CLSE: Closed-Loop Symbolic Execution

Rupak Majumdar[1,2], Indranil Saha[1], K. C. Shashidhar[2] and Zilong Wang[2]

[1] University of California Los Angeles
[2] Max-Planck Institute for Software Systems
{rupak, indranil}@cs.ucla.edu, {shashi, zilong}@mpi-sws.org

Abstract. We present CLSE, a *closed-loop* symbolic execution engine for control system implementations. CLSE takes as input the description of a physical plant represented by a system of linear ordinary differential equations, the software implementation and execution frequency for a discrete-time controller that senses and actuates the plant, and a time horizon, and symbolically executes the closed-loop system —the combination of the plant and the controller— up to the time horizon. The execution helps capture the bounded-time dynamics of the system in terms of the finite sequences of the plant's sampled state-sets and symbolic control inputs. We show the use of CLSE in symbolic execution of a set of control systems benchmarks. Using the symbolic execution engine, we also build a robustness analysis tool which computes the maximum deviation of the states of the plant due to measurement uncertainties in the controller up to the time horizon.

1 Introduction

Software controllers for physical systems are at the core of many safety-critical systems. The combination of physical behavior, given by the dynamics of continuous state variables, and discrete behavior, implemented in software, makes the end-to-end behavior of these systems hard to design and to reason about.

The need for effective analysis techniques for cyber-physical systems combining physical plants and software controllers has been long recognized. Most current techniques, though, take one of two approaches. In the first approach, the system is modeled as a hybrid automaton [1, 19, 8, 23] —a finite-state machine that is endowed with dynamics over continuous variables— and symbolic reachability analysis is performed on this model. This captures the intended semantics of the plant but usually the software-based controller is "abstracted away" to a mathematical function that is typically modeled as a second automaton running in parallel with the plant. While recent progress in reachability analysis for hybrid automata [14] have shown the potential for symbolic techniques to scale to systems with many continuous variables, the simplistic modeling of the controller, in particular, the omission of programming-language level features whose interaction with the physical system often leads to errors, makes it hard to provide any guarantees for the *implementation* of the feedback control system.

In the second approach, techniques used in analysis of programs, based on abstract interpretation, precisely model features of the controller program, but

usually "abstracts away" the plant's dynamics, assuming that the sensors can read arbitrary values from their range in each cycle. While tools like Astrée [4] and Fluctuat [17] have been successful in proving various safety properties of controller programs, such as the absence of arithmetic or buffer overflows, the absence of a plant model makes it hard to verify properties of the entire feedback control system that depend on the interaction between the plant and the controller.

In order to address the drawbacks in these two approaches, it is clear that analysis techniques need to model the full *closed-loop* control system —both the plant and the controller code— in analyzing embedded control software in cyber-physical systems. This need has been expressed before [7, 16, 3, 11], and some tools to perform closed-loop *simulation* of feedback control systems have been developed recently (cf. [2, 24, 27]).

We present CLSE, a symbolic execution engine for feedback control systems. CLSE takes as input the description of a feedback control system in two parts: a plant model given as a set of linear ordinary differential equations, and a software implementation of a controller for the plant. For a given time horizon and a sampling rate for the controller, and a given set of initial states of interest for the plant, CLSE performs symbolic simulation of the plant and the controller up to the time horizon. The simulation is guaranteed to provide a complete coverage of the initial state set, that is, the bounded-time evolution of the system starting from *any* state in the initial state set is included in the simulation. The symbolic analyses of the controller uses concolic execution techniques (cf. [15, 29]), together with decision procedures for non-linear arithmetic [20]. We have implemented CLSE for controller implementations in the C language and describe its application to examples of closed-loop control systems.

We also show how the symbolic execution engine of CLSE can be used to build additional analyzers for closed-loop control systems. We develop a symbolic robustness analyzer on top of CLSE. In addition to the closed-loop system and the time horizon, the robustness analyzer takes as input a bound on the disturbance on sensor measurements, and computes the maximum deviation between the plant state without disturbance and the plant state with disturbance up to the time horizon.

Outline of the paper. We first describe the class of closed-loop systems we address in our work in Section 2. CLSE's closed-loop analysis algorithm is then presented in Section 3. In Section 4 we show how such an analysis can be used to realize a robustness analyzer. Experimental evaluation of the analyses is provided on a few example closed-loop control systems in Section 5. We conclude the paper after a discussion of related work in Section 6.

2 Closed-Loop System Model

We consider the standard model of a closed-loop control system that is composed of a plant and a controller that are connected via sensors and actuators (see Fig. 1). The plant captures the continuous dynamics of the environment

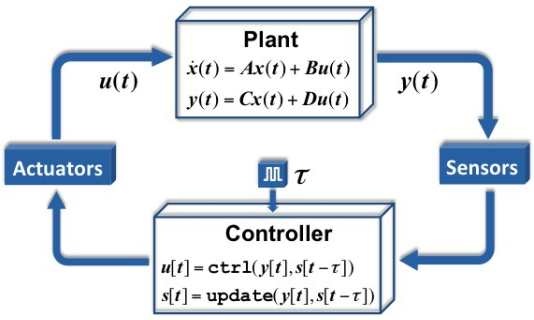


Fig. 1. Model of a closed-loop control system.

that is being controlled and the controller represents the software program that implements the control algorithm.

In an execution of the closed-loop system, the state of the plant is sensed by the controller at discrete time instants, called sampling times, based on which the control inputs to the plant are computed. Our analysis performs a symbolic simulation of the system up to a bounded-time horizon, $T = N \times \tau$, where $N \in \mathbb{N}$ and τ is the sampling period of the system. In what follows, we characterize the dynamics of the two main components in detail.

2.1 Plant Dynamics

We consider linear dynamical systems

$$\dot{x}(t) = Ax(t) + Bu(t), \quad x(0) = x_0 \tag{1}$$

where $x(t) \in \mathbb{R}^n$ is the continuous state vector, A and B are $n \times n$ and $n \times m$ matrices, respectively, and the *control input* $u(t)$ is a piecewise continuous function from $\mathbb{R}^+$ to a convex, compact set $U \subseteq \mathbb{R}^m$ of control actions. A function $\xi : (0, \infty) \to \mathbb{R}^n$ is said to be a *trajectory* of the dynamical system if $\xi(0) = x_0$ and there exists a control input υ such that $\dot{\xi}(t) = A\xi(t) + B\upsilon(t)$ for almost all $t \in (0, \infty)$.

In a *sampled-data* control system, there is an a priori fixed sampling time $\tau \in \mathbb{R}^+$. The plant state is sensed at the end of each sampling period of τ time units. The control inputs are computed based on the plant state and applied to the plant at the beginning of the next sampling period. The control input remains constant throughout the next sampling period. Thus, control inputs in a sampled-data control system are piecewise constant curves of duration τ, i.e., a control input $\upsilon : \mathbb{R}^+ \to U$ satisfies $\upsilon(t) = \upsilon((k-1)\tau)$ for all $t \in [(k-1)\tau, k\tau)$ and $k \in \mathbb{N}$. For a sampled-data control system $\dot{x}(t) = Ax(t) + Bu(t)$ with sampling time τ and control actions U, and a given set of initial states X_0, we define $\mathcal{X}(X_0, U, Ax(t) + Bu(t), \tau)$ to be the set of all possible trajectories starting from some state in X_0.

2.2 Controller Dynamics

We assume that the controller of the closed-loop system is implemented as a software program. The controller program may have state variables that retain their values at the end of an execution to be used in the next execution. We model controller software in a simple imperative language. In our implementation we handle more general features such as pointers, arrays, and function calls.

We represent a controller program as a *control flow graph (CFG)* $P = (Y, \mathcal{L}, \ell_i, \ell_o, \mathsf{op}, E)$ consisting of (1) a set of variables Y, containing disjoint subsets Y_0 of *input* variables, Y_s of *state* variables, and Y_u of *output* variables, (2) a set of control locations (or program counters) $\mathcal{L}$ which includes a special start location $\ell_i \in \mathcal{L}$ and an output location $\ell_o \in \mathcal{L}$, (3) a function op labeling each location $\ell \in \mathcal{L}$ with one of the following basic operations:

- an assignment $y := e$, where $y \in Y$ and e is an arithmetic expression over Y, and
- a conditional **if** (e) **then** ℓ' **else** ℓ'', where e is a side-effect free expression and ℓ', ℓ'' are locations in $\mathcal{L}$

and (4) a set of directed edges $E \subseteq \mathcal{L} \times \mathcal{L}$ defined as follows. The set of edges E is the smallest set such that (i) every node ℓ where $\mathsf{op}(\ell)$ is an assignment statement has exactly one node ℓ' with $(\ell, \ell') \in E$, (ii) every node ℓ such that $\mathsf{op}(\ell)$ is **if** (e) **then** ℓ' **else** ℓ'' has two edges (ℓ, ℓ') and (ℓ, ℓ'') in E. For a location $\ell \in \mathcal{L}$ where $\mathsf{op}(\ell)$ is an assignment operation, we write $N(\ell)$ for its unique neighbor. Thus, the locations of a CFG correspond to program locations with associated commands, and edges correspond to control flow from one operation to the next. The program ends on reaching the location ℓ_o and outputs the values for all variables $u \in Y_u$. A *path* is a sequence of locations $\ell^1, \ell^2, \ldots, \ell^n$ in the CFG. A location $\ell \in \mathcal{L}$ is reachable from $\ell' \in \mathcal{L}$ if there is a path $\ell', \ldots, \ell$ in the CFG. We assume that every node in $\mathcal{L}$ is reachable from ℓ_i and ℓ_o is reachable from every node. Note that, even though we do not include a loop construct here, our implementation handles static control loops via unrolling.

The concrete semantics of the program is given using a *memory* that maps variables in Y to values. For a memory M, we write $M[y \mapsto v]$ for the memory mapping y to v and every other variable $z \in Y \setminus \{y\}$ to $M(z)$. For an expression e, we denote by $M(e)$ the value obtained by evaluating e where each variable y occurring in e is replaced by the value $M(y)$.

Execution starts from a memory M_0 containing initial values for input variables in Y_0, final values for the state variables in Y_s from the execution on the previously sampled plant output and constant default values for variables in Y_u, at the entry location ℓ_i. When the program runs for the first time, the values of the variables in Y_s are equal to their initial values. Each operation updates the memory and the control location. Suppose the current location is ℓ and the current memory is M. If $\mathsf{op}(\ell)$ is $y := e$, then the new location is $N(\ell)$ and the new memory is $M[y \mapsto M(e)]$. If $\mathsf{op}(\ell)$ is **if** (e) **then** ℓ' **else** ℓ'' then e is evaluated based on the current memory M. If the evaluated value is 0, then the new location is ℓ'', otherwise the new location is ℓ'. In either case, the memory

remains unchanged. On reaching ℓ_o, the program terminates and outputs the values $M(v)$ for each $v \in Y_u$. Execution of the program starting from a memory M_0 defines a path in the CFG in a natural way. A path is executable if it is the path corresponding to program execution from some initial memory M_0.

3 Closed-Loop Symbolic Execution

In this section, we first discuss symbolic execution of the plant and concolic execution of the control program individually, and then discuss how their combination is handled by CLSE.

3.1 Symbolic Execution of the Plant

In sampled-data control systems, the evolution of the plant state over time can be viewed as a discrete-time sequence of sets $X_0, X_1, \ldots$, where X_0 is the set of initial states and X_i denotes the set of states reached in the ith sampling time (i.e., $t = i\tau$). Computation of the set X_i depends on the set of states X_{i-1} of the plant at the preceding sampling time, the continuous equations governing the dynamics of the plant, and the set of control inputs actuating the plant.

Suppose we are given a set X_k of states of a plant, corresponding to the sampling time $t = k\tau$, and suppose we let the plant evolve due to its continuous dynamics for one sampling period to reach the set X_{k+1}. For the given set of states X_k, the set of control inputs U_k, and the dynamics of the plant $\dot{x} = Ax(t) + Bu(t)$, we define $\texttt{Reach}(X_k, U_k, Ax(t) + Bu(t), \tau)$, the set of states X_{k+1} that the plant can be in after the elapse of a time period τ, as follows:

$$\begin{aligned} X_{k+1} &= \texttt{Reach}(X_k, U_k, Ax(t) + Bu(t), \tau) \\ &= \{\xi(\tau) \mid \xi \in \mathcal{X}(X_k, U_k, Ax(t) + Bu(t), \tau)\}. \end{aligned}$$

The set X_{k+1} may be hard to compute and represent exactly. Thus, in practice, we approximate X_{k+1}.

We have tried two techniques in CLSE. First, using symbolic reachability techniques for linear systems for a bounded time interval (cf. [23]), we can get arbitrarily precise approximations but at high computation costs. Second, for relatively coarse, but computationally practical approximations, we used continuous-time to discrete-time model conversion based on the *zero-order hold* method [12]. Since the control inputs in our system model are piecewise constant over the sampling period, for a given continuous time dynamics of the plant $\dot{x} = Ax(t)+Bu(t)$, and a sampling period, the method provides the discrete-time dynamics of the plant defined by $x[k+1] = A_d x[k] + B_d u[k]$.

3.2 Concolic Execution of the Controller

For closed-loop analysis of a control system we need to have the symbolic outputs and symbolic path constraints for all possible executable paths in the controller

program. We obtain them by analyzing the controller program via *concolic execution*, in which the program is executed on symbolic inputs in addition to concrete inputs [15, 29]. The concolic execution algorithm executes the program while maintaining two additional artifacts: a *symbolic memory* μ which maps variables in Y to symbolic expressions over a set of symbolic constants, and a *path constraint* κ, which collects predicates over symbolic constants along the execution path. The symbolic memory map and the symbolic path constraint are updated during the course of execution.

Concolic execution proceeds as follows. Starting at location ℓ_i in the control flow graph, the symbolic memory μ maps each input variable $y \in (Y_0 \cup Y_s)$ to a fresh symbolic constant α_y and each variable $z \in Y \setminus (Y_0 \cup Y_s)$ to some default constant value. Initially, the path constraint is *true*.

For an assignment $y := e$ at location ℓ, the symbolic memory μ is updated to $\mu[y \mapsto \mu(e)]$, where $\mu(e)$ denotes the symbolic expression obtained by evaluating e using the map μ. The path constraint is unchanged. The control location is updated to $N(\ell)$. For a conditional **if** (e) **then** ℓ' **else** ℓ'' at location ℓ, if none of these branches has been explored with a path that agrees with the current path up to location ℓ, a branch is arbitrarily chosen, otherwise the branch which has not been explored is taken. Based on which branch is chosen, the control location is updated to either ℓ' or ℓ''. If the new control location is ℓ', the path constraint is updated to $\kappa \wedge \mu(e) \neq 0$, and if the new control location is ℓ'', the path constraint is updated to $\kappa \wedge \mu(e) = 0$. In each case, the new symbolic memory is still μ. Execution ends when the control location is ℓ_o. At this point, κ is the path constraint, and the restriction $\mu|_{Y_u \cup Y_s}$ maps each $y \in Y_u \cup Y_s$ to $\mu(y)$. We denote $\mu|_{Y_u \cup Y_s}$ by λ.

At the end of an execution, a new execution is created by selecting a conditional ℓ : **if** (e) **then** ℓ' **else** ℓ'' along the path that was executed such that (1) the current execution took the **then** (respectively, **else**) branch of the conditional, and (2) the path that agrees with the current execution up to ℓ but then takes the **else** (respectively, **then**) branch of this conditional has not been explored before. In this way, each control path in the program is explored. At the end of symbolic execution, we get the set $controllerPaths$ of tuples $\langle \kappa, \lambda \rangle$ of path constraints and output maps for each explored path.

3.3 Combining the Two

Our closed-loop execution is based on the interaction of the symbolic execution of the plant and concolic execution of the controller implemented in software. A single iteration in our execution begins at the point the plant's state is sensed and ends after the plant evolves for one sampling period based on the controller's actions on the sensed data. We make the usual assumption that the time taken by the controller is negligible when compared to the chosen sampling period.

The algorithm for closed-loop execution is outlined in `closedLoopExecution` function in Algorithm 1. It takes the following as inputs: (1) t, the time instant at the current sampling (an integer multiple of τ), (2) X_t, the set of states of the plant at time instant t, (3) $S_{t-\tau}$, the set of states of the controller at time

Algorithm 1: Closed-loop execution of a system.

Input: A closed-loop system with the dynamics of the plant captured as *flow* and the controller captured in *controllerPaths*, sampling time τ, simulation time bound T, initial states of the plant X_0 and initial states of the controller $S_{-\tau}$.

Output: Reach set sequences and path sequences that cover the initial states up to time T.

function `closedLoopExecution`$(t, X_t, S_{t-\tau}, reachSetSeq, pathSeq)$
begin
  $intersectingPaths \leftarrow \{\langle\kappa,\lambda\rangle | \langle\kappa,\lambda\rangle \in controllerPaths, (X_t \wedge S_{t-\tau} \wedge \kappa)$ is satisfiable$\}$
  foreach $\langle\kappa,\lambda\rangle \in intersectingPaths$ **do**
    $X_{\text{init}} \leftarrow \exists Y_s, X_t(Y_0) \wedge S_{t-\tau}(Y_0, Y_s) \wedge \kappa(Y_0, Y_s)$
    $U_t \leftarrow \{u | u = \lambda(y)(x,s), y \in Y_u, x \in X_t, s \in S_{t-\tau}\}$
    $S_t \leftarrow \{s' | s' = \lambda(y)(x,s), y \in Y_s, x \in X_t, s \in S_{t-\tau}\}$
    $X_{t+\tau} \leftarrow$ `Reach`$(X_{\text{init}}, U_t, flow, \tau)$
    $reachSetSeq' \leftarrow$ `append`$(reachSetSeq, X_{t+\tau})$
    $pathSeq' \leftarrow$ `append`$(pathSeq, \langle\kappa,\lambda\rangle)$
    if $t + \tau < T$ **then**
      `closedLoopExecution`$(t+\tau, X_{t+\tau}, S_t, reachSetSeq', pathSeq')$
    else
      $reachSetSequences \leftarrow reachSetSequences \cup \{reachSetSeq'\}$
      $pathSequences \leftarrow pathSequences \cup \{pathSeq'\}$
begin
  global $reachSetSequences \leftarrow \emptyset$
  global $pathSequences \leftarrow \emptyset$
  `closedLoopExecution`$(0, X_0, S_{-\tau}, [X_0], [\,])$
  return $(reachSetSequences, pathSequences)$

instant $t - \tau$, (4) *reachSetSeq*, the sequence of plant's state sets $X_0, X_\tau, \ldots, X_t$ reached in the current execution at successive sampling instants until time t, and (5) *pathSeq*, the sequence of controller paths traversed in the current execution. Initially, `closedLoopExecution` function is called with $t = 0$, $X_t = X_0$, where X_0 is the set of initial states of the plant, $S_{t-\tau} = S_{-\tau}$, where $S_{-\tau}$ is the set of initial states of the controller, $reachSetSeq = [X_0]$ and $pathSequences = [\,]$.

At any time instant t, the `closedLoopExecution` function executes in the following manner. The algorithm first identifies *intersectingPaths*, a subset of paths in the controller program (*controllerPaths*), that can be executed starting from some state in X_t and $S_{t-\tau}$. This is computed by checking the satisfiability of $X_t \wedge S_{t-\tau} \wedge \kappa$, for each $path = \langle\kappa,\lambda\rangle \in controllerPaths$. Satisfiability implies that there exists a state with $x \in X_t$ and $s \in S_{t-\tau}$ that can execute the controller program along *path*. For each controller path, $path = \langle\kappa,\lambda\rangle$, we first compute three sets: X_{init}, U_t and S_t. The set X_{init} denotes the set of initial states of the plant for the evolution in the next sampling period when *path* is executed in the controller program due to the current sampled states of the plant. The sets U_t and S_t denote the next set of control inputs to the plant and the set

Algorithm 2: Robustness analysis of a closed-loop system.

Input: A closed-loop system with the dynamics of the plant captured as *flow* and the controller captured in *controllerPaths*, sampling time τ, simulation time bound T, initial states of the plant X_0 and initial states of the controller $S_{-\tau}$ and an upper-bound ε on the sensor errors.

Output: An upper-bound Δ on the deviation in the plant's states after time T.

function `computeDeviation`$(t, X_t, S_{t-\tau}, X'_t, S'_{t-\tau}, \delta_t)$

begin

$X_t^\varepsilon \leftarrow X'_t \oplus \mathcal{E}$

$intersectingPaths \leftarrow \{\langle\kappa,\lambda\rangle | \langle\kappa,\lambda\rangle \in controllerPaths, (X_t \wedge S_{t-\tau} \wedge \kappa) \text{ is satisfiable}\}$

$intersectingPaths^\varepsilon \leftarrow \{\langle\kappa^\varepsilon,\lambda^\varepsilon\rangle |$
$\langle\kappa^\varepsilon,\lambda^\varepsilon\rangle \in controllerPaths, (X_t^\varepsilon \wedge S'_{t-\tau} \wedge \kappa^\varepsilon) \text{ is satisfiable}\}$

foreach $\langle\kappa,\lambda\rangle \in intersectingPaths$ **do**

$X_{\text{init}} \leftarrow \exists Y_s, X_t(Y_0) \wedge S_{t-\tau}(Y_0, Y_s) \wedge \kappa(Y_0, Y_s)$

$U_t \leftarrow \{u | u = \lambda(y)(x,s), y \in Y_u, x \in X_t, s \in S_{t-\tau}\}$

$S_t \leftarrow \{s' | s' = \lambda(y)(x,s), y \in Y_s, x \in X_t, s \in S_{t-\tau}\}$

$X_{t+\tau} \leftarrow$ `Reach`$(X_{\text{init}}, U_t, flow, \tau)$

foreach $\langle\kappa^\varepsilon,\lambda^\varepsilon\rangle \in intersectingPaths^\varepsilon$ **do**

$X^\varepsilon_{\text{init}} \leftarrow \exists Y_s, X'_t(Y_0) \wedge S'_{t-\tau}(Y_0, Y_s) \wedge (\kappa(Y_0, Y_s) \oplus \mathcal{E})$

$U_t^\varepsilon \leftarrow \{u' | u' = \lambda^\varepsilon(y)(x,s), y \in Y_u, x \in X'_t, s \in S'_{t-\tau}\}$

$S'_t \leftarrow \{s'' | s'' = \lambda^\varepsilon(y)(x,s), y \in Y_s, x \in X'_t, s \in S'_{t-\tau}\}$

$X'_{t+\tau} \leftarrow$ `Reach`$(X^\varepsilon_{\text{init}}, U_t^\varepsilon, flow, \tau)$

$\delta_{t+\tau} \leftarrow$ `findOutputDeviation`$(X_t, X'_t, X_{t+\tau}, X'_{t+\tau}, \langle\kappa,\lambda\rangle, \langle\kappa^\varepsilon,\lambda^\varepsilon\rangle, \delta_t)$

if $t + \tau < T$ **then**

`computeDeviation`$(t+\tau, X_{t+\tau}, S_t, X'_{t+\tau}, S'_t, \delta_{t+\tau})$

else

$\Delta \leftarrow \mathbf{max}(\Delta, \delta_{t+\tau})$

begin

global $\Delta \leftarrow \mathbf{0}$

`computeDeviation`$(0, X_0, S_{-\tau}, X_0, S_{-\tau}, \mathbf{0})$

return Δ

of controller states at time t, respectively. Now we let time evolve for τ units to reach sampling time $t + \tau$ and obtain the set $X_{t+\tau}$ of the plant's states by applying `Reach` to X_{init} and U_t. The set $X_{t+\tau}$ thus computed is appended to the current sequence of reach sets $reachSetSeq'$, and the current controller path, *path*, is appended to the current sequence of paths $pathSeq'$.

If $t + \tau$ is less than the time horizon $T = N\tau$, we repeat the algorithm recursively, otherwise, the current sequence of reach sets $reachSetSeq'$ is added to the set *reachSetSequences* of all state set sequences and the current sequence of paths *pathSeq* is added to the set *pathSequences* of all path sequences.

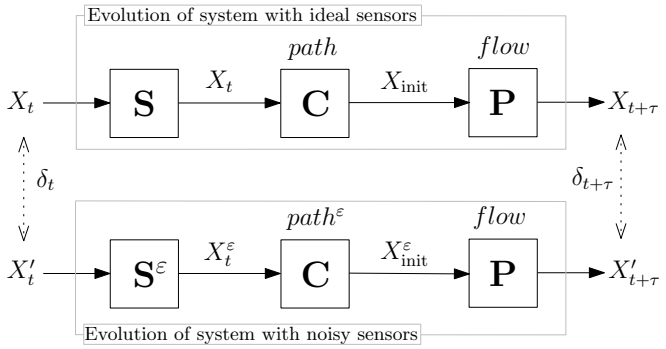


Fig. 2. Snapshot of an iteration instance in the computation of $\delta_{t+\tau}$.

4 Closed-Loop Robustness Analysis

We now describe a closed-loop robustness analyzer based on CLSE. Given a closed-loop control system, the robustness analyzer carries out two symbolic executions over plant's state variables: (1) the *reference evolution* in which all sensors are ideal so that they do not introduce any numerical errors in the inputs to the controller; and (2) the *perturbed evolution* in which all sensors can possibly introduce errors that are bounded by ε. The objective is to compute the maximum deviation Δ in the state of the plant due to sensor errors for a given set of initial states up to a bounded time horizon T (which is a multiple of τ).

The computation of the deviation Δ is outlined in Algorithm 2. Figure 2 shows one step of the computation. On the top, we perform closed loop symbolic execution on the plant and controller as described in the previous section. At the bottom, we again perform closed loop symbolic execution, but assume that instead of reading X'_t for the set of states, there is some sensor error that adds additional "noise" to the measurement, which leads to execution of an erroneous path in the controller, $path^\varepsilon$, instead of *path*. Hence, instead of continuing the simulation with X'_t, we continue with X^ε_t defined as follows.

We assume a simple sensor error model where each variable in the plant state x is sensed by a sensor with an error bound of ε. Then the bounds on the perturbations due to individual sensors define an *error box* $\mathcal{E} = [-\varepsilon, +\varepsilon]^n$. The perturbed set X^ε_t is defined by the Minkowski sum of X'_t and the error box, that is, $X^\varepsilon_t = X'_t \oplus \mathcal{E}$.

The closed-loop symbolic execution proceeds as previously described for the reference and perturbed evolutions by identifying the intersecting paths based on the sets X_t and X^ε_t, respectively. For each pair of intersecting paths from the two evolutions, the deviation in the next plant states, $\delta_{t+\tau}$, between $X_{t+\tau}$ and $X'_{t+\tau}$, is computed based on the deviation δ_t from the previous iteration. If $t + \tau < T$, we repeat the algorithm recursively. We now describe the different operations used in the above computation.

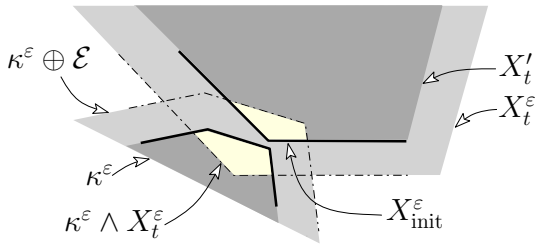
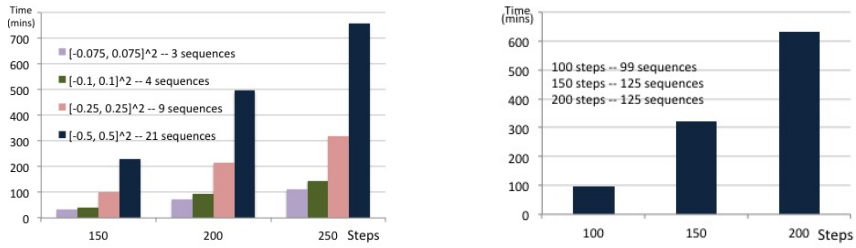


Fig. 3. Computation of the set $X^\varepsilon_{\text{init}}$.

Updating the initial sets. In each iteration of plant's evolution, we need to provide the initial set that is a subset of the reach set of the plant in the previous iteration. For the reference evolution, the controller senses the set X_t and the initial set is computed as described previously in Section 3.3 for *path*, the current path of the controller. For the perturbed evolution, however, the controller senses the set X^ε_t that is obtained by adding the effect of sensor noise $\mathcal{E}$ to the plant's reach state X'_t. Computing the initial set of states for the perturbed evolution has to take into account the fact that, $path^\varepsilon$, the current path of the controller is due to erroneously sensed states of the plant. Therefore, we need to identify the subset of states in X^ε_t, which when perturbed lead to the selection of $path^\varepsilon$, as follows. We first bloat the set κ^ε with $\mathcal{E}$ to obtain the set $\kappa^\varepsilon \oplus \mathcal{E}$. The intersection of this set with the reach set of the plant X'_t gives the set $X^\varepsilon_{\text{init}}$, which is the initial set for the next execution of the plant for the perturbed evolution. The different sets described above are illustrated in Fig. 3.

Finding deviation in the plant's state. The function `findOutputDeviation` computes the deviation in plant's state for the reference and perturbed evolution. It takes as inputs: (1) two sets of states of plant variables, X_t and X'_t, at time instant t for the reference and perturbed evolutions, respectively, (2) two sets of states of plant variables, $X_{t+\tau}$ and $X'_{t+\tau}$, at time instant $t+\tau$ for the reference and perturbed evolutions, respectively, (3) the pair of intersecting paths, *path* and $path^\varepsilon$, and (4) the current deviation δ_t, for a state $x_0 \in X_0$ between the two evolutions at time instant t. The computation of the deviation due to elapse of a time period τ is based on the relation between states in the sets X_t and $X_{t+\tau}$ that is given by a discretization of the plant dynamics in Equation 1. Suppose that the discrete-time dynamics is given by: $x[k+1] = A_d x[k] + B_d u[k]$, where $k \in \mathbb{N}$, then the computation of the deviation is formalized as a constraint solving



(a) Switching Control System (b) Cruise Control System

Fig. 4. Performance of CLSE on increasing initial state set sizes and simulation lengths for example systems.

problem as below:

$$
\begin{aligned}
&\delta_{t+\tau} := |x_{t+\tau} - x'_{t+\tau}| \\
&\text{where, } x_{t+\tau} = A_d x_t + B_d u_t, \quad \text{and} \quad x'_{t+\tau} = A_d x'_t + B_d u'_t, \\
&\qquad \mathit{path} = \langle \kappa, \lambda \rangle \in \mathit{intersectingPaths}, \\
&\qquad \mathit{path}^{\varepsilon} = \langle \kappa^{\varepsilon}, \lambda^{\varepsilon} \rangle \in \mathit{intersectingPaths}^{\varepsilon}, \\
&\qquad u_t = [\lambda(y_1), \ldots, \lambda(y_m)]^T, \\
&\qquad u'_t = [\lambda^{\varepsilon}(y_1), \ldots, \lambda^{\varepsilon}(y_m)]^T, \text{where } Y_u = \{y_1, \ldots y_m\}, \\
&\qquad x_{t+\tau} \in X_{t+\tau} \quad \text{and} \quad x'_{t+\tau} \in X'_{t+\tau}, \\
&\qquad |x_t - x'_t| = \delta_t, \quad x_t \in X_{\text{init}} \quad \text{and} \quad x'_t \in X^{\varepsilon}_{\text{init}}.
\end{aligned}
$$

The expression $|x_{t+\tau} - x'_{t+\tau}|$ denotes the vector with components obtained by taking the absolute values of the difference of the corresponding components in $x_{t+\tau}$ and $x'_{t+\tau}$.

Updating the maximum deviation. After the computation of $\delta_{t+\tau}$, if $t + \tau$ reaches the simulation time bound T, we compare $\delta_{t+\tau}$ with Δ that is the maximum deviation found so far. If $\delta_{t+\tau} > \Delta$, Δ is updated to $\delta_{t+\tau}$.

5 Experiments

We have implemented CLSE using `c2d`, a model discretization routine provided by MathWorks' Control System Toolbox, to capture plant dynamics, and Splat [31] for concolic execution of programs. In addition, CLSE calls upon iSAT [20] as the constraint solver of choice. We have run experiments using our implementation on a 64-bit Linux on a machine with Intel Xeon X5650 2.66GHz processor and 48GB memory. In what follows, we present performance of CLSE for two examples of feedback-control systems, followed by the results from the robustness analyzer for one of them.

Example 1: Switching Control System. We evaluate CLSE first on an example of a switching control system that is representative of the class of systems

No. of steps	Δ for $[-0.1, 0.1]^2$		time (mins)	Δ for $[-0.25, 0.25]^2$		time (mins)	Δ for $[-0.5, 0.5]^2$		time (mins)
	x_1	x_2		x_1	x_2		x_1	x_2	
50	0.041	0.059	20.28	0.053	0.064	40.12	0.071	0.084	65.12
75	0.068	0.081	60.13	0.076	0.093	74.07	0.134	0.151	91.27

Table 1. The dimension-wise maximum deviation computed by our robustness analyzer and the time taken to compute them.

that we handle. In this system, obtained from the literature [10], depending on the current state of the plant, one of two controller gains is selected. The plant is represented as a transfer function and the controller uses an integrator. We have substituted the transfer function model of the plant by its equivalent state-space model, and the continuous integrator in the controller by a discrete-time integrator. The plant model has two continuous state variables and the controller (about 50 lines of C code) has two paths. The performance of Clse on this example is as shown in Fig. 4(a) for increasing sizes of initial state sets. We observe that, in this example, the number of unique path sequences that covers each of the selected initial state sets does not increase with increase in the simulation length. This is due to the fact that the control system stabilizes quickly under the effect of the controller, and after that the controller can execute only one control path corresponding to the stable state of the plant.

Example 2: Cruise Control System. This automotive system is obtained from the demonstration suite of the Reactis test-generation tool [28]. The plant model has four inputs and one continuous state variable *speed*, whereas the controller has eight inputs and two state variables. The controller code is about 200 lines, with 24 unique control paths. Among the inputs to the controller, 6 are user inputs that are Boolean, which leads to 2^6 configurations in which the closed-loop system operates. We have simulated the system for all the configurations for varying simulation lengths for speed in the range $[0, 80]$. The worst case simulation time, which is observed for 2 of the 2^6 configurations, for each simulation length is as shown in Fig. 4(b).

Robustness of the switching control system. For robustness analysis of the switching control system, we choose an error bound of $[-0.01, 0.01]^2$ on the sensors for the two state of the plant. The maximum bounds obtained by our implementation for simulation lengths 50 and 75, for increasing sizes of initial sets, are as shown in Table 1.

6 Related Work

Clse is similar to several recent projects in model-based testing and verification of hybrid systems. Alur *et al.* [2, 21] have proposed a method for symbolic simulation of closed-loop Simulink/Stateflow (SL/SF) models for the purpose of test-case generation. Our method bears a resemblance to this work in that both

adopt the same notion of equivalence of closed-loop trajectories and provide coverage over the space of initial states of the plant. However, CLSE implements a forward analysis that guarantees full coverage for the selected initial set of states unlike the backward analysis implemented in [2]; and CLSE directly handles the software implementation of the controller.

Lerda *et al.* [24] present a closed-loop analysis technique to find bugs in control software which is coupled with a continuous plant. They perform systematic simulation of the controller program using an explicit state software model checker and perform numerical simulations on the plant in the Simulink environment. In a similar vein, Păsăreanu *et al.* [27] propose a framework which supports translation of different modeling formalisms, including SL/SF, into a common intermediate representation and then use model checking and symbolic execution tools for property verification and test-case generation. Given a set of initial states of the system, our objective is not to model-check the controller software, but to compute the set of all sequences of paths executed in the controller software due to the closed-loop evolution up to a bounded time. Due to this, CLSE provides coverage over the input space of the system and not over the structure of the controller software.

HybridFluctuat [5] is a static analyzer for closed-loop systems that deals with software implementations directly just as CLSE does. It provides an assertion-based mechanism for specifying interaction of the software controller with the plant that can help build custom analyzers. It is based on Fluctuat [17] and leverages techniques from abstract interpretation in order to deal with path explosion.

Now we turn to related work on robustness analysis. Robustness of control system has been studied widely in the last thirty years (cf. [32]). There are software tools available to help design robust control systems [22]. However, the above-mentioned theory and software tools only help in analyzing a mathematical model of the control system. They do not consider the case when the controller is implemented as software. Analyzing software programs for robustness has been undertaken in a few recent works [25, 6], where controller programs are analyzed independently without the presence of the plant.

Robustness of a trajectory of a system has been studied in the recent past. Fainekos and Pappas [9] introduce a notion of robust satisfaction of a Linear or Metric Temporal Logic formula which is interpreted over finite timed state sequences in some metric space. Fainekos et al. [10] present a framework for reporting points where a simulation of a Simulink model may not be robust in the presence of both uncertainties in the model and internal computation errors. Robustness of hybrid automaton models for control systems have been studied before [18, 13]. However, unlike these works, our algorithm and tool provides a quantitative guarantee on the robustness of the system's output through the reachability analysis of the continuous plant, and program path exploration by symbolic execution of the controller program.

7 Conclusions

We believe Clse is a step toward closed-loop static analyzers that incorporate both plant and software dynamics into the analysis. Our experiences suggest that symbolic execution-based techniques, while precise, suffer from path explosion. It will be interesting to design an algorithm combining symbolic execution (for precision) and abstract interpretation (for scalability). In particular, we would like to study how path merging techniques that have been developed in the abstract interpretation setting can help scale our method. While our tool accepts the description of the plant as a set of linear differential equations, it is possible (but not trivial) to "compile" a Simulink/Stateflow model of the system to such a description [30, 26]. Adding support for non-linear systems is also an open direction.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
2. R. Alur, A. Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In L. de Alfaro and J. Palsberg, editors, *EMSOFT*, pages 89–98. ACM, 2008.
3. A. Anta, R. Majumdar, I. Saha, and P. Tabuada. Automatic verification of control system implementations. In *EMSOFT*, pages 9–18. ACM, 2010.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *PLDI*, 2003.
5. O. Bouissou, E. Goubault, S. Putot, K. Tekkal, and F. Védrine. HybridFluctuat: A Static Analyzer of Numerical Programs within a Continuous Environment. In *Proc. CAV*, pages 620–626, 2009.
6. S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. In *SIGSOFT FSE*, pages 102–112. ACM, 2011.
7. P. Cousot. Integrating physical systems in the static analysis of embedded control software. In K. Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 135–138. Springer, 2005.
8. T. Dang, C. L. Guernic, and O. Maler. Computing reachable states for nonlinear biological models. In *CMSB 09*, pages 126–141. Springer, 2009.
9. G. Fainekos and G. Pappas. Robustness of temporal logic specifications. In *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 178–192. Springer Berlin / Heidelberg, 2006.
10. G. E. Fainekos, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Robustness of model-based simulations. In *IEEE RTSS*, pages 345–354, 2009.
11. E. Feron. From control systems to control software. *IEEE Control Systems Magazine*, 30(6):50–71, 2010.
12. G. F. Franklin, D. J. Powell, and M. Workman. *Digital Control of Dynamic Systems*. Prentice Hall, 1997.
13. E. Frazzoli, M. Dahleh, and E. Feron. Robust hybrid control for autonomous vehicle motion planning. In *Proceedings of IEEE Conference on Decision and Control*, volume 1, pages 821–826. IEEE, 2000.

14. G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *CAV 11: Computer-Aided Verification*. Springer, 2011.
15. P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.
16. E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *ERTS 06*, 2006.
17. E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: Principles and experiments. In S. Leue and P. Merino, editors, *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2007.
18. V. Gupta, T. Henzinger, and R. Jagadeesan. Robust timed automata. In *Hybrid and Real-Time Systems*, LNCS, pages 331–345. Springer, 1997.
19. T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
20. iSAT solver, AVACS project, Available from: `http://isat.gforge.avacs.org`.
21. A. Kanade, R. Alur, F. Ivančić, S. Ramesh, S. Sankaranarayanan, and K. C. Shashidhar. Generating and analyzing symbolic traces of Simulink/Stateflow models. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 430–445. Springer, 2009.
22. C. Y. Kao, A. Megretzki, U. J onsson, and A. Rantzer. A MATLAB toolbox for robustness analysis. In *Computer-Aided Control Systems Design*. IEEE, 2004.
23. C. Le Guernic and A. Girard. Reachability analysis of linear systems using support functions. *Nonlinear Analysis: Hybrid Systems*, 4(2):250–262, 2010.
24. F. Lerda, J. Kapinski, H. Maka, E. Clarke, and B. Krogh. Model checking in-the-loop: Finding counterexamples by systematic simulation. In *ACC*, 2008.
25. R. Majumdar and I. Saha. Symbolic robustness analysis. In *IEEE RTSS*, 2009.
26. K. Manamcheri, S. Mitra, S. Bak, and M. Caccamo. A step towards verification and synthesis from Simulink/Stateflow models. In *HSCC*, 2011.
27. C. S. Păsăreanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karsai, H. Nine, and S. Neema. Model based analysis and test generation for flight software. In *3rd Intl. Conf. on Space Mission Challenges for IT*, pages 83–90. IEEE, 2009.
28. Reactis, Reactive Systems, `http://www.reactive-systems.com`.
29. K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 263–272. ACM, 2005.
30. A. Tiwari. Formal semantics and analysis methods for Simulink/Stateflow models. Technical report, SRI International, 2002.
31. R.-G. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In B. G. Ryder and A. Zeller, editors, *ISSTA*, pages 27–38. ACM, 2008.
32. K. Zhou and J. C. Doyle. *Essentials of Robust Control*. Prentice-Hall, 1998.