HARE: A Hybrid Abstraction Refinement Engine for Verifying Non-linear Hybrid Automata

Nima Roohi[1(✉)], Pavithra Prabhakar[2], and Mahesh Viswanathan[1]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, USA
{roohi2,vmahesh}@illinois.edu

[2] Department of Computer Science, Kansas State University, Manhattan, USA
pprabhakar@ksu.edu

Abstract. HARE (Hybrid Abstraction-Refinement Engine) is a counterexample guided abstraction-refinement (CEGAR) based tool to verify safety properties of hybrid automata, whose continuous dynamics in each mode is non-linear, but initial values, invariants, and transition relations are specified using polyhedral constraints. HARE works by abstracting non-linear hybrid automata into hybrid automata with polyhedral inclusion dynamics, and uses dReach to validate counterexamples. We show that the CEGAR framework forming the theoretical basis of HARE, makes provable progress in each iteration of the abstraction-refinement loop. The current HARE tool is a significant advance on previous versions of HARE—it considers a richer class of abstract models (polyhedral flows as opposed to rectangular flows), and can be applied to a larger class of concrete models (non-linear hybrid automata as opposed to affine hybrid automata). These advances have led to better performance results for a wider class of examples. We report an experimental comparison of HARE against other state of the art tools for affine models (SpaceEx, PHAVer, and SpaceEx AGAR) and non-linear models (FLOW*, HSolver, and C2E2).

1 Introduction

Abstractions play an important role in the verification of cyber-physical systems, where complex continuous dynamics are abstracted into simpler dynamics that are amenable to automated analysis. This is because the general problem of safety verification is undecidable even for very simple class of continuous dynamics [2,4,20,25,32]. The success of the abstraction based method depends on finding the right abstraction, which can be difficult. One approach that tries to address this issue is the counter example guided abstraction refinement (CEGAR) framework [9] that tries to automatically discover the right abstraction through a process of progressive refinement based on analyzing spurious counter examples in abstract models. CEGAR has been found to be useful in a number of contexts [6,12,21,22], including hybrid systems [3,10,11,14,17,23,30,31].

© Springer-Verlag GmbH Germany 2017
A. Legay and T. Margaria (Eds.): TACAS 2017, Part I, LNCS 10205, pp. 573–588, 2017.
DOI: 10.1007/978-3-662-54577-5_33

In this paper, we present the tool HARE, which is a CEGAR based tool for safety verification of hybrid automata with non-linear hybrid systems. The input to HARE is the parallel composition of one or more hybrid automata, where the continuous dynamics in each control mode is described by non-linear ordinary differential equations, while the initial values, invariants, and transition relations are specified using polyhedral constraints. HARE abstracts such models into hybrid automata with polyhedral inclusion dynamics, i.e., in the abstract model, in each mode, the derivative of the continuous variables with respect to time is constrained to belong to a polyhedral set. In this sense, HARE is different from the other CEGAR based tool for non-linear hybrid systems, namely, HSolver [28], which abstracts hybrid automata by finite discrete transition systems. To perform validation of counter examples, HARE uses dReach and the δ-satisfiability procedure of dReal.

The tool described in this paper, is a significant improvement over the version reported in [29]. First, the old version only verified affine hybrid automata. The new version also considers non-linear dynamics. Second, the old version used rectangular automata to abstract concrete models. The new version uses polyhedral hybrid automata. We have observed a marked improvement in running time due to the change in abstract models—there are fewer refinement iterations on many examples because of the use of polyhedral hybrid automata. Third, the tool has been made robust. The implementation has migrated to C++ from Scala to improve its running time. We have changed some of the 3rd party tools that HARE uses internally. All these changes have enabled HARE to handle a larger class of examples (including more affine hybrid automata), with a faster running time (see results reported in Sect. 6). We have compared the performance of HARE against a number of state of the art model checkers for affine hybrid automata and non-linear hybrid automata— SpaceEx [19], PHAVer [18], SpaceEx AGAR [7], HSolver [28], C2E2 [16], and FLOW* [8]. We also compare against the old version of HARE [29]. We show that the new tool successfully proves safety when the others fail, and the running time is comparable to the other tools (see Sect. 6). A virtual machine for the new HARE, along with examples and scripts can be downloaded from https://uofi.box.com/v/HARE.

The rest of the paper is organized as follows. We introduce basic definitions and notation in Sect. 3. Our CEGAR framework, algorithms for abstraction, counter example validation, and refinement, that form the theoretical basis for HARE, are described in Sect. 4. The tool architecture and its internals are presented in Sect. 5, and Sect. 6 reports our experimental results.

2 Related Work

Doyen *et al.* consider rectangular abstractions for safety verification of affine hybrid systems in [15]. However, their refinement is not guided by counter example analysis. Instead, a reachable unsafe location in the abstract system is determined, and the invariant of the corresponding concrete location is split to ensure certain optimality criteria on the resulting rectangular dynamics. This,

in general, may not lead to abstract counter example elimination, as in our CEGAR algorithm. We believe that the refinement algorithms of the two papers are incomparable—one may perform better than the other on certain examples. Empirical evaluations could provide some insights into the merits of the approaches, however, the implementation of the algorithm in [15] was not available for comparison at the time of writing the paper.

Bogomolov *et al.* consider polyhedral inclusion dynamics as abstract models of affine hybrid systems for CEGAR in [7]. Their abstraction merges the locations, and refinement corresponds to splitting the locations. Hence, the CEGAR loop ends with the original automaton in a finite number of steps, if safety is not proved by then. Our algorithm splits the invariants of the locations, and hence, explores finer abstractions. Our method is orthogonal to that of [7], and can be used in conjunction with [7] to further refine the abstractions.

Nellen *et al.* use CEGAR in [26] to model check chemical plants controlled by programmable logic controllers. They assume that the dynamics of the system in each location is given by *conditional* ODEs, and their abstraction consists of choosing a subset of these conditional ODEs. The refinement consists of adding some of these conditional ODEs based on an unsafe location in a counter example. The method does not ensure counter example elimination in successive iterations. Their prototype tool does not automate the refinement step, in that the inputs to the refinements need to be provided manually. Hence, we did not experimentally compare with this tool.

Zutshi *et al.* propose a CEGAR-based search in [33] to find violations of safety properties. Here they consider the problem of finding a concrete counter example and use CEGAR to guide the search of the same. We instead use CEGAR to prove safety—the absence of such concrete counter examples.

3 Preliminaries

Numbers. Let $\mathbb{N}$, $\mathbb{Q}$, and $\mathbb{R}$ denote the set of *natural*, *rational*, and *real* numbers, respectively. Similarly, $\mathbb{N}_+$, $\mathbb{Q}_+$, and $\mathbb{R}_+$ are respectively the set of *positive* natural, rational, and real numbers, and $\mathbb{Q}_{\geq 0}$ and $\mathbb{R}_{\geq 0}$ are respectively the set of *non-negative* rational and real numbers. For any $n \in \mathbb{N}$ we define $[n] = \{0, 1, \ldots, n-1\}$.

Sets and Functions. For any sets A and B, $|A|$ is the size of A (the number of elements in A), 2^A is the power set of A, $A \times B$ is the Cartesian product of A and B, and B^A (similarly $A \to B$) is the set of all functions from A to B. In order to make the notations simpler, for any $n, m \in \mathbb{N}$, by A^n and $A^{n\times m}$, we mean $A^{[n]}$ and $A^{[n]\times[m]}$. The latter represents matrices of dimension $n \times m$ with elements from A. For any $f \in A \to B$ and set $C \subseteq A$, $f(C) = \{f(c) \mid c \in C\}$. Similarly, for any $\pi = a_1, a_2, \ldots, a_n$, a sequence of elements in A, we define $f(\pi)$ to be $f(a_1), f(a_2), \ldots, f(a_n)$.

Polytopes. For any set of variables $\mathtt{X}$, a function $c \in \mathbb{R}^{\mathtt{X}}$, and a constant $b \in \mathbb{R}$, $\Sigma_{x\in\mathtt{X}} c_x x \leq b$ is an *affine constraint* over the variables in $\mathtt{X}$. A *polyhedron* is a

conjunction of finite number of affine constraints. Every polyhedron P over $\mathtt{X}$, defines a set of points in $\mathbb{R}^{\mathtt{X}}$, namely the set of points that satisfy all constraints of P. We only consider non-strict inequalities, therefore P always defines a closed set. For any point $\nu \in \mathbb{R}^{\mathtt{X}}$, $\nu \in P$ means ν satisfies all the constraints in P. A polyhedron that defines a bounded set is called *polytope*. We denote the set of all polytopes over $\mathtt{X}$ by $\mathbb{P}^{\mathtt{X}}$.

3.1 Hybrid Automata

In this section, we present hybrid automata models for representing concrete and abstract hybrid systems.

Definition 1 (Hybrid Automata). *A hybrid automata $\mathcal{H}$ is a tuple* $(\mathtt{Q}, \mathtt{X}, \mathtt{I}, \mathtt{F}, \mathtt{E}, \mathtt{Q}^{\mathsf{init}}, \mathtt{Q}^{\mathsf{bad}})$ *in which*

- $\mathtt{Q}$ *is a non-empty finite set of* locations*.*
- $\mathtt{X}$ *is a non-empty finite set of* variables*. We let* $\mathtt{V} := \mathbb{R}^{\mathtt{X}}$ *be the set of all possible* valuations *of variables in* $\mathtt{X}$*. We also let* $\mathtt{X}'$ *to be the set of primed variables (*$\mathtt{X} \cap \mathtt{X}' = \emptyset$ *and* $|\mathtt{X}| = |\mathtt{X}'|$*). For every variable* $x \in \mathtt{X}$ *we use* x' *to denote the corresponding variable in* $\mathtt{X}'$*.*
- $\mathtt{I} \in \mathtt{Q} \to \mathbb{P}^{\mathtt{X}}$ *maps each location to a polytope over* $\mathbb{R}^{\mathtt{X}}$ *as* invariant *of that location.*
- $\mathtt{F} \in \mathtt{Q} \to 2^{\mathtt{V} \times \mathtt{V}}$ *maps each location q to the set of possible* flows *of that location. Each element in this set is a pair* $(\nu, \dot{\nu})$*. Intuitively it means, if the current continuous state is ν then $\dot{\nu}$ is a possible direction field.*
- $\mathtt{E}$ *is a set of* edges *of the form* $e = (s, d, r)$ *where*
  - $s, d \in \mathtt{Q}$ *are respectively* source *and* destination *of* e*,*
  - $r \in \mathbb{P}^{\mathtt{X} \cup \mathtt{X}'}$ *specifies relation of valuations before and after taking edge e as the* reset *relation.*

  We let $\mathtt{G}(e) := \exists \mathtt{X}' \bullet r$ *to be* guard *of e, as the set of valuations for which the reset relation is non-empty (note that* $\mathtt{G}(e)$ *can be represented by a polytope in* $\mathbb{P}^{\mathtt{X}}$*). We use* $\mathtt{S}(e)$*,* $\mathtt{D}(e)$ *and* $\mathtt{R}(e)$*, to denote different elements of guard e.*
- $\mathtt{Q}^{\mathsf{init}}, \mathtt{Q}^{\mathsf{bad}} \subseteq \mathtt{Q}$ *are respectively sets of* initial *and* unsafe *locations.*

We denote different elements of $\mathcal{H}$ by adding a subscript to their names. For example, we use $\mathtt{X}_{\mathcal{H}}$ *to denote the set of variables of $\mathcal{H}$. We may omit the subscript whenever it is clear from the context.*

In this paper, we use *non-linear hybrid automata* to specify a concrete system. In this class of automata, for any location $q \in \mathtt{Q}$, $\mathtt{F}(q)$ is specified by a *continuous* (possibly nonlinear) function f of type $\mathtt{I}(q) \to \mathtt{V}$. More precisely, $\mathtt{F}(q) := \{(\nu, f(\nu)) \mid \nu \in \mathtt{I}(q)\}$. Therefore, $\mathtt{F}(q)$ defines exactly one direction for any valuation in the invariant of that location. We abuse the notation and write $\mathtt{F}(q) = f$ when it causes no confusion. Next, in this paper, we use *polyhedral hybrid automata* to specify abstract systems. In this class of automata, for any location $q \in \mathtt{Q}$, $\mathtt{F}(q)$ is specified by a polytope $P \in \mathbb{P}^{\mathtt{X}}$. More precisely, $\mathtt{F}(q) := \{(\nu, \dot{\nu}) \mid \nu \in \mathtt{I}(q) \wedge \dot{\nu} \in P\}$. Therefore, $\mathtt{F}(q)$ is independent of the current

valuation. We abuse the notation and write $\mathtt{F}(q) = P$ when it causes no confusion. Note that affine hybrid automata and rectangular automata which we used in [29] for specifying concrete and abstract systems, are subclasses of non-linear automata and polyhedral automata we use in this paper.

The semantics of a hybrid automaton $\mathcal{H}$ is defined using an infinite transition system $[\![\mathcal{H}]\!]$ in the usual way. $\mathtt{S}_{[\![\mathcal{H}]\!]} := \mathtt{Q} \times \mathtt{V}$ is the state set of $[\![\mathcal{H}]\!]$. For any two states $(q_1, \nu_1), (q_2, \nu_2) \in \mathtt{S}_{[\![\mathcal{H}]\!]}$, we write $(q_1, \nu_1) \xrightarrow{t} (q_2, \nu_2)$ iff $q_1 = q_2$ and ν_1 goes to ν_2 at non-negative time t according to the continuous dynamics of location q_1. We also write $(q_1, \nu_1) \xrightarrow{e} (q_2, \nu_2)$ iff q_1 and q_2 are source and destination of the edge e and ν_1 and ν_2 satisfies invariants of source and destination locations as well as the transition relation. Finally, we use $\mathtt{S}^{\mathsf{init}}_{[\![\mathcal{H}]\!]}$ and $\mathtt{S}^{\mathsf{bad}}_{[\![\mathcal{H}]\!]}$ to refer to the set of initial and unsafe states respectively.

A *trajectory* is a sequence $\tau = s_0, (t_0, e_0), s_1, (t_1, e_1), s_2, (t_2, e_2), \ldots, s_n$ such that for any $i < n$ there is a state s'_i such that $s_i \xrightarrow{t_i} s'_i \xrightarrow{e_i} s_{i+1}$. We define τ_0 to be the initial state s_0 and τ_{lst} to be final state s_n. For any hybrid automaton $\mathcal{H}$, the *reachability problem* asks whether or not $\mathcal{H}$ has a trajectory τ such that $\tau_0 \in \mathtt{S}^{\mathsf{init}}_{[\![\mathcal{H}]\!]}$ and $\tau_{\mathsf{lst}} \in \mathtt{S}^{\mathsf{bad}}_{[\![\mathcal{H}]\!]}$. If the answer is positive, we say the $\mathcal{H}$ is *unsafe*. Otherwise, we say the $\mathcal{H}$ is *safe*.

For any hybrid automaton $\mathcal{H}$, set of states $S \subseteq \mathtt{S}_{[\![\mathcal{H}]\!]}$, and edge $e \in \mathtt{E}_{\mathcal{H}}$ we define the following functions:

- $\mathsf{dpost}^e_{\mathcal{H}}(S) = \{s' \mid \exists s \in S \bullet s \xrightarrow{e} s'\}$. Discrete post of S in $\mathcal{H}$ with respect to e is the set of states reachable from S after taking e.
- $\mathsf{dpre}^e_{\mathcal{H}}(S) = \{s \mid \exists s' \in S \bullet s \xrightarrow{e} s'\}$. Discrete pre of S in $\mathcal{H}$ with respect to e is the set of states that can reach a state in S after taking e.
- $\mathsf{cpost}_{\mathcal{H}}(S) = \{s' \mid \exists s \in S, t \in \mathbb{R}_{\geq 0} \bullet s \xrightarrow{t} s'\}$. Continuous post of S in $\mathcal{H}$ is the set of states reachable from S in an arbitrary amount of time using dynamics specified for the source locations.
- $\mathsf{cpre}_{\mathcal{H}}(S) = \{s \mid \exists s' \in S, t \in \mathbb{R}_{\geq 0} \bullet s \xrightarrow{t} s'\}$ Continuous pre of S in $\mathcal{H}$ is the set of states that can reach a state in S in an arbitrary amount of time using dynamics specified for the source locations.

4 CEGAR Algorithm for Safety Verification of Non-linear Automata

Every CEGAR-based algorithm has four main parts [9]: 1. abstracting the concrete system, 2. model checking the abstract system, 3. validating the abstract counter example, and 4. refining the abstract system. We explain parts of our algorithm regarding each of these parts in this section. Algorithm 1 shows at a very high level what the steps of our algorithm are.

Algorithm 1. High level steps of our CEGAR algorithm

Input: $\mathcal{C}$ a non-linear automaton ▷ $\mathcal{C}$ is called concrete hybrid automaton. Def 1
Output: Whether or not $\mathcal{C}$ is safe ▷ this is the reachability problem.
1. Add a self-loop to every location of $\mathcal{C}$
2. $P \leftarrow$ the initial partition of invariants in $\mathcal{C}$ ▷ Sec 4.1
3. $\mathcal{A} \leftarrow \alpha(\mathcal{C}, P)$ ▷ $\mathcal{A}$ is called abstract hybrid automaton. Def 3
4. $\tau = O^{\texttt{Poly}}(\mathcal{A})$ ▷ $O^{\texttt{Poly}}$ model checks polyhedral automata. Sec 4.2
5. ▷ τ is an annotated counter example. Sec 4.2
6. **while** $\tau \neq \emptyset$ **do** ▷ while abstract system is unsafe
7. **if** τ is valid in C **then return** 'unsafe' ▷ Sec 4.3
8. $(q, p) \leftarrow$ abstract location that should be split ▷ Sec 4.3
9. $p_1, p_2 \leftarrow$ sets that should be separated in (q, p) ▷ Sec 4.3
10. refine $P(q)$ such that p_1 and p_2 gets separated ▷ Sec 4.3
11. $A \leftarrow \alpha(\mathcal{C}, P)$ ▷ Sec 4.1
12. $\tau = O^{\texttt{Poly}}(\mathcal{A})$ ▷ Sec 4.2
13. **end while**
14. **return** 'safe'

For technical reasons (see Sect. 4.1 of [29]), we assume that in the concrete hybrid automaton, each location has a self loop transition that ensures that the duration between successive discrete steps is bounded. This assumption also makes defining the refinement step technically easier.

4.1 Abstraction

Input to our algorithm is a non-linear automaton $\mathcal{C}$ which we call the *concrete* hybrid automaton. The first step is to construct an *abstract* hybrid automaton $\mathcal{A}$ which is a polyhedral automaton. The abstract hybrid automaton $\mathcal{A}$ is obtained from the concrete hybrid automaton $\mathcal{C}$, by splitting the invariant of any location $q \in \mathtt{Q}_{\mathcal{C}}$ into a finite number of cells of type $\mathbb{P}^{\mathtt{X}}$ and defining an abstract location for each of these cells which over-approximates the non-linear dynamics in the cell by polyhedral dynamics. Definitions 2 and 3 formalizes the way an abstraction $\mathcal{A}$ is constructed from $\mathcal{C}$. Note that the construction guarantees that the behavior of $\mathcal{A}$ *over-approximates* behavior of $\mathcal{C}$ and therefore if $\mathcal{A}$ is found to be safe, $\mathcal{C}$ is guaranteed to be safe as well.

Definition 2 (Invariant Partitions). *For any hybrid automaton $\mathcal{C}$ and function $P \in \mathtt{Q} \to 2^{\mathbb{P}^{\mathtt{X}}}$ we say P partitions invariants of $\mathcal{C}$ iff the following conditions hold for any location $q \in \mathtt{Q}$:*

- $\bigcup P(q) = \mathtt{I}(q)$*, which means union of cells in $P(q)$ covers invariant of q.*
- *For any $p_1, p_2 \in P(q)$, $p_1 \neq p_2$ implies p_1 and p_2 have disjoint interior*[1].

Definition 3 (Abstraction Using Invariant Partitioning). *For any non-linear automaton $\mathcal{C}$ and invariant partition $P \in \mathtt{Q} \to 2^{\mathbb{P}^{\mathtt{X}}}$, $\alpha(C, P)$ returns polyhedral automaton $\mathcal{A}$ which is defined below:*

- $\mathtt{Q}_{\mathcal{A}} = \{(q, p) \mid q \in \mathtt{Q}_{\mathcal{C}} \wedge p \in P(q)\}$,
- $\mathtt{Q}^{\mathsf{init}}_{\mathcal{A}} = \{(q, p) \in \mathtt{Q}_{\mathcal{A}} \mid q \in \mathtt{Q}^{\mathsf{init}}_{\mathcal{C}}\}$,
- $\mathtt{X}_{\mathcal{A}} = \mathtt{X}_{\mathcal{C}}$,
- $\mathtt{I}_{\mathcal{A}}((q, p)) = p$,

[1] Interior of a polytope is obtained by making all its corresponding constraints strict.

- $\mathtt{Q}_{\mathcal{A}}^{\mathsf{bad}} = \{(q,p) \in \mathtt{Q}_{\mathcal{A}} \mid q \in \mathtt{Q}_{\mathcal{C}}^{\mathsf{bad}}\}$,
- $\mathtt{E}_{\mathcal{A}} = \{((s,p_1),(d,p_2),g,j,r) \mid (s,d,g,j,r) \in \mathtt{E}_{\mathcal{C}} \wedge (s,p_1),(d,p_2) \in \mathtt{Q}_{\mathcal{A}}\}$, *and*
- $\mathtt{F}_{\mathcal{A}}((q,p),\nu) = \mathsf{polyhull}(\bigcup_{\nu \in p} \mathtt{F}_{\mathcal{C}}(q,\nu))$, *where for any bounded set* $S \subset \mathbb{R}^{\mathtt{X}}$, $\mathsf{polyhull}(S)$ *is a polytope* W *such that* $\forall \nu \in S \bullet \nu \in W$ *and for any sequence of bounded sets* $S_1, S_2, \ldots$, *if the maximum distance of any two points in* S_n *converges to* 0 *then the maximum distance of any two points in the image of this sequence under* $\mathsf{polyhull}$ *converges to* 0 *as well.*

In addition, we define function $\gamma_{\mathcal{A}}$ *to map 1. every state in* $[\![\mathcal{A}]\!]$ *to a state in* $[\![\mathcal{C}]\!]$, *and 2. every edge in* $\mathtt{E}_{\mathcal{A}}$ *to an edge in* $\mathtt{E}_{\mathcal{C}}$. *Formally, for any* $s = ((q,p),\nu) \in \mathtt{S}_{[\![\mathcal{A}]\!]}$ *and* $e = ((q_1,p_1),(q_2,p_2),r) \in \mathtt{E}_{\mathcal{A}}$, *we define* $\gamma_{\mathcal{A}}(s)$ *to be* (q,ν) *and* $\gamma_{\mathcal{A}}(e)$ *to be* (q_1,q_2,r).

When $\mathtt{F}_{\mathcal{C}}(q)$ is an affine dynamic, there is unique minimum polytope for $\mathtt{F}_{\mathcal{A}}((q,p))$ that can be constructed exactly and efficiently. However, if the concrete flow is non-linear, abstraction even using the minimum rectangular hull might be very expensive. In our current implementation, when the flow is non-linear, we first find the rectangular hull for $\mathtt{I}_{\mathcal{C}}(q)$ and then use interval arithmetic to find a rectangular set that contains $\mathtt{F}_{\mathcal{A}}((q,p))$ as specified in Definition 3.

4.2 Counter Example and Model Checking Polyhedral Automata

After an abstract hybrid automaton is constructed (initially and after any refinement), we have to model check it. In this section we define the notion of a counter example and annotation of a counter example, which we assume is returned by the abstract model checker $O^{\mathtt{Poly}}$ when it finds that the input hybrid automaton is unsafe.

Definition 4. *For any hybrid automaton* $\mathcal{H}$, *a* counter example *is a path* $e_1, \ldots, e_n$ *such that* $\mathtt{S}e_1 \in \mathtt{Q}^{\mathsf{init}}$ *and* $\mathtt{D}e_n \in \mathtt{Q}^{\mathsf{bad}}$.

Definition 5. *A counter example* π *is called* valid *in* $\mathcal{H}$ *iff* $\mathcal{H}$ *has a trajectory* τ *and* τ *has the same sequence of edges as* π. *A counter example that is not valid is called* spurious.

Definition 6. *An* annotation for a counter example $\pi = e_1, \ldots, e_n$ *of hybrid automaton* $\mathcal{H}$ *is a sequence* $\tau = S_0 \rightarrow S'_0 \xrightarrow{e_1} S_1 \rightarrow S'_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} S_n \rightarrow S'_n$ *such that the following conditions hold:*

1. $\forall 0 \leq i \leq n \bullet \emptyset \neq S_i, S'_i \subseteq \mathtt{S}_{[\![\mathcal{H}]\!]}$,
2. $\forall 0 \leq i \leq n \bullet S_i = \mathsf{cpre}_{\mathcal{H}}\,(S'_i)$,
3. $\forall 0 \leq i < n \bullet S'_i = \mathsf{dpre}_{\mathcal{H}}^{e_{i+1}}\,(S_{i+1})$,
4. $S'_n = \mathtt{S}_{[\![\mathcal{H}]\!]}^{\mathsf{bad}} \cap (\{\mathtt{D}e_n\} \times \mathtt{V}_{\mathcal{H}})$.

Condition 1 means that each S_i and S'_i in τ are a non-empty set of states. Conditions 2 and 3 mean that sets of states in τ are computed using backward reachability. Finally, condition 4 means that S'_n is the set of unsafe states in destination of e_n. Note that these conditions completely specify $S_0, \ldots, S_n$ and

$S'_0, \ldots, S'_n$ from $e_1, \ldots, e_n$ and $\mathcal{H}$. Also, every S_i and S'_i is a subset of states corresponding to exactly one location.

In this paper, we assume to have access to an oracle $O^{\texttt{Poly}}$ that can correctly answer reachability problems when the hybrid automata are restricted to be polyhedral automata. If no unsafe location of $\mathcal{A}$ is reachable from an initial location of it, $O^{\texttt{Poly}}(\mathcal{A})$ returns 'safe'. Otherwise, it returns an annotated counter example of $\mathcal{A}$.

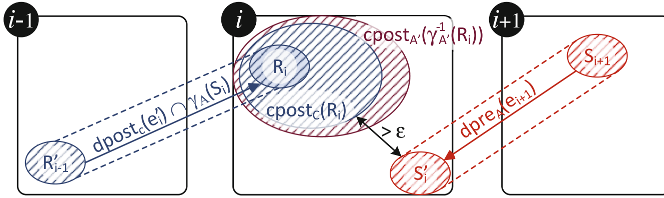

Fig. 1. Validation and refinement. There are three locations: $i-1$, i, and $i+1$. S_{i+1} and S'_i are elements of annotated counter example τ. R'_{i-1}, R_i, and $\mathsf{cpost}_C(R_i)$ are computed when τ is validated. i is the smallest index for which $\mathsf{cpost}_C(R_i)$ and $\gamma_{\mathcal{A}}(S'_i)$ are separated. Hence we need to refine $\mathcal{A}$ in location i. Refinement should be done in such a way that for the result of refinement $\mathcal{A}'$ we have $\mathsf{cpost}_{\mathcal{A}'}(\gamma^{-1}_{\mathcal{A}'}(R_i)) \cap \gamma_{\mathcal{A}'}(S'_i) = \emptyset$ (γ^{-1} is the preimage of γ).

4.3 Validating Abstract Counterexamples and Refinement

For any invariant partition P and non-linear automaton $\mathcal{C}$, if $O^{\texttt{Poly}}(\mathcal{A})$ (for $\mathcal{A} = \alpha(\mathcal{C}, P)$) returns 'safe', we know $\mathcal{C}$ is safe. So the algorithm returns $\mathcal{C}$ is 'safe' and terminates. On the other hand, if $O^{\texttt{Poly}}$ finds $\mathcal{A}$ to be unsafe it returns an annotated counter example τ of $\mathcal{A}$. Since $\mathcal{A}$ is an over-approximation of $\mathcal{C}$, we cannot be certain at this point that $\mathcal{C}$ is also unsafe. More precisely, if π is the path in τ, we do not know whether $\gamma_{\mathcal{A}}(\pi)$ is a valid counter example in $\mathcal{C}$ or it is spurious. Therefore, we need to validate τ in order to determine if it corresponds to any actual run from an initial location to an unsafe location in C.

To validate τ, an annotated counter example of $\mathcal{A} = \alpha(C, P)$, we run τ on C. More precisely, we create a sequence $\tau' = R_0 \to R'_0 \xrightarrow{e'_1} R_1 \to \cdots \xrightarrow{e'_n} R_n \to R'_n$ where

1. $e'_i = \gamma_{\mathcal{A}}(e_i)$,
2. $R_0 = \gamma_{\mathcal{A}}(S_0)$,
3. $R'_i = \mathsf{cpost}_C(R_i) \cap \gamma_{\mathcal{A}}(S'_i)$,
4. $R_i = \mathsf{dpost}^{e'_i}_C(R'_{i-1}) \cap \gamma_{\mathcal{A}}(S_i)$.

We proved the following proposition and lemma in [29].

Proposition 7. *$R'_n = \emptyset$ in τ' implies there exists i such that 1. $R'_i = \emptyset$, 2. $R_i \neq \emptyset$, 3. $\forall j < i \bullet R_j, R'_j \neq \emptyset$, and 4. $\mathsf{cpost}_C(R_i)$ and $\gamma_{\mathcal{A}}(S'_i)$ are nonempty disjoint sets.*

Lemma 8. *The counter example $\pi' = e'_1, \ldots, e'_n$ of C is valid iff $R'_n \neq \emptyset$.*

Refinement. Suppose the counterexample τ is spurious. There is a smallest index i such that $R'_i = \emptyset$. We will refine the location $(q,p) = \mathtt{De}_i$ of $\mathcal{A}$ by refining its invariant p. We know from Proposition 7, $\mathsf{cpost}_{\mathcal{C}}(R_i) \cap \gamma_{\mathcal{A}}(S'_i) = \emptyset$. However, the corresponding sets in the abstract system $\mathcal{A}$ are not disjoint, that is, $\mathsf{cpost}_{\mathcal{A}}(\gamma_{\mathcal{A}}^{-1}(R_i)) \cap S'_i \neq \emptyset$ (γ^{-1} is the preimage of γ). Our refinement strategy is to find a partition for the location (q,p) such that in the refined model $R = \alpha(\mathcal{C}, P')$ (for some P'), S'_i is not reachable from R_i (Fig. 1). Let us denote by $\mathcal{C}_{q,p}$ the restriction of $\mathcal{C}$ to the single location q with invariant p, *i.e.*, $\mathcal{C}_{q,p}$ has only one location q whose flow and invariant is the same as that of (q,p) in $\mathcal{A}$, and only transitions whose source and destination is q. We will say that an invariant partition P_r of $\mathcal{C}_{q,p}$ *separates* R_i from S'_i iff in the automaton $\mathcal{A}_1 = \alpha(\mathcal{C}_{q,p}, P_r)$, $\mathsf{reach}_{\mathcal{A}_1}(\gamma_{\mathcal{A}_1}^{-1}(R_i)) \cap \gamma_{\mathcal{A}_1}^{-1}(\gamma_{\mathcal{A}}(S'_i)) = \emptyset$. In other words, the states corresponding to S'_i in $\mathcal{A}_1$ are not reachable from $\gamma_{\mathcal{A}_1}^{-1}(R_i)$ in $\mathcal{A}_1$. Our refinement strategy will refine $\mathcal{A}$ by partitioning the control location (q,p) by the invariant partition P_r. Using results from [27], we observed [29] that such a partition P_r always exists. We also showed that such a refinement strategy ensures that any abstract counter example appears only finitely many times in the CEGAR loop.

The previous discussion, relies on the fact that we can compute $\mathsf{cpost}(.)$ exactly. Unfortunately this is not possible for the class of hybrid automaton we are considering. We use δ-complete decision procedures available through `dReach` and `dReal` to check whether R'_n will be empty for some n. If `dReach` returns `unsat`, we know the $R'_n = \emptyset$, and we can conclude that the counter example is spurious. However, if `dReach` returns δ-`sat`, we know δ-perturbation of the *syntax* of the formula defining R'_n makes it satisfiable. But this does not imply that R'_n itself isnon-empty. Hence, it is possible that because of our use of `dReach` for counter example validation, we may not be able to detect spurious counter examples.

5 Tool's Architecture

Figure 2 shows the flow and architecture of `HARE`. It also identifies 3$^{\text{rd}}$ party libraries/tools that are internally used by `HARE` at different steps. We use `Z3` [13] to check if a fix-point is reached in the abstract system model-checking, and also to check whether an unsafe state is reached. We use Boost Interval Arithmetic Library (`IAL`) [1] to abstract non-linear dynamics. We use `dReach` to validate a counter example (the validation a counter example of length n involves at most n invocations of `dReach`). Note that `dReach` calls `dReal`, internally. Also, `dReach`/`dReal` are not available in the form of libraries. Therefore, `HARE` executes `dReach` as a separate process and communicates with it through files. Finally, we use Parma Polyhedra Library (`PPL`) [5] to manipulate symbolic abstract states. This includes, computing discrete/continuous abstract posts, constructing annotated counter examples, finding rectangular hull of a polytope, abstracting affine flows, and checking if a parallelly composed location/edge has non-empty

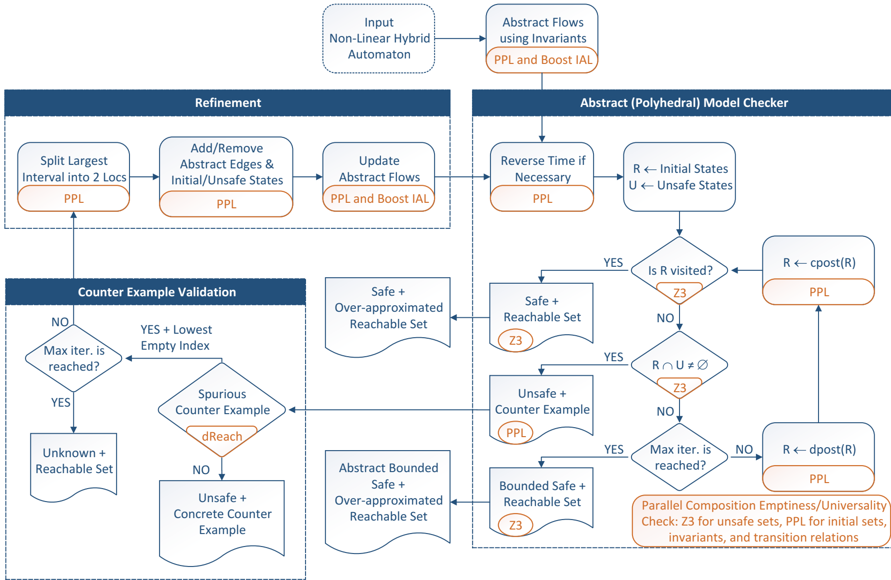

Fig. 2. Flow chart of HARE's CEGAR loop

invariant/transition relation. Compared to the old version of HARE in [29], we have replaced SpaceEx with dReach, since SpaceEx does not support non-linear dynamics. Also, we have implemented everything in C++ instead of Scala to improve performance.

The abstract model checker in HARE has a parameter `direction` with possible values `forward` and `backward`. It specifies whether the tool should perform forward or backward reachability. But PPL can only compute `cpost` and not `cpre`. This is the reason for the step "Reverse Time If Necessary". There is an optional integer parameter `max-iter` for each of the abstract and concrete model checkers. If the maximum number of iterations is reached in the abstract model checker, it returns `bounded-safe` as an answer. If abstract model checker returns this answer to the concrete model checker, `abstract bounded safe` will be returned as a result. If the maximum number of iterations is reached in the concrete model checker, it returns `unknown` as the answer. In addition to `Safe` or `Unsafe`, the user can also ask HARE to produce a `counter-example`, an `annotated-counter-example`, or the `reachable-set`. Clearly, the first two will only be produced if the system is found to be unsafe and the last one will only output the *abstract* reachable states. Note that abstract model checker can be directly called by user.

The model to be checked along with all the options for the model checker are specified in a single human readable text file according to INFO Parser from Boost Property Tree Library [24]. Every model, contains one or more hybrid automata and the safety problem is considered for their parallel composition

which is constructed on the fly. Continuous variables can be read by all hybrid automata. If the file specifies polyhedral automata, each hybrid automaton can write to all variables through transition relations and flow. On the other hand, if the file specifies a non-linear automaton, different hybrid automata can still write to a common variable through transition relations, but flow of a variable should be defined in exactly one hybrid automaton. Initial and unsafe states are specified after all hybrid automata using zero or more polyhedra for each composed location. Each edge has an optional label. If it is specified, it means that edge must be synced with an edge from other hybrid automata in the file. Otherwise, it will be interleaved. If a specified label does not end with '`?`', '`!`', or '`!!`', synchronization will be among all hybrid automata in the file (*i.e.* each hybrid automaton must take an edge with the exact same label). Characters '`?`', '`!`', and '`!!`' are used to specify input/output hybrid automata, where '`?`' is for an input edge, '`!`' is for an output edge, and '`!!`' is for a broadcast edge. Character '`*`' at the beginning of a location name means that location is transient and time cannot pass inside that location. Allowing transient locations in the model has three benefits 1. neither abstract nor concrete model checker will waste time by computing continuous post in transient locations, 2. the result automata will have one less variable, and 3. the model will be easier to understand. Finally, the current interface to the tool is only through the command line.

6 Experimental Results

The new version of `HARE` is available from https://uofi.box.com/v/HARE; the old version of the tool can be downloaded from https://uofi.box.com/cegar-hare-tacas-2016. Examples and scripts for running the examples can also be found on the links. Both these links contain a virtual machine to make repeatability straightforward.

We have run `HARE` with different set of examples with both affine and non-linear dynamics. Brief explanations of the affine benchmarks can be found in [29]. Table 1 contains the results for the affine examples. We compare the performance of `HARE`, its old version in [29], `SpaceEx` [19], `PHAVer` [18], and `SpaceEx AGAR` [7]s[2]. The first two tools are affine hybrid automata model checkers that are not CEGAR based, while the last is a CEGAR based tool for concurrent hybrid automata[3]. In the past [29], we also reported the performance of `HSolver` [28] on affine examples. However, since it performed poorly on affine examples, we have not included it for comparison in Table 1.

The new version of `HARE` proved all examples are safe, while the old version could not do this for four examples. Also the new version is faster on all examples, except one. `SpaceEx` almost never reached a fixed point. `PHAVer` could prove safety for only half of the examples, and it did it faster than new version of `HARE` in only one case. Abstraction in `SpaceEx AGAR` appears to be a very expensive

[2] By `SpaceEx` we mean `SpaceEx` with `Supp` as its scenario and by `PHAVer` we mean `SpaceEx` with `PHAVer` as its scenario.

[3] It is called "Assume Guarantee Abstraction Refinement" in [7].

operation—in four examples, the initial abstraction was not constructed even after 600 s (10 min) and we terminated the execution. Also, in three examples we could not find any set of locations that does not cause the tool to crash right at the beginning. Among 8 examples that worked for `SpaceEx AGAR`, it could prove safety for 5 of them and it was always slower than new version of `HARE`.

Table 1. Comparing `HARE` with its old version in [29] and other tools for affine dynamics. Dim. is the number of continuous variables. Size is the number of locations/edges in the input (concrete) model. Iters. is the number of iterations in our CEGAR loop before proving safety. FP. tells whether or not a tool reached a fixed-point. If a tool does not reach a fixed-point then even if it says the system is safe, the answer may not be true. As explained in [29], sometimes `SpaceEx` tells it reached a fixed-point, but before that it generates a warning that its result may not be complete. We continue to consider those cases as `SpaceEx` has not reached a fixed-point. Merged Locs. is the number of locations we initially merged for `SpaceEx AGAR`. Columns old and new for `HARE` contain results from the previous and current version of this tool. All times are in seconds and all examples were run on a laptop with Intel i5 2.50 GHz CPU and 6 GB of RAM.

			HARE						SpaceEx			PHAVer			SpaceEx AGAR			
Model	Dim.	Size	Time		Iters.		Safe		Time	FP.	Safe	Time	FP.	Safe	Merged Locs.	Time	FP.	Safe
			old	new	old	new	old	new										
Tank 16	3	3 / 6	< 1	< 1	1	1	✓	✓	3	✘	✘	1414	✘	✓	2	1133	✘	✓
Tank 17	3	3 / 6	< 1	< 1	1	1	✓	✓	5	✘	✓	1309	✘	✓	2	1041	✘	✓
Satellite 03	4	64 / 198	91	< 1	1	1	✘	✓	< 1	✘	✘	1804	✘	✘	28	> 600	---	---
Satellite 04	4	100 / 307	< 1	< 1	1	1	✓	✓	< 1	✘	✓	< 1	✓	✓	91	49	✓	✓
Satellite 11	4	576 / 1735	1	< 1	1	1	✓	✓	< 1	✘	✓	< 1	✓	✓	449	> 600	---	---
Satellite 15	4	1296 / 3895	2	< 1	1	1	✓	✓	< 1	✘	✓	< 1	✓	✓	264	> 600	---	---
Heater 03	3	4 / 6	> 600	54	---	1	---	✓	84	✘	✓	< 1	✓	✘	---	---	---	---
Heater 05	3	4 / 6	< 1	58	1	38	✘	✓	61	✘	✓	< 1	✓	✘	---	---	---	---
Heater 09	3	4 / 6	< 1	80	1	15	✘	✓	42	✘	✘	< 1	✓	✘	---	---	---	---
Nav 01	4	25 / 80	9	18	11	11	✓	✓	< 1	✓	✓	< 1	✓	✓	21	5	✓	✓
Nav 08	4	16 / 48	7	< 1	13	1	✓	✓	685	✘	✓	< 1	✓	✓	10	< 1	✓	✓
Nav 09	4	16 / 48	7	< 1	10	1	✓	✓	< 1	✘	✘	< 1	✓	✘	4	< 1	✓	✘
Nav 13	4	9 / 18	8	< 1	15	1	✓	✓	< 1	✘	✓	< 1	✓	✓	4	< 1	✓	✓
Nav 19	4	33 / 97	29	< 1	17	1	✓	✓	2	✘	✓	< 1	✓	✓	11	< 1	✓	✓

Table 2 contains results of comparing `HARE` with `C2E2` [16], `HSolver` [28], and `FLOW`* [8] on nonlinear examples. Note that `HARE` and `HSolver` support proving safety for unbounded time and unbounded number of discrete transitions. But both `C2E2` and `FLOW`* require bounded time and bounded number of discrete transitions. Also none of these two tools check whether the computed (unbounded) reachable set so far is a fixed-point. Therefore, no matter how big the time-bound is set, proving safety for this time bound in these tools does not guarantee unbounded time safety. In our experience, we set the bound for discrete number of transitions large enough so none of the tools reported maximum number of discrete transitions are reached. For the first 5 examples, we set the time bound equal to 1000 in `C2E2` and `HSolver`. For the last example, the time bound is 10 in all tools. `HARE` always finished faster than `C2E2`. On three examples `HARE` is faster than `FLOW`* and only in one example it is slower. On 3 examples `HARE` proved safety faster than `HSolver`, and in 2 examples `HSolver` was faster. `HSolver` comes with an example named `circuit` (not reported in

Table 2). The size of hybrid automaton in this example is small, but it has constants of the order 10^{12}, which turns out to be too big for C2E2 and dReach and trigger a bug in these two tools (and hence HARE). Only HSolver proves safety of this example. Finally, in our experiments, dReach performs much faster for the affine dynamics. Non-linear examples are also available at link for the new version of HARE we mentioned earlier.

Table 2. Comparing running time of HARE with other tools for non-linear dynamics. Dim. is the number of continuous variables. Size is the number of locations/edges in the input (concrete) model. Reached Abst. Size is the number of locations/edges in the final abstract model that are reached in HARE right before safety is proved. Time Bound is 10 for the "Sinusoid" model in all four tools. For all the other examples, there is no time bound in both HARE and HSolver. In other word, HARE and HSolver prove unbounded time safety for all but the last example. C2E2 and FLOW* on the other hand, require finite time bound, and we set it to be 1000 (except for the "Sinusoid" model which is 10). We have terminated all the runs that took more than 600 s (10 min). HSolver requires bounded invariants. So in the first four examples, we put 100 as an upper bound and −100 for as a lower bound of unbounded variables. FLOW* does not support trigonometric functions and C2E2 encounters an internal error on one of the examples. All times are in seconds and all examples were run on a laptop with Intel i5 2.50 GHz CPU and 6 GB of RAM.

			HARE			C2E2	HSolver	FLOW*
Model	Dim.	Size	Reached Abst. Size	Time Bound	Time	Time	Time	Time
Van der Pol	2	1 / 0	26 / 194	∞	< 1	56	3*	> 600
Jet Engine	2	1 / 0	189 / 1330	∞	55	56	2*	> 600
Cardiac Cell	2	2 / 2	249 / 1783	∞	16	50	< 1*	25
Cardiac Control	3	2 / 2	270 / 3974	∞	153	> 600	> 600*	41
Clock	3	1 / 0	9 / 56	∞	< 1	---	< 1	< 1
Sinusoid	2	1 / 0	32 / 62	10	< 1	1	7	---

6.1 Unbounded Invariants

The first 4 examples in Table 2 are taken from C2E2. Tools like C2E2 and FLOW* that try to compute the reachable set as precisely as possible, tend not to specify invariants. On the other hand, tools like HARE and HSolver that perform refinement by partitioning the state space tend to require bounded invariants. Another reason for HARE to prefer bounded invariants is that dReach, which HARE uses internally, only works for bounded variables. We had a few options to bound the invariants in those examples. The first option is to bound the invariants using large enough numbers (just like what we did for HSolver). This means we are guessing the invariant. If the guessed invariants are all closed sets, one can verify the guess by setting closure of complement of it as the unsafe states. If the unsafe states are not reachable then the guess is valid. Note that since HARE computes over-approximation of unsafe states, it is possible that HARE incorrectly says a guessed invariant is invalid. The second option is to first use tools like C2E2 or FLOW* and find a coarse invariant for all locations. Note that since these tools have bounded number of discrete transitions and they do not check

for fixpoint, one might still need to verify that invariants obtained using C2E2 or FLOW* are valid. The third option, which we have used for the current implementation, is noticing that the only part of the implementation that requires invariant to be bounded is where dReach is called. If this tool is called with an unbounded variable, then it will quickly raise an exception and terminate. In other words, it will terminate without saying that the counter example is valid. We take *not saying valid* as saying *invalid*. This approach makes it possible to use dReach even when invariants are not bounded. Note that during validation of a counter example of length larger than one, it is possible that only invariants after some step k are unbounded. Our current approach guarantees all variables are bounded when dReach is called for indices k or smaller. An example of such a system, *Automatic lane change system (driver assist)* that comes with C2E2. It is a system with affine dynamics and 10 unsafe sets. HARE proved unbounded safety for all these sets in about 190 s. During this time, dReach encountered exception in almost every iteration. But eventually, the abstract model checker reached a fixed point and found the system to be safe, so dReach was not called again. C2E2 needs to prove safety for each of these sets separately and it took this tool about 1163 s to prove them all when the time bound is set to 1000. HSolver and FLOW* could not prove safety for any of these sets within 600 s (10 min)[4]. A fourth option is one where we initially partition the state space blindly for a small number of times first, and then start the actual CEGAR loop. We used this option in all four examples in Table 2 from C2E2.

7 Conclusion

We presented a new version of the CEGAR-based model checker for non-linear hybrid systems called HARE. This version is a significant improvement over the previous version of HARE that was reported in [29]. First, HARE can now verify non-linear hybrid automata instead of hybrid automata with affine dynamics and rectangular constraints. Second, HARE now uses *polyhedral hybrid automata* as abstractions as opposed to rectangular hybrid automata. Finally, the implementation has been optimized. These changes have enabled the tool to handle a larger class of examples, in faster time. These observations have been substantiated by our experimental results reported here. While the use of dReach for counter example validation has improved the performance for affine hybrid automaton, our experiments show that dReach performs poorly for counter examples for non-linear automata (when compared with C2E2). In the future, we plan to explore if we can use C2E2 (instead of dReach) for counter example validation.

Acknowledgement. We gratefully acknowledge the support of the following grants—Nima Roohi was partially supported by NSF CNS 1329991; Pavithra Prabhakar was partially supported by NSF CAREER Award 1552668; and Mahesh Viswanathan was partially supported by NSF CCF 1422798 and AFOSR FA9950-15-1-0059.

[4] Time bound for HSolver and HARE are set to be the same. Similarly, time bound for FLOW* and C2E2 are set to be the same.

References

1. Boost Interval Arithmetic Library. http://www.boost.org/doc/libs/1_62_0/libs/numeric/interval/doc/interval.htm. Accessed 19 Oct 2016
2. 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. TCS **138**(1), 3–34 (1995)
3. Alur, R., Dang, T., Ivančić, F.: Predicate abstraction for reachability analysis of hybrid systems. ACM Trans. Embed. Comput. Syst. **5**(1), 152–199 (2006)
4. Asarin, E., Maler, O., Pnueli, A.: Reachability analysis of dynamical systems having piecewise-constant derivatives. TCS **138**(1), 35–65 (1995)
5. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1–2), 3–21 (2008)
6. Ball, T., Rajamani, S.K.: Bebop: a symbolic model checker for boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000). doi:10.1007/10722468_7
7. Bogomolov, S., Frehse, G., Greitschus, M., Grosu, R., Pasareanu, C., Podelski, A., Strump, T.: Assume-guarantee abstraction refinement meets hybrid systems. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 116–131. Springer, Heidelberg (2014). doi:10.1007/978-3-319-13338-6_10
8. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_18
9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). doi:10.1007/10722167_15
10. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and counterexample-guided refinement in model checking of hybrid systems. JFCS **14**(4), 583–604 (2003)
11. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Stursberg, O., Theobald, M.: Verification of hybrid systems based on counterexample-guided abstraction refinement. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 192–207. Springer, Heidelberg (2003). doi:10.1007/3-540-36577-X_14
12. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Zheng, H.: Bandera: extracting finite-state models from Java source code. In: ICSE, pp. 439–448 (2000)
13. Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_24
14. Dierks, H., Kupferschmid, S., Larsen, K.G.: Automatic abstraction refinement for timed automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 114–129. Springer, Heidelberg (2007). doi:10.1007/978-3-540-75454-1_10
15. Doyen, L., Henzinger, T.A., Raskin, J.-F.: Automatic rectangular refinement of affine hybrid systems. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 144–161. Springer, Heidelberg (2005). doi:10.1007/11603009_13
16. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2E2: a verification tool for stateflow models. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 68–82. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46681-0_5

17. Fehnker, A., Clarke, E., Jha, S., Krogh, B.: Refining abstractions of hybrid systems using counterexample fragments. HSCC **2005**, 242–257 (2005)
18. Frehse, G.: PHAVer: algorithmic verification of hybrid systems past HyTech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005). doi:10.1007/978-3-540-31954-2_17
19. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_30
20. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? J. Comput. Syst. Sci. 373–382 (1995). ACM Press
21. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. SIGPLAN Not. **37**(1), 58–70 (2002). doi:10.1145/565816.503279
22. Holzmann, G., Smith, M.: Automating software feature verification. Bell Labs Tech. J. **5**(2), 72–87 (2000)
23. Jha, S.K., Krogh, B.H., Weimer, J.E., Clarke, E.M.: Reachability for linear hybrid automata using iterative relaxation abstraction. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 287–300. Springer, Heidelberg (2007). doi:10.1007/978-3-540-71493-4_24
24. Kalicinski, M., Redl, S.: Boost Property Tree (2016). http://www.boost.org/doc/libs/1_62_0/doc/html/property_tree.html
25. Mysore, V., Pnueli, A.: Refining the undecidability frontier of hybrid automata. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 261–272. Springer, Heidelberg (2005). doi:10.1007/11590156_21
26. Nellen, J., Ábrahám, E., Wolters, B.: A CEGAR tool for the reachability analysis of PLC-controlled plants using hybrid automata. In: Bouabana-Tebibel, T., Rubin, S.H. (eds.) Formalisms for Reuse and Systems Integration. AISC, vol. 346, pp. 55–78. Springer, Heidelberg (2015). doi:10.1007/978-3-319-16577-6_3
27. Puri, A., Borkar, V.S., Varaiya, P.: Epsilon-approximation of differential inclusions. In: Hybrid Systems III: Verification and Control, pp. 362–376 (1995)
28. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation based abstraction refinement. ACM Trans. Embed. Comput. Syst. **6**(1), 8 (2007)
29. Roohi, N., Prabhakar, P., Viswanathan, M.: Hybridization based CEGAR for hybrid automata with affine dynamics. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 752–769. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49674-9_48
30. Segelken, M.: Abstraction and counterexample-guided construction of ω-automata for model checking of step-discrete linear hybrid models. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 433–448. Springer, Heidelberg (2007). doi:10.1007/978-3-540-73368-3_46
31. Sorea, M.: Lazy approximation for dense real-time systems. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT-2004. LNCS, vol. 3253, pp. 363–378. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30206-3_25
32. Vladimerou, V., Prabhakar, P., Viswanathan, M., Dullerud, G.: STORMED hybrid systems. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5126, pp. 136–147. Springer, Heidelberg (2008). doi:10.1007/978-3-540-70583-3_12
33. Zutshi, A., Deshmukh, J.V., Sankaranarayanan, S., Kapinski, J.: Multiple shooting, CEGAR-based falsification for hybrid systems. In: Proceedings of 14th International Conference on Embedded Software (2014)