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. We present the first steps towards a runtime verification framework for monitoring hybrid and cyber-physical systems (CPS) development tools—such as hybrid systems reachability analysis tools, model-based development environments like Simulink/Stateflow (SLSF), etc.—based on randomized differential testing. First, hybrid automaton models are randomly generated. Next, these hybrid automaton models are translated to a number of different tools (currently, dReach, Flow*, HyCreate, SpaceEx, 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 random differential testing approach for monitoring continues until the user terminates the process. We present preliminary results that have been useful in identifying several bugs in the analysis methods of the 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 outputs a truth value derived from a truth domain [3]. Runtime verfication can be used broadly in many purposes such as debugging, testing, verification, validation, fault protection, and online system repair. In this paper, we describe preliminary work towards 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 cyber-physical systems (CPS) analysis tools such as SpaceEx, dReach, Flow*, and Mathworks' Simulink/Stateflow (SLSF). A test subject is the hybrid automaton randomly generated in the input format for SpaceEx using a prototype tool called

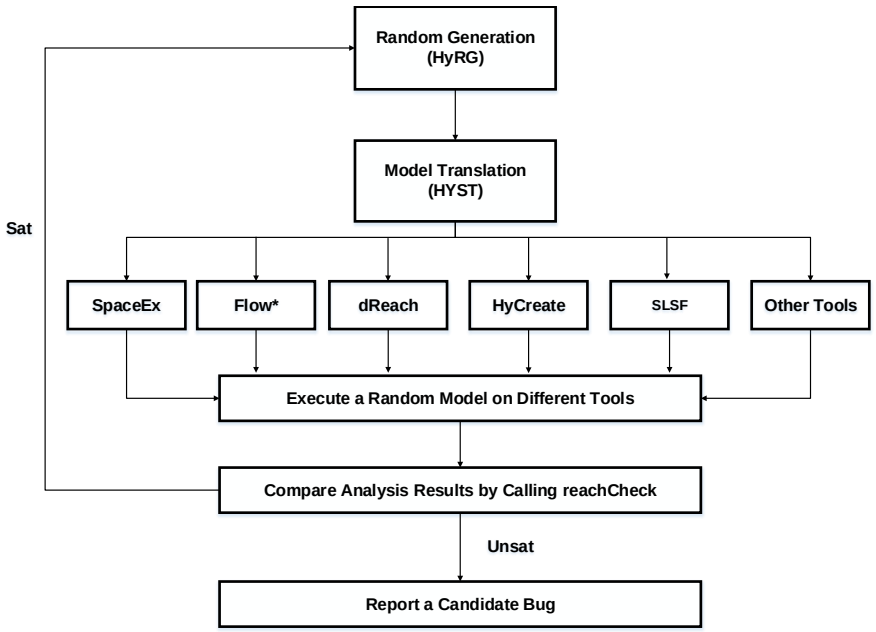


Fig. 1: Overview of monitoring framework for hybrid systems analysis tools with randomized differential testing.

HyRG [4][4], which is then translated to other formats including dReach, Flow*, and SLSF using the HyST model transformation tool [1]. Our contributions include the first steps toward a randomized differential testing framework to monitor CPS development and verification tools, and identifying several bugs in existing tools, including a semantic difference between SpaceEx and SLSF that we did not know about and a couple bugs in Flow* and dReach that have been corrected by the tool authors (for more details, see [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 random hybrid automaton randomly generated by HyRG will be translated to different formats of other tools with HyST. Next, we analyze the automaton with different verification tools such as SpaceEx, Flow*, dReach, and HyCreate, and simulate it with SLSF. Then we compare all analysis results by using a function reachCheck shown in Figure 3. The reachCheck function has an input Reach, which is a list

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

```
1  function randHA(m, n)
     A_R ← ∅
3    A_R.Var ← { x_1, ..., x_n }   // generate state variables
     {A_R.Loc, A_R.Trans} ← discreteStructure(m)   // gen. locations and transitions
5    foreach location l in A_R.Loc
       l.flow ← randFlow(A_R.Var)  // generate flows over state variables
7      l.inv ← randInv(A_R.Var)  // generate an invariant over state variables
       A_R.Flow ← A_R.Flow ∪ {l.flow}
9      A_R.Inv ← A_R.Inv ∪ {l.inv}
       foreach transition t in A_R.Trans
11       t.grd ← randGrd(A_R.Var)  // gen. guard condition over state variables
         t.rst ← randRst(A_R.Var)  // gen. update action over state variables
13       A_R.Trans.t ← t    // add guard and update to transition
     A_R.Init ← randInit(m, A_R.Var)   // generate an initial condition
15   return A_R
```

Fig. 2: Pseudo-code overview of HyRG method to randomly generate hybrid automata. The output hybrid automaton $\mathcal{A}_R$ is generated as a tuple of random locations, flows, invariants, transitions, guards, updates, and initial conditions.

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, at a given time $t \in [0, \beta]$, where β is the time bound. 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. The reachCheck function can check whether the reachable states or simulation traces computed by different tools at each time have non-empty intersections. If the reachable sets computed by these 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, the verification tools, or in SLSF. For the simulation traces, if some portion of a trace is not contained in any of the reachable states, reachCheck will return UNSAT. 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.

We define the structure of a hybrid automaton [2] and then describe a few details of the framework

Definition 1. *A hybrid automaton $\mathcal{H}$ is a tuple, $\mathcal{H} \triangleq \langle$*`Loc`*,* `Var`*,* `Flow`*,* `Inv`*,* `Trans`*,* `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 $\forall\, x \in$* `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 $\mathcal{Q} \triangleq$* `Loc` *$\times\, \mathcal{R}^n$ is the state space. (c)* `Inv`*: a finite set of invariants for each discrete location, $\forall\, l \in$* `Loc`*, $\texttt{Inv}(l) \subseteq \mathcal{R}^n$. (d)* `Flow`*: a finite set of derivatives for each continuous variable $x \in$* `Var`*, and $\texttt{Flow}(l, x) \subseteq \mathcal{R}^n$ that describes the continuous dynamics in each location $l \in$* `Loc`*. (e)* `Trans`*: a finite set of transitions between locations; each transition is a tuple $\tau = \langle$*`src`*,* `dst`*,* `Grd`*,* `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, $\texttt{Init} \subseteq \mathcal{Q}$.*

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

Fig. 3: reachCheck checks whether the set of reachable states and traces computed by different tools overlap (have non-empty intersection) at every time instant.

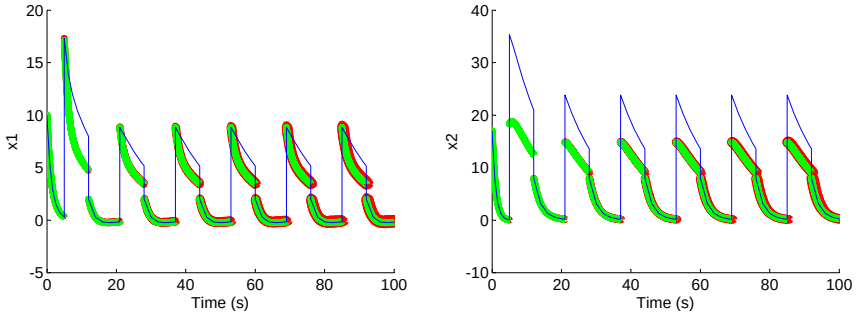
We denote a hybrid automaton that has been randomly generated by $\mathcal{A}_R$. The various syntactic components (Definition 1) of $\mathcal{A}_R$ are randomly generated as shown in Figure 2. We use the dot (.) notation to refer to different components of tuples, e.g., $\mathcal{A}_R.\texttt{Loc}$ refers to the set of locations $\texttt{Loc}$ of $\mathcal{A}_R$.

We randomly generate each syntactic component of the automaton to generate $\mathcal{A}_R$. The inputs include a number of locations m, and a number of variables n. First, we generate a set of state variables $\mathcal{A}_R.\texttt{Var} = \{x_1, \ldots, x_n\}$ (line 3). Next, we randomly generate sets of locations $\mathcal{A}_R.\texttt{Loc}$ and transitions $\mathcal{A}_R.\texttt{Trans}$ based on an arbitrary discrete structure (line 4). 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 interesting classes. While we assume affine functions making up the automaton, the general method may be extended to nonlinear functions. Next, for each location $l \in \mathcal{A}_R.\texttt{Loc}$ (line 5), we randomly generate its flow $l.\mathit{flow}$ (line 6) and invariant $l.\mathit{inv}$ (line 7) over the state variables $\mathcal{A}_R.\texttt{Var}$. Next, we iterate over each transition $t \in \mathcal{A}_R.\texttt{Trans}$ (line 10) to randomly generate a guard condition $t.\mathit{grd}$ and an update $t.\mathit{rst}$ as expressions over $\mathcal{A}_R.\texttt{Var}$ (lines 11 through 12). Finally, we generate a random initial condition $\mathcal{A}_R.\texttt{Init}$ for $\mathcal{A}_R$ (line 14). We highlight that *all* structural components of the automaton are selected randomly (i.e., the transitions 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 Appendix A.

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 $\mathcal{A}_R$ with the results shown in Figure 4 (details and the model are in Appendix A). The reachable states of x_1 and x_2 computed by the STC and LGG algorithms in SpaceEx do not contain a simulation trace for an equivalent SLSF model when $\mathcal{A}_R$ takes a transition. This happens because of semantic differences in resets between SpaceEx and SLSF. In SLSF, the variables x_1 and x_2 are updated in order, so that x_1 will be first updated to a new value, and then x_2 will be updated using the new (updated) value of x_1. However, these

[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 SpaceEx's LGG algorithm (red), and reachable states computed by SpaceEx's STC algorithm (green) for $\mathcal{A}_R$ showing x_1 and x_2 versus time, respectively. The SLSF simulation traces and the reachable states computed by SpaceEx's LGG and STC algorithms do not line up (i.e., have an empty intersection) at some time points (so reachCheck returns unsat) due to a semantic difference.

variables are updated concurrently in SpaceEx, so x_2 will be updated by using the previous value of x_1. Based on this, we fixed this translation error in HySST.

4 Conclusion and Future Work

In this paper, we describe our preliminary results toward building a randomized differential testing framework to monitor hybrid and cyber-physical systems (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 several bugs in Flow* and dReach that have been corrected by the tool authors. Based on our promising preliminary results, we plan to fully automate every step of the framework in the future.

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)

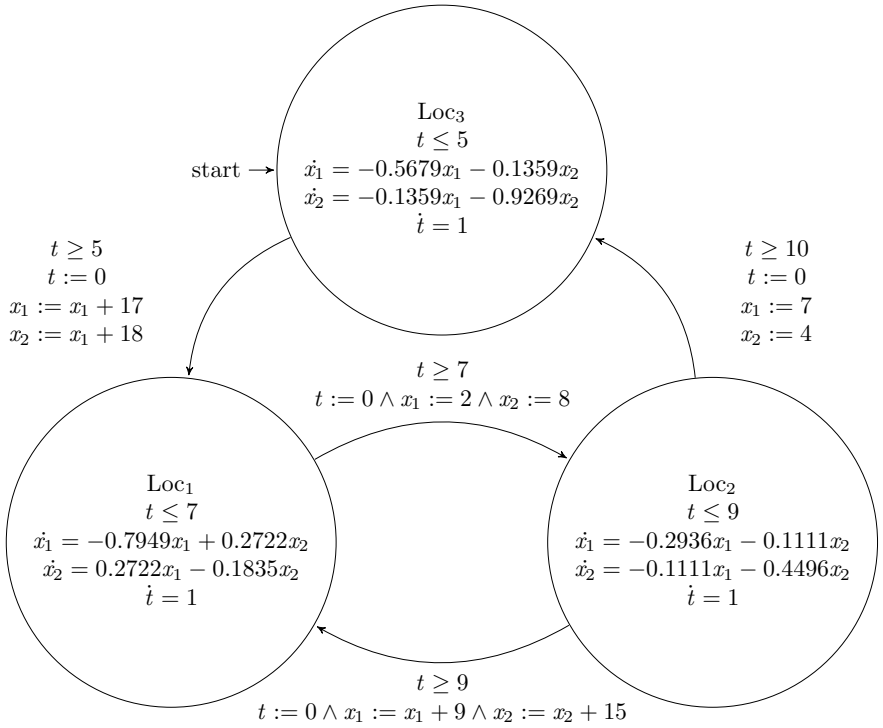


Fig. 5: An example of a random time-dependent switching hybrid automaton $\mathcal{A}_R$ randomly generated with HyRG using *negative definite* matrices.

A Appendix: Additional Random Generation Details

In this appendix, we describe the case study presented in the paper and additional details on the random generation methods used in HyRG, described previously in the high-level overview of Figure 2. The empirical results presented in Section 3 are for the automaton in Figure 5. The initial state of $\mathcal{A}_R$ is Loc_3, and the randomly initial values of its variables are respectively generated as $x_1 = 10$, $x_2 = 17$, and $t = 0$. $\mathcal{A}_R$ is nondeterministic. The continuous dynamics of each location in system $\mathcal{A}_R.\texttt{Loc}$ are randomly generated based on a *negative definite* matrix A, so it is exponentially asymptotically stable.

HyRG takes as inputs the options specified in Figure 2. As output, HyRG may produce a hybrid automaton in the input XML format of the SpaceEx toolAdditionally, HyRG is integrated with SLSF via a hybrid automaton to SLSF translation procedure.

Time-Dependent Switching Options A `Time` set includes all necessary components for randomly generating a time-dependent switched system

Definition 2. *A time-dependent switching set* `Time` *is a tuple* $\texttt{Time} \triangleq \langle \tau, \beta, \theta, \sigma, \phi \rangle$, *where (a)* τ*: a time variable. (b)* β*: a first order linear equation over the time variable,* $\beta \leftarrow \dot{\tau} = 1$*. (c)* θ*: an invariant randomly generated as* $\theta \leftarrow a\tau \leq b$*. (d)* σ*: a guard condition randomly generated as* $\sigma \leftarrow c\tau \leq d$*. (e)* ϕ*: an update*

map randomly generated as $\phi \leftarrow \tau = e$. *(f)* ι*: a initial condition generated as* $\iota \leftarrow \tau = f$*, where* a*,* b*,* c*,* d*,* e*,* f *are random constant numbers such that* $ab \geq 0$*,* $bc \geq 0$*, and* $e, f \geq 0$.

Randomly Generating Discrete Structure. The discrete structure of a hybrid automata is a set of locations and transition lines connected some pairs of locations. It can be randomly generated using random adjacency matrices.If a hybrid automata has a random m number of locations, so its transition graph is an $m \times m$ random adjacency matrix adjMatrix , whose elements equal to either 0 or 1. If an element adjMatrix$[i, j]$ is equal to 1, there is a transition from i^{th} location to j^{th} location. Otherwise, there is no connection between these two locations. An example of a discrete structure's graph randomly generated by a random adjacency matrix A_G is shown in Figure 6. If any diagonal element adjMatrix$[i, i]$ is equal to 1, the i^{th} location will be connected to itself. In other words, it has a self-loop transition. Moreover, a number of transitions can also be controlled by restricting the sum of rows and columns of adjacency matrix less than some arbitrary constants.

The pseudo-code for generating a random discrete structure by creating an arbitrary adjacency matrix shown in Figure 7—called from randHA (Figure 2, line 4). We first call the function randAdjMatrix (line 2) to get a random adjacency matrix adjMatrix. Next, we iterate over each row element i of an adjacency matrix adjMatrix (line 3), and then create a corresponding location l_i (line 5). For each row element i of adjMatrix , we iterate over each row element j of adjMatrix (line 6), and then generate a corresponding transition $t_{i,j}$ (line 8) when the value of adjMatrix$[i, j]$ is equal to one.

The pseudo-code of randomly generating an arbitrary adjacency matrix shown in Figure 8. A function randi$([0, 1], m)$ (line 6) generates an $m \times m$ random matrix whose elements are equal to either 0 or 1. We use a boolean variable *flag* to keep generating adjMatrix until we get our desired matrix (line 4), which provides at least one pair of ingoing and outgoing transitions for each location. We can generate this desired matrix by putting constraints on the sum of each row

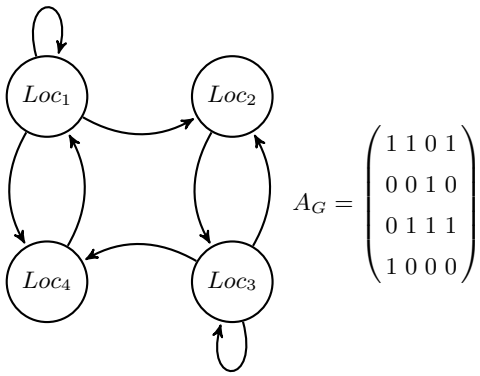


Fig. 6: An example of the transition graph of the hybrid automaton model randomly generated by the random adjacency matrix A_G.

```
    function discreteStructure(m)
2     adjMatrix ← randAdjMatrix(m, Opt)
      foreach row element i in adjMatrix
4       // generate a corresponding location
        A_R.Loc ← A_R.Loc ∪ {l_i}
6       foreach row element j in adjMatrix
          // generate a transition from location i to location j
8         if adjMatrix[i, j] = 1 then A_R.Trans ← A_R.Trans ∪ {t_{i,j}}
      return A_R.Loc, A_R.Trans
```

Fig. 7: Randomly generated discrete structure pseudo-code. The input is a number of locations m. And, the output are random sets of locations $\mathcal{A}_R$.Loc and transitions $\mathcal{A}_R$.Trans.

```
1   function randAdjMatrix(m, Opt)
      adjMatrix ← ∅
3     flag ← 1
      while flag = 1
5       flag ← 0
        adjMatrix ← randi([0, 1], m)   // Figure 6
7       foreach row element i in adjMatrix
          // without self-loop implementation, zero all diagonal elements
9         if Opt.F = ∅ then adjMatrix[i, i] ← 0
          // generate at least one ingoing transition
11        if ∑ adjMatrix[i, :] = 0 then flag ← 1
          // generate at least one outgoing transition
13      foreach column element j in adjMatrix
          if ∑ adjMatrix[:, j] = 0 then flag ← 1
15    return adjMatrix
```

Fig. 8: Randomly generated adjacency matrix pseudo-code. The input includes a number of locations m, and a set of options Opt. The output is an adjacency matrix adjMatrix.

```
1   function randFlow(n, A_R.Var, Opt, optFlw)
      X ← A_R.Var   // assign a vector of state variables
3     if Opt.F ≠ ∅ then randomly select ζ ∈ optFlw
        // return a random matrix for a corresponding random flow
5       A ← Λ(n, ζ)
        // generate a new continuous flow
7       flow ← {Ẋ = AX + B}
      return flow
```

Fig. 9: Randomly generated flow pseudo-code. The input includes the number of variables n, a set of state variables $\mathcal{A}_R$.Var, a set of options Opt, and a set of different classes of matrix's definiteness optFlw.

and column of adjMatrix (lines 7 through 14). Additionally, If we want to generate a random hybrid automaton without self-loop transition (line 9), we set all diagonal elements of adjMatrix equal to zero.

Randomly Generating Continuous Flow Dynamics. A randomly generated affine hybrid automaton $\mathcal{A}_R$ has continuous dynamics defined as $\dot{x} = Ax + B, x \in \mathcal{R}^n$, where n is a random number of state variables, x is an n-vector of state variables, and $\dot{x}$ is an n-vector of the derivatives of these variables w.r.t. time. Furthermore, A is an $n \times n$-matrix of real coefficients and B is an n-vector of real constants. We denote $\Psi(t)$ as a fundamental matrix of a linear system of differential equations $\dot{x} = Ax$, where t denotes time. Moreover, $\Psi(t) = e^{At}$

can be considered as one fundamental matrix of the system. By using the eigen-decomposition theorem, a matrix A can be written as $A = vDv^{-1}$, where D is an $n \times n$ diagonal matrix whose diagonal elements correspond to n eigenvalues λ_i of the matrix A, and v is an $n \times n$-matrix where each column v_i is a corresponding eigenvector of A. Note that v^{-1} is an $n \times n$ constant matrix and its determinant is non-zero. If $\Psi(t) = e^{At}$ is a fundamental matrix, so $\Psi(t) = e^{At}v = e^{vDv^{-1}t}v = ve^{Dt}v^{-1}v = ve^{Dt}$ is also a fundamental matrix. Therefore, the general solution of a system of differential equations $\dot{x} = Ax$ is $x(t) = ve^{Dt}C$, where C is an n-vector of real values determined by the initial conditions of $x(t)$. If $x(t_0) = x_0$ is a vector of initial conditions, then $C = \Psi(t_0)^{-1}x_0$. For linear systems, the continuous dynamics may be described as an exponential function of eigenvalues, and the eigen-decomposition theorem allows us to generate a random matrix A from sets of arbitrarily given eigenvalues and eigenvectors.

Suppose that A is a symmetric real matrix with all of its eigenvalues are real numbers. If A is considered as: (a) *positive definite* (*pd*), then all of its eigenvalues are positive, (b) *negative definite* (*nd*), then all of its eigenvalues are negative, (c) *semipositive definite*(*psd*), then all of its eigenvalues are non-negative, (d) *seminegative definite*(*nsd*), then all of its eigenvalues are non-positive, (e) and *indefinite* (*ind*), then its eigenvalues have both positive and negative values. More generally, if A is equal to its self-adjoin. Then A is a Hermitian matrix, and its definiteness is considered based on the real part of its eigenvalues

Definition 3. *Let $A \in \mathbb{C}^{n\times n}$ be an Hermitian matrix, and $U, U^* \in \mathbb{C}^n$ be a complex vector and its conjugate transpose vector respectively. (a) For every nonzero vector U (i) if $U^*AU > 0$, then A is pd (ii) if $U^*AU < 0$, then A is nd (b) For every vector U (i) if $U^*AU \geq 0$, then A is psd (ii) if $U^*AU \leq 0$, then A is nsd*

According to Definition 3, suppose that (λ, v) is an *eigenpair* of A, so $v^*Av = \lambda v^*v = \lambda$. Thus a sign of λ depends on a definiteness of A. For example, if A is *negative define*, then $\lambda = v^*Av < 0$ for all *eigenpairs* (λ, v) of A. In other words, a Hermitian matrix A is considered *negative define*. This type of Hermitian matrix is also considered as a Hurwitz matrix that has all negative real part eigenvaluesCorrespondingly, the continuous dynamic of each location in system $\mathcal{A}_R$ generated based this matrix will be exponentially asymptotically stableOtherwise, if A is randomly generated as a skew-Hamiltonian matrix, then all eigenvalues of A have only imaginary parts.

The pseudo-code of a randomly generated flow dynamic for each location $l \in \mathcal{A}_R.\texttt{Loc}$ shown in Figure 9—called from randHA(Figure 2, line 6). The inputs include a set of different classes of matrix's definiteness `optFlw` that includes all possible classes of flows for every location in $\mathcal{A}_R$. We define `optFlw` as a tuple $\texttt{optFlw} \triangleq \langle pd, nd, psd, nsd, ind\rangle$. For each definiteness $\zeta \in \texttt{optFlw}$, $\Lambda(n, \zeta)$ is a function that returns an $n \times n$ random matrix corresponding ζ. For randomly generating a flow of location l, we first generate the vector of state variable `X` (line 2). Next, we randomly select a different classes of definiteness in `optFlw`

```
    function randInv(d, A_R.Var, Opt, optInv)
2     if Opt.I ≠ ∅ then randomly select ρ ∈ optInv
        //returns a set of random linear inequalities
4       inv ← Γ(d, A_R.Var, ρ)
      return inv
```

Fig. 10: Randomly generated invariant pseudo-code. The input includes a dimension d of a invariant polytope, a set of variable x, a set of option choices `Opt`, and a set of different d dimensional polytopes `optInv`.

```
1   function randGrd(inv, A_R.Var, Opt)
      Opt.G ≠ ∅ then grd ← Ω(inv, A_R.Var)
3     return grd
```

Fig. 11: Randomly generated guard condition pseudo-code. The input are an invariant polytope *inv*, a set of option choices `Opt` and a set of variable $\mathcal{A}_R$.`Var`.

(line 3), and then assign a random matrix corresponding to this class of definiteness (line 5). The continuous dynamics *flow* is generated by the first order differential equation $\{\dot{\mathtt{X}} = A\mathtt{X} + B\}$ (line 7), where $\dot{\mathtt{X}}$ is an $n \times 1$ vector of the first derivatives of state variables $\mathtt{X}$, and B is an $n \times 1$ arbitrary constant vector.

Randomly Generating Invariants. An invariant for each location of $\mathcal{A}_R$ is randomly generated based on the concept of convex polytopes. Let $x \in \mathbb{R}^n$ is a vector of state variables of $\mathcal{A}_R$, then a convex polytope is defined as a solution set of a finite system of linear inequalities $Cx \leq D$ where C is an $k \times n$ constant matrix, k is a number of linear inequalities, D is either an $k \times 1$ vector of constants or symbolic expression algebra of state variables. Each linear inequality divides the whole space in two separately halves called a half-space.Suppose that we have an k number of half-spaces generated by an k random linear inequalities. An invariant $\mathtt{Inv} \in \mathbb{R}^d$ of a hybrid system $\mathcal{A}_R$ is an d dimensional convex polytope randomly generated as an intersection of k half-spaces. We investigate a polytope generated from system of linear inequalities, which is not full-dimensional. Then, there exists at least one state variable missing from all linear inequalities. Thus, this polytope contains a ray, and is unboundedAn unbounded polytope (*upo*) can be randomly generated as a slab between two arbitrary parallel hyperplanes, an arbitrary infinite prism, or an arbitrary infinite cone. On the other hand, we also investigate several bounded polytopes including: (a) d dimensional simplex polytope (*spo*): the convex hull of $d+1$ affinely independent points in $\mathbb{R}^d$, or an intersection of $d+1$ half-spaces. (b) d dimensional cubical polytope (*opo*): the family of polytopes that analogues to a cube, and is defined as an intersection of $2d$ half-spaces. (c) d dimensional cross polytope (*cpo*): the family of polytopes that analogues to a octahedron, and is defined as an intersection of $2d + 2$ half-spaces The pseudo-code of randomly generated invariant polytope for each location in $\mathcal{A}_R$ shown in Figure 10. If a location in $\mathcal{A}_R$ has an invariant, we randomly select one type of d dimensional polytope in `optInv` (line 2), and then assign a corresponding set of random linear inequalities to generate an arbitrary invariant *inv* (line 4).

```
1  function randRst(n, A_R.Var, Opt)
     X ← A_R.Var
3    Opt.R ≠ ∅ then randomly select ψ ∈ optRst
     rst ← {X = Ω(n, A_R.Var, ψ)}
5    return rst
```

Fig. 12: Randomly generated update map pseudo-code. The input are a number variables n, a set of option choices $\texttt{Opt}$ and a set of variable $\mathcal{A}_R.\texttt{Var}$.

```
1  function randInit(n, A_R.Var, Opt)
     X ← A_R.Var
3    init ← {X = rand(n, 1)}
     return init
```

Fig. 13: Randomly generated initial condition pseudo-code. The input are a number variables n, a set of option choices $\texttt{Opt}$ and a set of variable $\mathcal{A}_R.\texttt{Var}$.

Randomly Generating Guard Conditions. For each location $l \in \mathcal{A}_R.\texttt{Loc}$, its invariant inv is randomly generated as a d dimensional convex polytope P by the pseudo-code shown in Figure 10. If S is a random convex hull of any set of vertices of P, so S is considered as a d dimensional sub-polytope of PThen, a random outgoing guard condition of location l is a set of linear inequalities represented the complement between a vector space $\mathbb{R}^d$ and S. A function $\Omega(inv, \mathcal{A}_R.\texttt{Var})$ whose inputs are an invariant inv and a set of state variables $\mathcal{A}_R.\texttt{Var}$ returns a set of random linear inequalities $Jx \geq K$, where J, K are defined similar to C, and D respectively. The pseudo-code of randomly generated a guard condition for an outgoing transition of each location l shown in Figure 11. If there exists an outgoing transition from location l, then we will assign a corresponding set of random linear inequalities for its arbitrary guard condition grd by calling the Ω function (line 2).

Randomly Generating Update Map. A update map can be randomly generated by assigning either a random constant or an arbitrary symbolic expression algebra of state variables to each state variable in $\mathcal{A}_R.\texttt{Var}$. Suppose that a set of update map $\texttt{optRst}$ is a tuple $\texttt{optRst} \triangleq \langle const, symbo \rangle$. For each type of an update $\psi \in \texttt{optRst}$, $\Phi(n, \mathcal{A}_R.\texttt{Var}, \psi)$ is a function that returns an $n \times 1$ vector of random constants or symbolic expression algebra of state variables. Figure 12 shows the pseudo-code for randomly generating a update map. If any transition of system $\mathcal{A}_R$ has an update action, we first randomly select whether to update state variables to constants or assign them to any symbolic expression algebra of state variables (line 3). And then, we set an equality between a vector of state variables $\texttt{X}$ and a random vector returned by calling Φ function to be an update action rst (line 4).

Randomly Generating Initial Conditions. The pseudo-code for generating a random initial condition $init$ is shown in Figure 13. We use a random function $\texttt{rand}(n, 1)$ (line 3) to generate an $n \times 1$ vector of constants, and then assign it to a vector of state variables $\texttt{X}$.