Symbolic-Numeric Reachability Analysis of Closed-Loop Control Software

Aditya Zutshi Sriram Sankaranarayanan
University of Colorado, Boulder
aditya.zutshi,srirams@colorado.edu

Jyotirmoy V. Deshmukh and Xiaoqing Jin
Toyota Technical Center, USA
firstname.lastname@tema.toyota.com

ABSTRACT

We study the problem of falsifying reachability properties of real-time control software acting in a closed-loop with a given model of the plant dynamics. Our approach employs numerical techniques to simulate a plant model, which may be highly nonlinear and hybrid, in combination with symbolic simulation of the controller software. The state-space and input-space of the plant are systematically searched using a plant abstraction that is implicitly defined by "quantization" of the plant state, but never explicitly constructed. Simultaneously, the controller behaviors are explored using a symbolic execution of the control software. On-the-fly exploration of the overall closed-loop abstraction results in abstract counterexamples, which are used to refine the plant abstraction iteratively until a concrete violation is found. Empirical evaluation of our approach shows its promise in treating controller software that has precise, formal semantics, using an exact method such as symbolic execution, while using numerical simulations to produce abstractions of the underlying plant model that is often an approximation of the actual plant. We also discuss a preliminary comparison of our approach with techniques that are primarily simulation-based.

Keywords

Reachability; Hybrid Systems; Falsification; Program Analyses

1. INTRODUCTION

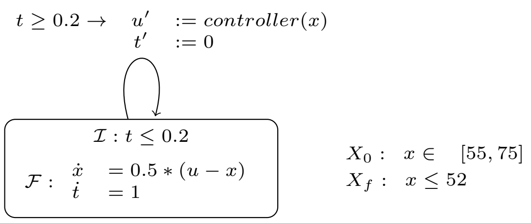
In this paper, we study the problem of searching for potential safety violations in real-time controller software by performing a closed-loop symbolic execution of the software in conjunction with a model of the plant dynamics being controlled. Such a closed-loop exploration is quite valuable as it incorporates controller software implementations rather than abstract, hybrid-automata-based models commonly used in formal reachability analysis tools. This allows us to model software centric issues such as fixed point arithmetic, overflows, division by zero and buffer overflows. At the same time, our approach uses a model of the plant dynamics that assures us that bugs found in this process are potentially realizable when deploying the control system.

However, closed-loop symbolic exploration of a controller with its accompanying plant model is quite challenging. Plant models are often nonlinear, and may exhibit hybrid behaviors due to discrete changes in the operating mode. Most modern controller software systems have different control regimes that are chosen based on the environmental conditions, which leads to several control-flow paths in the control code. Exhaustive exploration of all possible combinations of control-flow paths in the controller and plant dynamics can become prohibitively complex. Furthermore, in the control system development process, the control software is an artifact that is deployed in the final system. The plant model, on the other hand, is typically an (unsound) approximation of the actual physical environment, which is hard to characterize exactly. Plant models are often created for the purposes of evaluation and testing of specific aspects of the closed-loop system. Therefore, a precise treatment of the controller semantics is a desirable goal in control verification. However, a similar precise approach to the plant semantics is often prohibitively expensive. Finally, our approach focuses primarily on *falsification*– a best effort search for counterexamples, rather than proving the correctness of closed loop systems.

In this work, we consider the control software to be architected as a set of tasks operating at a fixed rate, where, in each run, the controller reads inputs from the sensors, computes a control value and outputs this to the plant. We call the period at which the controller software runs as the controller sampling period. The value output by the controller is held constant (zero-order hold) for the controller sampling period, before the controller updates it. The plant model is provided as a *black box* specified as a function $\textsc{sim}(\mathbf{x}, \mathbf{u}, \tau)$ that simulates the plant model for time $\tau > 0$ starting from a current state $\mathbf{x}$ and under an input $\mathbf{u}$. The underlying plant itself can be a nonlinear hybrid system, or even a data-driven model such as a *neural network* that maps a current state to a next state. We assume that the n-dimensional state of the plant is fully observable by our testing framework. We propose an abstraction of the closed-loop system where: (a) the set of states of the controller is represented by a formula in a suitable logical theory (such as real arithmetic with linear constraints), and (b) the plant is abstracted using a cell-to-cell mapping defined by quantizing real-valued plant states into a finite representation obtained by a *quantization operator* [30]. The plant abstraction is defined as a standard existential abstraction between quantized states and explored using numerical simulations. The controller abstraction is exact, as the operation of the controller is modeled as symbolic execution on the given set of controller states. As a result, our approach treats the controller semantics precisely while potentially missing out on possible plant behaviors. However, any "robust" behavior of the plant can be discovered by increasing the number of simulations used to build the plant abstraction [30].

The overall approach then explores the joint abstractions of the controller and plant using a depth-first search or breadth-first search strategy up to a given time horizon. If the process produces a violation of the time bounded safety property, we discover an abstract counterexample trace. A simplistic refinement scheme that increases

HSCC'16, April 12-14, 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-3955-1/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2883817.2883819



$t \geq 0.2 \rightarrow \quad u' := controller(x)$
$t' := 0$
$\mathcal{I} : t \leq 0.2$
$\mathcal{F} : \dot{x} = 0.5 * (u - x)$
$\dot{t} = 1$
$X_0 : \; x \in [55, 75]$
$X_f : \; x \leq 52$

Figure 1: The hybrid automaton for the room-heater-thermostat sampled data system with initial set X_0 and unsafe set X_f.

the number of bits retained by the quantization operator is then used to refine these abstract counterexample further to produce concrete counterexamples.

We implement the overall approach through the combination of the symbolic execution tool Pathcrawler [29] with a simulation infrastructure that can handle plant specifications in a variety of formats including Simulink$^{\text{TM}}$ /Stateflow$^{\text{TM}}$ models. Using Pathcrawler allows us to treat the control software *as is*, including the ability to handle fixed point arithmetic that is commonly used in embedded controllers. We compare our approach with simulation-based approaches on a few challenging controller benchmarks. By comparing our approach to the standard practice of performing Monte Carlo simulations with uniform random sampling of the initial states and inputs, we hope to highlight the underlying difficulty of arriving at an undesirable behavior purely by chance. We also compare our approach with S-Taliro, a falsification tool that uses simulations guided by an optimizer attempting to minimize the "distance" to a bug; where buggy behavior is specified using Metric Temporal Logic (MTL), and the distance to bug is defined using robust satisfaction semantics for MTL. We also compare with an earlier version of our tool, called S3CAM, that is based on an abstraction-refinement based approach where both the plant and controller abstractions are constructed using simulations.

On one hand, we find that simulation-based approaches are often much faster. This is because of the high overhead of symbolic executions, especially as symbolic execution tools typically use bitvector theories to reason about arithmetic. On the other, we find examples where corner-case behaviors in the controller software trigger reachability violations that cannot be found by the other simulation-based approaches. In doing so, we demonstrate preliminary evidence that despite the high cost of our approach, it can be potentially valuable in detecting corner-case violations that can be missed by approaches primarily based on simulations.

1.1 Motivating Example

Consider the example of a room heating system regulated by a thermostat that controls the operating mode of the heater. The plant dynamics are modeled as a single-mode hybrid automaton with linear dynamics as shown in Fig. 1. The controller action encapsulated inside the reset map of this automaton, can be one of three kinds: OFF, REGULAR HEATING, or FAST HEATING. The control software is a `C` program shown in Figure 3. This code incorporates extra control logic to prevent the heater from being switched between different modes frequently. The controller is run at 5 Hz (i.e, a controller time-step of 0.2 seconds). Assuming the initial temperature of the room to be $x \in [55, 75]^\circ$ F, we try to find a scenario where the temperature dips too low ($x < 52^\circ$ F) within 10s of system operation. Our implementation runs for about 9 seconds before discovering a violation. A setting where the initial room temperature is set to a small range around $69.9^\circ F$ exposes the poorly coded chatter protection. The system chatters, and the thermostat forces the heater to be non-responsive for too long, causing the room temperature dip too low. Using 100,000 random simulations running for 20 minutes (about 130x longer than our approach), we

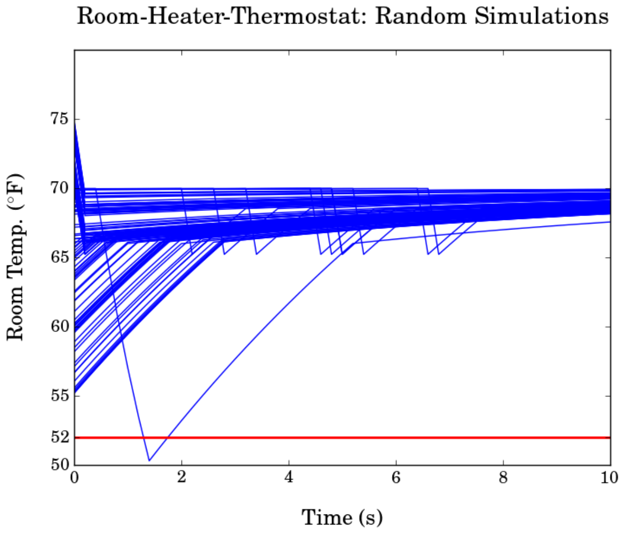


Figure 2: A plot showing around 100 biased random simulations with $T = 10s$. The unsafe regions is below red line at $(52^\circ F)$. Biasing towards $x \in [69.9, 70]$ helps magnify the unsafe behaviors.

find 45 violations (instances where temperature dips below 52°F). This corresponds to roughly a one in 1000 chance of discovering this violation by Monte Carlo simulations with uniform random sampling of the initial state.

2. RELATED WORK

Falsification for Hybrid Systems. In industrial-scale hybrid systems, tools based on falsification of formal safety requirements have recently emerged as a practical alternative to verification approaches which seldom scale well for complex systems. A key factor in this change is that falsification techniques are typically simulation-based and usually best-effort in nature, often providing only asymptotic or probabilistic correctness guarantees. S-Taliro [1] and Breach [10] are tools based on single-shooting based optimization techniques. These tools use the robust satisfaction semantics of a given temporal logic requirement as a cost function to guide the underlying optimizer to find initial states and a parameterized input signal that leads the system to a violation. The term "single shooting" refers to the fact that the optimizer picks a single initial state and an input signal, and then computes the cost of the objective function on the resulting output signal. These tools use a plethora of global optimizers with powerful heuristics to get around the problem that the cost surface is typically highly nonlinear, and occasionally discontinuous. In industrial-scale models, cost surfaces are a challenge for global optimizers, as they can be "flat," *i.e.*, lacking any gradient information to suggest a search-direction for the optimizer. In our experience, such scenarios are often encountered when the controller code has complex Boolean conditions over the continuous state variables or discrete state variables representing operating modes, as they result in mixed discrete-continuous optimization problems. Another dependency of the extant falsification tools is a good distance metric quantifying the distance to bug, which is a challenge to define in the presence of discrete state variables or operating modes, without deep internal knowledge of the model structure [24].

Some of the above challenges are mitigated in our previous work [30]. This technique uses a multiple shooting-based optimization technique, that does not depend on a distance metric on a hybrid state-space. Unlike most simulation based methods, multiple shooting utilizes multiple short simulation traces to search for a falsification. However, this work still treats the entire closed-loop system as a black-box.

```
#define MAX_TEMP (70.0)
#define MIN_TEMP (66.0)
#define CHATTER_LIMIT (2)

int controller(double room_temp){
  //**************************************************
  // on_ctr, off_ctr      :counters to track on/off cycles
  // chatter_detect_ctr  :counter to track chattering
  // previous_command    :previous command
  // command             :current command
  // u                   :control input to the heater
  //**************************************************
  static int on_ctr, off_ctr, chatter_detect_ctr;
  static int previous_command, command, u;

  // Compute command to heater based on room temperature
  if(room_temp >= MIN_TEMP && room_temp < MAX_TEMP)
    command = NORMAL_HEAT;
  else if(room_temp >= MAX_TEMP)
    command = NO_HEAT;
  else if(room_temp < MIN_TEMP)
    command = FAST_HEAT;
  else
    command = previous_command;

  // Chattering absent, reset the counter
  if(off_ctr >= 5 || on_ctr >= 5)
    chatter_detect_ctr = 0;

  // New command != previous command. Possible chattering
  if(command != previous_command)
    chatter_detect_ctr++;

  // Chattering detected, hold previous command
  if(chatter_detect_ctr > CHATTER_LIMIT)
    command = previous_command;

  // Increment counters
  if(command == NO_HEAT){
    on_ctr = 0;
    off_ctr++;
  }else{
    on_ctr++;
    off_ctr = 0;
  }

  // Translate command to control input
  if(command == NO_HEAT)      u = 20;
  if(command == FAST_HEAT)    u = 100;
  if(command == NORMAL_HEAT)  u = 70;

  return u;
}
```

Figure 3: C code for the Thermostat. All initial control states are 0.

Rapidly exploring Random Trees [19] (RRTs) have also been used to falsify safety properties for hybrid systems [2,11,17,23]. RRTs use a randomized tree-based algorithm to search for a finite sequence of discrete inputs which can lead to unsafe system states. Recent advances in the context of falsification include using a combination of sophisticated heuristics to maximize the exploration of reachable state-space (coverage), and biasing of the tree towards the goal using robust satisfaction measures over partial traces, Even though RRTs have been widely successful in planning, they are not as efficient in finding a violating trajectory of high dimensional search space with a highly constrained reachable space (as is the case with under-actuated systems). They too, operate on a black box assumption.

Symbolic Execution of Programs. Symbolic execution was first formally proposed in [18]. Since then, with increasingly powerful constraint solvers, it has evolved into an efficient code analysis technique, forming the basis for tools such as CUTE [28], KLEE [3], and Pathcrawler [29]. These tools are usually employed to generate inputs that maximize the coverage of control-flow paths in the program. For large programs, a purely symbolic approach can be quite inefficient, and a modified version of symbolic execution, where concrete states are maintained alongside symbolic states is usually

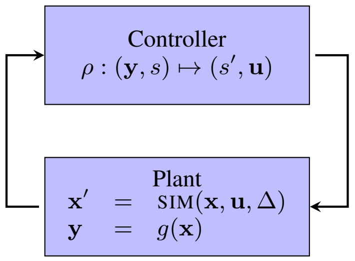

Controller
$\rho : (\mathbf{y}, s) \mapsto (s', \mathbf{u})$
Plant
$\mathbf{x}' = \textsc{sim}(\mathbf{x}, \mathbf{u}, \Delta)$
$\mathbf{y} = g(\mathbf{x})$

Figure 4: Closed loop composition of a plant and a controller model with controller sampling period Δ.

preferred. This enables program analyses in the presence of complex constraints that the constraint solvers can not handle, and efficient generation of test cases for path coverage, when 100% coverage is infeasible. Due to their symbolic nature, such techniques can be fused easily with CEGAR-like techniques [6], which we propose in this presentation. Recent surveys on symbolic execution can be found at [4,5].

Closed Loop Analyses Techniques. Closed loop falsification was first proposed by Lerda et al. [20,21], where model checking of the software was combined with simulation-based systematic exploration of the physical system. Majumdar et al. [22] proposed a verification mechanism for linear dynamical systems using symbolic execution of both the controller code and the plant (using over-approximation of reachable states). In contrast, we provide a more efficient but best effort approach where the physical dynamics are treated as a black-box.

3. SAMPLED DATA CONTROL SYSTEM

In this section, we provide useful definitions, including the problem setup along with the basic abstractions for the controller and the plant. We are interested in analyzing closed-loop systems consisting of a *plant* and a *controller*, wherein the plant is a physical process modeled as a dynamical system and the controller is implemented as a set of software tasks that execute repeatedly with a fixed period known as the *sampling period*.

DEFINITION 3.1 (PLANT MODEL). *The plant model is described by a set of plant states $\mathcal{X}$, plant inputs $\mathcal{U}$ and plant outputs $\mathcal{Y}$ along with two functions:*

- *A simulation function* $\textsc{sim} : \mathcal{X} \times \mathcal{U} \times \mathbb{R}_{\geq 0} \mapsto \mathcal{X}$, *where* $\textsc{sim}(\mathbf{x}, \mathbf{u}, \tau)$ *maps the current state* $\mathbf{x}$ *at time* t *to the next state* $\mathbf{x}'$ *at time* $t + \tau$ *(where* $\tau \geq 0$*) with the assumption that the input signal* $u(t)$ *is a constant* $\mathbf{u} \in \mathcal{U}$ *for* $t \in [0, \tau)$.
- *An observation function* $g : \mathcal{X} \mapsto \mathcal{Y}$ *that maps the current state* $\mathbf{x}$ *to the observable output* $\mathbf{y} = g(\mathbf{x})$.

The $\textsc{sim}$ function satisfies the property that $\textsc{sim}(\mathbf{x}, \mathbf{u}, 0) = \mathbf{x}$ for all $\mathbf{x} \in \mathcal{X}$ and $\mathbf{u} \in \mathcal{U}$.

The controller samples the output $\mathbf{y}$ of the plant at regular time instants, and updates the control input $\mathbf{u}$ before the next time instant. Controllers are assumed to have an internal state s that is updated by the execution of the controller.

DEFINITION 3.2 (CONTROLLER MODEL). *A controller is specified in terms of its input space $\mathcal{Y}$, its internal state space $\mathcal{S}$, and the controller sampling period Δ. Its semantics are provided by a function $\rho : \mathcal{Y} \times \mathcal{S} \mapsto \mathcal{U} \times \mathcal{S}$, where the function $\rho(\mathbf{y}, s)$ maps the controller input $\mathbf{y}$ (which is the plant output at time t) and internal state s (at time t) to $(s', \mathbf{u})$, where s' and $\mathbf{u}$ are the updated controller state and the input to the plant at time $t + \Delta$, respectively.*

The above mentioned parallel composition of a plant and a controller (Figure 4) is called a *Sampled Data Control System (SDCS)*.

DEFINITION 3.3 (SAMPLED DATA CONTROL SYSTEM). *A* sampled data control system *(SDCS) consists of two components, as illustrated in Figure 4. (a) A plant model P described by two functions* SIM *and g as in Definition 3.1, and (b) a controller implementation C described by a program whose semantics are described by a function ρ as in Definition 3.2. Finally, the closed-loop parallel composition assumes that the function* SIM *is always called with $\tau = \Delta$, i.e., the controller sampling period.*

In practice, sampled data control systems include A/D (analog-to-digital) and D/A converters for interfacing between the analog plant and the digital controller. Errors are often introduced due to the presence of measurement noise and the quantization of the A/D and D/A converters. Though we allow our models to have an exogenous input modeling a bounded controller disturbance, and allow searching over the disturbance-space during the falsification process, for simplicity of presentation, we omit this from the formalization.

The state of the closed-loop system is given by $(\mathbf{x}, s, \mathbf{u})$ where $\mathbf{x} \in \mathcal{X}$ denotes the plant state, $s \in \mathcal{S}$ denotes the internal control state and $\mathbf{u} \in \mathcal{U}$ the plant input. Let $\mathbf{x}_0$ be the initial plant state at $t = 0$, s_0 be the initial controller state and u_0 be the initial plant input or controller output. Given a controller sampling period Δ, the operational semantics of the closed-loop model can be described as a countable sequence of plant and controller moves as follows:

$$\begin{aligned}(\mathbf{x}_0, s_0, \mathbf{u}_0) \rightsquigarrow \quad & (\mathbf{x}_1, s_0, \mathbf{u}_0) \rightarrow (\mathbf{x}_1, s_1, \mathbf{u}_1) \rightsquigarrow \\ & (\mathbf{x}_2, s_1, \mathbf{u}_1) \rightarrow (\mathbf{x}_2, s_2, \mathbf{u}_2) \cdots\end{aligned}$$

In each of the above states, the index i denotes the real time $i\Delta$. The closed-loop model interleaves two types of moves:

- *Plant Moves:* $(\mathbf{x}_i, s_i, \mathbf{u}_i) \rightsquigarrow (\mathbf{x}_{i+1}, s_i, \mathbf{u}_i)$, where $\mathbf{x}_{i+1} = \textsc{sim}(\mathbf{x}_i, \mathbf{u}_i, \Delta)$ is the next state of the plant after time Δ has elapsed with input $\mathbf{u}_i$. The move has no effect on the controller state, or the control input to the plant that is held constant (zero-order hold).
- *Control Moves:* $(\mathbf{x}_{i+1}, s_i, \mathbf{u}_i) \rightarrow (\mathbf{x}_{i+1}, s_{i+1}, \mathbf{u}_{i+1})$ describes a move by the controller that denotes an instantaneous execution of the control program to yield $(s_{i+1}, \mathbf{u}_{i+1}) = \rho(g(\mathbf{x}_{i+1}), s_i)$. In our idealized semantics, no time elapse occurs during this computation.

Note: The idealized semantics ignores the time taken by the controller code to execute. However, when this time is assumed to be much smaller when compared to the overall time period Δ and the plant's dynamics are assumed to be "slow" enough, the idealized semantics can be justified due to their simplicity. Failing this, we may assume a small but known execution time $\hat{\Delta}$ for the controller and define the controller move to also allow the plant state to change while the controller finishes its computation:

$$(\mathbf{x}, s, \mathbf{u}) \rightarrow (\hat{\mathbf{x}}, \hat{s}, \hat{\mathbf{u}})$$

wherein $\hat{\mathbf{x}} = \textsc{sim}(\mathbf{x}, \mathbf{u}, \hat{\Delta})$ and the remaining parts of the definition remain intact.

3.1 Software-Centric View of the Controller

So far, we have used a map ρ to describe the controller. In most industrial embedded systems, implementations of controllers use imperative programming languages such as `C`. For the purpose of our analysis, we present the controller software as a control-flow graph (CFG), a structure that focusses on the structural organization of the execution paths in the controller software.

In the following presentation, we omit any discussion on features such as function calls, arrays, and pointers, but remark that as our technique uses off-the-shelf analysis tools, such features can be handled by our implementation. On the other hand, a programming language like `C`, allows dynamic memory allocation, recursive execution without known termination bounds, pointer arithmetic, and complex data structures. Such features are rarely found in real-time embedded control software, and we can safely assume that we do not encounter these in the controllers to be analyzed. We now formalize the control software as a CFG.

DEFINITION 3.4 (CONTROL-FLOW GRAPH). *A control-flow graph is defined as a tuple $\langle \mathcal{V}, \mathcal{V}_i, \mathcal{V}_o, L, E, \Phi, l_0, l_f \rangle$, where $\mathcal{V}$ is a set of variables, $\mathcal{V}_i \in \mathcal{V}$ and $\mathcal{V}_o \in \mathcal{V}$ are the input and output subset*[1]*. L is a set of nodes (control locations), l_0, l_f are unique start and end locations, representing entry and exit points of the given program respectively. $E \subseteq l \times l$ is a finite set of directed edges, Φ is a function labeling each edge$(l, l') \in E$ with two kinds of constraints:*

1. *An assignment constraint has the following form:*

$$(v_{l'} = e(\mathcal{V}_l)) \;\wedge \bigwedge_{w \in \mathcal{V}\setminus\{v\}} (w_{l'} = w_l).$$

It arises from an assignment statement $v := e$ in the program, where e is the symbolic expression signifying a function over some subset of variables in $\mathcal{V}$. The constraint itself relates the value of the modified variable v at location l' to the values of the variables at location l through the function e, and asserts that all other variables remain unchanged.

2. *A conditional constraint has one of the following forms:*

$$1.assume(b(\mathcal{V}_l)) \qquad 2.assume(\neg b(\mathcal{V}_l)).$$

It arises from a conditional statement of the form `if(`*b*`) then` *l'* `else` *l''. Here b is a Boolean-valued symbolic expression*[2] *over the variables. For the edge(l, l'), the first label is used, wheres for the edge(l, l''), the second label is used.*

DEFINITION 3.5 (CONTROL-FLOW PATH). *An entry-exit control-flow path π is a sequence of nodes, $l_0, \ldots, l_i, \ldots, l_f$, beginning with l_0 and ending in l_f, such that each location pair $(l_i, l_{i+1}) \in E$.*

During program execution an $edge(l, l')$ is taken if its label evaluates to *true*. The *conditional* label is assigned the valuation of its expression b or $\neg b$. The sequence of edges naturally partitions a program into a set of paths $\Pi = \{\pi_1, \ldots, \pi_N\}$. Let each path π_i be described by a path constraint $\rho_i(\mathcal{V}_i, \mathcal{V}_o)$ which sequentially composes the constraints along π_i. The path constraint ρ_i can be understood intuitively as a combination of (a) a path condition $\xi(\mathcal{V}_i)$ on the program inputs which decides if the path is feasible and (b) a path function $\mathcal{V}_o := f_i(\mathcal{V}_i)$ that describes the updates through the assignment statements along the path.

We can now summarize the program as the union of all possible control-flow paths $\rho = \bigcup_{i=1}^{N} \rho_i$. It can be easily shown that this union of path constraints exhaustively covers the entire set of values for $\mathcal{V}_i$, and thus, $\rho(\mathcal{V}_i, \mathcal{V}_o)$ can be written as a function on program inputs $\mathcal{V}_i := \rho_i(\mathcal{V}_i)$. In the present context, we can formulate the piecewise function which computes the controller move for a controller with N paths as follows:

$$\rho(\mathbf{y}, s) = \begin{cases} f_1(\mathbf{y}, s) & \text{if } \xi_1(\mathbf{y}, s) \\ \ldots & \\ f_N(\mathbf{y}, s) & \text{if } \xi_N(\mathbf{y}, s) \end{cases} \qquad (1)$$

[1]Note that some of these variables represent the internal state of the controller.

[2]It is assumed that b is side-effect free, i.e., it does not modify the values of the program variables.

This is accomplished by using symbolic execution to find the path condition ξ_i and path function f_i for every path π. In general, a software program might not terminate due to the presence of an infinite path. Additionally, the number of paths in a program can be infinite. It is also non-trivial to determine such cases. Fortunately, best practices in embedded control software discourage the use of jump statements and unbounded loops. Most loops have specified, fixed bounds, and we assume that the same rule is true for the controllers that we encounter. Under this assumption, there is a finite number of finite length control paths in the controller software.

4. IMPLICIT QUANTIZED ABSTRACTION

We now consider the abstraction of the closed loop system by defining abstractions of the plant and the controller state spaces.

4.1 Plant Abstraction

The abstraction of the plant is defined by a *tiling* of the state-state and the control-input space[3] $\mathcal{X} \times \mathcal{U}$ into a set of cells $\mathcal{C} : \{C_1, C_2, \ldots\}$. The set $\mathcal{C}$ is a partition on $\mathcal{X} \times \mathcal{U}$, i.e., the cells are pairwise disjoint $C_i \cap C_j \neq \emptyset$ if $i \neq j$, and their union $\bigcup_{C_j \in \mathcal{C}} C_j$ is equal to $\mathcal{X} \times \mathcal{U}$. Rather than performing an explicit construction of a tiling of $\mathcal{X} \times \mathcal{U}$, we define the tiling implicitly through *quantization*.

We introduce a quantization function which essentially truncates the decimal representation of the state and control-input to a given level of precision d (i.e., digits after the decimal point).

DEFINITION 4.1 (d-PRECISE QUANTIZATION). *A d-precise quantization is a function* QUANT_d *that maps a state-input pair* $(\mathbf{x}, \mathbf{u})$ *to a* quantized *pair* $(\hat{\mathbf{x}}, \hat{\mathbf{u}})$ *such that:*

$$\text{QUANT}_d(\mathbf{x}, \mathbf{u}) = \frac{1}{10^d} \left(\lfloor 10^d \mathbf{x} \rfloor, \lfloor 10^d \mathbf{u} \rfloor \right) \tag{2}$$

It is easy to see that the d-precise quantization function induces an equivalence relation $\equiv_d$, such that:

$$(\mathbf{x}_1, \mathbf{u}_1) \equiv_d (\mathbf{x}_2, \mathbf{u}_2) \text{ iff } \text{QUANT}_d(\mathbf{x}_1, \mathbf{u}_1) = \text{QUANT}_d(\mathbf{x}_2, \mathbf{u}_2)\,.$$

We observe that the quotient set $(\mathcal{X} \times \mathcal{U})/_{\equiv_d}$ is a set of cells defined as follows. Let $\mathbf{j}$ be a vector of integers, with dimension of $\mathbf{j}$ equal to the sum of dimensions of $\mathcal{X}$ and $\mathcal{U}$. Let $C_\mathbf{j}$ be defined as:

$$C_\mathbf{j} = \{(\mathbf{x}, \mathbf{u}) \mid \text{QUANT}_d(\mathbf{x}, \mathbf{u}) = \frac{\mathbf{j}}{10^d}\} \tag{3}$$

Then, the set $C_\mathbf{j}$ is an element of the quotient set $(\mathcal{X} \times \mathcal{U})/_{\equiv_d}$, with the representative element being $\frac{\mathbf{j}}{10^d}$. It is easy to see that the set of cells $C_\mathbf{j}$ forms a tiling; we call this *d-quantized tiling*, with $\equiv_d$ as the cell-equivalence relation. In simple terms, given a d, for any state-input pair $(\mathbf{x}, \mathbf{u})$, the index of the cell to which $(\mathbf{x}, \mathbf{u})$ belongs can be obtained by simply truncating the decimal representation of $(\mathbf{x}, \mathbf{u})$ to d digits and multiplying the result by 10^d. For a given d-quantized tiling, checking if two states belong to the same cell, simply involves checking if their first d digits in decimal notation are the same.

DEFINITION 4.2 (d-SAMPLING). *Given a d-quantized tiling $\mathcal{C}$ and a cell $C_\mathbf{j}$ in the tiling, a d-sampling is defined as a random sample of elements of $C_\mathbf{j}$.*

We observe that d-sampling is fairly trivial to obtain. If the number of dimensions of the representative element of $C_\mathbf{j}$ is $(n+m)$, then we take $(n+m)$ random strings of numbers and append them to each dimension in $\frac{\mathbf{j}}{10^d}$ to get one new random sample. We repeat this process till we get a set of desired size. If each random string generation uses a uniform random sampling, the resulting set contains points that have uniform distribution.

[3]The quantization is performed only for continuous values and discrete values are treated as is.

Given a tiling, the plant abstraction is now defined as an existential abstraction that connects two cells. We formally define this below:

DEFINITION 4.3. *Let $\mathcal{C}$ be a d-quantized tiling. The d-quantized tiling-based plant abstraction (denoted $\mathcal{P}_d$) is given by a graph (V, E), where $V = \mathcal{C}$, and the set of edges E is defined such that $(C_\mathbf{j}, C_\mathbf{k}) \in E$ iff: there is a state-input pair $(\mathbf{x}, \mathbf{u}) \in C_\mathbf{j}$ and a state-input pair $(\mathbf{x}', \mathbf{u}) \in C_\mathbf{k}$ such that $\mathbf{x}' = \text{SIM}(\mathbf{x}, \mathbf{u}, \tau)$.*

In other words, cells $C_\mathbf{j}$ and $C_\mathbf{k}$ are connected if there is some source state and source input such that the destination state can be reached from the source state using simulation under the action of the source input. It is easy to show that the abstraction $\mathcal{P}_{d'}$ refines $\mathcal{P}_d$ whenever $d' > d$. I.e, keeping more digits around produces a finer abstraction of the system.

So far, we have implicitly defined the abstraction of the plant. However, we have not yet provided the means to construct these abstractions for a given plant model. We now present the use of numerical simulations to explore $\mathcal{P}_d$.

On-the-fly exploration of the Plant Abstraction: A primitive operation involved in the exploration is to check for a given pair of cells $C_\mathbf{j}$ and $C_\mathbf{k}$ whether the edge from $C_\mathbf{j}$ to $C_\mathbf{k}$ exists in the abstraction. In the absence of simplifying assumptions about the nature of the plant model, this problem is equivalent to the general nonlinear reachability problem, and is undecidable in general. In our setup, we have made no assumptions beyond the efficient computation of the SIM function that defines the plant's discrete time behavior. Therefore, we resort to a numerical approach called *scatter-and-simulate*, first introduced in previous work by some of the authors [30]:

1. Obtain a d-sampling of $C_\mathbf{j}$ of size $\mathcal{N}$, $\mathcal{N} > 0$.
2. For each sampled state $\mathbf{x}_\ell$ in the d-sampling, compute $\mathbf{x}'_\ell$ = $\text{SIM}(\mathbf{x}_\ell, \mathbf{u}_\ell, \tau)$.
3. Check if $(\mathbf{x}'_\ell, \mathbf{u}_\ell) \in C_\mathbf{k}$ for any $\ell \in [1, \mathcal{N}]$. If yes, then $C_\mathbf{j}$ *must* have an edge to $C_\mathbf{k}$ in $\mathcal{P}_d$. Otherwise, $C_\mathbf{j}$ *may not* have an edge to $C_\mathbf{k}$.

In practice, the sampling of $\mathcal{N}$ states can either use a deterministic sampling scheme or the samples can be drawn uniformly at random from C_i. The value of $\mathcal{N}$ itself is a parameter that can be adjusted to achieve a tradeoff between time and precision.

An abstract edge from $C_\mathbf{j}$ to $C_\mathbf{k}$ is *robust* iff there exist a connected subset $C \subseteq C_\mathbf{j}$ of nonzero volume such that every state $(\mathbf{x}, \mathbf{u}) \in C$ leads to a state in $C_\mathbf{k}$ upon the application of the SIM function.

It is shown in our previous work (and elsewhere) that for uniform sampling of $C_\mathbf{j}$, as $\mathcal{N}$ increases all robust edges are found with probability 1. However, there may be non-robust plant edges that cannot be discovered by the procedure no matter how large $\mathcal{N}$ is. This is a key theoretical limitation caused by our reliance on black-box models and simulations. In practice, however, non-robust plant edges seldom exist. For instance, the problem of simulating a plant model with such non-robust edges is already nontrivial in the presence of numerical errors.

Another operation of interest over the d-quantized tiling is to find all neighbors $C_\mathbf{k} \in \mathcal{C}$ of a given cell $C_\mathbf{j}$ such that $(C_\mathbf{j}, C_\mathbf{k}) \in E$. The process of scatter and simulate is trivially modified to collect all the reached cells $C_\mathbf{k}$.

To define the controller abstraction, we need to similarly define a quantization function on the plant output $\mathbf{y}$, such that C^y = $\text{QUANT}_d(y)$. This will allow us to reason over abstract plant outputs in the same way as abstract plant states C.

4.2 Controller Abstraction

Given a CFG with N control-flow paths $\pi_1, \ldots, \pi_N$, recall that we have a set of concrete partial path functions $\rho_1, \ldots, \rho_N$ whose union yields the overall semantics ρ of the program as in Eq. (1).

Most real-world controllers use nonlinear expressions and conditions in assignment and conditional statements respectively. Performing analyses over such controller software (e.g. to perform symbolic execution), requires solvers capable of reasoning over such theories. Unfortunately, reasoning about non-polynomial arithmetic (such as transcendentals) is undecidable [26], while reasoning about polynomial arithmetic, while decidable, is computationally expensive. A common approach taken in traditional software verification (such as in abstract interpretation [7]) is to have conservative over-approximation of such operations using efficient logical theories (such as linear arithmetic). In this context, we need to over-approximate each path constraint ρ_i to obtain path constraint $\hat{\rho}_i$ that is expressible using the theory of linear arithmetic.

This is typically formalized using an abstraction function α that maps path constraints ρ_j into abstraction path constraints $\hat{\rho}_j : \alpha(\rho_j)$ that satisfy the following property: $\rho_j(\mathbf{y}, s) \subseteq \hat{\rho}_j(\mathbf{y}, s)$. The overall abstract controller relation $\hat{\rho}$ is simply the union of $\hat{\rho}_j$ for each path π_j in the control code. We can abuse notation and use $\hat{\rho}(C^y, \varphi)$ to denote the action of the (abstract) controller on a given cell C^y from the tiling-based plant abstraction, and an abstract set of controller states φ. The result yields a set of control inputs U and updated controller states ψ; we formalize the soundness of the abstraction in the following assumption:

ASSUMPTION 4.1 (CONTROLLER ABSTRACTION SOUNDNESS). *Suppose in the abstract semantics, we have $(U, \psi) = \hat{\rho}(C^y, \varphi)$, then for all plant outputs $\mathbf{y} \in C^y$ and all controller states $s \in \varphi$, the concrete values $(\mathbf{u}', s') = \rho(\mathbf{y}, s)$ are contained in their respective abstract states: $\mathbf{u}' \in U$ and $s' \in \psi$.*

Having defined an abstraction of the controller semantics, the goal is to compute $(U, \psi) = \hat{\rho}(C^y, \varphi)$. This is achieved through an abstract symbolic execution of the program. One way to achieve this is by precomputing each relation $\hat{\rho}_j$ for path π_j as a tuple of assertions (ξ_j, R_j, T_j), where ξ_j encodes the abstract path condition on C^y and φ, and R_j and T_j are projections of $\hat{\rho}(C^y, \varphi)$ on the controller states and controller outputs (i.e. plant inputs) respectively.

5. CLOSED LOOP FALSIFICATION

Having defined the plant and controller abstractions, we now explore a series of abstract states to analyze the safety of a plant and controller state combinations. The approach presented here can extend to more complex bounded-time temporal logic properties by instrumenting the system with a temporal logic monitor [13–15], and searching for the reachability of a target state in the monitor.

DEFINITION 5.1 (CLOSED LOOP ABSTRACT STATE). *The abstract state of the overall closed-loop system is a combination (C, φ), where C is an abstract plant cell and φ is an abstract controller state, i.e., a logical predicate specifying a set of controller states.*

The closed-loop system abstraction is explored in two phases:

- We consider an abstract plant move $(C, \varphi) \rightsquigarrow_A (C', \varphi)$ wherein $(C, C') \in E_A$ belongs to the abstract edge relation between cells. The new cell C' is generated using the simulate-and-scatter procedure that samples cell C and performs a concrete simulation using the plant's SIM function.
- Next, we consider a controller move $(C', \varphi) \rightarrow_A (C'', \psi)$ wherein we compute $(U, \psi) = \hat{\rho}(C^y, \varphi)$ and obtain a new cell $C'' = \textsc{Quant}_d(\mathbf{x}', \mathbf{u}')$ for some $\mathbf{x}' \in C'$, and some $\mathbf{u}' \in U$ by quantizing the new control inputs with the plant states. The updated controller states are now represented by ψ. Also, $C^y = \textsc{Quant}_d(g(x'))$.

The two moves combine to give a closed loop move of the system: $(C, \varphi) \xrightarrow{\Delta}_A (C'', \varphi')$.This can now be used for computing the set of reachable abstract states $R_\Delta : \{(C'', \varphi') \mid (C, \varphi) \xrightarrow{\Delta}_A (C'', \varphi')\}$ a one timestep ($\tau = \Delta$). We now present this combined closed loop step as an algorithm.

Algorithm 1: Closed Loop Execution on Abstract States

```
Input: Abstract states $(C, \varphi)$, plant input $\mathbf{u}$, abstract plant
       output $C_y$, plant simulator SIM, controller summary $\rho$
Output: Reachable abstract states $R_\Delta$
1 $X_u := \{(\mathbf{x}_1, \mathbf{u}_1), \ldots, (\mathbf{x}_\mathcal{N}, \mathbf{u}_\mathcal{N})\} = \mathsf{d\_sample}(C, \mathcal{N})$
2 $X'_u := \{\mathbf{x}' \mid \mathbf{x}' = \textsc{sim}(\mathbf{x}, \mathbf{u}, \Delta) \wedge (\mathbf{x}, \mathbf{u}) \in X_u\}$
3 $Y := \{C^y \mid C^y = \textsc{Quant}_d(g(\mathbf{x}')) \wedge \mathbf{x}' \in X'_u\}$
4 $S_{\varphi Y} := \{(U, \psi) \mid (U, \psi) = \hat{\rho}(C^y, \varphi) \wedge C^y \in Y\}$
5 $R_\Delta := \{(C'', \psi) \mid C'' = \textsc{Quant}_d(\mathbf{x}', \mathbf{u}') \wedge \mathbf{u}' =$
  $sample(U) \wedge (U, \psi) \in S_{\varphi Y} \wedge \mathbf{x}' \in X'_u\}$
```

To compute the plant step, first d-sampling is used to obtain $\mathcal{N}$ state-input pairs which are then supplied to SIM to compute a representative set of next plant states X'_u . The corresponding set of representative plant outputs are also computed using $g()$ and are then quantized to get Y. Using this set of abstract plant outputs, and controller function $\hat{\rho}$ new abstract controller states and outputs are computed. The controller inputs u are then sampled (using a constraint solver, as discussed in Sec. 6). Finally, R_Δ is built by quantizing plant state-input $(\mathbf{x}', \mathbf{u}')$ pair. The overall search algorithm scatter-and-simulate remains the same as explained in [30].

Thus, in practice, exploring the abstraction is a combination of exploring the plant abstraction through a numerical simulation procedure to compute the resulting cells and then running an abstract symbolic execution of the controller code. In doing so, the robust edges in the plant are captured whereas all possible moves by the controller are taken into account.

6. IMPLEMENTATION

We have prototyped the presented falsification technique as S3CAMX, a falsification tool which uses *scatter-and-simulate* combined with symbolic execution. The primary objective of S3CAMX is to demonstrate the feasibility of testing closed loop controller models designed in popular industrial frameworks like Matlab™ and Simulink™ , as well as legacy control systems.

S3CAMX takes in the controller source code in C, a SIM function representing the plant model, and the test description. The test description specifies the initial states and error states (safety property) for the system under test, its sampling period Δ and the time horizon for the test. It also specifies the initial abstraction parameters, a detailed discussion on their description and selection can be found in our previous work [30].

6.1 Controller Description

We assume a controller architecture where the interface to the controller is through a function with signature controller_step(input, output), corresponding to the controller execution on a set of plant outputs. *input* to the controller encapsulates the (a) plant output $\mathbf{y}$, (b) previous controller state s and (c) exogenous controller input $\mathbf{w}$ (disturbances/exogenous inputs). The controller returns a structured *output* consisting of (a) controller's next state and (b) control input (to the plant). This setup is quite generic, and follows a similar format used

```
int controller(){
    static int var;
    var += 1;
    return 0;
}
```

Listing 1: Present

```
int controller(int var){
    int var;
    var += 1;
    return var;
}
```

Listing 2: Absent

Figure 5: Listing with and without persistent variables.

by Embedded Coder™ , Matlab™ 's automatic code generation toolbox to generate C code.

We make a special distinction between the persistent ('global') state variables of the controller and 'local' variables that are *reinitialized* each time the controller is invoked. In typical C software, persistent variables are usually easy to identify as their declaration is qualified by *global* or *static*. In a Simulink™ model, persistent variables are associated with data-store blocks with dedicated read and write blocks to access their values. Given a control software with persistent states, it can be transformed to one without them by converting the persistent variables into function parameters. The above C code snippets in Fig. 5 illustrate the transformation of a function with persistent variables to an equivalent function with only local variables.

6.2 Controller Preprocessing

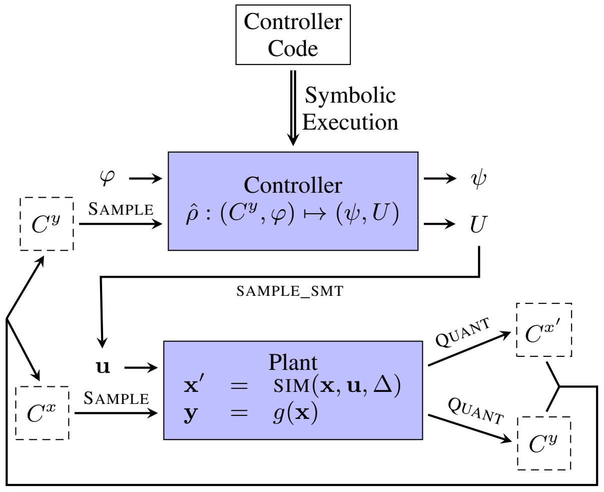
Before beginning the analyses, S3CAMX transforms the controller using a symbolic execution engine to a set of all possible control-flow paths in the controller. The paths are represented by their associated path constraints and update functions. This enumeration is expensive and carried only once for a given controller. Once pre-processed, a controller move involves checking the feasibility of each path constraint and if feasible, computing the valuations of updated state and output. We use Z3 [8], an SMT solver to check feasibility and compute a concrete satisfying assignment on the controller's input $(\mathbf{y}, s)$. The update function is then used to compute $(s', \mathbf{u})$. Multiple satisfying assignments can be obtained by adding blocking constraints.

Pathcrawler [29] is used as the symbolic executor of choice. It can work with constraints over floating points in C programs and uses ECLiPSe [27] as its constraint solver. At the time of implementation, Z3 did not support floating point arithmetic and the theory of reals was used instead. Pathcrawler and Z3 provide for efficient linear constraint solving, but limit the kind of non-linear constraints which can be handled in the controller. Using symbolic execution, we can extend the current implementation to catch program bugs like **division by zero**, **out of bounds access**, and other common programming bugs in the controller. As a remark, we also tried KLEE [3], but unlike Pathcrawler it uses bitvector theories to model C programs. This implies that it can not handle floating point arithmetic, and results in bitvector constraints which are comparatively inefficient to solve.

We note that static generation of *all* controller paths is not always efficient. In practice, controller code base can be quite large and enumerating all paths is an expensive and potentially wasteful process. This can be in part remedied by not doing a plant agnostic path enumeration which can result infeasible paths. Instead, the domain of the inputs to the controller can be constrained appropriately using knowledge about the relevant plant model, thus reducing the number of generated controller paths. A better solution will be a tighter integration of the symbolic execution engine to enable a budget constrained dynamic path exploration using suitable heuristics.

6.3 Plant Description

The current version of the prototype can directly use plants modeled in Python or Matlab™ by using the provided interface. Other frameworks can be easily accommodated by wrapping their respective simulation functionality within a layer of Python. The signature



Controller Code
Symbolic Execution
φ
C^y
SAMPLE
Controller
$\hat{\rho} : (C^y, \varphi) \mapsto (\psi, U)$
ψ
U
SAMPLE_SMT
$\mathbf{u}$
C^x
SAMPLE
Plant
$\mathbf{x}' = \textsc{sim}(\mathbf{x}, \mathbf{u}, \Delta)$
$\mathbf{y} = g(\mathbf{x})$
QUANT
$C^{x'}$
QUANT
C^y

Figure 6: Closed loop symbolic execution.

of the SIM function is of the form $\mathbf{x}' = \textsc{sim}(\mathbf{x}, \mathbf{u}, t)$ with $t, \mathbf{x}, \mathbf{u}, \mathbf{x}'$ as defined. The implementation details of the algorithm used to compute the plant abstraction can be found in our previous work [30].

6.4 Closed Loop Execution

The closed loop analyses involves computing the controller and the plant move in lock step Fig. 6. The controller code is first abstracted using symbolic execution to obtain controller $\hat{\rho}$. Given a set of abstract plant states C^x and inputs $\mathbf{u}$ and controller state φ, the closed loop step proceeds with the plant step. C^x is sampled and along with $\mathbf{u}$ simulated to obtain $\mathbf{x}'$ and $\mathbf{y}$, which are then quantized to get $C^{x'}$ and C^y respectively. Next, the controller step takes in C^y and the given φ to compute ψ and U. Using an SMT solver, we find satisfying assignments to the constraints and in essence, compute representative samples for U as $\mathbf{u}$. The implementation differs from the theory here, where we quantize the samples $\mathbf{u}$ and then sample again to get additional samples. This repeats for n steps, where $n = \lceil \frac{T}{\Delta} \rceil$ and T is the time horizon specified by the given property.

6.5 Constraint Blowup

The symbolic treatment of the controller code has benefits of being precise, but as the case with similar methodologies like bounded model checking, automatic test generation, and more, suffers from 'constraint blowup'. Each controller move requires an 'unrolling' of the controller in time, tracking the associated constraints. The constraints accumulate width each time step and become very inefficient to solve. Even though SMT solvers are quite efficient, large constraints can make the entire analyses unusable. We address this by providing an option of selecting the 'history depth' in S3CAMX. Once the given depth is exceeded, the symbolic controller states are concretized, and the accumulated symbolic constraints (history) are purged. This is done by selecting some representative concrete controller states instead of maintaining complex symbolic representation of all reachable controller states. To clarify, it does not limit the length of the abstract trajectories which can be discovered. For all the presented benchmarks, a depth of 1 was used.

7. EXPERIMENTAL RESULTS

We compare our tool S3CAMX in Table 2 against S-Taliro and our previous approach S3CAM which considers the entire composed sampled data controller system as a black-box. As in our previous work [30], we also provide a reference for the difficulty of falsification using random testing. As before, random testing uses 100,000 simulations on most benchmarks (except for AFC). Every time bounded property was tested to the time horizon specified next

Table 1: Summary of benchmarks. Each benchmark mentions the sampling period of the controller Δ and its description is split into constituent controller and plant. The controller is described by number of *States*, exogenous inputs (*Ex. Ip*), lines of code (*LOC*), symbolic paths in the (*Paths*) and time taken to generate them (*SymEx Time*) in minutes. The plant is described by the language used to implement its model (*Impl.*), number of modes if its a hybrid automaton or the number of blocks if its a Simulink^{TM} model (*Modes/Blocks*), continuous states (*C. States*)

Benchmark	Δ(s)	Controller					Plant		
		States	Ex. Ip	LOC	Paths	SymEx Time	Impl.	Plant Modes/Blocks	C. States
SPI	1	0	1	13	3	0	Python	1	1
Heater	0.2	4	0	59	26	0	Python	3	1
Heat	0.5	3	0	37	312	1	Python	6	3
DC Motor	0.02	1	1	29	3	0	Python	0	2
FuzzyC	0.01	0	0	218	208	236	Python	0	3
MRS	1	0	8	29	9	0	Python	0	4
AFC	0.01	7	0	243	120	40	Simulink	170	12

Table 2: Current tool *S3CAMX*, compared with *S-Taliro* and our previous tool *S3CAM*. All processes were run as single threaded on Ubuntu 12.04, running on an Intel i7-2820QM CPU @2.30GHz with 8GB RAM. **All times are in minutes unless mentioned as seconds(s)**.

Benchmark	Time Horizon	Random Testing		S-Taliro		S3CAM		S3CAMX	
	T (s)	Num. Vio	T	Num. Succ	T_{avg}	Num. Succ	T_{avg}	Num. Succ	T_{avg}
SPI($\mathcal{P}1$)	50	348/100k	17.5	10	0.36	10	0.01	10	0.22
SPI($\mathcal{P}2$)	200	33 /100k	60.9	10	19.59	10	0.03	10	1.31
SPI($\mathcal{P}3$)	500	0 /100k	154.4	0	TO	10	0.08	10	8.39
Heater	2	47 /100k	3.9	10	0.48	10	0.22	10	0.06
Heat	10	156/100k	39.0	10	11.78	10	0.06	10	0.16
DC Motor	1	0 /100k	45.0	0	TO	0	TO	10	0.63
FuzzyC	0.1	6 /100k	10.1	10	0.27	10	0.03	10	1.59
MRS 1	2	0 /100k	0.6	0	TO	0	TO	10	0.18
MRS 2	2	0 /100k	0.4	0	TO	0	TO	10	0.11
MRS 3	2	0 /100k	0.4	0	TO	0	TO	10	0.43
AFC	12	1 /100	238.7	10	0.25	10	16.15	10	13.45

to the benchmark. It should be noted that S3CAMX is a prototype in early stages and optimization and parallelization have been deferred to future releases.

To compare against random testing we used a fixed number of random simulations (ns) and recorded the number of violating (nv) traces. This is mentioned as (nv/ns) along with the time taken to run the simulations. Next to it, is the average time taken for 10 runs by S-Taliro, S3CAM and S3CAMX to find a falsification. This is required due to the randomized nature of all three. The run of the tools is defined as the time to find a falsification, permitting internal restart (multi start strategy is often combined with random search). If a run takes more than 1 hour to finish, we consider it as a time out. If all 10 runs time out, we mention TO as the time taken and 0 as the number of successful runs.

The implementation was benchmarked on several examples mentioned in Table 1, ranging from a simple PI controlled DC Motor to a complex air fuel controller for a powertrain described in Simulink^{TM} . Against each system tested, we mention its sampling period Δ and the controller and plant characteristics. For the former, this includes the number of states, exogenous inputs, lines of code, symbolic controller paths statically discovered by Pathcrawler and the time taken to do so. The plant is characterized by the implementation language for the simulator (Python or Simulink^{TM}), the number of discrete modes if the plant is a hybrid system, otherwise the number of Simulink^{TM} blocks, and lastly, the number of continuous states (plants can also have exogenous inputs [30]. A detailed description for each of the benchmark follows.

7.1 Sampled Polarity Integrator System

The SPI benchmark was used in [11] to highlight the difficulty faced by Markov-chain Monte-Carlo based random testing techniques, and optimization-guided techniques such as RRT-REX and S-TaLiRo. The system has an exogenous input w, and a single continuous state x which after every $\Delta = 1$ seconds gets reset to either -1, 0, or 1 if the input $w < 0$, $w = 0$ or $w > 0$ respectively. We split this system into a plant and a controller, where the controller computes $u = -1, 0, 1$ based upon its exogenous input w as $u = sign(w)$. The continuous state of the plant evolves as $\dot{x} = u$. We then check the three properties $\mathcal{P}1 : x < 20$, $\mathcal{P}2 : x < 50$ and $\mathcal{P}3 : x < 150$ for time horizons 50, 200, and 500 respectively. For all properties S3CAM takes only a few seconds. S3CAMX takes a bit longer due constraint solving, but, S-Taliro takes significantly longer and times out for $\mathcal{P}3$. Similar difficulty is faced by RRT-Rex [11]. This example brings out the difference in our iterative approach when compared to directed random search.

7.2 Heater

The heater system introduced in Sec. 1.1 consists of a room, and a heater controlled by a thermostat. The heater has 3 operating modes; *off*, *regular heating*, and *fast heating*. It can be switched between modes by the thermostat in order to reach and maintain a comfortable temperature in the room. Specifically, the thermostat is designed to sense the room temperature after every $\Delta = 0.2s$ and maintain a temperature of around $70°F$. It also has built-in logic to prevent chattering, i.e., avoiding rapid switching of the heater between modes. The heater is modeled as a hybrid system with linear dynamics, with 1 continuous state and three modes. The thermostat's software controller has 26 control-flow paths with 4 states which keep track of recent mode switchings. The property we seek to falsify is that the room-temperature T_F is always greater than $52°F$. We are able to find the falsification trace, which upon investigation indicates the failure of the chatter-prevention logic in a very narrow range of possible initial settings for the ambient room temperature (approx.) $T_{F0} \in [69.9, 70.0]$. Though all tools find the falsification in less than half a minute, S3CAMX is an order of magnitude faster than both S3CAM and S-Taliro.

7.3 Heat benchmarks

The heat benchmarks were proposed in [12], and describe a scenario where a limited number of heaters h are being used to heat r rooms, where $h < r$. The control system can shuffle the heaters or turn them on/off in order to maintain a comfortable temperature in all rooms. We are interested in finding scenarios where the controller fails and the temperature of any room dips below a certain threshold. We choose the first instance in the suite of heat benchmarks for case study; this instance has 3 rooms and 1 heater. The controller software has 312 control-flow paths and 3 states tracking each room's temperature. Correspondingly, the plant has 3 continuous states and 6 modes. Each mode is characterized by the heater's location and it's discrete state (on/off). We try to falsify the property that temperature of the first room does not drop below $17.23°C$. Interestingly, S3CAMX performs comparably to S3CAM even though the control software has 312 paths. Both tools are faster than S-Taliro and finish in a few seconds, where as S-Taliro requires a few minutes for falsification.

7.4 DC Motor

This example illustrates the search for errors in the presence of controller disturbance. The DC motor is a linear continuous system with armature current i and angular velocity $\dot{\theta}$. It is controlled by PI controller with saturation which results in 3 control-flow paths. The bounded additive disturbance in the controller induces error in the sensed plant outputs. We parameterize the disturbance as a piecewise constant signal. We wish the system to never enter the following region of the state-space: $i \in [1, 1.2], \dot{\theta} \in [10, 11]$. We choose this set by design; it is designed to be very hard to reach using random simulations. S3CAMX can, however, find a falsification, demonstrating the effectiveness of symbolic execution in finding a sequence of inputs that lead to a violation. In comparison, S-Taliro and our previous technique S3CAM fail to find a violation.

7.5 Fuzzy Control of Inverted Pendulum

Rule based controllers, such as ones implemented using fuzzy logic, are an interesting challenge for symbolic execution based analyses as they typically have a large number of control-flow paths. We consider an example from [25], where the controller tries to stabilize a nonlinear inverted pendulum balanced on a cart using a 5X5 rule matrix. The `C` code [4] has 218 lines describing 208 control-flow paths. The controller is stateless and computes the actuation force by classifying the current plant state (θ and $\dot{\theta}$) and selecting a corresponding control output from a lookup table. The safety property defines bounds on states, which when exceeded, indicate undesirable transients or possible unstable behaviors. S-Taliro finds a falsifying trajectory faster than S3CAMX, but not as fast as S3CAM. This result is nevertheless encouraging as it shows the ability of S3CAMX to analyze a large number of control-flow paths and yet be successful at finding a falsifying trajectory.

7.6 Mode-Specific Reference Selection (MRS)

These are a set of 3 benchmarks from [9, 11]. The benchmarks represent distinctive features from proprietary models of automotive controllers, and they highlight issues faced by optimization-guided methods. The systems have simple nonlinear dynamics but complex combinatorial Boolean logic over 8 exogenous inputs. To make the system amenable to S3CAMX, we split the discrete nonlinear dynamics into a plant, and the rest (linear combinatorial logic) becomes the controller. The controller remains the same across the 3 benchmarks, but the plant varies. The mode selection logic is now part of the controller which takes in 8 inputs $w_1, \ldots, w_8$, where each $w_i \in [0, 100]$ and computes u. The 3 plants have discrete time nonlinear dynamics where the evolution is governed by $x^+ = f(x, u)$, where $f_i(x, u)$ comprises of polynomials of up to degree 3 and trigonometric functions in x. The falsification can only be found in the mode which is triggered when $\bigwedge_{i\in[1..4]} w_{2i}(t) > 90 \wedge w_{2i-1}(t) < 10$. The probability of triggering the mode is thus a meager 10^{-8}. Such a combinatorial search is not amenable to sampling-based searches, and both S-Taliro and S3CAM fail. On the other hand, S3CAMX can falsify the property in under a minute.

[4]http://www2.ece.ohio-state.edu/~passino/fuzzycontrol.html

7.7 Powertrain Control Benchmark

We use the most complex version of the abstract fuel control system benchmark (AFC) presented in [16]. It represents a complex closed loop control system modeling a plant with hybrid dynamics controlled by a PI controller. The original benchmark has both the plant and the controller implemented in Simulink™ . To test it using our approach, we use Embedded Coder™ to generate `C` code for the controller block, which is then hand-tuned to satisfy controller interface requirements. Due to the observability requirement discussed in Sec. 6, the plant model is modified by simplifying the variable transport delay block to a first order filter. The property checked is a modified form of the 'Worst-case excursions in the normal mode' property in [16], i.e., we try to falsify $\mu >= 0.02$ while $t \in [0, 1.0]$. The search is over the original parameters: pedal frequency, pedal amplitude and engine speed. Both S3CAMX and S3CAM take longer than S-Taliro to find the falsification due to the inefficient Matlab™ interface, as explained below.

In summary, symbolic execution being an expensive operation, increases the time taken by S3CAMX to find a falsification when compared with S3CAM. However, it also enables it to find falsifications where S3CAM completely fails. Specifically, in cases where the control program has non-robust paths and corner cases (hard to cover behaviors using a uniform random distribution) which lead to the violation, S3CAMX should perform better than a purely sampling based search like S3CAM. This is due to the discussed exhaustive exploration by symbolic execution of all possible control paths for a given set of plant states. This is exemplified by the DC Motor and MRS benchmarks. On the other hand, when the above is not the case, and controller paths leading to the violation can be exercised using uniform random sampling alone. This is evident by the benchmarks: SPI, Heat and FuzzyC.

S3CAM fares better than S-Taliro in all benchmarks except AFC (which involves a plant model in Simulink™ /Matlab™). This can be partly attributed to the inefficient interface between S3CAM/S3CAMX and Matlab™ – our tool has a few layers of communication and indirection to get the Python implementation to communicate with the simulation infrastructure in Simulink™ /Matlab™ . Part of the inefficiency can be attributed to the Simulink™ SIM function. Each call to SIMin Simulink™ has a setup time required for model preprocessing and compilation; this makes repeated simulations very expensive. This can be clearly seen in the amount of time taken for 100 random simulations of the AFC model. The latter issue has been recently addressed in a newer version of Simulink™ (R2015b) through the introduction of a feature *fast restart* which reduces the setup time to some extent.

8. CONCLUSIONS

In conclusion we presented a technique to falsify properties of control systems described by implementations of control software and models of physical systems (plant). A combination of *numerical simulations* to explore the implicit plant abstraction and *symbolic execution* to maximize path coverage on control code was used to find abstract error traces. These were then iteratively concretized to yield reproducible error traces.

Results on benchmarks, ranging from the simple to the complex ones implemented in Simulink^{TM} were presented and compared with random testing, S-Taliro, and our previous, purely numerical technique S3CAM. We successfully demonstrated the effectiveness of symbolic execution in detecting corner cases in controller code. The implementation along with the benchmarks is made public at https://github.com/zutshi/S3CAMX.

Acknowledgements

We thank James Kapinski for countless useful discussions which helped shape this work. Zutshi and Sankaranarayanan were supported, in part, by the US National Science Foundation(NSF) under award number CNS-1319457 and, in part, by Toyota Engineering and Manufacturing North America(TEMA). All opinions are those of the authors and not necessarily of the NSF or TEMA.

9. REFERENCES

[1] Y. Annapureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. *Proc. TACAS*, pages 254–257, 2011.

[2] A. Bhatia and E. Frazzoli. Incremental search methods for reachability analysis of continuous and hybrid systems. *Proc. of HSCC*, pages 451–471, 2004.

[3] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[4] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071. ACM, 2011.

[5] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[6] E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *International Journal of Foundations of Computer Science*, 14(04):583–604, 2003.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[8] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[9] J. Deshmukh, X. Jin, J. Kapinski, and O. Maler. Stochastic local search for falsification of hybrid systems. In *Automated Technology for Verification and Analysis*, pages 500–517. Springer, 2015.

[10] A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Proc. CAV*, pages 167–170, 2010.

[11] T. Dreossi, T. Dang, A. Donzé, J. Kapinski, X. Jin, and J. V. Deshmukh. Efficient guiding strategies for testing of temporal properties of hybrid systems. In *NASA Formal Methods*, pages 127–142. Springer, 2015.

[12] A. Fehnker and F. Ivanĉić. Benchmarks for hybrid systems verification. In *Proc. of HSCC*, volume 2993, pages 326–341, 2004.

[13] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 412–416. IEEE, 2001.

[14] K. Havelund and G. Roşu. Monitoring programs using rewriting. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 135–143. IEEE, 2001.

[15] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356. Springer, 2002.

[16] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Powertrain control verification benchmark. In *Proceedings of the 17th international conference on Hybrid systems: computation and control*, pages 253–262. ACM, 2014.

[17] J. Kim, J. M. Esposito, and V. Kumar. An RRT-based algorithm for testing and validating multi-robot controllers. Technical report, DTIC Document, 2005.

[18] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[19] S. M. LaValle. Rapidly-exploring random trees a new tool for path planning. Technical Report TR 98-11, Computer Science Dept., Iowa State University, Ames, Iowa, 1998.

[20] F. Lerda, J. Kapinski, E. Clarke, and B. Krogh. Verification of supervisory control software using state proximity and merging. *Hybrid Systems: Computation and Control*, pages 344–357, 2008.

[21] F. Lerda, J. Kapinski, H. Maka, E. M. Clarke, and B. H. Krogh. Model checking in-the-loop: Finding counterexamples by systematic simulation. In *American Control Conference, 2008*, pages 2734–2740. IEEE, 2008.

[22] R. Majumdar, I. Saha, K. Shashidhar, and Z. Wang. Clse: Closed-loop symbolic execution. In *NASA Formal Methods*, pages 356–370. Springer, 2012.

[23] T. Nahhal and T. Dang. Test coverage for continuous and hybrid systems. In *Computer Aided Verification*, pages 449–462, 2007.

[24] T. Nghiem, S. Sankaranarayanan, G. Fainekos, F. Ivancić, A. Gupta, and G. J. Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, pages 211–220. ACM, 2010.

[25] K. M. Passino, S. Yurkovich, and M. Reinfrank. *Fuzzy control*, volume 42. Citeseer, 1998.

[26] J. Robinson. *The collected works of Julia Robinson*, volume 6. American Mathematical Soc., 1996.

[27] J. Schimpf and K. Shen. Ecl i ps e–from lp to clp. *Theory and Practice of Logic Programming*, 12(1-2):127–156, 2012.

[28] K. Sen, D. Marinov, and G. Agha. *CUTE: a concolic unit testing engine for C*, volume 30. ACM, 2005.

[29] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing-EDCC 5*, pages 281–292. Springer, 2005.

[30] A. Zutshi, S. Sankaranarayanan, J. V. Deshmukh, and J. Kapinski. Multiple shooting, cegar-based falsification for hybrid systems. In *Proceedings of the 14th International Conference on Embedded Software*, page 5. ACM, 2014.