Generating and Analyzing Symbolic Traces of Simulink/Stateflow Models

Aditya Kanade[1], Rajeev Alur[1], Franjo Ivančić[2], S. Ramesh[3], Sriram Sankaranarayanan[2], and K.C. Shashidhar[3]

1. University of Pennsylvania, 2. NEC Laboratories America,
3. GM India Science Lab

Abstract. We present a methodology and a toolkit for improving simulation coverage of Simulink/Stateflow models of hybrid systems using symbolic analysis of simulation traces. We propose a novel instrumentation scheme that allows the simulation engine of Simulink/Stateflow to output, along with the concrete simulation trace, the symbolic transformers needed for our analysis. Given a simulation trace, along with the symbolic transformers, our analysis computes a set of initial states that would lead to traces with the same sequence of discrete components at each step of the simulation. Such an analysis relies critically on the use of convex polyhedra to represent sets of states. However, the exponential complexity of the polyhedral operations implies that the performance of the analysis would degrade rapidly with the increasing size of the model and the simulation traces. We propose a new representation, called the *bounded vertex representation*, which allows us to perform under-approximate computations while fixing the complexity of the representation *a priori*. Using this representation we achieve a trade-off between the complexity of the symbolic computation and the quality of the under-approximation. We demonstrate the benefits of our approach over existing simulation and verification methods with case studies.

1 Introduction

Simulink/Stateflow [1] (SL/SF) models are currently the de-facto standard in the model-based development of real-time, embedded systems. They are widely used in many domains, including automotive and avionics. Simulink models are constructed by interconnecting blocks representing operations such as *gain*, *addition*, *multiplexors*, *lookup tables*, and *integrators*. Stateflow charts specify the control in the form of concurrent and hierarchical finite state machines that interact with the Simulink model. Together, they provide a powerful modeling framework that enables the development, testing, and rapid prototyping of control software, supported by automated code generation techniques.

The critical nature of the software developed with these models calls for the use of formal design verification tools. However, there are many challenges for

[1] Simulink and Stateflow are trademarks of The MathWorks Inc.

the construction of such verification tools. First of all, the semantics of these models is loosely defined in terms of a simulation engine. The lack of clearly specified semantics makes the construction of formal tools hard. Secondly, these models incorporate both the discrete state changes due to the Stateflow charts and the continuous evolution of the state due to the presence of blocks such as *integrators*. Hybrid automata [3] and related formulations are ideal for representing these models. However, the task of model-level translation from SL/SF to hybrid automata is difficult to automate.

In this paper, we present an approach for systematic exploration of distinct behaviors of SL/SF models. Our approach provides practical solutions to some of the key challenges using a novel approach to the problem of deciphering the semantics of these models. Rather than translating these models statically into hybrid automata [2] or Lustre programs [34], our approach performs an on-the-fly translation by instrumenting Simulink blocks and Stateflow transitions with *callback functions*. During numerical simulation, a callback function is called whenever the corresponding block is evaluated. The callback functions construct the symbolic transformer of the simulation step. An appropriate composition of these transformers gives us the *symbolic trace* of the simulation.

Given the concrete and symbolic traces of a k-step simulation starting from an initial model state m, our analysis constructs a set M of states that are *equivalent* to m up to the given simulation length k. Two model states m and m' are equivalent if they yield the same sequence of discrete components at each step of simulation. Systematic exploration of the model's behaviors is obtained by repeating simulation starting from an initial state outside the set M.

The performance of such analyses depends critically on the choice of a representation for sets of continuous states. While the representation of convex polyhedra as linear constraints with arbitrary coefficients [10] is expressive, it suffers from worst-case exponential time complexity. Various restrictions have been proposed to achieve tradeoff between precision and tractability. The typical restrictions include axis-parallel constraints (e.g. intervals [9]), difference constraints (e.g. octagons [23] and octahedra [7]), or an arbitrary but fixed set of linear constraints (e.g. template polyhedra [27]).

While most of these representations are designed for over-approximations, our approach requires under-approximations of state-sets to eliminate only redundant simulations. We therefore design an under-approximate *bounded vertex representation* of polyhedra using a set of direction vectors. The size of this representation is fixed by the number of direction vectors used to compute such a representation. As a result, the time and space complexity of the analysis can be controlled, allowing analysis of longer simulations for larger system models. This representation can potentially be useful to improve precision of over-approximate analyses, detect valid counter-examples, and in controller synthesis methods.

We have implemented our approach using model instrumentation and the bounded vertex representation of polyhedra for SL/SF models with *linear blocks*. The model instrumentation is implemented using the Simulink runtime API [31] and operations over the bounded vertex representation are performed using the GNU Linear Programming Kit (GLPK) [18].

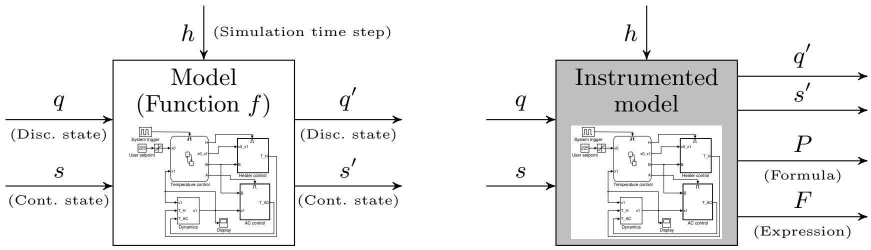
We illustrate the benefits of our approach on three case studies: a Simulink demo model from The MathWorks [1] and two instances of the room heating benchmark [12]. These models use several commonly used modeling features of SL/SF including hierarchical and concurrent Stateflow charts and lookup tables. Nevertheless, the symbolic traces generated by our instrumentation technique accurately capture the simulation semantics of these models. The experimental results indicate that the test cases generated by our tool successfully exercise distinct discrete mode switchings of these models, leading to better temporal coverage of model behaviors. Further, the use of the bounded vertex representation leads to scalable analysis. While most of the verification tools [21, 30, 5, 13] for hybrid systems are limited to systems with up to 10 continuous variables, the small runtime of our analysis on a case study with 10 continuous variables suggests that our technique can handle systems with large number of variables. We also compare our tool with Reactis [25, 8] and with random testing on a case study. The test cases generated by our tool explore significantly large number of inequivalent simulation behaviors than both Reactis and random testing.

Related work. Bisimulation metrics and expansion functions provide a mechanism of identifying states whose continuous trajectories stay close in space and time [17, 22, 11]. However, these techniques require transition guards to be planar and are not suitable for the case studies considered in this paper.

Our recent work on analysis of SL/SF models [4] also considers the problem of temporal coverage but requires static translation from SL/SF models to hybrid automata. The runtime technique of symbolic trace generation presented in this paper overcomes this drawback, enabling automated analysis of large and complex models. Further, the use of convex polyhedra may limit the scalability of the earlier approach. The algorithm presented in this paper exploits the scalability of linear programming solvers in under-approximate computations. Some related approaches that combine concrete and symbolic executions with constraint solving have been explored in software testing [19, 29].

There are many commercial and in-house tools for testing SL/SF models [14, 25, 28, 32, 35] which aim at structural coverage of model elements. These tools combine randomization with constraint solving techniques to generate test cases. Our notion of temporal coverage captures the simulation behaviors of models instead of structural properties (like blocks and branches) of model designs.

In hybrid systems verification, convex polyhedra are widely used to represent sets of states [21, 20, 30, 5, 13, 16]. However, the scalability of verification tools is limited by the worst-case exponential complexity of polyhedral operations. Restricted forms of polyhedra [6, 5, 33, 15, 26] have been designed for over-approximate computations. Our approach proposes the use of bounded vertex representations as under-approximations of sets of states. The bounded vertex representation bears some similarities to zonotopes [15, 16] but it need not be symmetric about a central vertex and computes under-approximations.



(a) Concrete simulation step (b) Concrete and symbolic simulation step

Fig. 1. Single-step concrete and symbolic simulation semantics of SL/SF models

2 Discrete-time Simulations of Simulink/Stateflow

An SL/SF model of an embedded system defines time-dependent mathematical relationships between the inputs, internal state variables, and outputs of the system. These models are represented graphically as dataflow diagrams of interconnected blocks. Simulink provides a diverse family of continuous, discrete, and logical building blocks. Stateflow complements these design features with concurrent and hierarchical state machines called Stateflow charts that are used for specifying discrete mode control logic.

2.1 Single-step Concrete and Symbolic Simulation Semantics

The SL/SF models are simulated on numerical data to generate concrete executions. The MathWorks simulation engine evaluates the blocks in a given model at discrete time steps. Each block has an associated sampling time which specifies the period at which the block is evaluated. Sampling times of the individual blocks are used to determine the *simulation time step* $h \in \mathbb{R}_+$ of the overall model, so that the model is evaluated every h time units. For simplicity, we assume that the sampling time of each block equals the simulation time step h.

Each block in a model represents a mathematical function. For example, an *integrator block* represents assignment to a *continuous state variable* of the model such that the time derivative of its output equals the value of its input. The output is computed by means of numerical integration. The precision of the integration can be controlled by choosing an integration routine and the simulation time step. We consider (explicit) fixed-step integration routines.

A *continuous state* $s \in \mathbb{R}^n$ of an SL/SF model consists of a valuation of the continuous state variables in the model. A *discrete state* $q \in Q$ of an SL/SF model consists of the set of *active states* in a Stateflow chart of the model along with the various choices of the conditional blocks in the model. Q denotes the set of discrete states of the model. The *model state* of an SL/SF model is denoted by $m = (q, s) \in M$ where $M = Q \times \mathbb{R}^n$.

For the numerical simulation, we fix an integration routine $\mathbb{S}$ and a simulation time step h. An SL/SF model defines a function $f : M \to M$. The inputs to the model are fixed at the beginning of a simulation run. A *concrete simulation step* of an SL/SF model applies the function f to a model state (q, s) to yield a model state (q', s') as shown in Figure 1(a). The transition between discrete states consists of a change in the choices of the conditional blocks and the transitions of the Stateflow charts. The transition between continuous states consists of an assignment to the continuous state variables. The function f is deterministic but it is defined *operationally* according to an evaluation order of the blocks in the model determined by the MathWorks simulation engine.

In order to overcome the opacity in the evaluation order semantics of SL/SF, we augment the concrete simulations of SL/SF models with symbolic simulations by instrumenting the model with callback functions. Our method discovers the function f defined by the model incrementally. The symbolic simulation emits the description of f applicable in each simulation step. A *concrete and symbolic simulation step* of the instrumented SL/SF model maps a model state (q, s) to a tuple (q', s', P, F) where $(q', s') = f(q, s)$, P is a quantifier-free formula and F is an expression over the continuous state variables. The simulation step of the instrumented model is shown in Figure 1(b). The symbolic transformers (P, F) generated by the simulation step satisfy the following properties: $[\![P]\!](s) = \mathit{true}$, $s' = [\![F]\!](s)$, and for all $v \in \mathbb{R}^n$, if $[\![P]\!](v) = \mathit{true}$ then $f(q, v) = (q', [\![F]\!](v))$.

2.2 Automated Instrumentation of Simulink/Stateflow Models

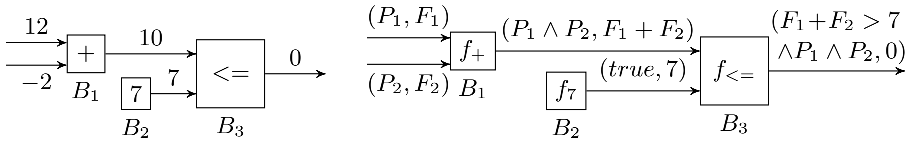
The ability to symbolically simulate SL/SF models presents many possibilities for automated analysis. It is however challenging due to their complex semantics. Factors such as concurrency and hierarchy in Stateflow charts, triggered or conditional subsystems, and virtual blocks complicate the semantics.

In this section, we present a technique that enables symbolic simulation of SL/SF models. It involves two steps: manual implementation of callback functions for the block types in the SL/SF language and automated technique of composing the callback functions through instrumentation of models.

Manual implementation of callback functions. For each block type T in a given model, we implement a callback function f_T that encodes the semantics of the block type T. The callback function for a block of type T takes a symbolic transformer (P_i, F_i) for each input port i of the block and generates a symbolic transformer (P'_j, F'_j) for each output port j of the block. For example, if (P_1, F_1) and (P_2, F_2) are the symbolic transformers of the inputs to a *sum block* then the symbolic transformer of the output is $(P_1 \wedge P_2, F_1 + F_2)$.

If a block type is conditional then we use the concrete input values to determine which conditional branch is being executed to generate corresponding symbolic transformer. Consider a *relational operator block* with relational operator <=. Suppose (P_1, F_1) and (P_2, F_2) are the input symbolic transformers. If the concrete input values satisfy the relation then the symbolic transformer of the output is $(F_1 \leq F_2 \wedge P_1 \wedge P_2, 1)$ otherwise it is $(F_1 > F_2 \wedge P_1 \wedge P_2, 0)$.

The symbolic transformer for a *constant block* with constant value c is (true, c). If the constant block corresponds to a continuous variable x_i of the model then



(a) Block evaluation

(b) Callback function evaluation

Fig. 2. Block-level compositional semantics

the symbolic transformer is $(true, x_i)$. An *integrator block* always corresponds to some continuous variable of the model. If it represents a variable x_i then the output symbolic transformer is $(P_1, \mathbb{S}(x_i, h, F_1))$ where (P_1, F_1) is the input symbolic transformer, h is the simulation time step, and $\mathbb{S}(x_i, h, F_1)$ is the expression for the numerical integration. For the Euler solver, the expression is $x_i + hF_1$.

Automated composition of callback functions. For every block B of type T in the model, we attach the callback function f_T with it. During simulation, the callback function of a block is evaluated immediately after the output of the block is computed by Simulink and before the next block is executed.

Consider Figure 2(a) which shows three inter-connected blocks and a valuation to the inter-connections during a simulation step. We use *the Simulink block runtime API* to access block data, such as block inputs and outputs, and parameters, during simulation. The key part of our scheme is to correctly access symbolic transformers of the blocks connected to the input of a given block. The inter-connections between blocks can be identified programmatically through the 'PortConnectivity' parameter of each block. This allows us to compose the callback functions of the individual blocks in a model as shown in Figure 2(b).

The block evaluation order determined by the MathWorks simulation engine is guided by data dependencies between the blocks so that a block is evaluated only after its inputs are computed. The technique of callback functions works with any valid block evaluation order. For instance, $\langle B_1, B_2, B_3 \rangle$ and $\langle B_2, B_1, B_3 \rangle$ are the valid block evaluation orders for the model in Figure 2. The callback functions f_+, f_7, and $f_{<=}$ are also evaluated in the order chosen by the simulator.

The presence of virtual blocks and enabled, triggered, or conditional subsystems requires special care. For instance, a virtual block (e.g. an *inport*) cannot be attached a callback function because it is compiled away before simulation. For a block whose input port is attached to a virtual block, we need to walk the inter-connections to find the correct non-virtual input block.

The callback function for a Stateflow chart uses *the Stateflow API* to access the elements of the chart. We annotate the transitions of a Stateflow chart to monitor which transitions are taken in a simulation step. The guards on these transitions are used for generating the corresponding symbolic transformers for the simulation step. We check consistency of symbolic transformers with the corresponding concrete simulation values by evaluating the transformers independently in MATLAB and matching the output with the concrete values.

Limitations. Our current implementation is restricted to linear transformers. While we support several block types and their configuration settings, the library of callback functions that we built is not complete, given the rich set of modeling features that Simulink provides. Most of these limitations can be overcome by means of simple extensions to the basic scheme proposed here. We assume that there are no local variables and functions in Stateflow charts and that all the output variables take discrete values. While we handle many commonly used Stateflow features including hierarchy and concurrency, extension to advanced features such as recursive broadcasts and completion semantics is future work.

3 Under-approximations of Convex Polyhedra

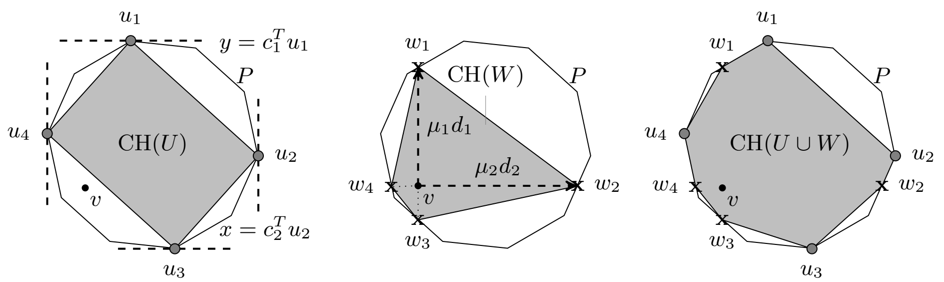
Polyhedra form a natural representation of sets of continuous states. However, the worst-case exponential complexity of polyhedral operations limits the scalability of analysis tools. Several restricted forms of domains support efficient over-approximate computations. Since the goal of our analysis is to eliminate redundant simulations, it requires exact or under-approximate computations.

We now present a novel representation of polyhedra for under-approximate computations. The representation called, the *bounded vertex representation*, consists of a finite number, say k, of vectors where k is bounded *a priori*. The polyhedron represented by these vectors is their convex hull. In other words, the vectors form the vertices of the polyhedron. The rapid increase in the number of vertices with repeated operations affects the performance of an iterative analysis. The approach of working with a bounded number of vertices overcomes this bottleneck. In our analysis, the under-approximation of the set of states equivalent to a given state s should contain the state s itself to ensure that s is not chosen as an initial state subsequently. Therefore, given a vector v in a polyhedron P, the under-approximation of P with respect to v should contain v itself.

3.1 Bounded Vertex Representation

Consider the polyhedron $P \subseteq \mathbb{R}^n$ shown in Figure 3(a) (for $n = 2$) and a vector $v \in P$. An under-approximation of P can be obtained by selecting a collection $U = [u_1, \ldots, u_4]$ of vectors from the vertex set of P. To obtain these vertices, we pick a collection $C = [c_1, \ldots, c_4]$ of vectors and maximize the objective functions ${c_i}^T x$ over the polyhedron P where x is the $n \times 1$ vector of real valued variables. For instance, Figure 3(a) shows the vertices obtained by selecting c_i parallel to positive and negative axes of the 2-dimensional space. While the intersection of the half-spaces ${c_i}^T x \leq {c_i}^T u_i$ gives an over-approximation (bounding box) of P, the convex hull of the vertices u_i gives an under-approximation. We denote the convex hull of a collection U of vectors by $\mathrm{CH}(U)$. However, the given vector $v \in P$ may not belong to the convex hull of U as shown in Figure 3(a).

Another under-approximation of P is shown in Figure 3(b). It is formed with respect to the vector v by extending rays starting from v along the directions given by vectors in $D = [d_1, d_2, -d_1, -d_2]$ where d_1 and d_2 are axis-parallel. Let



(a) u_i are vertices of P (b) w_i lie on facets of P (c)

Fig. 3. Under-approximations of a polyhedron P with respect to a vector $v \in P$

$W = [w_1, \ldots, w_4]$ be the collection of vectors that lie at the intersection of these rays $v + \mu_i d_i$, where $\mu_i \in \mathbb{R}^+$ are the scaling factors, and facets of the polyhedron. The convex hull of W is a subset of P. If there are at least two direction vectors $d_i, d_j \in D$ such that $d_i = -d_j$, the vector v is guaranteed to belong to CH(W).

While the collection W ensures that the vector v is included in the under-approximation, the collection U is potentially useful in obtaining a better under-approximation. The under-approximation given by the union $U \cup W$ of the two collections U and W is shown in Figure 3(c). We combine these two observations to define the bounded vertex representation of P.

Consider two collections $C = [c_1, \ldots, c_m]$ and $D = [d_1, \ldots, d_r]$ of $n \times 1$ real vectors, called the *coefficient* and *direction* vectors respectively. We require that there exist at least two vectors $d_i, d_j \in D$ such that $d_i = -d_j$. Given C and D, the *bounded vertex representation* (BVR) of a convex polyhedron $P \subseteq \mathbb{R}^n$ with respect to a vector $v \in P$ consists of a collection $V = U \cup W$ where $U = [u_1, \ldots, u_m]$ and $W = [w_1, \ldots, w_r]$ of vectors, called *vertices*, such that

1. The vector u_i, $i \in [1, m]$, maximizes the function ${c_i}^T x$ over P and
2. The vector $w_i = v + \mu_i d_i$, $i \in [1, r]$, where $\mu_i = \mathsf{max}\,\{\lambda \geq 0 \mid w_i = v + \lambda d_i \in P\}$.

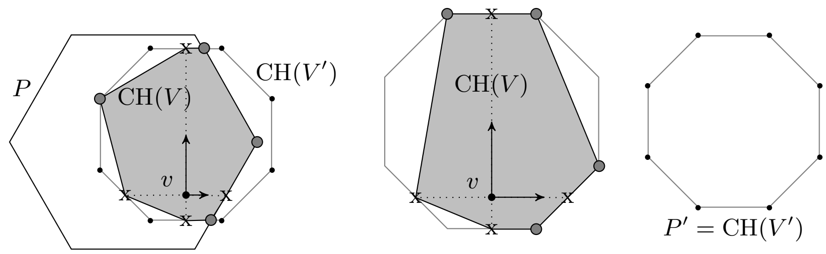
A bounded vertex representation V of a polyhedron $P \subseteq \mathbb{R}^n$ with respect to a vector $v \in P$ satisfies the following properties: CH(V) $\subseteq P$ (under-approximation) and $v \in$ CH(V) (membership).

3.2 Computing Bounded Vertex Representations

In the following discussion, we consider collections $C = [c_1, \ldots, c_m]$ and $D = [d_1, \ldots, d_r]$ of coefficient and direction vectors.

Construction. We define a procedure $\mathsf{BVR}(A, b, v, C, D)$ to compute a bounded representation $V = U \cup W$ of a convex polyhedron $P = \{x \colon Ax \leq b\}$ with respect to a vector $v \in P$ using linear programming (LP). For each $c_i \in C$, the vertex $u_i \in U$ is computed by maximizing the linear objective function ${c_i}^T x$ over P.

$$u_i = \text{maximize } {c_i}^T x \text{ subject to } Ax \leq b \tag{1}$$



(a) $\mathsf{Intersect}(V', P, v, C, D)$ (b) $\mathsf{PreImage}(V', F, v, C, D)$

Fig. 4. Operations on the bounded vertex representation

An optimal solution to a linear objective function over a convex polyhedron occurs at a vertex of the polyhedron. A vector u_i is thus a vertex of the polyhedron P. The optima with respect to ${c_i}^T x$ may not be unique but, given a deterministic LP solver, the set U is uniquely defined.

For each direction vector $d_i \in D$, the procedure BVR maximizes a scalar variable μ such that $v + \mu\, d_i \in P$.

$$\mu_i = \text{maximize} \;\; \mu \;\; \text{subject to} \quad (A\, d_i)\, \mu \leq (b - Av) \wedge \;\mu \geq 0 \tag{2}$$

The vector $w_i \in W$ is obtained from the optimal value μ_i as: $w_i = v + \mu_i\, d_i$.

Intersection. We define a procedure $\mathsf{Intersect}(V', P, v, C, D)$ that takes as input a bounded vertex representation V', a convex polyhedron $P = \{x\colon Ax \leq b\}$, and $v \in P \cap \mathrm{CH}(V')$. It computes a bounded vertex representation $V = U \cup W$ of the intersection of P and $\mathrm{CH}(V')$ with respect to the vector v.

Let $|V'| = k$. For convenience, we treat the collection V' as $k \times n$ matrix. The vertices in V' form the rows of the matrices. Let $R_{(V')}$ be the constraint representation of $\mathrm{CH}(V')$ using a $k \times 1$ vector λ of real-valued auxiliary variables. Informally, R represents vectors x and λ such that $x \in \mathrm{CH}(V')$ and λ are multipliers that certify the membership of x.

$$R_{(V')}(x, \lambda):\;\; x = {V'}^T \lambda \wedge \lambda \geq \mathbf{0} \;\wedge\; \mathbf{1}^T \cdot \lambda = 1$$

A vertex $u_i \in U$ is computed by solving the following linear program:

$$u_i = \text{maximize } {c_i}^T x \text{ subject to } Ax \leq b \;\wedge\; R_{(V')}(x, \lambda)$$

Consider Figure 4(a) as an example where $|C| = 4$ and $|D| = 4$. The vectors in V' are shown as small black dots. The vectors in the set U are shown as small gray circles and are vertices of $P \cap \mathrm{CH}(V')$. A vector $w_i \in W$ is obtained by solving the following linear program:

$$\mu_i = \text{maximize } \mu \text{ subject to } x = v + \mu\, d_i \wedge Ax \leq b \;\wedge\; R_{(V')}(x, \lambda) \wedge \;\mu \geq 0$$

The optimal solution for x directly yields the required element w_i of the collection W. In Figure 4(a), the vectors in the set W are shown as ‘x’s.

Preimages of Affine Functions. We define a procedure $\mathsf{PreImage}(V', F, v, C, D)$ that takes as input a bounded vertex representation V', an affine function $F(x) = Ax + b$, and a vector v such that $F(v) \in \mathrm{CH}(V')$. It computes a bounded vertex representation $V = U \cup W$ of the preimage of $\mathrm{CH}(V')$ under F with respect to v. For each $c_i \in C$, the following LP is solved to yield $u_i \in U$:

$$u_i = \text{maximize } {c_i}^T x \text{ subject to } x' = Ax + b \ \wedge \ R_{(V')}(x', \lambda)$$

Similarly, each direction vector $d_i \in D$ yields an element $w_i = v + \mu_i d_i \in W$:

$$\mu_i = \text{maximize } \mu \text{ subject to } x = v + \mu \, d_i \ \wedge x' = Ax + b \ \wedge \ R_{(V')}(x', \lambda) \wedge \ \mu \geq 0$$

Figure 4(b) illustrates the preimage computation using BVR.

Complexity. Consider bounded vertex representations $V = U \cup W$ where $|U| = m$ and $|V| = r$. The construction of a BVR of a polyhedron $P \subseteq \mathbb{R}^n$ involves solving m LP instances in n variables and r LP instances in a single variable. The intersection and preimage operations involve solving m instances in $(m + r + n)$ variables and r instances in $(m + r + 1)$ variables. Each LP instance can be solved in time polynomial in the number of variables.

We have implemented BVR using the GNU linear programming kit (GLPK). We present preliminary evaluation of BVR in the context of a case study with 10 continuous variables in Section 5.

4 Symbolic Analysis of Simulation Traces

The SL/SF models are typically analyzed through numerical simulation. However, the usual practice of random testing does not guarantee coverage of all the model behaviors. In this section, we propose an automated testing technique to systematically explore distinct behaviors of SL/SF models. Our technique is based on an analysis of simulation traces of the model.

Consider an SL/SF model instrumented for symbolic simulation (Section 2). We fix a simulation framework $\langle \mathbb{S}, h, k \rangle$ where $\mathbb{S}$ is the integration routine, h is the simulation time step, and $k > 0$ is the bound on the number of simulation steps. The *concrete trace* of the simulation of the model starting from an initial model state (q_1, s_1) is $\langle (q_1, s_1), \ldots, (q_{k+1}, s_{k+1}) \rangle$ where (q_i, s_i) and (q_{i+1}, s_{i+1}) are the model states before and after the ith simulation step, for $i \in [1, k]$. The *symbolic trace* of the simulation is a sequence $\langle (P_1, F_1) .... (P_k, F_k) \rangle$ where (P_i, F_i) is the symbolic transformer of the ith simulation step. We consider models with linear transformers, that is, P_i is a conjunction of linear predicates over the continuous state variables of the model and F_i is a linear transformation of the state variables. During a simulation step, the blocks in the model are evaluated in a deterministic order chosen by the MathWorks simulation engine. Thus, the choice of the initial model state completely determines the simulation trace of the model. Presently, we do not allow time-varying inputs to the models.

Algorithm 1: Iterative preimage computation

Input :

1. A k-step concrete simulation trace $\langle (q_1, s_1), \ldots, (q_{k+1}, s_{k+1}) \rangle$
2. A k-step symbolic simulation trace $\langle (P_1, F_1), \ldots, (P_k, F_k) \rangle$
3. A polyhedron $X = \{x \colon Ax \le b\}$ of the continuous state-space of the model
4. Collections $C = [c_1, \ldots, c_m]$ and $D = [d_1, \ldots, d_r]$ of coefficients and directions

Output: A BVR V such that for all $s \in \mathrm{CH}(V)$, $(q_1, s) \equiv_k (q_1, s_1)$

1 $V := \mathsf{BVR}(A, b, s_{k+1}, C, D)$
2 **for** $i := k;\ i > 0;\ i := i - 1$ **do**
3 $\quad V' := \mathsf{PreImage}(V, F_i, s_i, C, D)$
4 $\quad V := \mathsf{Intersect}(V', P_i, s_i, C, D)$
5 **return** V

We propose the notion of *equivalence of states* to improve effectiveness of simulation-based analysis. If two states are equivalent, it is sufficient to simulation the model from only one of them and search for different model behaviors starting with states that are not equivalent to the already chosen one.

Definition 1. *Consider an* SL/SF *model SL with a simulation framework $\langle \mathbb{S}, h, k \rangle$. Two model states (q, s) and (q', s') are* equivalent up to k simulation steps*, denoted by $(q, s) \equiv_k (q', s')$, if the discrete components of the concrete traces starting from them agree at each step of the simulation.*

Note that the notion of equivalence is valid even for models which do not have any Stateflow chart because the discrete component of a model state also consists of the outcomes of the conditional blocks in the model (Section 2.1).

Our overall testing technique proceeds as follows. We choose an initial state (q_1, s_1) and simulate the model for k steps. Using the concrete and symbolic traces of the simulation, we infer the set of states that are equivalent to (q_1, s_1) up to k steps. Even with a single simulation, we can thus declare a non-trivial set of model states as covered. Below we discuss the algorithm for computation of equivalent states. The initial model state for the next simulation is then chosen randomly from outside the covered region.

Algorithm. Given the concrete trace $\langle (q_1, s_1), \ldots, (q_{k+1}, s_{k+1}) \rangle$ and the symbolic trace $\langle (P_1, F_1), \ldots, (P_k, F_k) \rangle$ of a k-step simulation of an SL/SF model, Algorithm 1 computes the bounded vertex representation V of the set of continuous states equivalent to s_1. Thus, for each continuous state $s \in \mathrm{CH}(V)$, (q_1, s) is equivalent to (q_1, s_1) up to k simulation steps.

Let V_{i+1} be the bounded vertex representation at the beginning of an iteration with the loop counter equal to i (Lines 2–4). Initially, V_{k+1} is the BVR of the continuous state-space X of the model (Line 1). The loop body computes the intersection of the convex polyhedron P_i and the preimage of $\mathrm{CH}(V_{i+1})$ under the function F_i, where (P_i, F_i) is the symbolic transformer of the ith simulation

Model	Variables	SL blocks	SL conn.	SF states	SF trans.	Discrete states	State-space
VCC	2	75	83	12	28	106	$[172, 373]^2$
RHB{3}	3	33	42	1	18	12	$[16.5, 23]^3$
RHB{10}	10	33	42	1	18	3360	$[15, 23]^{10}$

Fig. 5. Characteristics of the case studies

step. The bounded vertex representation V_i is the under-approximation of the intersection. The symbolic operations are computed as defined in Section 3 using an LP solver. The number of vertices in the bounded vertex representations is bounded by the number of coefficient and direction vectors chosen as input to the algorithm. For each $i \in [1, k+1]$, $|V_i| = m + r$ where $m = |C|$ and $r = |D|$.

Performance optimizations. The quality of the result in the iterative analysis of long simulation runs may reduce due to successive under-approximations. We achieve a tradeoff between runtime and quality of under-approximations by reducing the number of preimage computations. We select a parameter called the *width of preimage computation*. If w is the width then we perform a preimage computation at every wth simulation step instead of every step. For example, if $w = 2$ and the symbolic trace is $\langle (P_1, F_1), \ldots, (P_{2k}, F_{2k}) \rangle$ then the symbolic transformer for the ith preimage computation is (P'_i, F'_i) where $P'_i = P_j \wedge P_{j+1}[F_j(x)/x]$, $F'_i = F_{j+1} \circ F_j$, and $j = 2k - 2i + 1$. The expression $e[y/x]$ denotes substitution of y for x in e and $G \circ H$ is the function composition. The number of variables in a preimage computation (a linear program) is independent of the width but the number of constraints is proportional to the width.

The quality of an under-approximate bounded vertex representation can be improved by selecting more coefficient and direction vectors but at higher computational cost. In Section 5, we discuss the effect of varying these parameters on performance of the algorithm and quality of analysis results.

5 Experimental Results

We evaluated our model instrumentation and analysis tool on the following case studies: a Simulink demo model (VCC) from The MathWorks [1] and two instances, with 3 and 10 continuous state variables, of a parametrized hybrid systems verification benchmark (RHB) [12]. Figure 5 shows their characteristics. The VCC model uses many modeling features of SL/SF including hierarchical and concurrent Stateflow and lookup tables. RHB{10} models a system with 4 heaters that can be distributed among 10 rooms. Each heater can be either on or off. The number of discrete states is $\binom{10}{4} \cdot 2^4$. The runtime technique of symbolic trace generation enables analysis of such large models.

An instrumented model is simulated from a randomly chosen initial state (q, s) and a set of states equivalent to (q, s) up to 100 simulation steps is computed. The initial state for the next simulation is chosen from outside the states

Model	Total simulations	BVR size	Width	Avg. constraints per simulation	Inequivalent simulations	Avg. analysis time per simulation(sec.)
VCC	592	8	10	1170	575 (97%)	0.1697
RHB{3}	957	10	10	500	531 (55%)	0.1503
RHB{10}	324	24	50	1292	298 (92%)	7.1514

Fig. 6. Analysis results for the case studies: Total runtime is 1 hour for each case study.

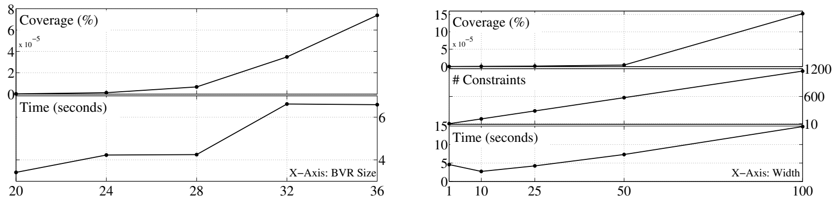
covered in all the preceding simulations. Figure 6 summarizes the test results. For VCC, over 97% initial states chosen by our tool explore inequivalent simulation traces, each exhibiting distinct discrete mode switches. The tool however cannot eliminate all equivalent initial states because of the use of under-approximations in the analysis and the presence of disjunctive guards in the models, as can be seen for the results on RHB{3} and RHB{10} models. The small runtime of our analysis for RHB{10} suggests that our technique of using the bounded vertex representation can scale to systems with large number of variables.

We compared effectiveness of our tool with Reactis [25, 8] and with random testing for the VCC model on 100 test cases generated by these tools individually. Given the model, Reactis selects test inputs at different simulation steps. We treat the test inputs generated by Reactis as distinct initial states and also choose a set of initial states by random sampling. Our tool successfully explores 96 inequivalent simulations as compared to 56 by Reactis and 73 by random testing as summarized in Figure 7. The last 4 columns report the typical structural coverage metric as percentages of the total number of conditions, decisions, modified condition/decision (MC/DC), and lookup table rows covered by the tools.

We evaluate the effect of various parameters to the analysis algorithm in the context of the RHB{10} case study. The quality of under-approximations and the computational cost can be balanced by varying the size of the bounded vertex representation and the width of preimage computation. Figure 8 shows the relative growth in computation time and coverage with increasing BVR size and width. The coverage plots show the percentage of the volume of the entire initial continuous state-space covered by a single simulation. The volumes are estimated using the MATLAB interface to the QuickHull algorithm [24]. While the number of variables is independent of the width, the number of constraints for a preimage computation increases with the width. The number of calls to the LP solver increase for decreasing widths. For small widths, the cost of calls to the LP solver may slow down the analysis as shown in Figure 8(b).

Tool	Inequivalent simulations	Conditions	Decisions	MC/DC	Lookup table rows
Our tool	96	58	84	16	29
Reactis	56	55	86	17	30
Random	73	58	84	16	29

Fig. 7. Comparison with Reactis and random testing on 100 step simulations



(a) Effect of BVR size (b) Effect of width

Fig. 8. Results are measured for the RHB{10} model and are averages over 10 runs of 100 simulation steps each. In (a), the width is 25; and in (b), the size of BVR is 24.

6 Conclusions

We have presented an analysis technique for systematic exploration of distinct behaviors of SL/SF models. The completely automated analysis of SL/SF models is made possible via symbolic trace generation using model instrumentation. Through the use of under-approximate bounded vertex representation, we can analyze models with large number of continuous variables. We have demonstrated the benefits of our approach with case studies, including a model that has 10 continuous variables. Our current implementation covers many SL/SF modeling features and extending it is future work.

Acknowledgments. The work by authors at University of Pennsylvania was partially supported by NSF award CNS 0524059 and a grant from General Motors.

References

1. Simulink demos `http://www.mathworks.com/products/simulink/demos.html`.
2. A. Agrawal, G. Simon, and G. Karsai. Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. *ENTCS*, 109:43–56, 2004.
3. R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
4. R. Alur, A. Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *EMSOFT*, pages 89–98, 2008.
5. E. Asarin, T. Dang, and O. Maler. The d/dt tool for verification of hybrid systems. In *CAV*, LNCS 2404, pages 365–370, 2002.
6. O. Bournez, O. Maler, and A. Pnueli. Orthogonal polyhedra: Representation and computation. In *HSCC*, LNCS 1569, pages 46–60, 1999.
7. R. Clarisó and J. Cortadella. The octahedron abstract domain. *Science of Computer Programming*, 64(1):115–139, 2007.
8. R. Cleaveland, S. A. Smolka, and S. Sims. An instrumentation-based approach to controller model validation. In *ASWSD*, pages 84–97, 2006.
9. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. of the Second International Symp. on Programming*, pages 106–130, 1976.

10. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.
11. A. Donzé and O. Maler. Systematic simulation using sensitivity analysis. In *HSCC*, LNCS 4416, pages 174–189, 2007.
12. A. Fehnker and F. Ivancic. Benchmarks for hybrid systems verification. In *HSCC*, LNCS 2993, pages 326–341, 2004.
13. G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In *HSCC*, LNCS 2289, pages 258–273, 2005.
14. A. A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik, and K. C. Shashidhar. AutoMOTGen: Automatic Model Oriented Test Generator for Embedded Control Systems. In *CAV*, LNCS 5123, pages 204–208, 2008.
15. A. Girard. Reachability of uncertain linear systems using zonotopes. In *HSCC*, LNCS 3414, pages 291–305, 2005.
16. A. Girard and C. L. Guernic. Zonotope/hyperplane intersection for hybrid systems reachability analysis. In *HSCC*, LNCS 4981, pages 215–228, 2008.
17. A. Girard and G. J. Pappas. Verification using simulation. In *HSCC*, LNCS 3927, pages 272–286, 2006.
18. GLPK (GNU Linear Programming Kit). `http://www.gnu.org/software/glpk/`.
19. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.
20. N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Form. Meth. in Sys. Design*, 11(2):157–185, 1997.
21. T.A. Henzinger and P. Ho. HyTech: The Cornell hybrid technology tool. In *Hybrid Systems II*, LNCS 999, pages 265–293, 1995.
22. A. Agung Julius, G. E. Fainekos, M. Anand, I. Lee, and G. J. Pappas. Robust test generation and coverage for hybrid systems. In *HSCC*, pages 329–342, 2007.
23. A. Miné. The octagon abstract domain. In *WCRE*, pages 310–, 2001.
24. Implementation of Qhull. `http://www.qhull.org`.
25. Reactis, Reactive Systems, Inc., `http://www.reactive-systems.com`.
26. S. Sankaranarayanan, T. Dang, and F. Ivancic. Symbolic model checking of hybrid systems using template polyhedra. In *TACAS*, LNCS 4963, pages 188–202, 2008.
27. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, pages 25–41, 2005.
28. Simulink Design Verifier, The Mathworks, Inc., `http://www.mathworks.com/products/sldesignverifier`.
29. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.
30. B.I. Silva, K. Richeson, B.H. Krogh, and A. Chutinan. Modeling and verifying hybrid dynamic systems using CheckMate. In *ADPM*, 2000.
31. Simulink Reference, The Mathworks, Inc. `http://www.mathworks.com`.
32. Safety Test Builder, TNI-Software., `http://www.tni-software.com/en/produits/safetytestbuilder`.
33. O. Stursberg and B. H. Krogh. Efficient representation and computation of reachable sets for hybrid systems. In *HSCC*, LNCS 2623, pages 482–497, 2003.
34. S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Trans. Embedded Comput. Syst.*, 4(4):779–818, 2005.
35. T-VEC Tester, T-VEC Technologies, Inc.,. `http://www.t-vec.com/solutions/simulink.php`.