> (v′, x̃′), e = (v, v′), x̃ ⊢ act(e)
       and α ⊆ L_P2H(⟨g, Υ⟩) and τ ⊆ L_H2P(v) do
4          if r and d are dependent then
5              {LPDS and HA must transition together}
               Δ_sync ← Δ_sync ⋃ {⟨(g, q), Υ⟩ → ⟨(g′, q′), ω⟩}
6          else
7              {LPDS transition and HA self-loops}
               Δ_vert ← Δ_vert ⋃ {⟨(g, q), Υ⟩ → ⟨(g′, q), ω⟩}
8              {HA transition and LPDS self-loops}
               Δ_hori ← Δ_hori ⋃ {⟨(g, q), Υ⟩ → ⟨(g, q′), Υ⟩}
9  Δ_P ← Δ_sync ⋃ Δ_hori ⋃ Δ_vert
10 return Δ_P;
```

V. Constructing *HAPS* for Cyber-Physical System

In this section, we discuss how to construct a *HAPS* model. As illustrated in Fig. 5, our formal framework has: software model, plant model and cyber/physical interface. There are three steps to construct a *HAPS* model from the software and plant model: (1) define the cyber/physical interface; (2) instrumenting the software model based on the cyber/physical interface; (3) instrumenting the plant model based on the cyber/physical interface. In order to facilitate the understanding of our approach, we illustrate it with the TableSat example.

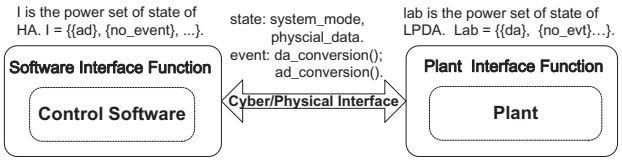

Figure 5: Formal framework for co-verification.

A. Cyber/Physical Interface

CPS contains both the physical components being controlled and the software implementation of control algorithms. The software controller and its control plant are mostly asynchronous and only transition synchronously when they interact through their interface. The cyber/physical concurrency describes such situations:

- Mostly, software and plant transition asynchronously, where their states do not affect each other;
- When software and plant interact with each other, their synchronous transition will be decided by the states of both software and plant.

The cyber/physical interface describes how software controller and its controlled plant should transition synchronously when they interact with each other. A cyber/physical interface has two parts: interface states and interface events. Interface states are state variables provided either by software or plant and accessible by both. Interface events have two types: software or plant. When software updates the plant interface states, a software interface event occurs, and vice versa. For example, when software writes command to the plant, the cyber/physical interface will set the related actuator accordingly. In general, the cyber/physical interface describes the synchronous transitions of software and plant when an interface event occurs.

Consider the example of TableSat (see Fig. 5). Interface states are `system_mode` and `physical_data`, which represent motion of fan (Eq. 3 or Eq. 4) and senor read value respectively. A plant discrete transition is a set of *HA* transitions labeled by the symbol from *lab*; and an atomic software statement is a set of *LPDS* rules labeled by the same input symbol from I. The input alphabet of software model is $\{ad\}$ and $\{no\_event\}$; the *lab* of plant model is $\{da\}$ and $\{no\_evt\}$. The labeling function L_{H2P} (resp. L_{P2H}) maps plant (resp. software) states to input symbols in I (resp. *lab*). For *HA*, there are symbols such as $\{da\}$, $\{no\_evt\} \in lab$, where the propositional variable $\{da\}$ represents the software interface events when software writes the command to set the related actuator; and the propositional variable $\{no\_evt\}$ represents that there is no software interface event. On the other direction, for *LPDS*, there are input symbols such as

{*ad*}, {*no_event*} ∈ *I*, where the propositional variable {*ad*} represents the plant interface event, i.e., A/D conversion; and the propositional variable {*no_event*} represents that there is no plant interface event.

Fig. 6 illustrates an example of a software interface event function in response to a software read sensor value operation. The function `ad_convertion` is labeled by the symbol, {*ad*} ∈ *I*. Conceptually, the software interface event happens, i.e., {*ad*} is evaluated true, when entry stack symbol of `ad_convertion` is reached. When software writes command, plant should be switched in response to a software write actuator operation. Fig. 7 simulates this process. The function `da_convertion` are labeled by {*da*} ∈ *lab*. Therefore, the plant discrete transition is enabled when software writes command.

```
1  //software function labeled by the symbol {ad}
2  int ad_convertion(float sensor_data)
3  {
4    ...
5    //AD conversion
6    switch(system_mode){
7    case (system_mode == On):{
8      physical_data = physical_dynamic1(t_i-js);
9      break;}
10   case(system_mode == Off):{
11     physical_data = physical_dynamic2(t_i-js);
12     break;}
13   ...
14   }
15   ...
16 }
```

Figure 6: An implementation of a software interface event.

```
1  //plant function labeled by the symbol {da}
2  int da_convertion(int voltage)
3  {
4    ...
5  //if command voltage=12, switch to 12V mode
6    if (voltage==12) system_mode = On;
7  //Otherwise, switch to 0V mode
8    else system_mode = Off;
9    ...
10 }
```

Figure 7: An implementation of a plant interface event.

B. Instrumenting HA

The physical plant model describes the behaviors of physical dynamic when it transitions asynchronously with software, i.e., when there is no D/A conversion. There are two steps to construct a *HA* model: first, we instrumenting the plant model based on the cyber/physical interface, which adds the synchronization label imposed by *HAPS*. Second, we convert hybrid automata into an equivalent C program. We utilize SMT to abstract the physical plant and serve as the semantic foundation for abstraction.

In the following, we describe the encoding of a *HA* with disjoint sets of continuous variables. The $HA = (\tilde{x}, \tilde{x_0}, V, v_0, inv, dif, E, act, syn, lab)$, where

- $\tilde{x}$ is encoded as global variables.
- V is encoded with a set of module functions.
- Initial state is given by the initialization function;
- *inv* is encoded with a set of inequalities over data variables in a module function;
- The continuous variables evolve according to dif, which is defined within a module function.
- E is encoded with the function call. E_{entry} and E_{exit} are the function entry and function exit, respectively. E_{entry} determines when *HA* performs a continuous evolution, while E_{exit} determines when *HA* stutters (i.e. no transition). The E_{entry} is chosen when the value of data variables satisfies $act(e), e \in E$;
- *act* is encoded with a set of inequalities over data variables;
- Synchronization constraint, *lab*, is encoded with a set of boolean variables;
- *syn* is encoded with a set of equalities over boolean variables.

Take TableSat as a example. We convert its plant model (Fig. 4) into a C program (Fig. 8). The code in Fig. 8 works as follows:

Lines 1: Defines three float variables `physical`, `nu` and `t`, which represent three continuous states (ω, v, t).

Line 3: Realizes Eq. 1 and Eq. 3 during the time interval `TimeInterval`. Here we use Euler method to solve ordinary differential equations with a given initial value.

Line 5-29: Models the state transitions for TableSat when the plant executes asynchronously with the controller. During each execution of the transaction function, the motion of TableSat is selected by the controller through interface state `system_mode` (Line 7 and 25). So only one module executes and its related state variables will be update.

Line 8, 16, 20: Checks whether the value of time variable `t` satisfies $act(e)$.

line 11, 17, 21: Executes plant dynamic by calling function `physcial_dynamic1` or `physcial_dynamic2`.

C. Instrumenting Software

The software model describes the behaviors of the controller when it transitions asynchronously with the physical plant, i.e., when there is no sampling. It describes the desired operation sequences for the controller to control the plant. The left side of Fig. 9 shows the skeleton of the control software for TableSat. When initializing the controller, the software will set a timer to invoke the control loop periodically. After that, the controller will wait to invoke the main control loop. In the control loop, the controller first reads senor values from the channels, then calculates

```
1  float physical, nu, t;
2  //plant function labeled by the symbol no_evt
3  float physcial_dynamic12(float TimeInterval);
4  //plant function labeled by the symbol no_evt
5  void physical_run(void){
6    switch(system_mode){
7      case (system_mode == On):{
8        if((0 < t) && (t <= t_i)){
9          ...
10       //physical dynamic in system mode 1
11         physical = physical_dynamic1(t);
12       //get senor value
13         ad_convertion();
14         ...
15       }
16       else if((t_i < t) && (t <= t_o)){
17         physical = physical_dynamic1(t);
18         ...
19       }
20       else if((t_o < t) && (t <= T)){
21         physical = physical_dynamic1(t);
22         ...
23       }break;
24     }
25     case(system_mode == Off):{
26       ...
27     }
28   }
29 }
```

Figure 8: C program for TableSat plant.

outputs according to the control logic, writes commands to related actuator and update state in the end. The `initial`, `wait_clock`, `calculate_output`, `command_motor` and `update_state` function are labeled by $\{no\_event\} \in I$, and the function `read_sensors` is labeled by $\{ad\} \in I$.

```
int main()
{
  ...
  initial();
  While(1){
    wait_clock();
    read_sensors();
    calculate_output();
    command_motor();
    update_state();
  }
  ...
}

void commandMotor(double *v)
{
  ...
if( ( result = da_convertion(voltage) ) != DE_NONE )
  ...
}
```

physical_run(void);

Figure 9: Left side: C program of control software; right side: execution of plant interleaved with software statements.

Given the cyber/physical interface, plant model, and software, we combine them into a C program. After software functions (`wait_clock`, `calculate_output` and `command_motor`), we invoke the plant transaction function, `physical_run`, to let the asynchronous plant model execute.

VI. Verification Algorithm for Cyber-Physical System

A. Bounded Model Checking of Safety Properties of HAPS

In order to verify CPS, it is necessary to specify properties to be be checked, i.e., whether the implementation of CPS design may exhibit behaviors that satisfy certain conditions, specifying some desired or undesired interactions among the components. Events in CPS are basically boolean propositions over physical variables. Assertions are desired correlations among physical plant and software events.

In our co-verification framework, the formal model is a *HAPS* and the property ϕ are specified using assertions. When a safety property is specified, we check properties of the form $AG\ \phi$ through reachability analysis. Fig. 10 shows the co-verification flow as supported by our approach.

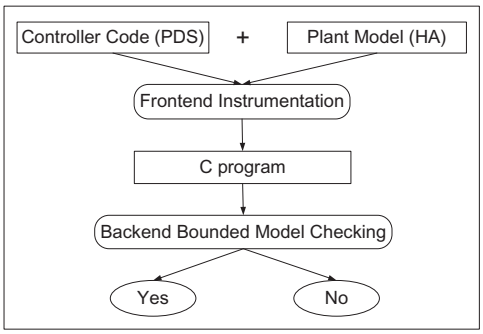

Figure 10: Cyber/Physical co-verification flow.

We convert a *HAPS* into a C program, which we refer to as the verification model, so that model checkers for C can be readily utilized. The backend engine, CBMC [18], checks the safety properties on the C program. The CBMC can directly reason about floating point arithmetic by bit-blasting, following the IEEE 754 standard.

B. Symbolic Execution

Bounded Model Checking (BMC) is a technique that finds a violation of a property ϕ in a transition system. Thus, BMC finds a violation state for ϕ if there is a path which witnesses the violation in at most k steps. x_i ($i \in [0,k]$) represents the configuration of *HAPS* in step k. In this paper, we express that violation state will be reached within k-step execution as

$$[HAPS]^k = init(x_0) \wedge \Delta_{P'}(x_0, x_1) \wedge \cdots \wedge \Delta_{P'}(x_{k-1}, x_k) \wedge \phi. \quad (5)$$

As we discussed in section III-A, temporal determinism is very important for system design. Designers normally assume equidistant sampling and negligible or constant latency. However, this situation can rarely be achieved in practice. To circumvent the theoretically difficult temporal determinism problem, we defined the latency as a symbolic variable: instead of asking whether a system is safe or not by a precise latency, a symbolic variable asks whether a system

is safe or not within a loose bound. The symbolic variable is a bounded interval $[m, n] = \{r \in \mathbb{R} \mid m \leq r \leq n\}$ $(m, n \in \mathbb{R})$, which is a set of real numbers.

Proposition 1 For a Hybrid Automata Pushdown System ($HAPS$), suppose $[HAPS]^k$ is a satisfiability formula encoded for k steps as described above. If $[HAPS]^k$ is unsatisfiable, then $HAPS$ is safe for k-step execution.

VII. Evaluation

To evaluate our proposed approach, especially the effectiveness of CPS verification, we conduct two case studies to evaluate its applicability to real-world control systems. In our case studies, we want to check whether the controller is safe or not with slight perturbations in the inputs and outputs to the system. All experiments were performed on a machine with 2.93GHz Intel(R) Xeon and 8G memory.

A. TableSat

We started by implementing the controller in C. Then, we synchronized the software and physical plant model and convert it into a C program. Fig. 11 shows a simulation run of TableSat.

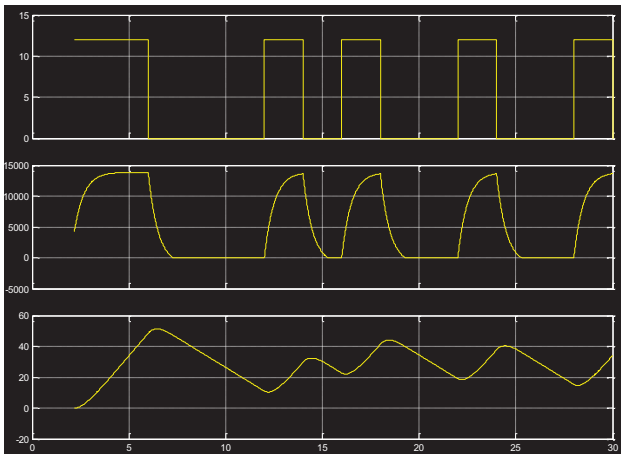

Figure 11: Simulation result (Input 30deg/s; from top to bottom the charts show the voltage, fan speed, angular velocity over the simulated time).

We formulated the properties of the system and its components, and conducted bounded model checking. Assertions are desired correlations among discrete computation and continuous control plant events. In summary, the behaviors of TableSat must comply with the following safety rules:

Safety Rule 1: The controller never accelerates the TableSat over the rotary velocity limit $VelocityUpBound$. Formally, $(Rotary.Velocity \leq VelocityUpBound)$.

Safety Rule 2: When running more than a threshold $TimeBound$, the controller will always accelerate the TableSat over the rotary velocity limit $VelocityDownBound$. Formally, $(Time \geq TimeBound) \Rightarrow (Rotary.Velocity \geq VelocityDownBound)$.

Safety Rule 3: When the rotary velocity exceeds 1.5 times of its expected value $TargetVelocity$, the controller will set the fans to 0 volt. Formally, $(Rotary.Velocity > 1.5 * TargetVelocity) \Rightarrow (Actuator.FanVoltage \equiv 0)$.

Safety Rule 4: When the rotary velocity below 0.4 times of its expected value $TargetVelocity$, the controller will set the fans to 12 volt. Formally, $(Rotary.Velocity < 0.4 * TargetVelocity) \Rightarrow (Actuator.FanVoltage \equiv 12)$.

Safety Rule 5: When running more than a threshold $TimeBound$, the rotary velocity must be stable at $TargetVelocity$ within the error $SteadyStateError$. Formally, $(Time \geq TimeBound) \Rightarrow (|Rotary.Velocity - TargetVelocity| \leq SteadyStateError)$.

We use the following initial set of parameters: the fixed sampling interval ($T = 2s$), the A/D conversion instant ($t_i = 0.4s$), and the D/A conversion instant ($t_o = 1.6s$). We specified a target rotary velocity ($TargetVelocity = 30deg/s$) as TableSat input. We set the initial value of angular velocity ([0, 40]) as a symbolic variable. Bounded model checking revealed a simple bug of the controller that: If the initial value of angular velocity is $39.960621deg/s$, then the rotary velocity reaches ($63.649414deg/s$) above a threshold ($VelocityUpBound = 60deg/s$) at $2.324336s$. The verification run took 7260.86s.

We used the same initial set and target rotary velocity to conduct another verification run. The initial value of angular velocity was $0deg/s$. We set sampling jitter ($j_s \in [0, 0.2]$) and input-output jitter ($j_{io} \in [0, 0.1]$) as symbolic variables. The verification result shows that the controller satisfies the specification within $6s$. The verification run took 10415s. The running time largely depends on the backend solver.

B. Cruise Control System

The second experiment is cruise control system (see Fig. 12). Cruise control is an example of a feedback control system found in many modern vehicles. The purpose of the cruise control system is to automatically controls the speed of an vehicle. The driver sets the speed and the system takes over the throttle of the car to maintain the speed according to a control law. We obtain the car model from the benchmark of iSAT [19].

The top-level diagram of the model shown in Fig. 12 is composed of a module which represents the vehicle dynamic, with an additional cruise control block to control the vehicle speed. The dynamics of the car consists of: the continuous speed variable $v = \dot{x}$ $(m \cdot s^{-1})$, the continuous inputs that are the engine torque u_t (Nm) and the braking force u_b (N), plus six binary inputs g_1, g_2, g_3, g_4 and g_5 $\in \{0, 1\}$ corresponding to the selected gear.

1) Car Model: The car model in the benchmark is the model of Renault Clio 1.9 dTi RXE. The dynamic motion

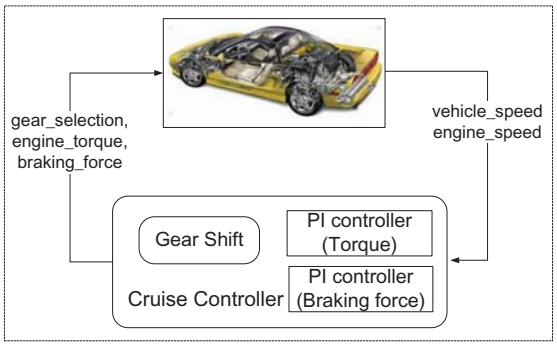

Figure 12: Cruise control system.

equation of the car is as follows:

$$m\ddot{x} = F_e - F_b - F_\Upsilon \tag{6}$$

where $\ddot{x}$ is the vehicle accelerated speed, m is the vehicle mass, F_e is the traction force, $F_b = u_b$ is the braking force, F_Υ is the friction force. $F_\Upsilon = \beta v$, where β is a constant that takes into account all the frictions.

The traction force F_e and engine speed ω can be written as $F_e = F_{e1}+F_{e2}+F_{e3}+F_{e4}+F_{e5}$ and $\omega = \omega_1+\omega_2+\omega_3+\omega_4+\omega_5$, where

$$F_{ei} = \begin{cases} \frac{R_g(i)}{k_{loss}} u_t, & \text{if } g_i = 1 \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

$$\omega_i = \begin{cases} \frac{R_g(i)}{k_{loss}} \dot{x}, & \text{if } g_i = 1 \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

where $R_g(i)$ is the gear ratio corresponding to the ith gear, u_t (Nm) is the engine torque, and k_{loss} is the drive train efficiency level.

The main parameters of the car are: $R_g(1) = 3.7271$, $R_g(2) = 2.048$, $R_g(3) = 1.321$, $R_g(4) = 0.971$, $R_g(5) = 0.756$, $K_{loss} = 0.925$, $\beta = 25(kg \cdot m \cdot s^{-1})$, $m = 1020kg$.

2) Cruise Controller: The controller commands throttle position, braking force and selected gear, based on the desired vehicle speed and measurements of the actual car speed and engine speed. It consists of three controllers: gear shift controller, torque PI controller and braking force PI controller.

The gear shift controller performs the function of gear selection in the automatic transmission. The automatic transmission adjusts gear ratios as the vehicle moves. Each gear is followed by different dynamic law (traction force and engine/vehicle speed). The model describes controls for a 5-speed transmission system. Fig. 13 illustrates the functionality.

The gear shift controller defines the input as the actual vehicle speed and the output as the desired gear number. Shifting is only possible between adjacent gears and is triggered when engine speed leaves a certain interval. If

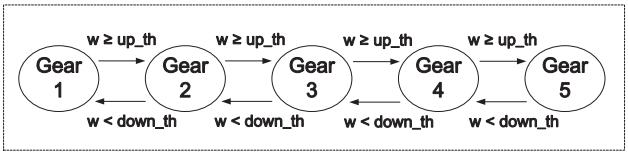

Figure 13: Diagram of gear shift logic.

the engine speed is faster than `up_th` (3500 RPM) then the gear is shifted up. Similarly, if the speed is lower than `down_th` (1500 RPM), the gear is shifted down.

The throttle and the brakes are operated by the PI controller, which is characterized by:

$$u_t(k) = \begin{cases} k_t \Delta v(k) + i_t e(k), & if\ v(k) \le v_r(k) + 1 \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

$$u_b(k) = \begin{cases} k_b \Delta v(k), & if\ v(k) \le v_r(k) + 1 \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

where $e(k)$ is integral error and $\Delta v(k)$ is the difference between desired speed and actual vehicle speed.

3) Co-Verification of Cruise Control System: We applied our co-verification approach to the transmission shift control system with the same process as TableSat. We synchronized the software and physical plant model and converted it into a C program, formulated the properties of the system and its components, and then conducted bounded model checking.

We checked a safety property: the cruise controller will never accelerate the car over the speed limit. We used the following initial set $\{T = 3s,\ t_i = 1.5s,\ t_o = 2.1s\}$ and specified a target speed $13.89m/s$. We set sampling jitter ($j_s \in [0, 0.1]$) and input-output jitter ($j_{io} \in [0, 0.2]$) as symbolic variables. Bounded model checking thus revealed a simple bug of the controller that was, however, subtle enough not to be detected when designing the model: when the car speed near the speed limit (14.5m/s), instead of applying the maximum braking force, the controller still sets engine torque to $112.220001Nm$, allowing the car to accelerate and exceed the speed limit. This happens since the controller re-computes the cruise control setting only every $3s$ with sampling jitter of $0.091578s$ and input-output jitter of $0.132955s$. The error trace is shown in Table I. Each row in the table shows a system state at a time instant. For instance, the last row shows that the car speed has exceeded its speed limit at 7.5s with the set of parameters: selected gear = 1, engine torque = $112.220001Nm$, and braking force = $0N$. The verification run took 7242.26s.

VIII. Conclutions

This paper proposes an automata-theoretic approach to cyber/physical co-verification. The core of this approach is a formal model for co-verification, Hybrid Automata Pushdown System ($HAPS$). We have demonstrated the process

Table I: Error Trace.

Time (s)	Gear No.	Engine Torque (Nm)	Braking Force (N)	Car Speed (m/s)
0.000000	1	0	0	0
0.248437	1	0	0	0
1.500122	1	112.220001	0	0
2.115534	1	112.220001	0	0
3.000000	1	112.220001	0	0
4.562532	1	112.220001	0	0.313083
5.917902	1	112.220001	0	6.386552
7.500000	1	112.220001	0	14.687162

of constructing a *HAPS* from the software, plant, and the interface between cyber/physical through the synchronization of a PDS and a HA and converting it into a C program. So reachability analysis of C program algorithms can be readily utilized to analyze *HAPS*. We have implemented this approach in our co-verification tool and successfully applied it to co-verify real-world control systems. The evaluation has shown that *HAPS* is an effective model for co-verification and our approach has major potential in verifying CPS.

Our work is an ongoing project. On one hand, we are continuously improving our algorithms to reduce verification complexity. On the other hand, we are currently extending the integration of numerical solving of ODEs in order to directly handle ODEs in the solver without numerical approximation.

ACKNOWLEDGMENT

This research received financial support from the National Science Foundation of the United States (Grant #: 0720546 and Grant #: 0916968), the National High-Tech Research and Development Plan of China (Grant #: 2011AA010105 and Grant #: 2011AA010102), and the National Infrastructure Software Plan of China (Grant #: 2012ZX01041-002-003).

REFERENCES

[1] E. A. Lee, "Cps foundations," in *Proc. of the 47th Design Automation Conference (DAC)*. ACM, June 2010, pp. 737–742.

[2] R. Thacker, K. Jones, C. Myers, and H. Zheng, "Automatic abstraction for verification of cyber-physical systems," in *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*. ACM, 2010, pp. 12–21.

[3] P. Kumar, D. Goswami, S. Chakraborty, A. Annaswamy, K. Lampka, and L. Thiele, "A hybrid approach to cyber-physical systems verification," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, june 2012, pp. 688 –696.

[4] L. Bu, Q. Wang, X. Chen, L. Wang, T. Zhang, J. Zhao, and X. Li, "Toward online hybrid systems model checking of cyberphysical systems time-bounded short-run behavior," *ICCPS11 Work-in-Progress Session*, 2011.

[5] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical computer science*, vol. 138, no. 1, pp. 3–34, 1995.

[6] A. Eggers, N. Ramdani, N. Nedialkov, and M. Fränzle, "Improving sat modulo ode for hybrid systems analysis by combining different enclosure methods," *Software Engineering and Formal Methods*, pp. 172–187, 2011.

[7] T. Henzinger, P. Ho, and H. Wong-Toi, "Hytech: A model checker for hybrid systems," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 110–122, 1997.

[8] K. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 134–152, 1997.

[9] E. Goubault, "Static analyses of the precision of floating-point operations," in *Proceedings of the 8th International Symposium on Static Analysis*, ser. SAS '01. London, UK, UK: Springer-Verlag, 2001, pp. 234–259.

[10] F. Balarin, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, "Formal verification of embedded systems based on cfsm networks," in *Design Automation Conference Proceedings 1996, 33rd*. IEEE, 1996, pp. 568–571.

[11] L. Cortes, P. Eles, and Z. Peng, "Formal coverification of embedded systems using model checking," in *Euromicro Conference, 2000. Proceedings of the 26th*, vol. 1. IEEE, 2000, pp. 106–113.

[12] E. Goubault, S. Putot, P. Baufreton, and J. Gassino, "Static analysis of the accuracy in control systems: Principles and experiments," *Formal methods for industrial critical systems*, pp. 3–20, 2008.

[13] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas, "Discrete abstractions of hybrid systems," *Proceedings of the IEEE*, vol. 88, no. 7, pp. 971–984, 2000.

[14] A. Cimatti, S. Mover, and S. Tonetta, "Smt-based verification of hybrid systems," in *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[15] S. Gao, J. Avigad, and E. M. Clarke, "Delta-decidability over the reals," *CoRR*, vol. abs/1204.6671, 2012.

[16] K. Årzén, A. Cervin, and D. Henriksson, "Implementation-aware embedded control systems," *Handbook of networked and embedded control systems*, pp. 377–394, 2005.

[17] S. Schwoon, "Model-checking pushdown systems," Ph.D. dissertation, Technische Universität München, Universitätsbibliothek, 2002.

[18] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *In Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 168–176.

[19] iSAT benchmark, "Automatic verification and analysis of complex systems," http://isat.gforge.avacs.org/benchmarks.