Runtime Verification for Hybrid Analysis Tools

Luan Viet Nguyen[1], Christian Schilling[2], Sergiy Bogomolov[3], and Taylor T. Johnson[1]

[1] University of Texas at Arlington, USA
[2] Albert-Ludwigs-Universität Freiburg, Germany
[3] IST Austria, Austria

Abstract. In this paper, we present the first steps toward a runtime verification framework for monitoring hybrid and cyber-physical systems (CPS) development tools based on randomized differential testing. The development tools include hybrid systems reachability analysis tools, model-based development environments like Simulink/Stateflow (SLSF), etc. First, hybrid automaton models are randomly generated. Next, these hybrid automaton models are translated to a number of different tools (currently, SpaceEx, dReach, Flow*, HyCreate, and the MathWorks' Simulink/Stateflow) using the HyST source transformation and translation tool. Then, the hybrid automaton models are executed in the different tools and their outputs are parsed. The final step is the differential comparison: the outputs of the different tools are compared. If the results do not agree (in the sense that an analysis or verification result from one tool does not match that of another tool, ignoring timeouts, etc.), a candidate bug is flagged and the model is saved for future analysis by the user. The process then repeats and the monitoring continues until the user terminates the process. We present preliminary results that have been useful in identifying a few bugs in the analysis methods of different development tools, and in an earlier version of HyST.

1 Introduction

Runtime verification is an approach to ensure the correctness and reliability of a system during its execution. It can check and analyze executions of a system under scrutiny that violate or satisfy a given correctness property by using a decision procedure called a monitor. A monitor can also be considered as a device that can read finite traces and output a truth value derived from a truth domain [3]. Runtime verification can be used broadly for many purposes such as debugging, testing, verification, validation, fault protection, and online system repair. In this paper, we describe a preliminary work toward a randomized differential testing framework [5] that may be used as a runtime monitor for various components (from parsers to analysis algorithms) in hybrid and CPS analysis tools such as SpaceEx, dReach, Flow*, HyCreate and the Mathworks' Simulink/Stateflow (SLSF). A test subject is the hybrid automaton randomly generated in the input format for SpaceEx using a prototype tool called HyRG [4][4], which is then translated to other formats including dReach,

[4] The tool and examples are available online: http://www.verivital.com/hyrg/

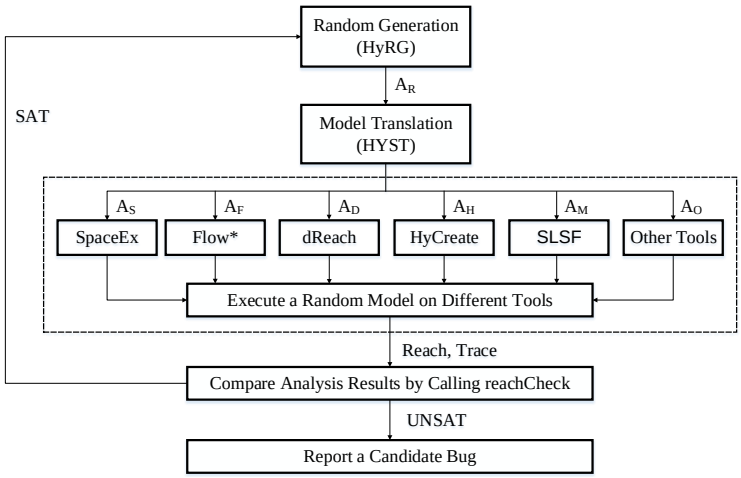


Fig. 1: Overview of monitoring framework for hybrid systems analysis tools with randomized differential testing.

Flow*, HyCreate and SLSF using the HyST model transformation tool [1]. Our contributions include (a) the first steps toward a randomized differential testing framework to monitor CPS development and verification tools, and (b) identifying some bugs in existing tools, including a semantic difference between SpaceEx and SLSF that we did not know about and some soundness bugs in the verification tools that have been corrected by the tool authors [1].

2 Monitoring with Randomized Differential Testing

We first describe how the hybrid systems are randomly generated in HyRG so they have diverse continuous and discrete behaviors. We then analyze these examples with different hybrid systems development and verification tools, and then compare their outputs to identify possible bugs in the tools. Figure 1 shows the overview of our framework for randomized differential testing to monitor hybrid systems development tools. First, a hybrid automaton $\mathsf{A_R}$ is randomly generated by HyRG, then $\mathsf{A_R}$ is translated using HyST to equivalent automata in different tools' formats, denoted $\mathsf{A_S}$, $\mathsf{A_F}$, $\mathsf{A_D}$, $\mathsf{A_H}$, $\mathsf{A_M}$, and $\mathsf{A_O}$. Next, the automata can be analyzed using the different tools, such as SpaceEx, Flow*, dReach, and HyCreate, or simulated in SLSF. Then we compare all analysis results by using a function reachCheck shown in Figure 2.

The reachCheck function has three inputs: Reach, Trace, and β, where β is the reachability analysis and simulation time bound. Reach is a list of sets of time-bounded reachable states computed by different tools (e.g., the output of SpaceEx, Flow*, etc.). Each set of reachable states, $\mathcal{R}(t)$, is the set of states that may be visited by following the model's trajectories and transitions, for any time $t \in [0, \beta]$. That is, for a given time t, $\mathcal{R}(t)$ is the set of states reachable at time t (sometimes referred to as a time-slice). The input Trace is a set of all simulation traces produced by SLSF up to a maximum simulation time β.

```
1  function reachCheck(Reach, Trace, β)
     foreach set of reachable states R_i in Reach
3      foreach set of reachable states R_j in Reach
         if i ≠ j and ∀t ∈ [0, β] R_i(t) ∧ R_j(t) is UNSAT then return UNSAT
5    foreach execution trace T_k in Trace
       if ∀t ∈ [0, β] T_k(t) ∧ R_i(t) is UNSAT then return UNSAT
7  return SAT
```

Fig. 2: reachCheck checks whether the set of reachable states and traces computed by different tools overlap (have non-empty intersection) at every time instant.

The reachCheck function can check whether the reachable states or simulation traces computed by different tools at each time have non-empty intersections. Although all of the reachable states and simulation traces are described in different formats such as support functions, Satisfiability Modulo Theories (SMT) formulas, convex sets, etc., there still exists an equivalence among them. For example, reachable sets computed by SpaceEx's LGG algorithm are a representation of convex sets (support functions), but these could be compared to the Taylor models of Flow*. If the reachable sets computed by different tools have a non-empty intersection (pairwise over all the tools), then reachCheck will return SAT, and the monitoring continues by generating a different random model. Otherwise, there is possibly a bug in the HyST translation or the verification tools. For the simulation traces, if some portions of a trace are not contained in any of the reachable states, reachCheck will return UNSAT and there is again possibly a bug in HyST, the verification tools, or SLSF. Obviously all these tools have numerous parameters, so numerical issues, accuracies, etc. must be taken into account by the user to determine whether a candidate bug is real.

Next, we define the structure of a hybrid automaton [2] and then summarize the random generation framework.

Definition 1. *A hybrid automaton* $\mathcal{H}$ *is a tuple,* $\mathcal{H} \triangleq \langle \texttt{Loc}, \texttt{Var}, \texttt{Flow}, \texttt{Inv}, \texttt{Trans}, \texttt{Init} \rangle$*, consisting of following components: (a)* Loc*: a finite set of discrete locations. (b)* Var*: a finite set of* n *continuous, real-valued variables, where* $\forall\, x \in \texttt{Var}$*,* $v(x) \in \mathcal{R}$ *and* $v(x)$ *is a valuation—a function mapping* x *to a point in its type—here,* $\mathcal{R}$*; and* $\mathcal{Q} \triangleq \texttt{Loc} \times \mathcal{R}^n$ *is the state space. (c)* Inv*: a finite set of invariants for each discrete location,* $\forall\, l \in \texttt{Loc}$*,* $\texttt{Inv}(l) \subseteq \mathcal{R}^n$*. (d)* Flow*: a finite set of derivatives for each continuous variable* $x \in \texttt{Var}$*, and* $\texttt{Flow}(l, x) \subseteq \mathcal{R}^n$ *that describes the continuous dynamics in each location* $l \in \texttt{Loc}$*. (e)* Trans*: a finite set of transitions between locations; each transition is a tuple* $\tau = \langle \texttt{src}, \texttt{dst}, \texttt{Grd}, \texttt{Rst} \rangle$*, which can be taken from source location* src *to destination location* dst *when a guard condition* Grd *is satisfied, and a state is updated by an update map* Rst*. (f)* Init*: an initial condition,* $\texttt{Init} \subseteq \mathcal{Q}$*.*

We denote a hybrid automaton that has been randomly generated by $\mathsf{A_R}$. We randomly generate each syntactic component of the automaton $\mathsf{A_R}$. Rather than picking only random matrices and vectors for the affine functions used in flows, guards, invariants, assignments, etc., we instead partition these affine functions into classes. While we assume affine functions making up the automaton, the general method may be extended to nonlinear functions. We highlight that *all* structural components of the automaton are selected randomly (i.e., the tran-

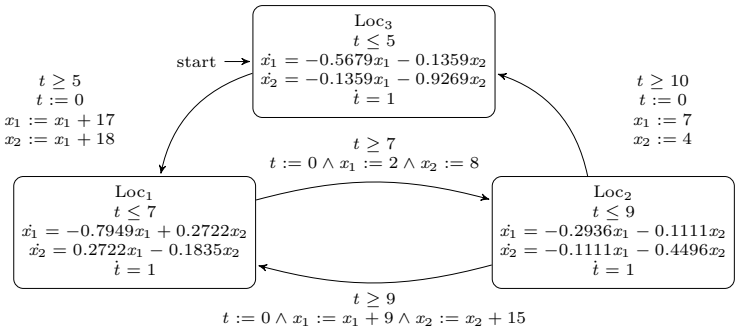


Fig. 3: An example hybrid automaton $\mathsf{A_R}$ with time-dependent switching that was randomly generated using HyRG.

sitions and continuous dynamics), and are not simply parameters. For brevity, we do not describe in detail the random generation of all structural components here, but refer to our other preliminary results [4].

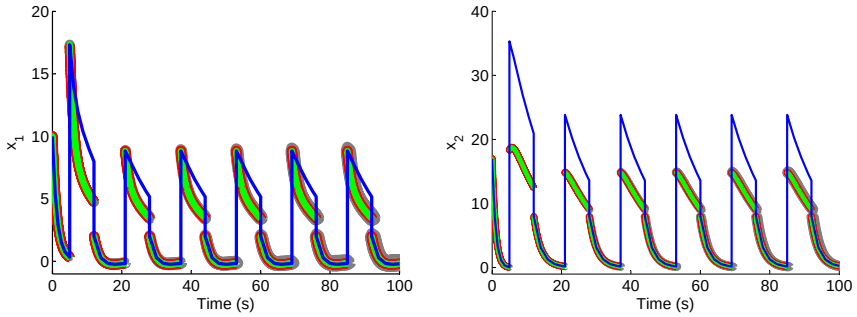
3 Preliminary Experimental Results

We evaluate our preliminary[5] monitoring framework in several scenarios to compare differences among several hybrid systems verification tools including SpaceEx, dReach, and Flow*, as well as SLSF simulation. Consider a randomly generated hybrid automaton $\mathsf{A_R}$ shown in Figure 3. The initial state of $\mathsf{A_R}$ is Loc_3, and the randomly generated initial values of its variables are respectively $x_1 = 10$, $x_2 = 17$, and $t = 0$. Note that $\mathsf{A_R}$ is nondeterministic. The results of simulations and reachability analysis on $\mathsf{A_R}$ are shown in Figure 4. The reachable states restricted to x_1 and x_2 computed by Flow* as well as the STC and LGG algorithms in SpaceEx do not contain a simulation trace for a supposedly equivalent SLSF model created using HyST when $\mathsf{A_R}$ takes a transition. In this case, the reachCheck function in Figure 2 will return UNSAT. This happens because of semantic differences in resets among Flow*, SpaceEx, and SLSF. In SLSF, the variables x_1 and x_2 are updated sequentially, so that x_1 will first be updated to a new value, and then x_2 will be updated using the new (already updated) value of x_1. However, these variables are updated concurrently in Flow* and SpaceEx [2], so x_2 will be updated by using the previous value of x_1. Based on this, we fixed this translation error in HyST.

4 Conclusion and Future Work

In this paper, we describe our preliminary results toward building a randomized differential testing framework to monitor hybrid and CPS development tools like SLSF and verification tools like SpaceEx, dReach, Flow*, etc. Our preliminary results include identifying semantic mismatches between tools automatically that have been integrated into subsequent versions of HyST. Additionally, we have found a couple bugs in some of the verification tools that have been corrected by

[5] Some of the steps are currently manual, particularly the parsing of reachable states and comparison thereof, but the generation with HyRG and translation with HyST is fully automatic.



(a) (b)

Fig. 4: SLSF simulation (blue), reachable states computed by Flow* (green), SpaceEx's STC algorithm (red), and SpaceEx's LGG algorithm (gray) for $\mathsf{A_R}$ showing x_1 and x_2 versus time, respectively. The SLSF simulation traces and the reachable states computed by Flow*, SpaceEx's LGG and STC algorithms do not line up (i.e., have an empty intersection) at some points in time (so reachCheck returns UNSAT) due to a semantic difference.

the tool authors. Based on our promising preliminary results, we plan to fully automate every step of the framework in the future.

Acknowledgments

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-15-1-0105. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government. This work was also partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center Automatic Verification and Analysis of Complex Systems (SFB/TR 14 AVACS, http://www.avacs.org/), by the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award).

References

1. Bak, S., Bogomolov, S., Johnson, T.T.: HyST: A source transformation and translation tool for hybrid automaton models. In: Proc. of the 18th Intl. Conf. on Hybrid Systems: Computation and Control (HSCC). ACM (2015)
2. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Computer Aided Verification (CAV). LNCS, Springer (2011)
3. Leucker, M., Schallhart, C.: A brief account of runtime verification. Journal of Logic and Algebraic Programming 78(5), 293–303 (May 2009)
4. Nguyen, L.V., Schilling, C., Bogomolov, S., Johnson, T.T.: Poster: Hyrg: A random generation tool for affine hybrid automata. In: 18th International Conference on Hybrid Systems: Computation and Control (HSCC 2015) (2015)
5. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 283–294. PLDI '11, ACM, New York, NY, USA (2011)