Sandboxing Controllers for Cyber-Physical Systems

Stanley Bak, Karthik Manamcheri, Sayan Mitra, and Marco Caccamo

Abstract—**Cyber-physical systems bridge the gap between cyber components, typically written in software, and the physical world. Software written with traditional development practices, however, likely contains bugs or unintended interactions among components, which can result in uncontrolled and possibly disastrous physical-world interactions. Complete verification of cyber-physical systems, however, is often impractical due to outsourced development of software, cost, software created without formal models, or excessively large or complex models where the verification process becomes intractable.**

Rather than mandating complete modeling and verification, we advocate sandboxing of unverified cyber-physical system controllers by augmenting the system with a verified safety wrapper that can take control of the plant in order to avoid violations of formal safety properties. The focus of this work is an automatic method, based on reachability and time-bounded reachability of hybrid systems, to generate verified sandboxes. The method is shown to be both more general than previous work, and allows the trade-off of increased computation time for improved reachability accuracy. We also present an end-to-end toolkit which performs the low-level computation to generate the sandbox source code from Simulink/Stateflow models of a cyber-physical system.

I. Introduction

This paper addresses the problem of verifying properties of cyber-physical systems with partly unknown software components. Consider, for example, a firm A which manufactures autonomous vehicles outsourcing the development of the embedded controller to firm B. A provides to B a set of specifications, such as the details of the available sensor-actuator signals, certain performance criteria, etc., based on which B then designs the controller and hands it back to A. In doing so, B may wish to provide the controller as a black-box without revealing the source-code or the core algorithms used for implementing it. How does A then guarantee the key safety properties of the autonomous vehicle that it manufactures without having complete access to B's controller model? In an analogous situation, the controller model provided by B may be untrusted, or it may be too complicated to be analyzed in conjunction with the rest of the autonomous vehicle.

As a more concrete example, consider the off-road vehicle industry, where vehicles traditionally made with hydraulics are starting to be built using steer-by-wire and drive-by-wire control. Software-based controllers may implement advanced features, such as locking the inner wheels on a sharp turn, to improve operator productivity. However, the unverified software may run on an unverified operating system which communicates to the actuators over a non-real-time network shared with other untrusted nodes. Complete verification is prohibitively expensive, yet an uncontained fault in the system could result in arbitrary actuation of both the steering and acceleration while the vehicle is in motion.

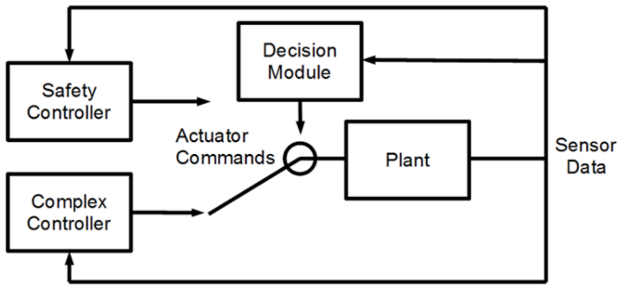


Fig. 1. The Simplex Architecture includes the physical plant, and three cyber components: a verified safety controller, a verified decision module, and an unverified complex controller.

Sandboxing is a popular technique for addressing these issues in the context of software and web-based security [1]. A *sandbox* is a testing environment that isolates untested and untrusted code and protects critical resources, such as live servers and their data, from changes that could be damaging. In this paper we present a technique for sandboxing controller software for cyber-physical systems in order to maintain formal safety invariants and, furthermore, automate the most difficult steps in the sandboxing process.

Our sandboxing technique uses the Simplex architecture [2]. As shown in Figure 1, Simplex, given a complex, untrusted or unverified controller (CC), creates a protected environment for the plant by adding a safety controller (SC) and a decision module (DM) such that the DM activates the SC and deactivates the CC *whenever* the CC may jeopardize the safety of the system. The key challenge in using the Simplex approach is determining the switching conditions for the DM, that is, the states of the system in which it is safe to allow the CC to remain active and the states in which SC must take control in order to maintain safety, without being excessively conservative. Previously, Lyapunov-function-based techniques have been used to determine switching conditions for purely continuous systems [2]. Alternatively, for discrete systems, model checking can be effective for creating and verifying the switching logic [3]. For combined continuous and discrete systems, *hybrid systems*, we introduced an algorithm [4] which computes the switching set by performing reach-set computations on certain classes of hybrid automata. [5], [6], [7].

At a high level, our earlier approach [4] computes the switching set as follows: Let U be the set of unsafe states. No SC with bounded actuation strength can guarantee safety from every state outside of U, U^c. If the system is at the boundary of U then, because of delays and inertia, SC may not be able to prevent entry into U. Instead, we strive to find a smaller set $R \subsetneq U^c$ from which SC is guaranteed to drive the system in a way which avoids U. Formally, R is computed as the backwards reachable set, or backreach set, from U using the SC behavior and plant dynamics. Furthermore, the switching from CC to SC is not instantaneous as the DM can only observe the plant state and make decisions at most once every δ time, for some positive time interval δ. Thus, a still smaller set $R' \subsetneq R$ is computed where CC can be active for δ-time without threatening safety of the system. Once outside R', SC must be activated or safety may be violated. Formally, R' is computed as the δ-time-bounded backreach set from R with the CC behavior and plant dynamics.

Computing exact reach sets for hybrid systems is, in general, an undecidable problem [8], but various restrictive subclasses have been identified for which it is decidable [8], [9], [10]. The applications we have in mind cannot be modeled naturally using these decidable classes, and hence, we resort to overapproximating the reach sets. In our earlier approach [4], the backreach sets R and R' were over-approximated by first creating discrete abstractions (simulations) of the system models, and then computing the backreach set and time-bounded backreach set from the abstractions. One advantage of the algorithm was that it could be used for systems with some nonlinear dynamics. The earlier approach was used to guide the design and verification of the DM for a Simplex-based off-road vehicle system in order to prevent vehicle rollover.

Although the algorithm from our previous work was a good first approach, several disadvantages were present. First, the models for which a discrete abstraction could be constructed were restricted. Second, even when a safe overapproximation of the backreach set could be produced, it was often excessively pessimistic (for example, the overapproximation computed of backreach of U in the SC model would be the entire state space). Lastly, no technique was provided to reduce the pessimism in the backreach computation.

In this paper, we address these core technical limitations of the earlier algorithm and integrate it within a software tool suite for sandboxing controllers for cyber-physical systems.

Specifically, the key contributions of this paper are:

- A fixpoint algorithm for computing backreachabililty which is applicable for a more general class of hybrid systems. The algorithm can be used with nonlinear dynamics as long as a function is provided which bounds each variable's derivative in a section of the state space.
- Three accuracy-increasing strategies which can be used to decrease the pessimism of the backreach computation. These are shown to be effective with an example, and an error-bounding theorem is provided.
- A Simulink/Stateflow-based toolkit for generating the switching set for a given plant, CC, and SC, which automates the sandboxing process. The toolkit is applied on a case study of a skid-steer system, which illustrates the steps involved in system modeling and switching-set generation.

The organization of this paper is as follows. First, in Section II, we briefly review background material relevant to the discussion of our work. Next, Section III presents our algorithm for computing backreachability and time-bounded backreachability. The text is accompanied by pseudocode and accuracy-improving strategies and an associated error-bounding theorem is provided. Finally, we discuss a Simulink/Stateflow-based toolkit which uses our algorithm to automatically generate the source code for the Simplex switching set in Section IV. This is presented in the context of case study of a skid-steer vehicle system with nonlinear dynamics. The paper finishes with related work in Section V and conclusions in Section VI.

II. Preliminaries

In this section, a brief review is provided about the Simplex Architecture and hybrid systems.

The Simplex Architecture enables verification of control systems where the controller cannot be modeled completely. This feature is particularly advantageous where the model of the complex controller is unavailable or untrusted, or where it is too complicated to be tractable by verification procedures. The key components of a Simplex system (Figure 1) are (a) an abstract model of the complex controller (CC), (b) a safety controller (SC) and (c) a decision module (DM). Once every δ-time, the DM observes the plant and makes a decision about which controller (SC or CC) to activate. Roughly, if there is a possibility of entering an unrecoverable state within the next δ interval, then DM activates SC. Once SC restores the plant to a state from which there is no possibility of violating safety in the next δ-interval, it reactivates CC.

We model each controller/plant combination in the Simplex system as a *hybrid automaton*. A hybrid automaton is a combination of differential equations with a finite state machine, where the state of the system can evolve both continuously and discretely. The continuous evolution of a hybrid system typically models the evolution of the physical variables in the plant, while the discrete transitions typically model software behavior. Formally, a hybrid automaton (HA) $\mathcal{H} = (V, \mathcal{L}, S, \theta, \mathcal{D}, \mathcal{T})$ consists of: (1) a set V of variables which define the dimensions of the system, (2) a finite set $\mathcal{L}$ of *locations* and $\ell_0 \in \mathcal{L}$, the initial location, (3) a set S of continuous states and a non-empty subset $\theta \subseteq S$ of start states, (4) a set $\mathcal{D} \subseteq \mathcal{L} \times \mathcal{L}$ of *discrete transitions* and (5) a set $\mathcal{T}$ of *trajectories* (or flows) for V which define the continuous behavior of the system in a given location. Each location $\ell \in \mathcal{L}$, has an *invariant condition* $\mathcal{I}_\ell$ associated with it. $\mathcal{I}_\ell$ should be satisfied for the system to be in the location ℓ. Each discrete transition $\tau \in \mathcal{T}$ has a guard condition g_τ and a reset map R_τ associated with it. A hybrid automaton can transition from ℓ_1 to ℓ_2, through a transition τ_1, if and only if the following conditions are satisfied: (a) $\mathcal{I}_{\ell_2}$ should be satisfied, and (b) g_τ should be satisfied. The complete formalism of hybrid systems has been further described in earlier work [5], [6], [7].

Let $\mathcal{U}$ be the set of unsafe states of the plant model. $\mathsf{BackReach}(\mathcal{U}, \textsc{sc})$ is defined as the set of states from which $\mathcal{U}$ can be reached within the SC hybrid automaton. $\mathcal{G} = \mathsf{BackReach}_{\leq\delta}(\mathsf{BackReach}(\mathcal{U}, \textsc{sc}), \textsc{cc})$ is the set of states from which $\mathsf{BackReach}(\mathcal{U}, \textsc{sc})$ can be reached in *up to* δ time within the CC hybrid automaton. Previously [4], we showed that if SC is activated when the plant state is first detected to be in $\mathcal{G}$ then the overall Simplex system remains safe. Therefore, in principle, if we can compute the backwards reachable set of states with respect to SC, and time-bounded backwards reachable sets with respect to CC, we can generate a DM that is correct by construction. Unfortunately, as discussed earlier, the problem of computing backwards reachable sets for hybrid systems has been well studied and is, in general, undecidable [8]. Furthermore, an accurate model for CC may not be available for the reasons presented in the introduction.

We can circumvent these issues by computing an overapproximation $\mathcal{G}' \supseteq \mathsf{BackReach}_{\leq\delta}(\mathsf{BackReach}(\mathcal{U}, \textsc{sc}), \textsc{cc}')$ based on an abstract model $\textsc{cc}'$ of the complex controller. For example, the actual outputs generated by CC can be abstracted by the range of values that are valid outputs for the actuators. Since $\mathcal{G}'$ overapproximates $\mathcal{G}$, using $\mathcal{G}'$ as the switching set also guarantees overall system safety.

III. Computing Backreachability

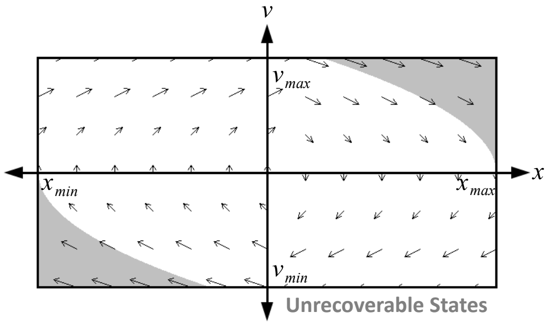
In this section, we describe the details of the algorithm used to construct Simplex switching set by overapproximating the equation $\mathsf{BackReach}_{\leq\delta}(\mathsf{BackReach}(\mathcal{U}, \textsc{sc}), \textsc{cc}')$. Throughout this section, we will augment the discussion with a demonstrative Simple-Vehicle System, which is introduced in Section III-A. Section III-B presents the assumptions of our backreachability algorithm. Next, Subsection III-C presents the algorithm for computing bounded and unbounded backreach sets. Finally, in Section III-D, three strategies are proposed which together can bound the error of a $\mathsf{BackReach}_{\leq\delta}$ computation to an arbitrary constant.

Additionally, although we are concerned with backreachability for Simplex, the explanations are easier to understand, and therefore presented, in terms of forward reachability. The two notions can be shown to be computationally equivalent.

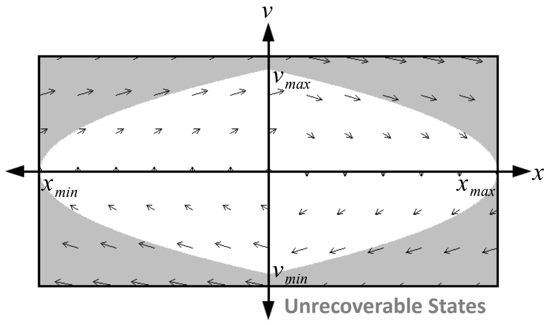
A. Simple-Vehicle System Example

Consider a vehicle which moves along a one-dimensional line, modeled as a point x on the x-axis which moves according to the input acceleration a generated by the controller. Physical constraints require that $a \in [a_{min}, a_{max}]$, and velocity the velocity v of the vehicle remains in the range $[v_{min}, v_{max}]$. The safety property requires that the point x remains in the range $[x_{min}, x_{max}]$, where $x_{min} < 0 < x_{max}$.

For this system, one possible safety controller is a bang-bang controller which outputs the maximal negative acceleration $a = a_{min}$ for $x > 0$ and outputs $a = a_{max}$ if $x \leq 0$. Two



(a) if $(v_{max})^2/(2 * a_{min}) < x_{max}$



(b) if $(v_{max})^2/(2 * a_{min}) > x_{max}$

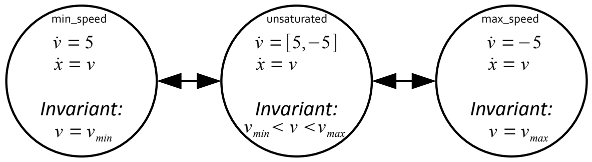
Fig. 2. The gray area indicates $\mathsf{BackReach}(\mathcal{U}, \textsc{sc})$, states where using the safety controller leads to safety violations for the example from Section III-A. Both figures assume $x_{min} = -x_{max}$, $v_{min} = -v_{max}$ and $a_{min} = -a_{max}$.

possible sets of unrecoverable states (depending on the exact parameters of the plant) are shown in Figure 2. This region can be computed directly by examining the backreachability using the SC automaton from set of unsafe states, $\mathcal{U}$, or formally $\mathsf{BackReach}(\mathcal{U}, \textsc{sc})$.

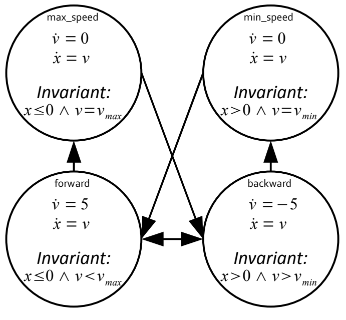
Next, we specify the hybrid automata for the CC' system and SC system, which are show in Figure 3. The hybrid automaton for the CC' system, in Figure 3(a), has three locations (discrete modes). Under unsaturated operation, the location invariant is $v_{min} < v < v_{max}$ and the derivative equations are $\dot{x} = v$ and $\dot{v} = [a_{min}, a_{max}]$ ($\dot{v}$ is nondeterministic). There are two other locations corresponding to when the point has reached its minimum and maximum velocity (labeled min_speed and min_speed) where $\dot{v}$ is restricted to be either nonnegative or nonpositive. The hybrid automaton for the SC, in Figure 3(b), contains two locations corresponding to the two states of the controller (labeled forward and backward), and two more locations for when the velocity has reached saturation (labeled min_speed and max_speed). When we compute an overapproximation of $\mathsf{BackReach}$ or $\mathsf{BackReach}_{\leq\delta}$, it will be with respect to one of these automata.

B. System Assumption

In order to apply our algorithm, we have two assumptions, which we outline and elaborate on in the context of the Simple-Vehicle System below.



(a) the CC' (complex controller abstraction and plant) automaton



(b) the SC (safety controller and plant) automaton

Fig. 3. The hybrid automata describing the Simple-Vehicle System have no transition guard restrictions, so the discrete location switching is done solely based on the invariants.

Assumption 1: For any rectangular set of states $H \subseteq S$, for any continuous variable x_i, there exist functions $db_{x_i}^{min}$ and $db_{x_i}^{min}$, that bound the derivative of x_i with respect to time in H. That is, $db_{x_i}^{min}(H) \le \frac{dx_i}{dt} \le db_{x_i}^{max}(H)$, for every x_i.

Assumption 2: We make a distinction between two types of derivative dependencies, explicit ones directly extracted from the differential equations in each location of the hybrid automaton (for example, $\dot w = v$ would create an directed edge from the node corresponding to v to the node corresponding to w), and implicit dependencies which arise because as time advances, the continuous state may cause a change in hybrid-automaton locations which causes the differential equations of variables to be changed. Assumption 2 restricts the systems we consider to those where the *explicit* dependency graph of the state-variable derivatives does not have cycles, except for self-loops. This restriction is more relaxed than in our previous work, where the combined explicit / implicit derivative dependency graph was required to be acyclic, except for self-loops.

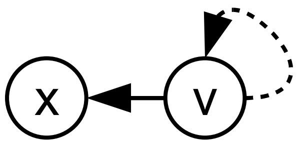
Example: In the context of the Simple-Vehicle System, to meet Requirement 1, we must provide the derivative bounds functions for each variable, x and v, which can be automatically extracted from the hybrid automata. These functions takes as input a rectangle of the state space defined by upper and lower bounds on each variable, $([v^{lower}, v^{upper}] \times [x^{lower}, x^{upper}])$.

For the CC' automaton, $db_x^{min} = v^{lower}$, $db_x^{max} = v^{upper}$,

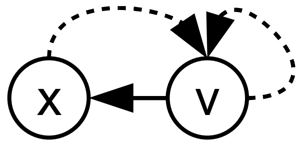
$$db_v^{min} = \begin{cases} -5 & \text{if } v^{upper} > v_{min} \\ 0 & \text{otherwise} \end{cases}$$

and

$$db_v^{max} = \begin{cases} 5 & \text{if } v^{lower} < v_{max} \\ 0 & \text{otherwise} \end{cases}$$



(a) the CC' derivative dependency graph



(b) the SC derivative dependency graph

Fig. 4. The derivative dependency graphs for the Simple-Vehicle System have explicit dependencies (solid arrows) and implicit dependencies (dashed arrows).

For the CC' automaton, again, $db_x^{min} = v^{lower}$, $db_x^{max} = v^{upper}$, and

$$db_v^{min} = \begin{cases} -5 & \text{if } x^{upper} > 0 \wedge v^{upper} > v_{min} \\ 0 & \text{if } x^{upper} > 0 \wedge v^{upper} \le v_{min} \\ 0 & \text{if } x^{upper} \le 0 \wedge v^{upper} \ge v_{max} \\ 5 & \text{otherwise} \end{cases}$$

and

$$db_v^{max} = \begin{cases} 5 & \text{if } x^{lower} \le 0 \wedge v^{lower} < v_{max} \\ 0 & \text{if } x^{upper} > 0 \wedge v^{upper} \le v_{min} \\ 0 & \text{if } x^{upper} \le 0 \wedge v^{upper} \ge v_{max} \\ -5 & \text{otherwise} \end{cases}$$

To meet Requirement 2, we construct and check the derivative dependency graphs (shown in Figure 4). For both controllers, in every location, the value of $\dot x$, explicitly depends on v, and the value of $\dot v$ does not explicitly depend on any variables. However, due to the possibility of a change in automaton location, there is an implicit dependence of $\dot v$ on v in the CC' automaton. In the SC, there are two implicit dependencies: one implicit dependence of $\dot v$ on v, and a second implicit dependence of $\dot v$ on x. Since both of the explicit dependency graphs (solid arrows) with self-loops removed are acyclic, the system meets the second requirement. Notice, however, that the SC automaton would not meet the restrictions of our previous algorithm since the combined explicit / implicit derivative dependency graph contains a cycle that is not a self-loop.

C. Algorithm for Overapproximating Reach and $\text{Reach}_{\le\delta}$

In this section, we outline our proposed algorithm for computing overapproximations of Reach and $\text{Reach}_{\le\delta}$. We start by presenting the pseudocode of the algorithm, and then elaborate on each of the functions.

The pseudocode uses the following notation: The expression $D.i$ refers to the ith element of a finite set D in an arbitrary fixed ordering. The expression $H_{/i}$ refers to the projection of the ith dimension of a hyperrectangle H. The minimum value in this one-dimensional projection is referred to by $H_{/i}^{min}$ and the maximum value is referred to by $H_{/i}^{max}$. The hybrid automaton has n continuous variables, $x_1, x_2, \dots, x_n$, which are ordered according to the topological sort of the explicit derivative dependency graph with self-loops removed. The values q_{x_i} for each variable x_i are quanta used in the computation which are fixed constants.

In the pseudocode, several functions are also used:

- `getLocations` takes a hyperrectangle state space as input and outputs a set of integers corresponding to the set of locations in which the hybrid automaton may be.
- `getFirstImplicitDerivativeDependency` takes as input an integer corresponding to an automaton location, and returns the implicit derivative dependency of the node corresponding to x_i that comes first in the topologically-sorted variable order.
- α is an abstraction function which takes as input a set of states, and outputs a set of abstract states represented an integer for each variable. The α^{-1} function is the concretization function which, given an abstract state, returns the set of corresponding concrete states of the system. Formally, if the continuous state space is $\mathcal{X} = \mathbb{R}^n$ and the abstract state space is $\mathcal{Y} = \mathbb{Z}^n$, then $\alpha : \mathcal{X} \longrightarrow \mathcal{Y}$. $\alpha(x) = y_1, y_2, \ldots, y_n$, where each $y_i = x_i/q_{x_i}$ and x_i refers to the ith coordinate of a state $x \in \mathcal{X}$. In the abstraction function, q_{x_i} are the constant quanta for each variable, as mentioned above. As usual, $\alpha^{-1} : \mathcal{Y} \longrightarrow 2^{\mathcal{X}}$. In the pseudocode, we use the natural lifting of these functions from a single input state to a set of input states.

The pseudocode for the reachability and time-bounded reachability algorithms is below:

```
1  %%% ComputeReach outputs an overapproximation of reachability from an initial set
2  ComputeReach($\mathcal{I}$}
3  {
4    declare $D := \alpha(\mathcal{I})$;
5    declare $C := \alpha^{-1}(D)$;
6    declare $m := |D|$; % number of abstract states
7    declare array[$m$] E;
8    declare $C' := \emptyset$;
9
10   for $i = 1$ to $m$
11     E[$i$] $:= \alpha^{-1}(D.i)$; % initialize the $m$ exact reach sets
12
13   while $(C \neq C')$ % loop until fixpoint
14     $C' := C$;
15
16     for $(i = 1; i < m; i := i + 1)$
17       $(\mathcal{H}, \mathcal{E}) :=$ ComputeDeltaReach(E[$i$]);
18       E[$i$] $:= \mathcal{E}$; % update exact reach set
19       $C := C \cup \mathcal{H}$; % accumulate reachability
20
21   return $C$;
22 }
23
24 %%% ComputeDeltaReach outputs $(\mathrm{Reach}_{\leq\delta}, \mathrm{Reach}_{\delta})$ from an input hyperrectangle
25 ComputeDeltaReach($\mathcal{H}$)
26 {
27   declare $\mathcal{E} := \mathcal{H}$;
28   declare $\mathcal{L} :=$ getLocations($\mathcal{H}$);
29
30   for $(i = 1; i < n; i = i + 1)$ % loop over every variable
31     declare $l = MinReach(i, \mathcal{H})$;
32     declare $u = MaxReach(i, \mathcal{H})$;
33
34     $\mathcal{E}_{/i} := [l, u]$; % update exact delta reach
35     $\mathcal{H}_{/i} := [min(\mathcal{H}^{min}_{/i}, l),\ max(\mathcal{H}^{max}_{/i}, u)]$; % update delta reach
36
37     if $(\mathcal{L} \neq$ getLocations($\mathcal{H}$))
38       $\mathcal{L} :=$ getLocations($\mathcal{H}$);
39       $i :=$ getFirstImplicitDerivativeDependency($i$) $- 1$; % backtrack
40
41   return $(\mathcal{H}, \mathcal{E})$
42 }
43
44 %%% MinReach overapproximates the minimum value of $x_i$ reached after $\delta$ time
45 MinReach($i$, $\mathcal{H}$)
46 {
47   return MinReachRecursive($\mathcal{H}$, $i$, $\delta$, $\mathcal{H}^{min}_{/i}$, $db^{min}_{x_i}$, true);
48 }
49
50 %%% MinReachRecursive overapproximates the minimum value of $x_i$ reached
51 MinReachRecursive($i$, $\mathcal{H}$, time, start, db, isFirstInterval)
52 {
53   $\mathcal{H}_{/i} :=$ [start $-q_{x_i}$, start]; % consider the current interval
54   declare der $:= db(\mathcal{H})$;
55   declare nextIntervalTime $:=$ (der $= 0$ ? time : $-q_{x_i}/der$);
56
57   if (nextIntervalTime $< 0$) % switch direction
58     if (!isFirstInterval) % can not safely reverse direction; be pessimistic
59       return start;
60     else
61       return MaxReachRecursive($i$, $\mathcal{H}$, time, start, db, false);
62   else if (nextIntervalTime $\geq$ time) % time expires in this interval
63     return start + nextIntervalTime $*$ der;
64   else % continue to next interval
65     return MinReachRecursive($i$, $\mathcal{H}$, time $-$ nextIntervalTime, start $-q_{x_i}$, db, false);
66 }
```

The top-level of the algorithm, the `ComputeReach` function (lines 1-22), starts by dividing the state space into hyperrectangles based on a provided constant quantum size for each variable through the abstaction function α (line 4). The fixpoint computation loop for computing reachability occurs on lines 13-19. Starting from the hyperrectangle corresponding to every discrete state which contains an unsafe state, we use the `ComputeDeltaReach` function to compute both the states reachable in up to δ time, $\mathsf{Reach}_{\leq\delta}$, and the states reachable in exactly δ time, $\mathsf{Reach}_{=\delta}$ (line 17). The $\mathsf{Reach}_{=\delta}$ result is used as the initial set of states for the next iteration of the loop (line 18), while the $\mathsf{Reach}_{\leq\delta}$ result is added to the global reachable set (line 19). The intuition behind the correctness of this function is that any state reachable in up to $k < \delta$ time will necessarily pass through a state reachable in exactly δ time (specifically, after $\delta - k$ time). Therefore, we need not consider all of $\mathsf{Reach}_{\leq\delta}$ as the initial states in the next iteration. In terms of termination, notice that this algorithm will clearly not terminate if the computed reach set is infinite. If termination is desired, one can bound the reachable state space with the hybrid automaton, and assure the the variable derivatives do not asymptotically approach 0 (for example, by adding an ϵ amount of overapproximation in the $db^{min}_{x_i}$ and $db^{max}_{x_i}$ functions).

The `ComputeDeltaReach` function (lines 24-42) is used to overapproximate both $\mathsf{Reach}_{\leq\delta}$ and $\mathsf{Reach}_{=\delta}$. Starting from an input initial hyperrectangle, the sets of states are computed for each variable in the order of the topological sort of the explicit derivative dependency graph with self-loops removed (lines 30-39). This ensures that when the bounds is being computed for a particular variable, all (non-self) dependent variables already have valid computed $\mathsf{Reach}_{\leq\delta}$ bounds, ensuring correctness. After a $\mathsf{Reach}_{\leq\delta}$ bound for a variable is computed, this bound is used as input to the derivative bounds function that is used to compute $\mathsf{Reach}_{\leq\delta}$ for the subsequent dependent variables (because of the assignment on line 35). The $\mathsf{Reach}_{=\delta}$ set is also maintained (line 34). After we have computed $\mathsf{Reach}_{\leq\delta}$ for every variable, the cross product is an overapproximation of $\mathsf{Reach}_{\leq\delta}$ from the initial state (which is iteratively constructed on line 35).

The proposed algorithm does allow models with cycles caused by implicit derivative dependencies, restricting only that the explicit-only dependency graph of derivatives be acyclic except for self-loops. After we compute the $\mathsf{Reach}_{\leq\delta}$ for a variable with an implicit derivative dependency which creates a cycle in the combined explicit / implicit derivative dependency graph, we go back and check if we may have entered a new state of the automaton (line 37). If we may have, the algorithm backtracks and recomputes the reachability of all the variables that could have possibly been affected (line 39). Since there are only a finite number of discrete locations in the hybrid automaton, the backtrack process can only happen a finite number of times, and the algorithm remains terminating.

Example: Consider computing $\mathsf{Reach}_{\leq\delta}$ in the Simple-Vehicle System with respect to the safety controller / plant automaton (Figure 3(b)), starting from $(x = [-2, -1], v = [4, 5])$. We fix the quanta of both dimensions to be 1, fix δ to be 0.5 time units, fix v_{max} to be 10, and fix $a_{min} = -5$ and $a_{max} = 5$. The algorithm will compute in the order of the explicit dependency graph, first v then x. First the reachability for v is computed using `MinReach` and `MaxReach`. These two functions return 6.5 and 7.5, respectively. Due to the capping of the minimum value (line 35) the $\mathsf{Reach}_{\leq\delta}$ range is set as $v = [4, 7.5]$. Second, the reachability for x is computed to be $x = [-2, 2.75]$. The algorithm can not terminate here, since the change in x may result in a change in hybrid-automaton location (the condition on line 37 is true). The algorithm goes back and recomputes $\mathsf{Reach}_{\leq\delta}$ for v to be $v = [1.5, 7.5]$. Next, the reachability of x is recomputed once again to be $x = [-2, 2.75]$. Although there is an implicit dependency, the set of possible discrete locations has not changed since the last iteration (the condition on line 37 is false due to the prior assignment on line 38), and the loop terminates. The result of the algorithm is an overapproximation of the actual $\mathsf{Reach}_{\leq\delta}$ states, shown in Figure 5.

The `MinReach` and `MinReachRecursive` functions (lines 44-66) (and symmetric `MaxReach` and `MaxReachRecursive`, not shown), overapproximate $\mathsf{Reach}_{=\delta}$ for one variable, under the assumption that the passed-in hyperrectangle contains valid $\mathsf{Reach}_{\leq\delta}$ bounds for all dependent dimensions. This is done by starting at the minimum value of the variable in the initial hyperrectangle (line 47), and proceeding at the minimum derivative for δ time, considering new derivatives as we enter new intervals (line 53). Intuitively, this is correct because the all the bounds of the dependent dimensions is correct are assumption so the call to db (line 54) will yield a correct bound on the current variable's derivative. If the derivative is nondeterministic, we will still overapproximate $\mathsf{Reach}_{=\delta}$ by considering the minimum in each interval. Special care is taken if the derivative is zero (line 55), or if the minimum derivative is positive (lines 58-61), which avoids infinite loops caused by Zeno behavior through the `isFirstInterval` variable.

Example: To help illustrate this algorithm, we compute $\mathsf{Reach}_{\leq\delta}$ from a state in the Simple-Vehicle System, in the

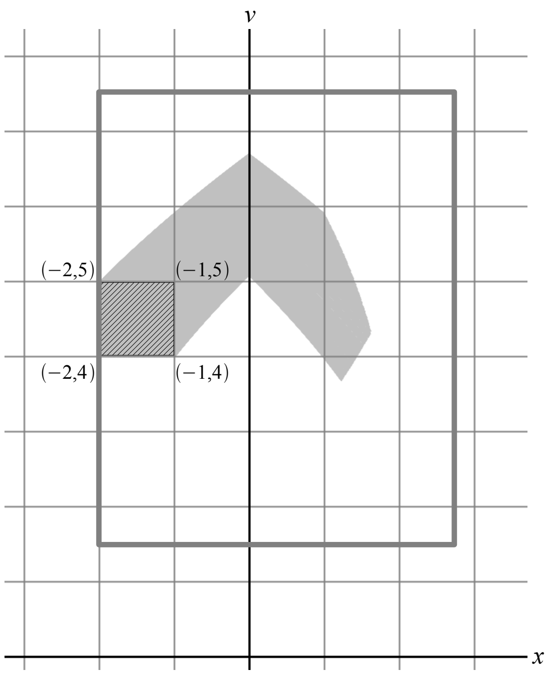


Fig. 5. An estimate of $\mathsf{Reach}_{\leq\delta}$ in the computed using 10,000 simulations (light gray region) for the Simple-Vehicle System is shown in comparison with the overapproximation computed with proposed $\mathsf{Reach}_{\leq\delta}$ algorithm (solid gray line). This computation is done with respect to the SC automaton. Here, δ is 0.5 and the initial state is $(x = [-2, -1], v = [4, 5])$.

CC' automaton (Figure 3(a)). Now, we will compute $\mathsf{Reach}_{\leq\delta}$ from the hyperrectangle $(x = [1, 2], v = [8, 9])$. The topological sort of the self-loop-removed derivative-dependency graph (Figure 4(a)) is v, x, so we start with the v dimension. The inner loop of the algorithm, `MinReach` and `MaxReach`, will compute the minimum and maximum values of v that can be reached after δ time. In the functions, the variable `time` is used to keep track of the time elapsed in terms of the execution of the system when computing these values.

Initially, `time` $= \delta = 0.5$. To compute the minimum reachable velocity, we start by invoking db_v^{min} with the hyperrectangle $(x = [1, 2], v = [7, 8])$, which outputs the derivative bounds -5. At the minimum derivative, -5, the next interval is reached in 0.2 time units. We update `time` to be $0.5 - 0.2 = 0.3$. The process then repeats for the next interval, $v = [6, 7]$. The minimum derivative for v is again -5, and `time` is updated to 0.1. On the third iteration, `time` is less than the time it would take to reach the next interval, and the minimum $\mathsf{Reach}_{\leq\delta}$ value is computed as $v = 4.5$ (line 63).

To compute the maximum velocity we again, initialize `time` to be $\delta = 0.5$. First, db_v^{max} with the hyperrectangle $(x = [1, 2], v = [9, 10])$ outputs 5 as the maximum derivative for the v variable. The `time` variable is updated to 0.3. On the second iteration, however, db_v^{max} with the hyperrectangle $(x = [1, 2], v = [10, 11])$ outputs 0 as the maximum derivative for the v variable, so the next interval is unreachable in 0.3 time units. The maximum $\mathsf{Reach}_{=\delta}$ value is therefore $v = 10$ (line 63). The $\mathsf{Reach}_{=\delta}$ values for v are $[5, 10]$

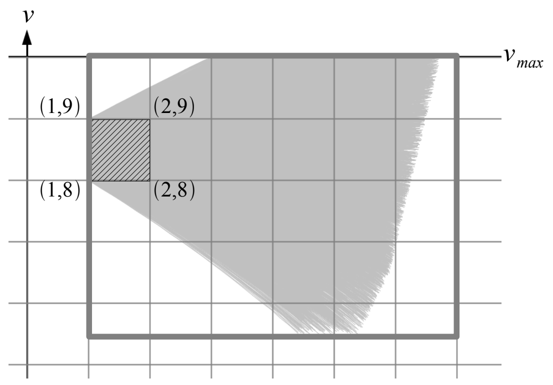


Fig. 6. An estimate of $\mathsf{Reach}_{\leq\delta}$ computed using 10,000 simulations (light gray region) for the Simple-Vehicle System is shown in comparison with the overapproximation computed with the proposed $\mathsf{Reach}_{\leq\delta}$ algorithm (solid gray line). This computation is done with respect to the CC' automaton. Here, δ is 0.5 and the initial state is $(x = [1, 2], v = [8, 9])$.

Now the `ComputeDeltaReach` function would proceeds to compute the reachable values for the next variable, x, using the values $[5, 10]$ for the v-dimension of the hyperrectangle when calling `MinReach` and `MaxReach`.

Computing the maximum value that can be reached for x, the states of the computation proceed as: ($t = 0.5$, $x = [2, 3]$, $db_x^{max}(x = [2, 3], v = [5, 10]) = 10$),
($t = 0.4$, $x = [3, 4]$, $db_x^{max}(x = [3, 4], v = [5, 10]) = 10$),
($t = 0.3$, $x = [4, 5]$, $db_x^{max}(x = [4, 5], v = [5, 10]) = 10$),
($t = 0.2$, $x = [5, 6]$, $db_x^{max}(x = [5, 6], v = [5, 10]) = 10$),
($t = 0.1$, $x = [6, 7]$, $db_x^{max}(x = [6, 7], v = [5, 10]) = 10$).
where t is the value of `time`. At this point, the next interval can not be entered in 0.1 time, and the maximum $\mathsf{Reach}_{\leq\delta}$ value is $x = 7$.

Computing the minimum value x can reach, we start with $t = 0.5$ and invoke db_x^{min} with the hyperrectangle ($x = [0, 1], v = [5, 10]$). This returns a minimum derivative in the x dimension of 5, which is nonnegative, so we switch directions and call `MaxReachRecursive` (line 61), with the db_x^{min} function as a parameter. This will eventually reach a minimum $\mathsf{Reach}_{=\delta}$ value of $x = 6$. The $\mathsf{Reach}_{=\delta}$ values are $x = [6, 7]$. This example is shown in Figure 6

D. Accuracy Convergence

One important concern for using the proposed algorithm to compute the Simplex switching set is the accuracy of the $\mathsf{Reach}_{\leq\delta}$ overapproximation, which in turn affects the accuracy of the global-reachability overapproximation that it computes. In this subsection, we propose three strategies that can be used to reduce the error of the proposed algorithm. Then, an important accuracy theorem is stated which allows us, under some reasonable assumptions, to reduce the computed $\mathsf{Reach}_{\leq\delta}$ error for each variable to an arbitrarily small constant. Finally, through example, the proposed strategies are shown to, in fact, reduce the computed error.

We propose three strategies to reduce the error of the $\mathsf{Reach}_{\leq\delta}$ region from an initial hyper-rectangle, which we call the quantum rule, the refine rule and the split rule. We later show that these, when used in combination, can reduce the error in each variable of the$\mathsf{Reach}_{\leq\delta}$ overapproximation to an arbitrary constant.

The **quantum rule** uses the fact that the inner loop (`MinReach` and `MaxReach`) of the algorithm considers intervals one quantum in size. Since no restrictions are placed on the minimum size of the quantum, the quantum rule, which evenly splits the size of this quantum for one variable, is always safe to apply. Intuitively, this helps with accuracy by, at each time, providing a smaller hyperrectangle to the derivative bounds function which allows it to output a more accurate derivative bound.

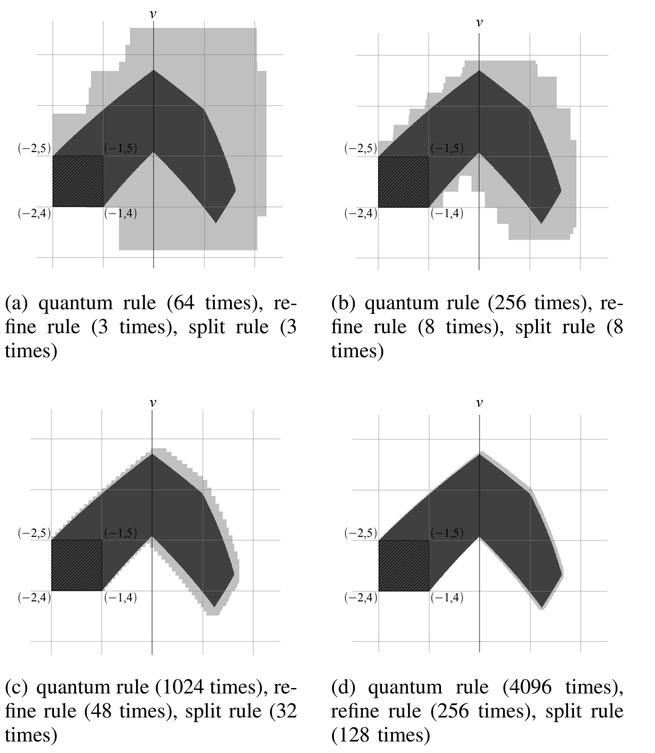
The **refine rule** is based on the fact that $\mathsf{Reach}_{\leq\delta}$ from an initial set, w, is equal to the union of $\mathsf{Reach}_{\leq\delta}$ from multiple smaller sets, as long as the union of those smaller sets is equal to the initial set w. Formally, if $w = w_1 \cup w_2 \cup \ldots \cup w_n$, then $\mathsf{Reach}_{\leq\delta}(w)$ = $\mathsf{Reach}_{\leq\delta}(w_1) \cup \mathsf{Reach}_{\leq\delta}(w_2) \cup \ldots \cup \mathsf{Reach}_{\leq\delta}(w_n)$. This allows us to refine the initial hyperrectangle into smaller hyperrectangles and still obtain a safe overapproximation by taking the union of the results. The rule itself will be applied to a particular variable, and will split the initial hyperrectangle into equal-sized hyperrectangles along that variable. As with the quantum rule, this intuitively helps with accuracy by providing a smaller hyperrectangle to the derivative bounds function which allows it to output a more accurate derivative bound.

The final rule, the **split** rule, is based on the fact that $\mathsf{Reach}_{\leq\delta}$ can be decomposed in a way similar to the way in which we described computing full reachability the `ComputeReach` function. Formally, $d_1 + d_2 + \ldots + d_n = \delta$ implies that $\mathsf{Reach}_{\leq\delta}(w)$ = $\mathsf{Reach}_{\leq d_1}(w) \cup \mathsf{Reach}_{\leq d_2}(\mathsf{Reach}_{=d_1}(\mathrm{w})) \cup \mathsf{Reach}_{\leq d_3}(\mathsf{Reach}_{=d_2}(\mathsf{Reach}_{=d_1}(\mathrm{w}))) \cup \ldots$. When we say this rule is applied n times, we split δ with n equal constants, $d_1 = d_2 = \ldots = d_n = \frac{\delta}{n}$. Intuitively, this rule will improve accuracy by, when computing the reachability for a particular variable, reducing the variable ranges of the dependent dimensions. This provides the derivative bounds function with a smaller hyperrectangle, which allows it to output a more accurate derivative bound.

The application of these three rules can be used to reduce the pessimism of the error in the $\mathsf{Reach}_{\leq\delta}$ overapproximation to an arbitrarily small constant in each dimension. This is reflected in the following theorem.

Theorem 1: By applying the quantum rule, the refine rule and the split rule finitely many times, the maximum error in each variable x of the computed $\mathsf{Reach}_{\leq\delta}$ overapproximation, for a fixed δ and from an initial hyperrectangle α, can be reduced to below an arbitrary positive constant e_x.

The proof of this theorem has two primary assumptions. First, the derivative bounds function should not itself output errors of the actual derivative bounds, as the computed set relies on the accuracy of this function. Second, the derivative function for each variable x, $\dot{x} = f(y_1, y_2, \ldots, y_n)$ should be a Lipschitz continuous function with respect to each input



(a) quantum rule (64 times), refine rule (3 times), split rule (3 times)

(b) quantum rule (256 times), refine rule (8 times), split rule (8 times)

(c) quantum rule (1024 times), refine rule (48 times), split rule (32 times)

(d) quantum rule (4096 times), refine rule (256 times), split rule (128 times)

Fig. 7. An estimate of $\text{Reach}_{\leq\delta}$ computed using 10,000 simulations (dark gray region) for the Simple-Vehicle System is shown in comparison with the overapproximation computed with the $\text{Reach}_{\leq\delta}$ algorithm (light gray region) with various amounts of accuracy-increasing strategies applied.

variable, and the derivative value $\dot{x}$ should be bounded. This essentially means that we can bound the rate of change of the derivative $\dot{x}$ (and therefore indirectly the rate of change of the value of x) in a finite amount of time.

Due to space limitations, the proof of this theorem is presented in this paper's companion technical report [11].

As a demonstration of the implications of this theorem, we show an example in the Simple-Vehicle System using the safety controller / plant with an initial hyperrectangle of $(x = [-2, -1], v = [4, 5])$ with a value of $\delta = 0.5$. The original computed $\text{Reach}_{\leq\delta}$ result, with no applied accuracy-increasing strategies, was previously shown in Figure 5. By applying the accuracy-increasing strategies, we can approach the actual $\text{Reach}_{\leq\delta}$ set with arbitrary precision, as shown in Figure 7.

IV. Case Study: Waypoint Tracking System

We now discuss the proposed sandboxing approach for an autonomous waypoint tracking system (WTS) with a short case study. This system model is inspired by applications such as automated lawn mowers or skid-steer loaders. The autonomous vehicle is required to follow a (predefined) sequence of waypoints while remaining within a fixed safe distance of the line joining successive waypoints.

The controller software periodically senses the position (x, y), the velocity v, and the heading θ, of the vehicle and sets the acceleration $(\dot{v})$ and the steering $(\dot{\theta})$ based on the current waypoint (x^*, y^*) of the system. The vehicle models a skid-steer system which can turn in place, i.e., the heading θ can change even when the velocity v is 0. The equations of motion for the vehicle's position are given by the following nonlinear differential equations:

$$\dot{x} = v\cos\theta, \quad \dot{y} = v\sin\theta$$

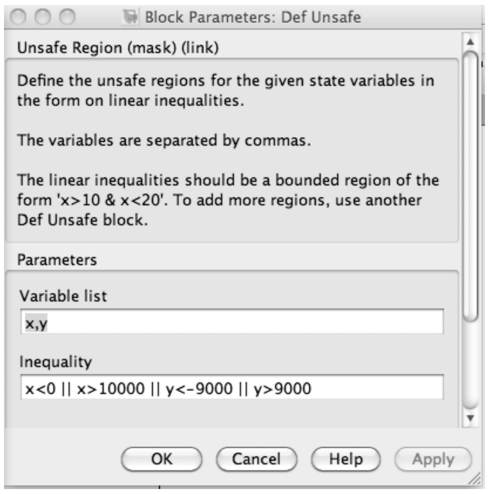
We assume that there is no information available for the complex controller (CC) we wish to sandbox, except that it operates within the physical limits of the actuator. That is, $\dot{v} \in [a_{min}, a_{max}]$ and $\dot{\theta} \in [\phi_{min}, \phi_{max}]$. Recall that the safety requirement is to keep the vehicle within some distance of the line joining the waypoints. Thus, a simple safety controller (SC) strategy is to '*slow down and stop the vehicle as fast as possible*'.

The Embedded Safety Critical Programming Environment (ESCAPE) toolkit is a set of tools and design methodology we are developing which is uses the Simplex Architecture to generate cyber-physical system sandboxes. ESCAPE consists of two parts, 1) HyLink, and 2) SimplexGen. HyLink is a translation tool which takes as its input a Simulink/Stateflow model and translates the model into an hybrid system intermediate format. SimplexGen is an implementation of the algorithm described in this paper which takes as its input, 1) the safety controller / plant (SC) model, 2) the abstract complex controller / plant (CC') model 3) the safety invariant to verify 4) computation constants (the size of the quanta, the value of the control interval δ) and generates the Simplex switching set. ESCAPE provides a set of Simulink blocks for defining the safety invariant and computation constants, as shown in Figure 8(a).

The CC and SC were modeled as hybrid systems using Mathwork's Simulink environment (Figure 8(b)). HyLink is used to extract the hybrid automata from the Simulink/Stateflow models and create an input format which SimplexGen can use. The SimplexGen tool then uses the algorithm described in Section III-C to generate the switching set. This switching set can then be automatically encoded into a source code file to be used in the Simplex decision module during operation.

For computing the backreach set, the algorithm in Section III-C requires the derivative bounds for each variable. These bounds are obtained from the translated hybrid automata models of the CC and SC as follows: given a rectangular set H of states, the derivative bounds of each variable in H are determined from the trajectory definitions (differential equations) of the locations of the hybrid automaton whose invariants intersect H. The differential equations for each variable x_i are of the form $\dot{x}_i = f(\mathbf{x})$. We then maximize and minimize the function $f(\mathbf{x})$ over the set H.

The size of the discrete state space for this example consisted of 1,536,000 states, and the algorithm ran to termination in about 20 minutes. The switching set generated for the WTS is a four-dimensional set (x, y, v, θ) which cannot be easily visualized. We can, however, analyze the output set by fixing two of the dimensions (for example, v and θ) and plotting the other two dimensions. Figure 9 shows two plots of the



Define the unsafe regions for the given state variables in the form on linear inequalities.

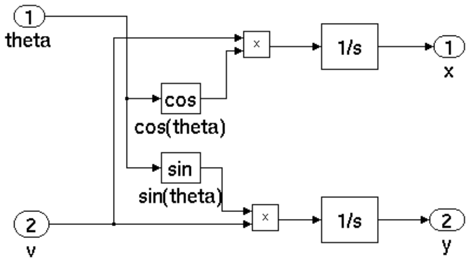
The variables are separated by commas.

The linear inequalities should be a bounded region of the form 'x>10 & x<20'. To add more regions, use another Def Unsafe block.

Variable list: x,y

Inequality: x<0 || x>10000 || y<-9000 || y>9000

(a) Custom ESCAPE blocks were created in Simulink to provide the non-model input for SimplexGen.



(b) Simulink blocks are used to specify the system dynamics.

Fig. 8. The ESCAPE toolkit uses Simulink/Stateflow as a front end.

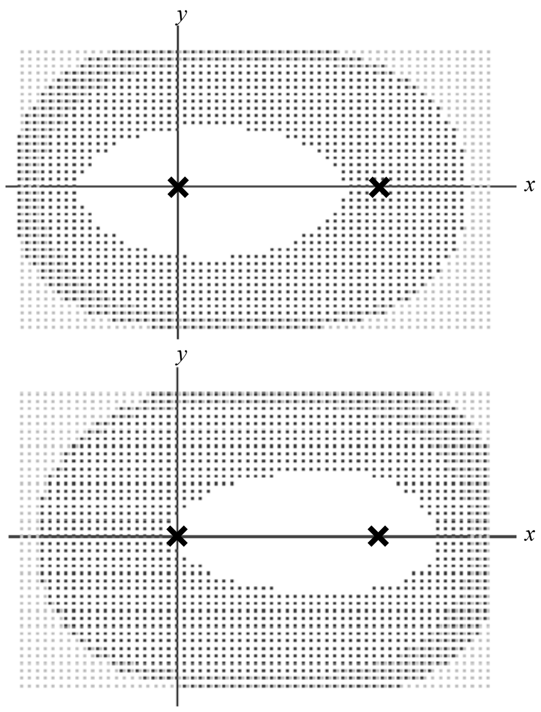


Fig. 9. The switching set of points output by our algorithm is shown projected onto fixed v and θ dimensions. Here, a single segment between two waypoints (crosses) is analyzed. The light gray states indicate the unsafe regions, whereas the dark gray states are states where the safety controller must be immediately used in order to maintain safety. The top figure is for $v = 1000mm/s$ and bottom figure is for $v = -1500mm/s$. In both figures, θ is fixed at 0.

switching set for two values of v and a fixed θ. By changing the values of v and θ, we can verify our intuition about the switching set: that going away from the waypoint-connecting line segment at high velocity will more quickly switch to the safety controller.

V. Related Work

There are several algorithms and tools for computing reachable states and approximate reachable states for hybrid systems and a complete survey of these techniques is beyond the scope of this paper [12], [13], [14], [15], [16], [17], [18], [19], [20].

Work related to the PESSOA [21] tool for synthesizing embedded controllers is similar in spirit to our work. PESSOA generates a finite state abstraction of a given system and uses these abstractions for synthesizing controllers that are guaranteed certain restricted class of LTL properties such as $\Box\varphi$, $\Diamond\varphi$, $\Diamond\Box\varphi$ and $\Diamond\Box\varphi \wedge \Box\varphi'$. The controller synthesis uses earlier techniques [22], [23], [24], [25]. The finite state abstractions used in PESSOA are approximate simulations [26], [27] of the original continuous system. In contrast, the finite state abstractions we use are simulations of the original system in the classical sense. Furthermore, our algorithm (and tool) can handle plants which are described with hybrid automata with nonlinear differential equations. To the best of our knowledge, PESSOA currently does not support such models.

Checkmate [14] is a tool for verifying control systems modeled as a hybrid automaton. Checkmate computes the reach set for a linear and non-linear systems by using the flow pipe approximation technique [18] to approximate the reachable sets by a sequence of convex polyhedra. Checkmate restricts the class of hybrid systems that can be verified to polyhedral-invariant hybrid systems. Our approach, however, can verify hybrid systems without this restriction.

The Simplex Architecture [2] has been applied as a sandboxing technique for various systems ranging from a fleet of remote-controlled cars [28], to a pacemaker [3], to a set of advanced aircraft maneuvers [29]. Some approaches rely on heuristics and testing to create the Simplex decision module [30]. In contrast, we propose verification based on system models. For a limited class of hybrid systems, we have previously proposed an automatic method to verify the Simplex decision module [4]. The algorithm presented in this paper accepts more general models than our previous work, and provides ways to increase reachability accuracy. Additionally, we provide the ESCAPE toolkit to synthesize automatically the decision module, rather than only checking a decision module after it has been created, as in our previous work.

VI. Conclusions

In this paper, we have addressed the problem of verifying properties of cyber-physical systems with partly unknown software components. The main technique to achieve this was to sandbox unverified components by using the Simplex Architecture. In this Simplex Architecture, however, the decision module, which switches between the unverified complex controller and verified safety controller, must be verified as correct.

This paper presented a hybrid-systems reachability approach for creating this Simplex switching logic. The low-level algorithm was shown to be more general than previous approaches, with three techniques provided to improve accuracy. The approach was integrated into an end-to-end toolkit where Simulink/Stateflow models of the system can be used to create source code for the Simplex decision module, and then the toolkit was successfully applied to a skid-steer vehicle system model.

The cyber-physical system sandbox approach presented in this paper, however, should not be regarded as a silver bullet that mitigates all possible system faults. For example, hardware failure is not considered in our designs, so practical systems may need to also incorporate redundancy. Additionally, as with all model-based verification techniques, the result of the verification is only as accurate as the models themselves. Runtime monitoring may be necessary to ensure that the deployed system behaves in accordance with the verification model. The proposed algorithm is remains inapplicable for hybrid systems with circular explicit derivative dependencies, although this will be investigated as future work. Additionally, we will explore techniques to reduce the computation time of the method, which due to the accuracy-increasing rules, can lead to improved accuracy of the reachability computation, which will construct a less pessimistic decision module.

References

[1] C. Reis, A. Barth, and C. Pizano, "Browser security: lessons from google chrome," *Commun. ACM*, vol. 52, pp. 45–49, August 2009.

[2] L. Sha, "Using simplicity to control complexity," *IEEE Softw.*, vol. 18, no. 4, pp. 20–28, 2001.

[3] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, "The system-level simplex architecture for improved real-time embedded system safety," in *RTAS '09: Proceedings of the 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 99–107.

[4] S. Bak, A. Greer, and S. Mitra, "Hybrid cyberphysical system verification with simplex using discrete abstractions," *Real-Time and Embedded Technology and Applications Symposium, IEEE*, vol. 0, pp. 143–152, 2010.

[5] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical Computer Science*, vol. 138, no. 1, pp. 3–34, 1995. [Online]. Available: citeseer.nj.nec.com/alur95algorithmic.html

[6] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, *The Theory of Timed I/O Automata*, ser. Synthesis Lectures on Computer Science. Morgan Claypool, November 2005, also available as Technical Report MIT-LCS-TR-917.

[7] S. Mitra, "A verification framework for hybrid systems," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA 02139, September 2007. [Online]. Available: Available at http://people.csail.mit.edu/mitras/research/thesis.pdf

[8] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" in *ACM Symposium on Theory of Computing*, 1995, pp. 373–382. [Online]. Available: citeseer.nj.nec.com/henzinger95whats.html

[9] G. Lafferriere, G. J. Pappas, and S. Yovine, "A new class of decidable hybrid systems," in *HSCC '99: Proceedings of the Second International Workshop on Hybrid Systems*. London, UK: Springer-Verlag, 1999, pp. 137–151.

[10] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[11] S. Bak, K. Manamcheri, S. Mitra, and M. Caccamo, "Sandbox generation for hybrid cyber-physical systems," www.cs.uiuc.edu/homes/sbak2/files/simplex_tr_2010.pdf, 2010.

[12] G. Frehse, "Phaver: Algorithmic verification of hybrid systems past hytech." in *HSCC*, ser. LNCS, M. Morari and L. Thiele, Eds., vol. 3414. Springer, 2005, pp. 258–273.

[13] E. Asarin, O. Bournez, T. Dang, and O. Maler, "Approximate reachability analysis of piecewise-linear dynamical systems," in *Hybrid Systems: Computation and Control*, ser. LNCS, B. Krogh and N. Lynch, Eds., vol. 1790. Hybrid Systems: Computation and Control, 2000, pp. 20–31.

[14] J. Kapinski and B. H. Krogh, "A new tool for verifying computer controlled systems," in *IEEE Conference on Computer-Aided Control System Design*, pp. 98–103.

[15] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "Hytech: A model checker for hybrid systems," in *Computer Aided Verification (CAV '97)*, ser. LNCS, vol. 1254, 1997, pp. 460–483.

[16] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL in 1995," in *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1996, pp. 431–434.

[17] K.-D. Kim, S. Mitra, and P.R.Kumar, "Computing bounded epsiolon-reach set with finite precision computations for a class of linear hybrid automata."

[18] A. Chutinan and B. H. Krogh, "Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations," *LNCS*, vol. 1569, pp. 76–91, 1999. [Online]. Available: citeseer.nj.nec.com/chutinan99verification.html

[19] Z. Han and B. H. Krogh, "Reachability analysis of large-scale affine systems using low-dimensional polytopes," in *HSCC*, pp. 287–301.

[20] A. Chutinan and B. Krogh, "Computational techniques for hybrid system verification," in *Auto-matic Control, IEEE Transactions*, pp. 64–75.

[21] M. M. Jr., A. Davitian, and P. Tabuada, "Pessoa: A tool for embedded controller synthesis," in *Proceedings of the 22nd International Conference on Computer Aided Verification*, 2010.

[22] R. Kumar and V. Garg, "Modeling and control of logical discrete event systems." Kluwer Academic Publishers, 1995.

[23] C. Cassandras and S. Lafortune, "Introduction to discrete event systems." Kluwer Academic Publishers, 1999.

[24] L. de Alfaro, T. A. Henzinger, and R. Majumdar, "Symbolic algorithms for infinite-state games," in *CONCUR 01: Concurrency Theory, 12th International Conference*, 2001.

[25] R. Alur, T. Henzinger, O. Kupferman, and M. Vardi, "Alternating refinement relations," in *Proceedings of the 8th International Conference on Concurrence Theory*. Springer, 1998.

[26] Paulo Tabuada, *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, 2009.

[27] A. Girard and G. J. Pappas, "Approximation metrics for discrete and continuous systems," in *IEEE Transactions on Automatic Control*, 2005.

[28] T. L. Crenshaw, E. Gunter, C. L. Robinson, L. Sha, and P. R. Kumar, "The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures," in *RTSS '07*, Washington, DC, USA, 2007, pp. 400–412.

[29] D. Seto, E. Ferreira, and T. F. Marz, "Case study: Development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis)," Technical Report Cmu/ sei-99-Tr-020. [Online]. Available: citeseer.ist.psu.edu/ 606539.html

[30] E. D. Ferreira and B. H. Krogh, "Switching controllers based on neural network estimates of stability regions and controller performance," in *Lecture Notes on Computer Science, Special Issue: Hybrid Systems VI*. Springer Verlag, 1999.