HyRG: A Random Generation Tool for Affine Hybrid Automata

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

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

Abstract. In this paper, we describe methods for randomly generating hybrid automata with affine differential equations, invariants, guards, and updates. Selecting an arbitrary affine function from the set of all affine functions results in a low likelihood of generating hybrid automata with diverse and interesting behaviors, as there are an uncountable number of elements in the set of all affine functions. Instead, we partition the set of all affine functions into potentially interesting classes and randomly select random elements from these classes. For example, we partition the set of all affine differential equations by using restrictions on eigenvalues such as those that yield stable, unstable, etc. equilibrium points. We partition the components describing discrete behavior (guards, updates, and invariants) to allow either time-dependent or state-dependent switching, and in particular provide the ability to generate subclasses of piecewise-affine hybrid automata. Our preliminary experimental results with a prototype tool called HyRG (Hybrid Random Generator) illustrate the feasibility of this generation method to automatically create standard hybrid automaton examples like the bouncing ball and thermostat with acceptable likelihoods.

1 Introduction

In this paper, we describe methods for randomly generating hybrid automata with affine (linear) differential equations, invariants, guards, and updates implemented in a prototype tool called HyRG. While random generation of affine vector fields (i.e., continuous linear systems, or a hybrid automaton with one location and no transitions) has been used to evaluate reachability algorithms [1,2], to the best of our knowledge, there has been no effort to randomly generate hybrid automata with more complex discrete structure. Additionally, existing methods for generating random continuous linear systems are relatively unsophisticated.[3] We highlight there are many tools and methods for random program generation in various languages (C, Java, etc.) [16]. Random generation of models is useful for: (a) evaluating reachability algorithms, (b) testing various components (from parsers to analysis algorithms) in hybrid analysis tools,

[3] For instance, MathWorks Matlab includes a function `rss` to generate random linear systems, http://www.mathworks.com/help/control/ref/rss.html.

```
1  function randHA(m, n, d, Time, Opt)
     A_R ← ∅
3    // generate state variables
     A_R.Var ← { x_1, ..., x_n}
5    // generate sets of locations and transitions
     {A_R.Loc, A_R.Trans} ← discreteStructure(m, Opt)
7    foreach location l in A_R.Loc
       // generate a flow over state variables
9      l.flow ← randFlow(n, A_R.Var, Opt)
       // generate time-dependent switched systems
11     if Opt.T ≠ ∅ then l.flow ← l.flow ∪ {Time.β}  // add time-flow
         l.inv ← Time.θ  // generate an invariant over time
13     // generate state-dependent switched systems
       else
15       // generate an invariant over state variables
         l.inv ← randInv(d, A_R.Var, Opt)
17     A_R.Flow ← A_R.Flow ∪ {l.flow}
       A_R.Inv ← A_R.Inv ∪ {l.inv}
19     foreach transition t in A_R.Trans
         if Opt.T ≠ ∅ then t.grd ← Time.σ // generate a time-guard
21         t.rst ← Time.φ  // generate a time-reset
         else
23         // generate a guard condition over state variables
           t.grd ← randGrd(l.inv, A_R.Var, Opt)
25         // generate an update action over state variables
           t.rst ← randRst(n, A_R.Var, Opt)
27       // overwrite transition with associated guard and update
         A_R.Trans.t ← t
29   // randomly generate an initial condition
     A_R.Init ← randInit(n, A_R.Var, Opt)
31   If Opt.T ≠ ∅ then A_R.Init ← A_R.Init ∧ Time.ι
     return A_R
```

Fig. 1: 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.

(c) testing translators from hybrid systems modeling languages to other tools like Mathworks Simulink/Stateflow (SLSF) [4], and (d) developing libraries of examples with diverse continuous and discrete behaviors.

2 HyRG: Randomly Generating Hybrid Automata

We begin by defining the structure of a hybrid automaton.

Definition 1. *A hybrid automaton [3, 7]* $\mathcal{H}$ *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* $\texttt{Flow}(l, x) \subseteq \mathcal{R}^n$ *describes the continuous dynamics of 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 $\mathcal{H}$ that has been randomly generated by $\mathcal{A}_R$. The various syntactic components (as described in Definition 1) of $\mathcal{A}_R$ are randomly generated as shown in Figure 1. We use the standard dot (.) notation to refer to different components of tuples, e.g., $\mathcal{A}_R.\texttt{Loc}$ refers to the set of locations `Loc` of $\mathcal{A}_R$. The inputs include: a number of locations m, a number of variables n, a dimension of invariant polytopes d, a set of options `Opt` (Definition 2) for generating different classes (i.e., to assign different classes of flows, invariants, guards, and updates) of $\mathcal{A}_R$, and a set of time-dependent switching options `Time` (Definition 3).

Definition 2. *An option set* `Opt` *is a tuple* $\texttt{Opt} \triangleq \langle \texttt{T}, \texttt{F}, \texttt{L}, \texttt{I}, \texttt{G}, \texttt{R} \rangle$ *of different tool options to generate: (a)* `T`*: time-dependent switched systems, (b)* `F`*: different classes of flows, (c)* `L`*: self-loop transitions, (d)* `I`*: different classes of invariants, (e)* `G`*: different classes of guards, and (f)* `R`*: different classes of update maps.*

We randomly generate each syntactic component of the automaton to generate $\mathcal{A}_R$. First, we generate a set of state variables $\mathcal{A}_R.\texttt{Var} = \{x_1, \ldots, x_n\}$ (line 4). 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 6). For each location $l \in \mathcal{A}_R.\texttt{Loc}$ (line 7), we randomly generate its flow $l.flow$ over the state variables $\mathcal{A}_R.\texttt{Var}$ (line 9). If $\mathcal{A}_R$ is a time-dependent switched system, then the first order differential equation of time variable $\texttt{Time}.\beta$ is added into $l.flow$ (line 11). An invariant $l.inv$ will be assigned as a linear inequality $\texttt{Time}.\theta$ of time variable τ (line 12). If $\mathcal{A}_R$ is a state-dependent system, $l.inv$ is only generated over $\mathcal{A}_R.\texttt{Var}$ (line 16). Different classes of random flows and invariants can be assigned for each location using additional options. Next, we iterate over each transition $t \in \mathcal{A}_R.\texttt{Trans}$ (line 19). If $\mathcal{A}_R$ is time-dependent, then a random linear inequality $\texttt{Time}.\sigma$ is assigned to a guard condition $t.grd$ (line 20). An update action $t.rst$ is created as an equation $\texttt{Time}.\phi$ (line 21). On the other hand, if $\mathcal{A}_R$ is state-dependent, then $t.grd$, and $t.rst$ are randomly generated over $\mathcal{A}_R.\texttt{Var}$ (lines 24 through 26). Finally, we generate a random initial condition $\mathcal{A}_R.\texttt{Init}$ for $\mathcal{A}_R$ (line 30). If we generate a random time-dependent switched system, we give an initial value for a time variable (line 31).

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 a vector of the first 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 parameters. A linear differential equation $\dot{x} = Ax$, using the eigen-decomposition theorem [10], may be expressed in the form: $\dot{x} = Ce^{\lambda}vx$ where λ is an n-vector containing the eigenvalues λ_i of matrix A, v is an $n \times n$-matrix in which each column v_i is a corresponding eigenvector of A, and C is an n-vector of coefficients. Continuous dynamics of linear systems may be described as an exponential function of eigenvalues, so different stability scenarios for each location of $\mathcal{A}_R$ can be generated by adding constraints over the eigenvalues of a randomly generated matrix A. Hence, we can randomly generate many classes of continuous dynamics based on different sets of given eigenvalues.

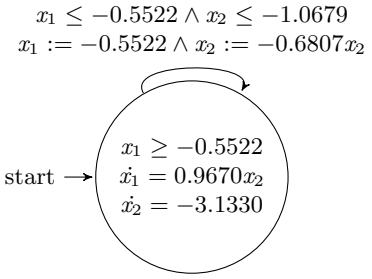


Fig. 2: A hybrid automaton randomly generated by HyRG with similar behavior to the bouncing ball (BB) example.

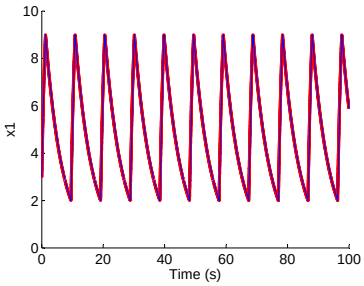
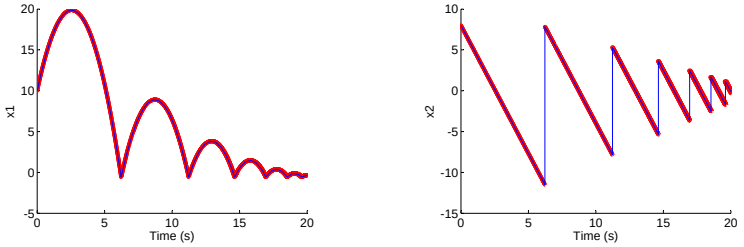


Fig. 3: The behavior of a randomly generated thermostat system.



(a) (b)

Fig. 4: The reachable states showing x_1 and x_2 computed by the LGG algorithm in SpaceEx (red) contain their SLSF simulation traces (blue) in Example 1.

3 HyRG Implementation and Experimental Results

We implemented the prototype HyRG tool in Java and Matlab and evaluated it in in several scenarios.[4]

Example 1. For the input $m = 1$ (number of locations) and $n = 2$ (number of variables), if we run the function randHA (in Figure 1) long enough, we can randomly generate a hybrid system $\mathcal{A}_R$ whose behavior is similar to the bouncing ball (BB) system. A randomly generated instance is shown in Figure 2. The initial values of its state variables are randomly generated as $x_1 = 10$, $x_2 = 8$. Figure 4 shows the SpaceEx reachability analysis and SLSF simulations of $\mathcal{A}_R$, where x_1 and x_2 represent the position and velocity of the BB system. We also randomly generated 100 models similar to the BB system and collected data of the generation process, shown in Table 1. As a result, for an automaton with a single location and two variables, on average we will generate a BB model after running the randHA function about 112 times. The average time to generate each BB model is 17.022 seconds.

Example 2. Again, if we run the function randHA (in Figure 1) long enough, this time with the input $m = 2$ (number of locations) and $n = 1$ (number of variables), we can randomly generate a hybrid system $\mathcal{A}_R$ whose behavior is similar to the thermostat system. A random instance is shown in Figure 5. The system $\mathcal{A}_R$ starts at location Loc_1, and an initial value of x_1 is equal to 3. The

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

Table 1: HyRG trial table for randomly generating 100 bouncing ball examples.

	Mean	Median	Std.Dev	Min	Max	Total
Number of trials	111.63	65	120.26	1	661	11163
Generation time per trial (s)	17.022	9.824	18.615	0.0946	101.23	1702.2

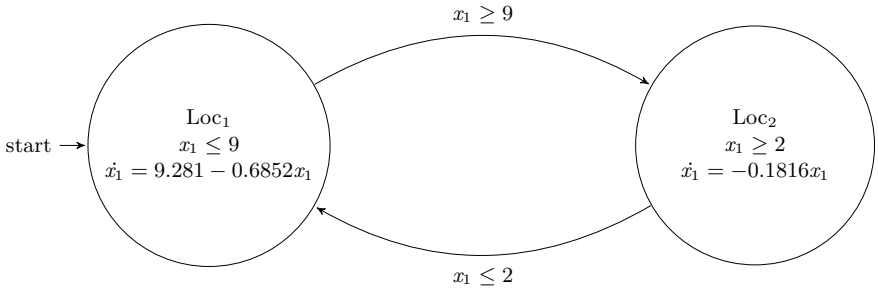


Fig. 5: A hybrid automaton randomly generated by HyRG with similar behavior to the standard thermostat system example.

Table 2: HyRG trial table for randomly generating 100 thermostat examples.

	Mean	Median	Std.Dev	Min	Max	Total
Number of trials	2126.5	1481	2152.2	24	10710	212650
Generation time per trial (s)	216.35	152.13	219.15	2.4855	1091.5	21635

SpaceEx reachability analysis and SLSF simulation of $\mathcal{A}_R$ are shown in in Figure 3, where x_1 represents the temperature of a thermostat system. Again, we generated 100 random hybrid models similar to the thermostat system. Table 2 shows the data collected from the generation process. The average number of unsuccessful trials before we get one hybrid model similar to the thermostat system is approximately 2127 trials, and an average generation time for this model is 216.35 seconds. By comparison with the trial data for generating the BB example in Table 1, we can see that the random generation function randHA runs more than ten times longer to produce the thermostat system. Since increasing the number of locations of $\mathcal{A}_R$ leads to a larger number of choices for other components (e.g., flow dynamics, invariants, transitions, etc.), more instances of $\mathcal{A}_R$ needed to be generated.

4 Conclusion

In this paper, we described methods for randomly generating hybrid automata with affine differential equations, invariants, guards, and updates implemented in a prototype software tool called HyRG. Rather than picking only random matrices and vectors for the affine functions used in flows, guards, invariants, updates, etc., we instead partition the classes of affine functions into interesting classes, for example, by restricting the eigenvalues of the differential equations to fall into stable or unstable classes. We illustrated the capability of the approach and HyRG implementation by randomly generating hybrid automata with the same qualitative behavior as standard examples like the bouncing ball and thermostat (heater) with reasonable numbers of iterations and runtime. In future

work, we plan to improve the HyRG prototype and define theoretical notions of qualitatively similar behavior using e.g., simulation relations. With such notions of qualitatively similarity, we plan to investigate possible completeness of random generation methods to enumerate every qualitatively equivalent automata with a fixed number of variables and locations.

References

1. Althoff, M., Krogh, B.: Zonotope bundles for the efficient computation of reachable sets. In: Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on. pp. 6814–6821 (Dec 2011)
2. Althoff, M., Krogh, B.H., Stursberg, O.: Analyzing reachability of linear dynamic systems with parametric uncertainties. In: Rauh, A., Auer, E. (eds.) Modeling, Design, and Simulation of Systems with Uncertainties, Mathematical Engineering, vol. 3, pp. 69–94. Springer Berlin Heidelberg (2011)
3. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
4. Bogomolov, S., Johnson, T.T., Nguyen, L.V., Schilling, C.: Compositional design of Simulink/Stateflow models with hybrid automata. In: NASA Formal Methods. Springer Berlin / Heidelberg (2015 (Under Review))
5. Cobb, J.D., DeMarco, C.L.: The minimal dimension of stable faces required to guarantee stability of a matrix polytope. Automatic Control, IEEE Transactions on 34(9), 990–992 (1989)
6. Eppstein, D., Löffler, M.: Bounds on the complexity of halfspace intersections when the bounded faces have small dimension. Discrete & Computational Geometry 50(1), 1–21 (2013)
7. 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)
8. Godsil, C., Royle, G.: Algebraic graph theory, volume 207 of Graduate Texts in Mathematics. Springer-Verlag, New York (2001)
9. Grunbaum, B., Klee, V., Perles, M.A., Shephard, G.C.: Convex polytopes. Springer, second edn. (1967)
10. Horn, R.A., Johnson, C.R.: Matrix analysis. Cambridge university press, second edn. (2012)
11. Kloetzer, M., Belta, C.: A fully automated framework for control of linear systems from temporal logic specifications. Automatic Control, IEEE Transactions on 53(1), 287–297 (2008)
12. Liberzon, D.: Switching in Systems and Control. Birkhäuser, Boston, MA, USA (2003)
13. Michel, A.N., Hou, L., Liu, D.: Stability of dynamical systems: continuous, discontinuous, and discrete systems. Springer (2007)
14. Soh, C.: Necessary and sufficient conditions for stability of symmetric interval matrices. International Journal of Control 51(1), 243–248 (1990)
15. Waterhouse, W.C.: The structure of alternating-hamiltonian matrices. Linear Algebra and its Applications 396(5), 385–390 (2005)
16. 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)

A Appendix: Additional Random Generation Details

In this appendix, we describe additional details on the random generation methods used in HyRG, described previously in the high-level overview of Section 2, Figure 1. HyRG takes as inputs the options specified in Figure 1. As output, HyRG may produce a hybrid automaton in the input XML format of the SpaceEx tool [7]. Additionally, HyRG is integrated with SLSF via a hybrid automaton to SLSF translation procedure [4].

Time-Dependent Switching Options A `Time` set includes all necessary components for randomly generating a time-dependent switched system [12].

Definition 3. *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 [8]. If a hybrid automata has a random m number of locations, so its transition graph is an $m \times m$ random adjacency matrix $\mathsf{adjMatrix}$, whose elements equal to either 0 or 1. If an element $\mathsf{adjMatrix}[i, j]$ 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 $\mathsf{adjMatrix}[i, i]$ 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.

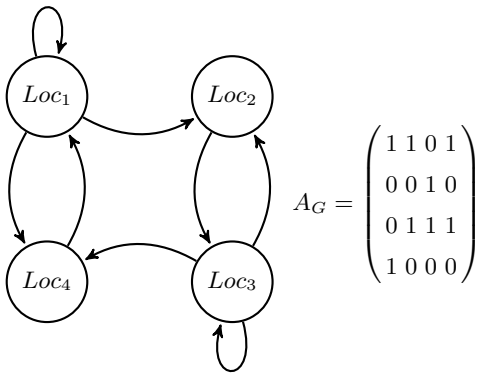


Fig. 6: An example of the transition graph of the hybrid automaton model randomly generated by the random adjacency matrix A_G.

```
    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 $\mathcal{A}_R.\texttt{Loc}$ and transitions $\mathcal{A}_R.\texttt{Trans}$.

```
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.

The pseudo-code for generating a random discrete structure by creating an arbitrary adjacency matrix shown in Figure 7—called from randHA (Figure 1, line 6). 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 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.

The continuous dynamic of each location can be randomly generated by different classes of matrices. 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*),

```
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.

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 [10, 13].

Definition 4. *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 4, 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 eigenvalues [14]. Correspondingly, the continuous dynamic of each location in system $\mathcal{A}_R$ generated based this matrix will be exponentially asymptotically stable [13]. Otherwise, if A is randomly generated as a skew-Hamiltonian matrix [15], then all eigenvalues of A have only imaginary parts.

The pseudo-code of a randomly generated flow dynamic for each location $l \in \mathcal{A}_R$.Loc shown in Figure 9—called from randHA(Figure 1, line 9). 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 $\zeta \in$ 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 (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 X, and B is an $n \times 1$ arbitrary constant vector.

```
    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`.

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 [9]. Suppose that we have an k number of half-spaces generated by an k random linear inequalities. An invariant $\texttt{Inv} \in \mathbb{R}^d$ 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 unbounded [9]. An 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 [6]. 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).

Randomly Generating Guard Conditions. For each location $l \in \mathcal{A}_R$.`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 P [5, 11]. Then, a random outgoing guard condition of location l is a set of linear inequal-

```
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 `Opt` and a set of variable $\mathcal{A}_R$.`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 `Opt` and a set of variable $\mathcal{A}_R$.`Var`.

ities 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 `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 `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 `X`.

B Appendix: Additional Experimental Results

In this appendix, we describe additional experimental results using our prototype HyRG implementation for randomly generating hybrid automaton models.

Example 3. Randomly generate the time-dependent switching system $\mathcal{A}_R$ with two locations, two transitions and two state variables x_1, x_2, and a dwell-time

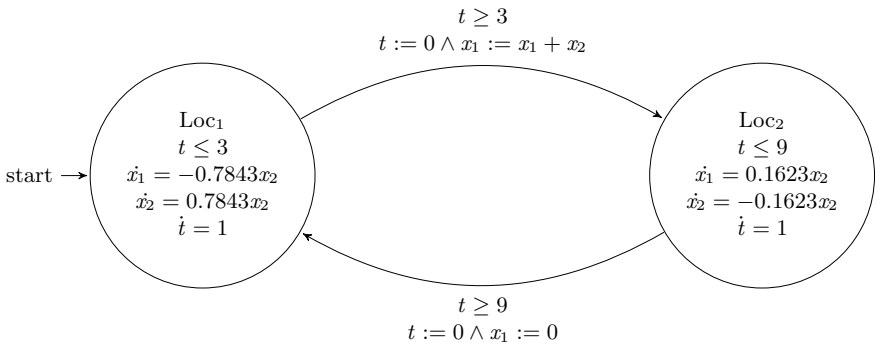
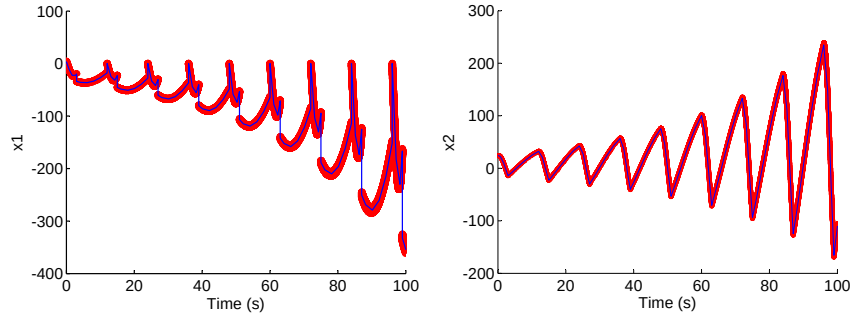


Fig. 14: An example of a random time-dependent switching hybrid automaton system $\mathcal{A}_R$ randomly generated by using skew-Hamiltonian matrices.



(a) (b)

Fig. 15: SLSF simulation (blue) and SpaceEx reachability (red) of $\mathcal{A}_R$ showing x_1 and x_2 versus time in Example 3, respectively. Overall, their simulation traces in SLSF are contained in the reachable states computed using the LGG algorithm in SpaceEx. Exceptionally, the SLSF simulation settles into a vertical trace at the reset instances of x_1.

variable t, shown in Figure 14. The system has only reset action on x_1 after each transition between two locations Loc_1, and Loc_2. Assume the system starts at location Loc_1, and its randomly generated initial condition is $x_1 = 4$, $x_2 = 24$, and $t = 0$. The continuous dynamics of each location is randomly generated based on a skew-Hamiltonian matrix A, then all eigenvalues of matrix A are purely imaginary. As a result, the trajectory of each state variables in $\mathcal{A}_R$.`Var` will show some oscillatory behaviors, and is considered to be marginally stable. Although the continuous dynamics of each location in $\mathcal{A}_R$.`Loc` is stable, the whole system $\mathcal{A}_R$ is not necessarily stable, shown in Figure 15. Figure 15 illustrates the trajectory of x_1 and x_2 versus time from a SLSF simulation trace and the reachable states computed by LGG algorithm in SpaceEx. It shows that the set of reachable states computed by SpaceEx contains totally the simulation trace produced by SLSF. Since, the SLSF simulation is different from reachable states computed by SpaceEx that it connects discrete reset points of x_1 by a vertical line.

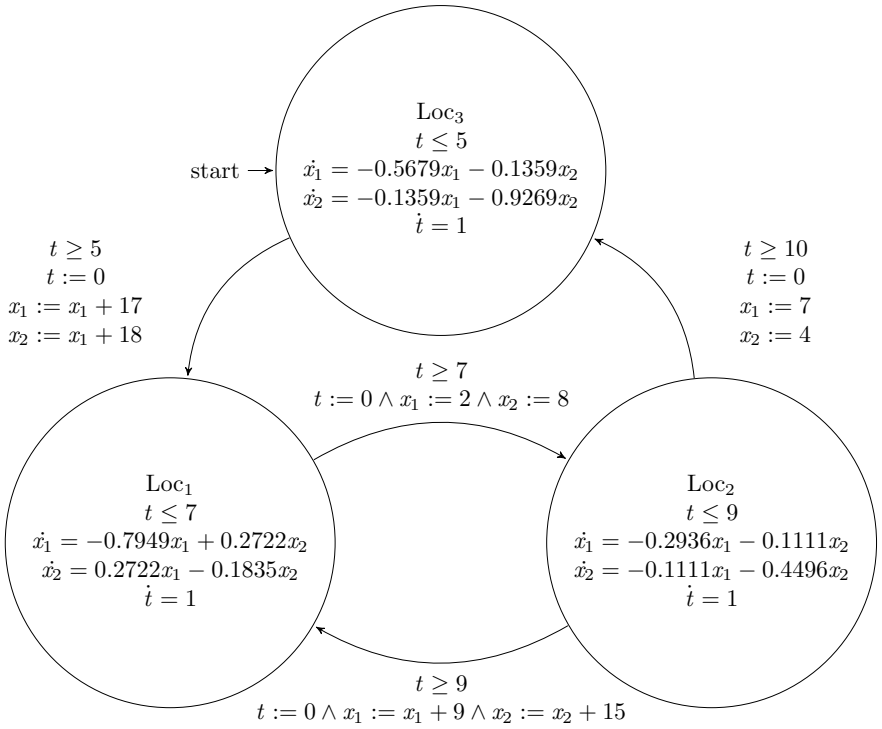
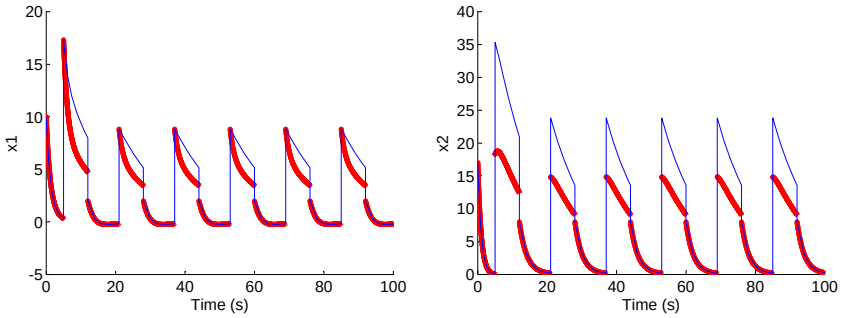


Fig. 16: An example of a random time-dependent switching hybrid automaton system $\mathcal{A}_R$ randomly generated by using *negative definite* matrices.

Example 4. Consider the randomly generated hybrid system $\mathcal{A}_R$ shown in Figure 16. 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 non-deterministic since the transition from location Loc_2 to location Loc_3 is never happened. The continuous dynamic 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 [13], shown in Figure 17. The reachable states of x_1 and x_2 computed by LGG algorithm in SpaceEx do not contain their simulation trace in SLSF when the system $\mathcal{A}_R$ takes a transition from Loc_3 to location Loc_1. It's happened due to the semantic difference in reset mechanism between SpaceEx and SLSF. In SLSF, the variables x_1 and x_2 are updated in order, it means that x_1 will be first updated to a new value, and then x_2 will be updated based on a new value of x_1. However, these variables are updated concurrently in SpaceEx, so x_2 will be updated by using an old value of x_1.



(a) (b)

Fig. 17: SLSF simulation (blue) and SpaceEx reachability (red) model $\mathcal{A}_R$ showing x_1 and x_2 versus time in Example 4, respectively. The SLSF simulation traces and the reachable states computed by LGG algorithm in SpaceEx do not line up, that depicts the semantic difference between these two tools . After each reset action, the variables x_1 and x_2 are updated sequentially in SLSF, while they are updated concurrently in SpaceEx.