Symbolic Analysis of Linear Hybrid Automata – 25 Years Later

Goran Frehse[1][0000−0002−5441−0481], Mirco Giacobbe[2][0000−0001−8180−0904], and Enea Zaffanella[3][0000−0001−6388−2053]

[1] U2IS, ENSTA Paris, Institut Polytechnique de Paris, France
goran.frehse@ensta-paris.fr
[2] University of Birmingham, UK m.giacobbe@bham.ac.uk
[3] Department of Mathematical, Physical and Computer Sciences, University of Parma, Italy enea.zaffanella@unipr.it

Abstract. We present a collection of advances in the algorithmic verification of hybrid automata with piecewise linear derivatives, so-called Linear Hybrid Automata. New ways to represent and compute with polyhedra, in combination with heuristic algorithmic improvements, have lead to considerable speed-ups in checking safety properties through set propagation. We also showcase a CEGAR-style approach that iteratively constructs a polyhedral abstraction. We illustrate the efficiency and scalability of both approaches with two sets of benchmarks.

1 Introduction

Hybrid automata are a modeling paradigm that combines finite state machines with differential equations in order to capture processes in which discrete, event-based, behavior interacts with continuous, time-based behavior. They came to rise in the beginning of the 1990s, through of a collaboration of scientists from various disciplines, notably computer scientists and control theorists. By that time, formal methods such as abstract interpretation [15] and model checking [14, 35] had demonstrated their potential to increase the trustworthiness of safety critical software and digital hardware designs. The goal was to develop similar techniques for discrete systems that interact with processes that be described by differential equations, like some mechanical or biological processes, so-called *hybrid systems*. In *The Theory of Hybrid Automata*, whose first version was published 25 years ago in 1996, Tom Henzinger pointed out a class of hybrid automata that hit a particular sweet spot for the purposes of symbolic (set-based) analysis: *linear hybrid automata* (LHA). LHA are characterized by linear predicates over the continuous variables and the evolution of the continuous variables is governed by differential inclusions that depend only on the discrete state, not the continuous variables themselves. LHA readily lend themselves as sound abstractions of complex natural and technical processes and as asymptotically complete approximations of a large class of hybrid automata [27]. While properties like safety are not decidable for LHA, the states reachable over a given finite path can be computed exactly and symbolically, in the form of continuous

sets associated to discrete states. In a sense, the continuous time domain can be abstracted away for LHA, so that the symbolic analysis resembles that of linear programs. Consequently, techniques from linear program analysis, such as the polyhedral computations in [23], could be applied. This lead to symbolic analysis tools such as the pioneering model checker HyTech [25]. Since then, much research effort has been invested in making symbolic analysis more efficient, in order to scale up to systems of practical interest.

In this paper, we present a selection of techniques that, applied to the symbolic analysis of LHA, have lead to performance improvements of several orders of magnitudes since the days of HyTech. We focus entirely on safety properties encoded as *reachability* problems, i.e., whether a state is reachable from any state in a given set of initial states. We start with a simple fixed-point algorithm for computing reachable sets of states, using convex polyhedra as set representations. We then present various advances in polyhedral computations as well as efficient abstractions that serve as heuristics to speed up the fixed-point algorithm, and illustrate the performance gains with experiments. As an alternative approach, we also present a technique based on the CEGAR (Counter-Example Guided Abstraction Refinement) paradigm. Starting from an initial, coarse abstraction, the finite-path encoding of LHA is used to iteratively refine the abstraction until either a counterexample has been found or the system is proved safe.

Our overview is far from exhaustive and limited to work by the authors. We point the reader to the references in [31, 1, 40] for related work. Other CEGAR approaches are implemented, e.g., in the tools HARE [36] and HyCOMP/IC3 [11]. To take an instance of an entirely different approach, we point to the work in [34], where LHA are encoded as linear programs, which are then analyzed by the software model checker ARMC. Bounded model checking for LHA has been implemented in the tool BACH [12].

The remainder of the paper is structured as follows. In Section 2, we give a formal definition of linear hybrid automata and their semantics and briefly present the fundamentals of set-based reachability. In Section 3 we show how set-based reachability can be implemented, either exactly or by resorting to overapproximations, when using convex polyhedra for the representation of symbolic states; in doing this, we highlight several optimization techniques and heuristics that are able to significantly improve the efficiency of the approach. In Section 4 we show how stronger forms of abstraction, leading to coarser overapproximations, can be successfully adopted in a CEGAR framework, so as to further enhance scalability. In Section 5 we recall the results of some recent experimental evaluations, providing evidences for the feasibility of the proposed approaches; in particular, our analysis focuses on the practical impact of the enhanced implementation techniques described in Section 3. Finally, we conclude in Section 6.

2 Symbolic Analysis of Linear Hybrid Automata

Hybrid automata describe the evolution of a set of real-valued variables over time. In this section, we give a formal definition of hybrid automata and their

behaviors, and illustrate the concept with an example. But first, we introduce some notation for describing real-valued variables and sets of these values in the form of predicates and polyhedra.

2.1 Preliminaries

Variables: Let $X = \{x_1, \ldots, x_n\}$ be a finite set of *variables*. A *valuation* over X is written as $x \in \mathbb{R}^X$ or $x : X \to \mathbb{R}$. We use the primed variables $X' = \{x'_1, \ldots, x'_n\}$ to denote successor values and the dotted variables $\dot{X} = \{\dot{x}_1, \ldots, \dot{x}_n\}$ to denote the derivatives of the variables with respect to time. Given a set of variables $Y \subseteq X$, the *projection* $y = x \downarrow_Y$ is a valuation over Y that maps each variable in Y to the same value that it has in x. We may simply use a vector $x \in \mathbb{R}^n$ if it is clear from the context which index of the vector corresponds to which variable. We denote the i-th element of a vector x as x_i or $x(i)$ if the latter is ambiguous. In the following, we use $\mathbb{R}^n$ instead of $\mathbb{R}^X$ except when the correspondence between indices and variables is not obvious, e.g., when valuations over different sets of variables are involved.

Predicates: A *predicate* over X is an expression that, given a valuation x over X, can be evaluated to either true or false. A *linear constraint* is a predicate

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \leq b,$$

where $a_1, \ldots a_n$ and b are real-valued constants, and whose sign may be strict ($<$) or nonstrict ($\leq$). A linear constraint is written in vector notation as

$$a^{\mathsf{T}} x \leq b,$$

with coefficient vector $a \in \mathbb{R}^n$ and inhomogeneous coefficient $b \in \mathbb{R}$. A *halfspace* $\mathcal{H} \subseteq \mathbb{R}^n$ is the set of points satisfying a linear constraint. A predicate over X defines a continuous set, which is the subset of $\mathbb{R}^X$ on which the predicate evaluates to true.

Polyhedra: A conjunction of finitely many linear constraints defines a polyhedron in *constraint form*, also called *$\mathcal{H}$-polyhedron*,

$$\mathcal{P} = \left\{ x \;\middle|\; \bigwedge\nolimits_{i=1}^{m} a_i^{\mathsf{T}} x \bowtie_i b_i \right\}, \text{ with } \bowtie_i \in \{<, \leq\},$$

with *facet normals* $a_i \in \mathbb{R}^n$ and *inhomogeneous coefficients* $b_i \in \mathbb{R}$. A bounded polyhedron is called a *polytope*. Note that the constraints defining $\mathcal{P}$ are not necessarily unique. The representation of a polyhedron has a big impact on the computational cost of different geometric operations. Other representations for polyhedra can be more efficient for model checking, and will be discussed in detail in Sect. 3.

2.2 Linear Hybrid Automata

We now give a formal definition of a hybrid automaton and its run semantics.

Definition 1 (Hybrid automaton). *[2, 28] A* hybrid automaton $H = (\mathsf{Loc}, \mathsf{Lab}, \mathsf{Edg}, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump})$ *consists of*

- *a finite set of* locations $\mathsf{Loc} = \{\ell_1, \ldots, \ell_m\}$ *representing the discrete states;*
- *a finite set of* synchronization labels Lab*;*
- *a finite set of edges* $\mathsf{Edg} \subseteq \mathsf{Loc} \times \mathsf{Lab} \times \mathsf{Loc}$*, also called* transitions*, which determines which discrete state changes are possible using which label;*
- *a finite set of* variables $X = \{x_1, \ldots, x_n\}$*, partitioned into uncontrolled variables U and controlled variables Y; a* state *of H consists of a location ℓ and a value for each of the variables, and is denoted by $s = (\ell, x)$;*
- *a set of states* Inv *called* invariant *or* staying condition*; it restricts for each location the values that x can possibly take and so determines how long the system can remain in the location;*
- *a set of* initial *states* $\mathsf{Init} \subseteq \mathsf{Inv}$*; every behavior of H must start in one of the initial states;*
- *a* flow relation Flow*, where* $\mathsf{Flow}(\ell) \subseteq \mathbb{R}^{\dot{X}} \times \mathbb{R}^X$ *gives for each state (ℓ, x) the set of possible derivatives $\dot{x}$, e.g., using a differential equation such as*

$$\dot{x} = f(x);$$

  given a location ℓ, a trajectory *of duration $\delta \geq 0$ is a continuously differentiable function $\xi : [0, \delta] \to \mathbb{R}^X$ such that for all $t \in [0, \delta]$, $(\dot{\xi}(t), \xi(t)) \in \mathsf{Flow}(\ell)$; the trajectory* satisfies the invariant *if for all $t \in [0, \delta]$, $\xi(t) \in \mathsf{Inv}(\ell)$;*
- *a* jump relation Jump*, where* $\mathsf{Jump}(e) \subseteq \mathbb{R}^X \times \mathbb{R}^{X'}$ *defines for each transition $e \in \mathsf{Edg}$ the set of possible successors x' of x; jump relations are typically given by a* guard *set $\mathcal{G} \subseteq \mathbb{R}^X$ and an assignment (or reset) $x' = r(x)$ as*

$$\mathsf{Jump}(e) = \{(x, x') \mid x \in \mathcal{G} \wedge x' = r(x)\}.$$

In a linear hybrid automaton *(LHA), all continuous sets and relations in* $\mathsf{Inv}, \mathsf{Init}, \mathsf{Flow}, \mathsf{Jump}$ *are defined by linear constraints (polyhedra), and the flow relation is independent of the continuous state. Typically, the flow relation in a LHA is given in the form of differential inclusions $\dot{x} \in \mathcal{P}(\ell)$, where $\mathcal{P}(\ell)$ is a polyhedron.*

We define the behavior of a hybrid automaton with a *run*: starting from one of the initial states, the state evolves according to the differential equations whilst time passes, and according to the jump relations when taking an (instantaneous) transition. Special events, which we call *uncontrolled assignments*, model an environment that can make arbitrary changes to the uncontrolled variables.

Definition 2 (Run semantics). *A* run *of H is a sequence*

$$(\ell_0, x_0) \xrightarrow{\delta_0, \xi_0} (\ell_0, \xi_0(\delta_0)) \xrightarrow{\alpha_0} (\ell_1, x_1) \xrightarrow{\delta_1, \xi_1} (\ell_1, \xi_1(\delta_1)) \ldots \xrightarrow{\alpha_{N-1}} (\ell_N, x_N),$$

with $\alpha_i \in \mathsf{Lab} \cup \{\tau\}$, satisfying for $i = 0, \ldots, N-1$:

1. *The first state is an initial state of the automaton, i.e.,* $(\ell_0, x_0) \in \mathsf{Init}$.
2. Trajectories: *In location* ℓ_i, ξ_i *is a trajectory of duration* δ_i *that satisfies the invariant.*
3. Jumps: *If* $\alpha_i \in \mathsf{Lab}$, *there exists a transition* $(\ell_i, \alpha_i, \ell_{i+1}) \in \mathsf{Edg}$ *with jump relation* $\mathsf{Jump}(e)$ *such that* $(\xi_i(\delta_i), x_{i+1}) \in \mathsf{Jump}(e)$ *and* $x_{i+1} \in \mathsf{Inv}(\ell_{i+1})$.
4. Uncontrolled assignments: *If* $\alpha_i = \tau$, *then* $\ell_i = \ell_{i+1}$, $\xi_i(\delta_i) \downarrow_Y = x_{i+1} \downarrow_Y$, *and* $x_{i+1} \in \mathsf{Inv}(\ell_{i+1})$. *This represents arbitrary assignments that the environment might perform on the uncontrolled variables* $U = X \setminus Y$.

A state (ℓ, x) *is* reachable *if there exists a run with* $(\ell_i, x_i) = (\ell, x)$ *for some* i.

The existence of a run can be reduced to satisfiability of a conjunction of linear constraints. This has been exploited to synthesise parameters [20] and in Counter Example Guided Abstraction Refinement (CEGAR) frameworks [30], which we will discuss in more detail in Sect. 4. It also follows from these semantics that with a simple model transformation,[4] a LHA can be verified by model checkers able to handle linear constraints over the rationals, see [34].

2.3 Symbolic Analysis

A standard method to compute the reachable states is to iterate the following *one-step successor* operators for discrete and continuous transitions. Given a set of states S, let $\mathsf{post}_C(S)$ be the set of states reachable by letting time elapse from any state in S,

$$\mathsf{post}_C(S) = \Big\{ \big(\ell, \xi(\delta)\big) \;\Big|\; \exists \delta \in \mathbb{R}^{\geq 0}, (\ell, x) \in S : (\ell, x) \xrightarrow{\delta, \xi} \big(\ell, \xi(\delta)\big) \Big\}.$$

Let $\mathsf{post}_D(S)$ be the set of states resulting from a jump from any state in S,

$$\mathsf{post}_D(S) = \Big\{ (\ell', x') \;\Big|\; \exists (\ell', x') \in S, \exists \alpha \in \mathsf{Lab} \cup \{\tau\} : (\ell, x) \xrightarrow{\alpha} (\ell', x') \Big\}.$$

Starting from the initial states, post_C and post_D are applied to obtain the sequence

$$\begin{aligned} R_0 &= \mathsf{post}_C(\mathsf{Init}), \\ R_{i+1} &= R_i \cup \mathsf{post}_C(\mathsf{post}_D(R_i)). \end{aligned} \tag{1}$$

If the sequence reaches a fixed-point, i.e., when $R_{i+1} = R_i$, then R_i is the set of reachable states.

In model checkinger such as HyTech [25], PHAVer [17] and SpaceEx [21], the sequence (1) is computed using *symbolic states* $s = (\ell, \mathcal{P})$, where $\ell \in \mathsf{Loc}$ and $\mathcal{P}$ is a continuous set, e.g., a polyhedron. Computing the timed successors post_C of a symbolic state $s = (\ell, \mathcal{P})$ produces a new symbolic state $s' = (\ell, \mathcal{P}')$. Computing the jump successors post_D of $s = (\ell, \mathcal{P})$ involves iterating over all outgoing transitions of ℓ, and produces a set of symbolic states $\{s'_1, \ldots, s'_N\}$, each in one of the target locations. A *waiting list* contains the symbolic states whose successors still need to be explored, and a *passed list* contains all symbolic states computed so far. The fixed-point computation proceeds as follows:

[4] It suffices to introduce a variable for the elapsed time in each location.

1. Initialization: Compute the continuous successors of the initial states and put them on the waiting list.
2. Pop a symbolic state s from the waiting list and compute its one-step successors $\{s'_1, \ldots, s'_N\} = \mathsf{post}_C(\mathsf{post}_D(s))$.
3. Containment checking: Discard the s'_i that have previously been encountered, i.e., those contained in any symbolic state on the passed list. Add the remaining symbolic states to the passed and waiting list.
4. If the waiting list is empty, terminate and return the passed list as the reachable states. Otherwise, continue with step 2.

Avoiding the processing of redundant states

An improvement on the fixed point algorithm outlined above can be obtained by modifying step 3 as follows:

3.1 Containment checking: discard the s'_i that are contained in any symbolic state on the passed list.
3.2 *Waiting list filtering*: remove from the waiting list those states that are (strictly) contained in any remaining s'_i.
3.3 State addition: add the remaining s'_i to the passed and waiting lists.

The change with respect to the original algorithm is in step 3.2: any state s_i which is removed from the waiting list need not be processed, as it is made redundant by the newly added s'_i. Note that this is a heuristic optimization, since we are going to perform some more containment checks, causing a potential overhead, in the hope that the filtering phase will actually remove some states; when successful, it leads to greater savings in computational time, compensating any previous overhead.

3 Implementing Symbolic Analysis using Polyhedra

In this section we briefly describe how to implement the symbolic analysis outlined in Section 2.3 when adopting a domain of convex polyhedra for the representation of symbolic states.

The Double Description method

Even though there exist polyhedra libraries that are exclusively based on the constraint form,[5] the classical approach [16] is based on the Double Description (DD) method [33], where the constraint form is paired with a *generator form* and conversion algorithms [13] allow for obtaining each representation from the other, removing redundancies so as to obtain minimal descriptions, as well as keeping them in synch after an incremental update. Polyhedra libraries based on the DD method include PolyLib (`www.irisa.fr/polylib/`), ELINA [39], NewPolka in Apron [29], PPL (Parma Polyhedra Library) [4], and PPLite [8]; the last three also support strict linear constraints.

[5] For instance, VPL (Verified Polyhedron Library) [10].

Generator form and $\mathcal{V}$-polyhedra. The classical definition of generators for closed polyhedra has been extended in [4] to the case of NNC (not necessarily closed) polyhedra. Namely, an $\mathcal{H}$-polyhedron can be equivalently represented in generator form by three finite sets (P, C, R), where $P \subseteq \mathbb{R}^n$ is a set of *points* of $\mathcal{P}$ (including its vertices), $C \subseteq \mathbb{R}^n$ is a set of *closure points*, and $R \subseteq \mathbb{R}^n$ is a set of *rays*. The generator form defines a *$\mathcal{V}$-polyhedron* as

$$\mathcal{P} = \left\{ \sum_{p_i \in P} \pi_i \cdot p_i + \sum_{c_j \in C} \gamma_j \cdot c_j + \sum_{r_k \in R} \rho_k \cdot r_k \;\middle|\; \begin{array}{l} \pi_i \geq 0, \gamma_j \geq 0, \rho_k \geq 0, \\ \sum_i \pi_i + \sum_j \gamma_j = 1, \sum_i \pi_i \neq 0, \end{array} \right\}$$

which consists of the convex hull of points and closure points, extended towards infinity along the directions of the rays; the requirement that at least one point p_i positively contributes to the convex combination means that the closure points, which are in the topological closure of $\mathcal{P}$, are not necessarily contained in $\mathcal{P}$.

Both NewPolka and PPL, following the approach outlined in [23, 24] and further developed in [3], use an additional slack variable (usually named ϵ) to encode the strict constraints as nonstrict ones, obtaining closed *ϵ-representations* of the NNC polyhedra. While allowing for a simple reuse of the classical conversion algorithms, this choice easily leads to a significant computation overhead. In contrast, the PPLite library is based on a *direct representation* for the strict constraints, leveraging on enhanced versions of the Chernikova procedures [5, 6] fully supporting the use of strict constraints and closure points.

Converting between $\mathcal{H}$ and $\mathcal{V}$ representations. No matter if using the direct or the slack variable representation, the core algorithmic step of the DD method $\langle \mathcal{H}, \mathcal{V} \rangle \xrightarrow{\beta} \langle \mathcal{H}', \mathcal{V}' \rangle$ modifies a DD pair by adding a single constraint (resp., generator) β. From this, the conversion procedure computing the generator form $\mathcal{V} = \mathcal{V}_m$ for a given constraint form $\mathcal{H} = \{\beta_0, \ldots, \beta_m\}$ is obtained by *incrementally* processing the constraints, starting from a DD pair $\langle \mathcal{H}_0, \mathcal{V}_0 \rangle \equiv \mathbb{R}^n$ representing the whole vector space:

$$\langle \mathcal{H}_0, \mathcal{V}_0 \rangle \xrightarrow{\beta_0} \ldots \xrightarrow{\beta_{k-1}} \langle \mathcal{H}_k, \mathcal{V}_k \rangle \xrightarrow{\beta_k} \langle \mathcal{H}_{k+1}, \mathcal{V}_{k+1} \rangle \xrightarrow{\beta_{k+1}} \ldots \xrightarrow{\beta_m} \langle \mathcal{H}_m, \mathcal{V}_m \rangle.$$

The conversion from generators to constraints works similarly, starting from a DD pair representing the empty polyhedron and incrementally adding the generators. The same approach can also be used to compute the *set intersection* $\mathcal{P}_1 \cap \mathcal{P}_2$ (resp., the *convex polyhedral hull* $\mathcal{P}_1 \uplus \mathcal{P}_2$) of polyhedra $\mathcal{P}_1 \equiv \langle \mathcal{H}_1, \mathcal{V}_1 \rangle$ and $\mathcal{P}_2 \equiv \langle \mathcal{H}_2, \mathcal{V}_2 \rangle$: the constraints in $\mathcal{H}_2$ (resp., the generators in $\mathcal{V}_2$) are incrementally added to the DD pair describing $\mathcal{P}_1$.

Cartesian factoring. In general, converting between $\mathcal{H}$ and $\mathcal{V}$ polyhedra has a worst case complexity which is exponential in the size of the input representation; it is therefore essential that these representations are kept as small as possible (hence the interest in detecting and removing redundancies as soon as possible). Cartesian factoring [22] is another technique that, in some interesting

and rather common cases, can greatly reduce the space needed to represent a $\mathcal{V}$-polyhedron. Informally, the space dimensions $X = \{x_1, \ldots, x_n\}$ of the polyhedron $\mathcal{P}$ are *partitioned* into a sequence of blocks $(B_1, \ldots, B_k)$ so that each linear constraint in the $\mathcal{H}$-representation mentions the dimensions of a single block B_i; then, the $\mathcal{H}$-polyhedron $\mathcal{P}$ is factored in the corresponding sequence of k polyhedra $(\mathcal{P}_1, \ldots, \mathcal{P}_k)$; if needed, these $\mathcal{H}$-polyhedra are converted to a sequence of $\mathcal{V}$-polyhedra, achieving significant time and space efficiency improvements with respect to the direct conversion of polyhedron $\mathcal{P}$. The ELINA library [39] is characterized by a very efficient implementation of Cartesian factoring; the technique is also implemented in PPLite.

Implementation of containment checks

The overall efficiency of the procedure computing the set of reachable states is deeply affected by the efficiency of the polyhedra containment check.

When using the DD method, the inclusion test $\mathcal{P}_1 \subseteq \mathcal{P}_2$ is implemented by checking that all the m_1 generators of $\mathcal{P}_1$ satisfy all the m_2 constraints of $\mathcal{P}_2$. In the worst case, i.e., when the inclusion holds, this amounts to the computation of $m_1 \cdot m_2$ scalar products, each one requiring $\mathcal{O}(n)$ arbitrary precision multiplications and additions, where n is the number of variables.

As shown in [7], impressive efficiency improvements can be obtained by exploiting the fact that each one-step successor state s'_i is checked against all the states stored in the passed list before being added to the passed and waiting lists. It is therefore possible to compute, and cache for reuse, simpler abstractions of the polyhedra that are enough to quickly semi-decide the containment check. The *boxed polyhedra* proposal in [7] uses a two level scheme, where each polyhedron $\mathcal{P}_i$ is abstracted into its bounding box $\mathcal{B}_i$, which in turn is further abstracted in the *pseudo-volume* information.[6]

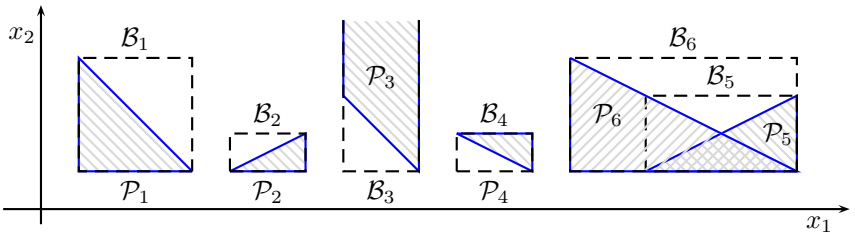


Fig. 1. Incomplete decision procedures speed up the containment checks.

While referring the interested reader to [7] for the formal definitions and technical details, a few examples are provided in Figure 1, where for each polyhedron $\mathcal{P}_i$ we draw, using a dashed black border, the corresponding bounding

[6] Roughly speaking, the volume of the box, in the case of a polytope; or the number of rays of the box, in the case of an unbounded polyhedron.

box $\mathcal{B}_i$. Intuitively, we know that $\mathcal{P}_1 \not\subseteq \mathcal{P}_2$ holds because $\text{vol}(\mathcal{B}_1) > \text{vol}(\mathcal{B}_2)$; we know that $\mathcal{P}_3 \not\subseteq \mathcal{P}_1$ holds because $\text{num\_rays}(\mathcal{B}_3) = 1 > 0 = \text{num\_rays}(\mathcal{B}_1)$; we know that $\mathcal{P}_2 \not\subseteq \mathcal{P}_4$ holds because $\mathcal{B}_2 \not\subseteq \mathcal{B}_4$ (even though $\text{vol}(\mathcal{B}_2) = \text{vol}(\mathcal{B}_4)$ and $\text{num\_rays}(\mathcal{B}_2) = \text{num\_rays}(\mathcal{B}_4)$); finally, when checking whether or not $\mathcal{P}_5 \subseteq \mathcal{P}_6$, since $\mathcal{B}_5 \subseteq \mathcal{B}_6$ holds no semi-decision procedure applies and we need to resort to the more expensive polyhedra containment check to discover that neither this last inclusion holds.

Implementing the continuous post operator

For a fixed location ℓ, the flow relation of an LHA is specified by a polyhedron $\mathcal{Q} \subseteq \mathbb{R}^n$ describing the possible values of the first time derivatives of the system variables. The possible trajectories starting from the states in polyhedron $\mathcal{P}$ are obtained by the *time-elapse* operator:

$$\mathcal{P} \nearrow \mathcal{Q} = \{\, p + t \cdot q \mid p \in \mathcal{P}, q \in \mathcal{Q}, t \in \mathbb{R}, t \geq 0 \,\}.$$

Assuming that $\mathcal{Q}$ is a closed $\mathcal{V}$-polyhedron described by generators (P, C, R), where $C = \emptyset$, the set $\mathcal{P} \nearrow \mathcal{Q}$ is a convex polyhedron that can be computed by (incrementally) adding to $\mathcal{P}$ the finite set of *rays* $R' = P \cup R$ [23].[7]

An example of application of the continuous post operator to a symbolic state $(\ell, \mathcal{P})$ is shown on the left hand side of Figure 2. Suppose that $\mathcal{I}$ and $\mathcal{Q}$ are the polyhedra representing the invariant and the flow condition for location ℓ. Then, $\mathsf{post}_C(\mathcal{P}) = (\mathcal{P} \nearrow \mathcal{Q}) \cap \mathcal{I} = \mathcal{R}$,[8] is computed by first adding rays r_1 and r_2 to $\mathcal{P}$ and then computing set intersection to restore the invariant $\mathcal{I}$.

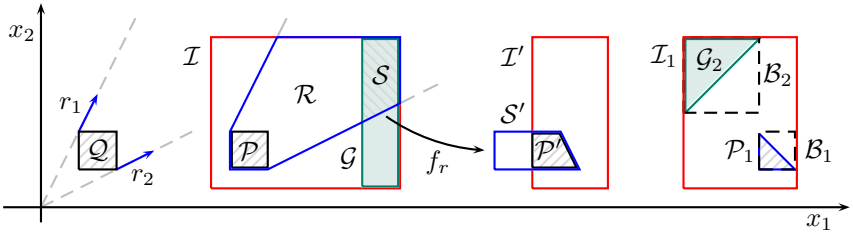


Fig. 2. Continuous and discrete post operators.

Implementing the discrete post operator

The discrete post operator can be implemented by combining several lower level operators on the polyhedra domain.

[7] Not all polyhedra libraries directly support this operator: PPL/PPLite provide an operator named *time_elapse_assign*; the Apron interface defines an equivalent function named *add_ray_array*; the operator is not available in ELINA and VPL.

[8] Polyhedron $\mathcal{R}$ is shown with a blue border; it contains both $\mathcal{P}$ and $\mathcal{S}$.

Uncontrolled assignments. Consider first the case of an uncontrolled assignment to the variables in the set $U \subseteq X$. In order to avoid projection (which would imply a change of space dimension), this can be implemented by the existential quantification of the variables in U, followed by the intersection with the location invariant. When using the DD method, existential quantification is obtained by (incrementally) adding the set of rays $R_U = \{ e_u, -e_u \mid u \in U \}$, where each e_u is the standard basis vector for variable u.[9]

Guarded assignments. Let $e \in \mathsf{Edg}$ be a transition composed by a polyhedral guard $\mathcal{G}$ and a reset $x' = r(x)$ only containing affine assignments. Then, the image of relation $\mathsf{Jump}(e)$ on input $\mathcal{P}$ can be computed as $\mathcal{P}' = f_r(\mathcal{P} \cap \mathcal{G})$, where $f_r : \mathbb{R}^n \to \mathbb{R}^n$ is the linear transformation modeling the reset r. To avoid inefficiencies, particular care has to be taken when implementing the linear transformation f_r. Most polyhedra libraries implement a *sequential assignment* operator, which can be directly used when the (parallel) reset operator r does not contain cyclic dependencies, so that the assignments can be topologically sorted without affecting their semantics. The sequential assignment further distinguishes between invertible and non-invertible assignments. In the invertible case (e.g., $x_1' = 2 \cdot x_1 + x_2$), both the constraint and the generator forms can be updated by simply applying f_r^{-1} and f_r, respectively. In the non-invertible case (e.g., $x_1' = x_2 + 3$), only the generator form is updated (using f_r), while the constraint form has to be recomputed from scratch using the conversion procedure. An alternative approach, better exploiting the incrementality of the DD method, implements the non-invertible assignment by temporarily adding a fresh variable. Letting $X' = X \setminus \{x_1\} \cup \{x_1'\}$, the assignment $x_1' = \mathrm{rhs}$ can be computed as[10]

$$\big((\mathcal{P} \uparrow_{\{x_1'\}}) \cap \{x_1' - \mathrm{rhs} \leq 0, \mathrm{rhs} - x_1' \leq 0\}\big) \downarrow_{X'},$$

followed by a renaming of x_1' into x_1. This approach has been extended in [7] so as to be also applicable to parallel assignments having cyclic dependencies: the parallel assignment is compiled into an equivalent sequence of sequential assignments, taking care to introduce a minimal number of fresh variables (only when breaking a dependency cycle).

An example of application of the discrete post operator is shown in the middle of Figure 2. Suppose that there exists a single transition (ℓ, e, ℓ') exiting from source location ℓ, having the polyhedron $\mathcal{G}$ as guard component and a reset component modeled by affine transformation f_r (which combines a rotation and a translation); let also $\mathcal{I}'$ be the invariant for the target location ℓ'. Then, starting from $\mathcal{R}$, we obtain $\mathsf{post}_D(\mathcal{R}) = f_r(\mathcal{R} \cap \mathcal{G}) \cap \mathcal{I}' = f_r(\mathcal{S}) \cap \mathcal{I}' = \mathcal{S}' \cap \mathcal{I}' = \mathcal{P}'$. On the right hand side of Figure 2 we also show an example where, by using the boxed polyhedra proposal of [7], it is sometimes possible to efficiently detect

[9] Polyhedra libraries often directly support the existential quantification operator; e.g., the *unconstrain* operator in PPL/PPLite and the *forget* operator in Apron.

[10] We denote by $\mathcal{P} \uparrow_Y$ the addition to polyhedron $\mathcal{P}$ of the fresh, i.e., unconstrained, variables in Y, where $X \cap Y = \emptyset$.

disabled transitions. Namely, the cheaper (but incomplete) check for disjointness $\mathcal{B}_1 \cap \mathcal{B}_2 = \emptyset$ on the bounding boxes $\mathcal{B}_1$ and $\mathcal{B}_2$ for the polyhedron state $\mathcal{P}_1$ and the polyhedral guard $\mathcal{G}_2$, when successful, is enough to conclude that $\mathcal{P}_1$ and $\mathcal{G}_2$ are disjoint too.

Computing overapproximations for scalability

While some verification tasks require the *exact* symbolic computation of the set of reachable states, there exists cases where an overapproximation may be good enough (e.g., when trying to prove a safety property of the hybrid automaton). One possibility is to choose a less precise symbolic domain, such as octagons [32] or template polyhedra [38]. A less radical alternative is to maintain the full generality of the domain of convex polyhedra and give up some precision in specific contexts or on specific operators. As a classical example, in [24] all symbolic states $(\ell_i, \mathcal{P}_i)$ for location ℓ_i are merged into a single state $(\ell_i, \uplus\{\mathcal{P}_i\})$. Since the computation of the convex polyhedral hull might still be expensive, it can be further approximated by computing, for instance, their *constraint hull* $(\ell_i, \uplus_{\mathrm{w}}\{\mathcal{P}_i\})$ (also called *weak join* [37]): this resembles the join operator defined on template polyhedra, since it is restricted to only use those constraint slopes that already occur in the arguments. Figure 3 shows a simple example of the different levels of overapproximation obtained.

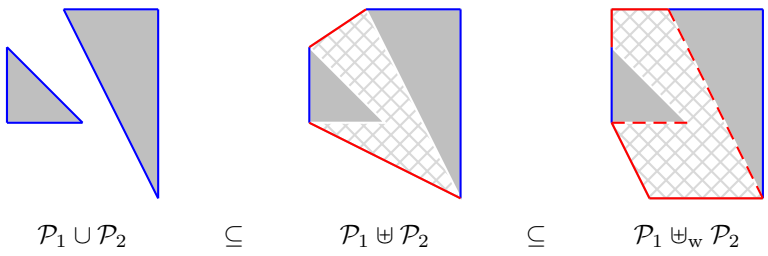


Fig. 3. Set union overapproximations: polyhedral hull vs. constraint hull.

At the implementation level, the constraint hull of a set of polyhedra can be computed either by solving many Linear Programming problems or by enumerating the generators of the arguments. The latter approach is adopted in PPLite, which is the only library based on the DD method directly supporting this operator. Some tools (e.g., PHAVer) allow for the user to choose *if* and *how* to approximate set union by using a single polyhedron per location.

4 Symbolic Analysis using CEGAR

Polyhedral representations enable the exact symbolic analysis of linear hybrid automata. Exact analysis provides strong soundness guarantees in the sense that,

if a counterexample to a safety property is identified over the symbolic representation, then a trajectory corresponding to this counterexample must exist also over the system dynamics. Also, if the analysis terminates by finding a fixed point and without identifying any counterexample, then the system is safe. However, safety verification by exact symbolic analysis may be limiting in some cases for the following two reasons. First, it is computationally costly in general. Computing the result of a post operator amounts to computing a projection of a system of linear inequalities, that is, the projection over the output variables of a system that represents time-elapse (in the continuous case) or discontinuous update (in the discrete case). Methods for computing these projections include quantifier elimination methods as, e.g., the Fourier-Motzkin algorithm [41], or double-description methods (see Sect. 3). Whereas modern solvers for computing projections are efficient in many practical instances, they may suffer from exponential complexity blow-ups in the worst case. Moreover, a second limitation of exact analysis is that it may produce overly tight representations of the state space. It may in some cases prevent the overall safety analysis from identifying a fixed point and, therefore, produce an answer at all. A method that tackles both shortcomings is abstraction.

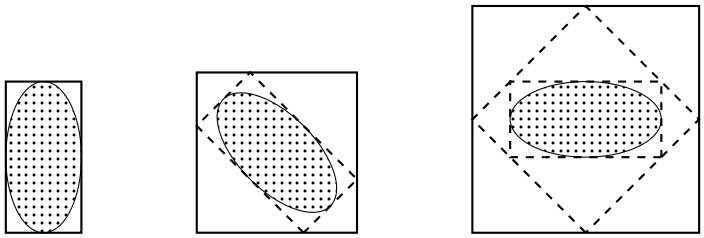
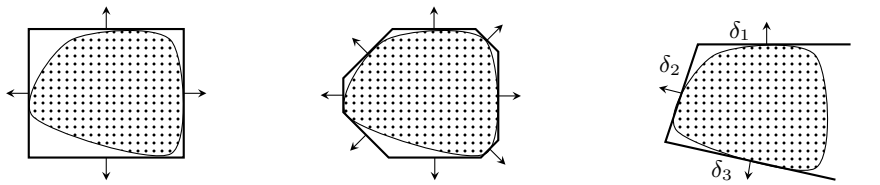


Fig. 4. The wrapping effect.

Abstractions for hybrid systems come with a wide variety of flavors which typically depend both on the kind of systems under analysis and the safety specifications of interest. Examples are abstractions based on interval arithmetic, which enjoy a high generality as they can even account for system dynamics described using polynomial and transcendental functions (that is, more general than LHA) and also enjoy high efficiency. On the other hand, interval analysis suffers from the wrapping effect. An example for the wrapping effect is shown in Fig. 4. The system in this example rotates an ellipse counterclockwise by 45 degrees in discrete time steps, that is, a two-dimensional systems whose dynamic is governed by a linear difference equation. An abstraction based on interval analysis constructs rectangles (1) whose facets are orthogonal to the axes of the state space of interest and (2) that over-approximate the initial set of states (the init set) and the result of every computation of the post operator. In this instance, at the first step (Fig. 4a) the abstraction constructs a rectangle that

encloses the init but also includes states that do not belong to it, introducing a small error. At the second step (Fig. 4b) the post operator is first applied to the abstract set of states (depicted with dashed lines) and then abstracted again within a larger rectangle, which introduces a further error with respect to the original set of states (the ellipse). The process is repeated over the third (Fig. 4c) and all successive steps, and this causes an ever increasing accumulation of over-approximation error.

Abstract safety analysis based on over-approximation conserves soundness with respect to exact analysis in the sense that, upon termination, if the abstract reach set is disjoint from an unsafe region then the system is safe. However, it loses the property for which counterexamples are always genuine. An example can be constructed over Fig. 4, considering a bad region that intersect an abstract set of states (a rectangle) but does not intersect the concrete set of states (the ellipse). In this case, an abstract safety analyser would produce a spurious counterexample to safety. On the other hand, it should be clear that if a bad region is disjoint from the abstract states then it is also disjoint from the concrete states.



Rectangle (a) Octagon (b) $\{\delta_1, \delta_2, \delta_3\}$-polyhedron (c)

Fig. 5. Template polyhedra.

Abstractions are lightweight to compute, but may produce spurious counterexamples. Exact safety analysis always produces genuine counterexamples, but relies on heavy machinery. The approach that capitalises over the advantages of both worlds is counterexample-guided abstraction refinement (CEGAR) []. It consists of two phases, one that abstracts the system and another which refines the abstraction, which interact in a loop. The fundamental ingredient of a CEGAR loop is an abstraction that admits refinement, that is, an abstraction whose precision can be made tighter and tighter by changing some parameters. One successful example of parameterised abstraction is that of *template polyhedra* []. This is a generalisation of the classic rectangular and octagonal to polyhedra whose facets are normal to the vectors in a finite set (see Fig. 5), which we call the template. More formally, where $X \subseteq \mathbb{R}^n$ is a set of states and $\Delta \subset \mathbb{R}^n$ is a template, we call the Δ-polyhedron of X the set defined as follows:

$$\cap\{\underbrace{\{x \colon \langle x, \delta\rangle \leq \rho_X(\delta)\}}_{\text{supporting halfspace}} \colon \delta \in \Delta\}, \tag{2}$$

where $\rho_X(d) = \sup\{\langle x, \delta\rangle : x \in X\}$ denotes the support function of X in direction δ. In other words, a template polyhedron a finite intersection of supporting halfspaces of X; a supporting halfspace in the tightest halfspace that is normal to a given direction. The intersection of these halfspaces is thus defined by the directions template Δ. Rectangular and octagonal abstractions are special cases of templates polyhedra, as Fig. 5 exemplifies. Abstract safety analysis can be seen as the fixed point iteration of Eq. 1, but where post_D and post_C over-approximate sets of states using template polyhedra.

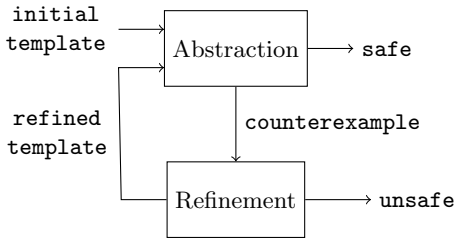


Fig. 6. Architecture of a CEGAR loop.

Refining a template polyhedron amounts to adding directions to the template. Intuitively, the more the directions are, the tighter the abstraction is. The objective of an abstraction refinement scheme for template polyhedra is identifying a template that avoids finding any spurious counterexamples. A CEGAR loop constructs this template incrementally. As depicted in Fig. 6, the initial phase computes an abstraction using some initial template. If the abstraction is determined safe then the system is also safe and then the loop terminates and returns safe. If the abstraction identifies a counterexample then this is passed to the refinement phase. Refinement determines whether the counterexample is genuine or proposes a new template. In the earlier case the loop terminates and returns unsafe. In the latter case, refinement computes a template, that is, adds new directions to the existing template, which excludes the latest spurious counterexample from the abstraction. This refined template is passed to the abstraction phase and the loop is repeated. As a result, the loop enumerates spurious counterexamples and adds directions to the template until either all counterexamples are eliminated or some genuine counterexample is found. While the loop may in general never terminate, we ensure that the same counterexample is never encountered twice; this ensures progress.

A counterexample is a path over the control graph of the hybrid automaton for which abstract symbolic analysis encounters a bad state. The region that results from this path can be seen as a sequence of post operators from initial to bad state. Refining the template so as to eliminate the counterexample amounts to identifying exactly one direction for each step along the counterexample. In turn, this amounts to identifying a sequence of *halfspace interpolant* along

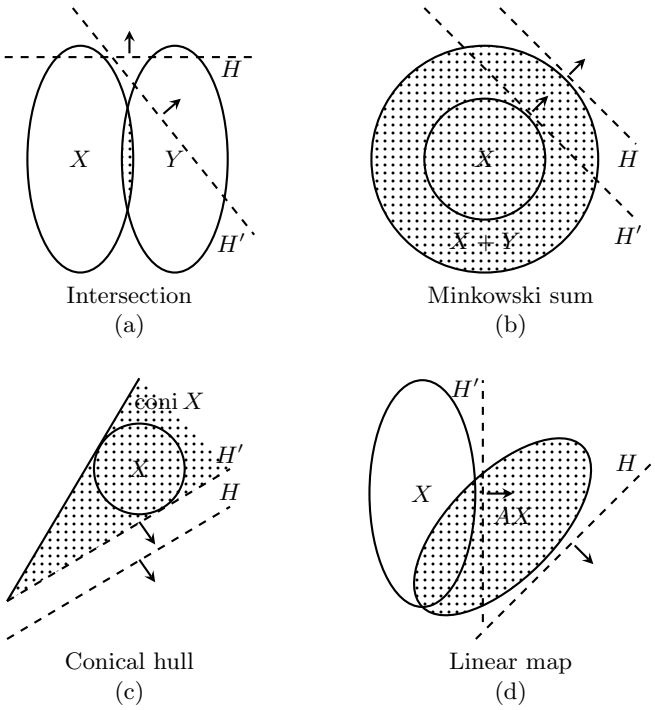


Fig. 7. Halfspace interpolants.

these steps such that the last halfspace separates the result from the bad region []. More precisely, if X_0 is the set of initial states, B is the bad state, and $\mathsf{post}_1, \ldots, \mathsf{post}_k$ is the sequence of post operators, we construct a sequence of halfspaces $H_1, \ldots, H_k \subseteq \mathbb{R}^n$ such that

$$\mathsf{post}_1(X_0) \subseteq H_1, \mathsf{post}_2(H_1) \subseteq H_2 \ldots, \mathsf{post}_k(H_{k-1}) \subseteq H_k, H_k \cap B = \emptyset. \quad (3)$$

To compute these halfspaces we break down these post operators into combinations of basic operations over sets. As indicated in Sect. 3, four operations are sufficient for the symbolic analysis of LHA: intersection between sets $X \cap Y$, Minkowski sum $X + Y = \{x + y \colon x \in X, y \in Y\}$, conical hull $\operatorname{coni} X = \{\lambda x \colon x \in X, \lambda > 0\}$, and linear map $AX = \{Ax \colon x \in X\}$ where $A \in \mathbb{R}^{n \times m}$. Then, we reason inductively over every operation. In other words, for any halfspace H that contains the result of an operation, i.e., $X \cap Y \subseteq H$, $X + Y \subseteq H$, $\operatorname{coni} X \subseteq H$, or $AX \subseteq H$, we compute a second halfspace H' that includes one operand, i.e., $X \subseteq H'$, and abstracts it so as to preserve inclusion of the result into H, i.e., $H' \cap Y \subseteq H$, $H' + Y \subseteq H$, $\operatorname{coni} H' \subseteq H$, or $AH' \subseteq H$ respectively. Exemplars are depicted in Fig. 7. As it turns out, these sequences of halfspace interpolants always exists if and only if the counterexample is spurious and can be computed efficiently by solving a large linear program [9]. The performance of this CEGAR approach will be illustrated by some experiments in the next section.

5 Experiments

We now provide some experimental evidence of the feasibility of the proposed approaches for the analysis of Linear Hybrid Automata and of the practical impact of the enhanced implementation techniques.

One of the most important aspects of an experimental evaluation is the selection of a suitable set of benchmarks. In particular, in order to obtain significant experimental results, it is necessary to collect a sufficiently varied set of benchmarks, possibly originating from different applicative contexts. At the same time, the complexity of the benchmarks needs to be tunable, e.g., by changing the values of a few parameters: tunable benchmarks allow to easily switch from simple instances, useful to test prototype implementations based on novel analysis approaches, to challenging instances that are more effective when testing mature implementations, for instance to record their incremental progresses in terms of efficiency and scalability. A step in this direction has been taken by the ARCH (Applied Verification for Continuous and Hybrid System) workshops,[11] which have hosted several editions of the ARCH-COMP friendly competition on the analysis and verification of hybrid systems. Here we focus our attention on a subset of the tests collected for the category of hybrid system with piecewise constant dynamics [11], which includes several (variants of) benchmarks that have been proposed in the relevant literature. In total, the 2020 edition of the

[11] `https://cps-vo.org/group/ARCH`

competition was considering 28 instances of verification problems,[12] divided in 19 *safe* instances (where the goal is to prove that the system definitely satisfies a given safety property) and 9 *unsafe* instances (where the goal is to prove that the system definitely violates a safety property).

In the following, we present experimental results obtained with tools that implement fixed-point algorithm in Sect. 2.3, the polyhedral computations from Sect. 3 to different degrees, and the CEGAR approach from Sect. 4:

- PHAVer [17] is a formal verification tool for computing reachability and equivalence (simulation relation) of hybrid systems. PHAVer uses standard operations on polyhedra for the reachability computation in a way similar to HyTech, but calls on the Parma Polyhedra Library [3] and its infinite precision arithmetic do the heavy lifting.[13] PHAVer/SX is a subset of PHAVer, included as a plugin in the tool platform SpaceEx [21].
- PHAVerLite is a variant of PHAVer that uses the polyhedra library PPLite [6], which employs a novel representation and conversion algorithm [5] for NNC (Not Necessarily Closed) polyhedra. PHAVer-lite is an earlier version, implemented as a SpaceEx plugin.
- Lyse [19] is a tool for the reachability analysis of convex hybrid automata, namely hybrid automata with piecewise constant dynamics, whose constraints are possibly non-linear but required to be convex. In this class are LHA but also HA whose flow is contrained in ellipses and parabolae. Lyse performs forward reachability analysis by means of template-polyhedra, whose directions are incrementally extracted from spurious counterexamples. The extraction is performed by a technique that generates interpolants by means of convex programming [9], as outlined in Sect. 4.

5.1 Distributed Controller

In Table 1 we provide some evidence of the incremental efficiency improvements that have been obtained in recent years. To this end, we consider the Distributed Controller (DISC) benchmarks [26], which model a distributed controller for a robot, reading data from multiple sensors and processing them according to multiple priorities. The instances DISCn are parametric on $n \in \{2, 3, 4, 5\}$, which is the number of sensors: the product automaton has $1 + 4n$ variables and $4 \times (1 + n) \times 4^n$ locations. The verification goal is to prove a safety property, so that overapproximations are allowed. The rows in Table 1 are labeled by a year corresponding to the edition of the competition; they provide the overall execution time spent by the corresponding model checking tool (we focus on the tools derived from PHAVer). In editions 2017 and 2018 the tools were both configured to compute the exact set of reachable states, so that instances with

[12] We count instances of *unbounded* verification problems; each of these was paired by a *bounded* instance, where the search is limited up to a certain computation depth.

[13] While PHAVer provides overapproximation and widening operators that can force termination at the cost of reduced precision, these operators were not used in the experiments presented in this section.

Table 1. The progress on computation times for the DISC benchmarks.

		DISC2	DISC3	DISC4	DISC5
edition	tool	computation time in [s]			
2017	PHAVer/SX	1.1	—	—	—
2018	PHAVer-lite/SX	0.1	548.0	—	—
2019	PHAVerLite-0.1	0.04	0.68	77.51	—
2020	PHAVerLite-0.3.1	0.04	0.35	2.59	27.99

$n > 3$ were timing out.[14] In edition 2019, PHAVerLite-0.1 was configured to over-approximate set unions using the constraint hull operator, thereby also solving the instance with $n = 4$; finally, in edition 2020 the adoption of the Cartesian factoring representation allowed to solve the instance with $n = 5$. Note that, in Table 1, the computation times obtained by the different versions of PHAVerLite in editions 2018, 2019 and 2020 can be meaningfully compared, since they have been obtained on the very same computer hardware.

In Table 2, which considers the specific instance DISC3, we present a more detailed view of the efficiency contributions provided by the implementation improvements and techniques discussed in Section 3. The first four columns of the table show the tool configuration indicating: (filter w-list) whether redundant polyhedra are removed from the waiting list; (boxing) whether polyhedra are boxed to speed up inclusion tests; (con-hull) whether set union is approximated using the constraint hull; and (factoring) whether polyhedra are represented using Cartesian factoring. For each of the considered combinations, in the next four columns we show: (iter) the number of iterations of the algorithm; (p-list) the final length of the passed list of polyhedra; (r-loc) the number of the reachable locations of the automaton; (time) the overall time spent by the tool.

The first three rows in Table 2 show that, when the *exact* reachable set needs to be computed, the filtering and boxing techniques are quite effective in improving efficiency. The last two rows show that, for the considered benchmark, the constraint hull approximation provides another significant efficiency improvement; note that precision is degraded (for instance, we obtain 78 reachable locations instead of the 67 recorded in the exact case), but the overapproximation is precise enough to prove the required safety property. For this specific instance, the Cartesian factoring technique only yields a marginal improvement (in absolute terms); its effects become more relevant when considering the bigger instances: for instance, as shown in Table 1, for $n = 4$ the time drops from 77.51 to 2.59 seconds.

Note that this incremental progress is not really specific of the considered benchmark: while referring the interested reader to the series of reports of the competition, we note that in the 2017 edition the considered tool was able to

[14] The time improvement in 2018 is due to replacing the PPL library with PPLite.

Table 2. The effect of implementation techniques on DISC3.

filter	w-list	boxing	con-hull	factoring	iter	p-list	r-loc	time
no		no	no	no	63805	63805	67	1379.4
yes		no	no	no	9652	5506	67	93.1
yes		yes	no	no	9652	5506	67	10.3
—		—	yes	no	189	78	78	0.7
—		—	yes	yes	189	78	78	0.4

successfully prove/disprove 13 of the 20 verification tasks in about 4 hours and 40 minutes, whereas in the 2020 edition 27 of the 28 verification tasks were completed in less than 3 minutes.

5.2 Adaptive Cruise Controller

With this next benchmark, we compare the performance of the set-propagation approach implemented in PHAVer with the CEGAR approach implemented in the tool Lyse. The adaptive cruise controller is a distributed system for assuring that safety distances in a platoon of cars are satisfied [9]. For n cars, the number of discrete states is 2^n and the number of continuous variables is n. Each variable x_i encodes the relative position of the i-th car. The relative velocity of each car has a drift $|\dot{x}_i - \dot{x}_{i+1}| \leq 1$ when cruising and $|\dot{x}_i - \dot{x}_{i+1} - \varepsilon| \leq 1$ when recovering, where ε is a slow-down parameter. The cars can stay in cruise mode as long as the distance to the preceding vehicle is greater 1. The can go into recovery mode when the distance is smaller than 2. The specification is that the distance between adjacent cars should be positive.

Table 3 shows the computation times for instances of different complexity. First, we observe that the set propagation approach shows similar performance characteristics as in the DISC benchmark: The advances associated with the polyhedra library PPLite, as well as the heuristic improvements to the fixed-point algorithm lead to drastic gains in speed. The CEGAR approach clearly outshines the early versions of the set propagation approach. It also has a clear advantage in unsafe instances, where a counterexample can be found by solving a SAT instance. Somewhat surprisingly, however, the latest generation of set propagation tools seems to outperform CEGAR.

6 Conclusions

In this paper, we tried to draw the arc from straightforward to more sophisticated symbolic analysis methods for linear hybrid automata (LHA). We presented two flavors, one based on set propagation and one based on counterexample-guided abstraction refinemend (CEGAR). The set propagation approach starts from an initial set and then repeatedly computes one-step successors until no new states

Table 3. Computation Times of the Adaptive Cruise Controller [19, 18].

edition	instance (n)	5-safe	5-unsafe	6-safe	6-unsafe	7-safe	7-unsafe	8-safe	8-unsafe
	#locs.	32	32	64	64	128	128	256	256
	tool	computation time in [s]							
2017	Lyse	1.08	≈ 0	–	–	573.35	0.233	–	–
2017	PHAVer/SX	9.4	13.7	461	13430	∞	∞	–	–
2018	PHAVer-lite	1.0	0.9	38.1	22.4	–	–	–	–
2019	PHAVerLite	0.10	0.06	0.55	0.27	4.26	1.39	47.10	7.15

are found. The performance of this approach hinges on the chosen set representation and how efficiently the required operations can be realized. A natural choice for LHA are convex polyhedra in constraint representation. Despite the fact that convex polyhedra are used, e.g., in program analysis, since the seventies, we remark that several advances were made over the years that lead to progressively more efficient libraries for standard operations. In particular, a novel representation for polyhedra with strict as well as non-strict inequalities has lead to gains on this fundamental level. Further gains have been achieved through heuristics in the fixed-point computation algorithm. The common denominator of these heuristics is the use of multiple levels of abstraction: A property like containment is decided by going progressively through different levels of abstraction, so that the most precise and expensive checks are only carried out when cheaper checks have failed. The accumulated gains from both efficient encodings and heuristics are substantial.

The CEGAR approach constructs an abstraction in the form of polyhedra iteratively, by checking whether the abstraction admits a path from the initial states to a given bad set of states, and then refining the abstraction to exclude this path if it turns out to be spurious. CEGAR easily outperformed earlier versions of the set propagation approach and in our experiments it outperforms for unsafe instances, quickly returning an unsafe path as a witness. Compared to more recent implementations of set propagation that leverage efficient encodings and a series of heuristics, CEGAR seems to lose some of the advantage.

This paper provided a small sample of implementations and benchmark instances in order to outline some of the improvements that can be had through clever encodings and heuristics. Further experimentation is needed to evaluate in which application domains such gains translate to successful analysis results.

References

1. R. Alur. Formal verification of hybrid systems. In S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, editors, *EMSOFT*, pages 273–278. ACM, 2011.

2. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
3. R. Bagnara, P. M. Hill, and E. Zaffanella. Not necessarily closed convex polyhedra and the double description method. *Formal Aspects Comput.*, 17(2):222–257, 2005.
4. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
5. A. Becchi and E. Zaffanella. A direct encoding for NNC polyhedra. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 230–248. Springer, 2018.
6. A. Becchi and E. Zaffanella. An efficient abstract domain for not necessarily closed polyhedra. In A. Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 146–165. Springer, 2018.
7. A. Becchi and E. Zaffanella. Revisiting polyhedral analysis for hybrid systems. In B. E. Chang, editor, *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11822 of *Lecture Notes in Computer Science*, pages 183–202. Springer, 2019.
8. A. Becchi and E. Zaffanella. PPLite: Zero-overhead encoding of NNC polyhedra. *Inf. Comput.*, 275:104620, 2020.
9. S. Bogomolov, G. Frehse, M. Giacobbe, and T. A. Henzinger. Counterexample-guided refinement of template polyhedra. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 589–606, 2017.
10. S. Boulmé, A. Maréchal, D. Monniaux, M. Périn, and H. Yu. The verified polyhedron library: an overview. In *20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2018, Timisoara, Romania, September 20-23, 2018*, pages 9–17. IEEE, 2018.
11. L. Bu, A. Abate, D. Adzkiya, M. S. Mufid, R. Ray, Y. Wu, and E. Zaffanella. ARCH-COMP20 category report: Hybrid systems with piecewise constant dynamics and bounded model checking. In *ARCH20. 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20), Berlin, Germany, July 12, 2020*, volume 74 of *EPiC Series in Computing*, pages 1–15. EasyChair, 2020.
12. L. Bu, Y. Li, L. Wang, X. Chen, and X. Li. BACH 2 : Bounded reachability checker for compositional linear hybrid systems. In *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*, pages 1512–1517, 2010.
13. N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.
14. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, page 52–71, Berlin, Heidelberg, 1981. Springer-Verlag.

15. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.
16. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In A. V. Aho, S. N. Zilles, and T. G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.
17. G. Frehse. PHAVer: algorithmic verification of hybrid systems past HyTech. *STTT*, 10(3):263–279, 2008.
18. G. Frehse, A. Abate, D. Adzkiya, A. Becchi, L. Bu, A. Cimatti, M. Giacobbe, A. Griggio, S. Mover, M. S. Mufid, I. Riouak, S. Tonetta, and E. Zaffanella. Arch-comp19 category report: Hybrid systems with piecewise constant dynamics. In G. Frehse and M. Althoff, editors, *ARCH19. 6th International Workshop on Applied Verification of Continuous and Hybrid Systems*, volume 61 of *EPiC Series in Computing*, pages 1–13. EasyChair, 2019.
19. G. Frehse, A. Abate, D. Adzkiya, L. Bu, M. Giacobbe, M. S. Mufid, and E. Zaffanella. Arch-comp18 category report: Hybrid systems with piecewise constant dynamics. In G. Frehse, editor, *ARCH18. 5th International Workshop on Applied Verification of Continuous and Hybrid Systems*, volume 54 of *EPiC Series in Computing*, pages 1–13. EasyChair, 2018.
20. G. Frehse, S. K. Jha, and B. H. Krogh. A counterexample-guided approach to parameter synthesis for linear hybrid automata. In M. Egerstedt and B. Mishra, editors, *HSCC*, volume 4981 of *LNCS*, pages 187–200. Springer, 2008.
21. G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *LNCS*, pages 379–395. Springer, 2011.
22. N. Halbwachs, D. Merchat, and L. Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods Syst. Des.*, 29(1):79–95, 2006.
23. N. Halbwachs, Y. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. L. Charlier, editor, *Static Analysis, First International Static Analysis Symposium, SAS'94, Namur, Belgium, September 28-30, 1994, Proceedings*, volume 864 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 1994.
24. N. Halbwachs, Y. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods Syst. Des.*, 11(2):157–185, 1997.
25. T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
26. T. A. Henzinger and P. Ho. HYTECH: the cornell hybrid technology tool. In P. J. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems II, Proceedings of the Third International Workshop on Hybrid Systems, Ithaca, NY, USA, October 1994*, volume 999 of *Lecture Notes in Computer Science*, pages 265–293. Springer, 1994.
27. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43:540–554, 1998.
28. T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

29. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.
30. S. K. Jha, B. H. Krogh, J. E. Weimer, and E. M. Clarke. Reachability for linear hybrid automata using iterative relaxation abstraction. In *HSCC*, pages 287–300, 2007.
31. O. Maler. Algorithmic verification of continuous and hybrid systems. In *Int. Workshop on Verification of Infinite-State System (Infinity)*, 2013.
32. A. Miné. The octagon abstract domain. *High. Order Symb. Comput.*, 19(1):31–100, 2006.
33. T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The double description method. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games – Volume II*, number 28 in Annals of Mathematics Studies, pages 51–73. Princeton University Press, Princeton, New Jersey, 1953.
34. A. Podelski and A. Rybalchenko. Armc: The logical choice for software model checking with abstraction refinement. In *PADL*, pages 245–259, 2007.
35. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming*, pages 337–351, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.
36. N. Roohi, P. Prabhakar, and M. Viswanathan. Hybridization based cegar for hybrid automata with affine dynamics. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 752–769. Springer, 2016.
37. S. Sankaranarayanan, M. Colón, H. B. Sipma, and Z. Manna. Efficient strongly relational polyhedral analysis. In E. A. Emerson and K. S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, volume 3855 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2006.
38. S. Sankaranarayanan, T. Dang, and F. Ivancic. Symbolic model checking of hybrid systems using template polyhedra. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2008.
39. G. Singh, M. Püschel, and M. T. Vechev. Fast polyhedra abstract domain. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 46–59. ACM, 2017.
40. P. Tabuada. *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, 2009.
41. H. P. Williams. Fourier's method of linear programming and its dual. *The American mathematical monthly*, 93(9):681–695, 1986.