Symbolic Analysis of Linear Hybrid Automata – 25 Years Later

Goran Frehse[1(✉)], Mirco Giacobbe[2], and Enea Zaffanella[3]

[1] U2IS, ENSTA Paris, Institut Polytechnique de Paris, Paris, France
goran.frehse@ensta-paris.fr
[2] School of Computer Science, University of Birmingham, Birmingham, UK
m.giacobbe@bham.ac.uk
[3] Department of Mathematical, Physical and Computer Sciences, University of Parma, 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 led 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, throughout a collaboration of scientists from various disciplines, notably computer scientists and control theorists. By that time, formal methods such as abstract interpretation [19] and model checking [18,44] 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 can 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 [35]. While properties like safety are not decidable for LHA, the states reachable over a given

J.-F. Raskin and K. Chatterjee (Eds.): Principles of Systems Design, LNCS 13660, pp. 39–60, 2022.
https://doi.org/10.1007/978-3-031-22337-2_3

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 [30], could be applied. This led to symbolic analysis tools such as the pioneering model checker HyTech [33]. 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 led to performance improvements of several orders of magnitude 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 [1,3,40,51] for related work. Other CEGAR approaches are implemented, e.g., in the tools HARE [46] and HyCOMP/IC3 [13]. To take an instance of an entirely different approach, we point to the work in [43], 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 [14].

The remainder of the paper is structured as follows. In Sect. 2, we define linear hybrid automata and give a brief overview on set-based reachability. In Sect. 3 we show how set-based reachability can be implemented efficiently, either exactly or by resorting to overapproximations. In Sect. 4 we present a CEGAR framework than can further enhance scalability. In Sect. 5, a series of experiments illustrates the impact of the different approaches and techniques on performance. Finally, we conclude in Sect. 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 former 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 $a_1x_1 + a_2x_2 + \cdots + a_nx_n \bowtie b$, where $a_1, \ldots a_n$ and b are real-valued constants, and whose sign may be strict or nonstrict (i.e., $\bowtie \in \{<, \leq\}$). A linear constraint is written in vector notation as $a^{\mathsf{T}}x \bowtie b$, 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 set of constraints defining $\mathcal{P}$ is 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 linear hybrid automaton and its run semantics.

Definition 1 (Linear Hybrid Automaton). *[2,32,36] A* linear hybrid automaton $H = (X, \mathsf{Loc}, \mathsf{Edg}, \mathsf{Lab}, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump}, \mathsf{Event})$ *consists of*

- *a finite set of variables $X = \{x_1, \ldots, x_n\}$, partitioned into uncontrolled variables U and controlled variables Y;*
- *a finite directed multigraph* $(\mathsf{Loc}, \mathsf{Edg})$*, called the control graph, which consists of a set of locations* $\mathsf{Loc} = \{\ell_1, \ldots, \ell_m\}$ *that represent discrete modes, and a set of edges* Edg *that represent discrete transitions between modes;*

- *a finite set of synchronization labels* Lab*;*
- *a polyhedral constraint over variables* $\mathsf{Inv}(\ell) \in \mathbb{R}^X$ *called invariant or staying condition, which restricts the values the variables can possibly take over location* $\ell \in \mathsf{Loc}$*; a* state *of* H *consists of a location* ℓ *and a value* $x \in \mathsf{Inv}(\ell)$ *for the variables, and is denoted by* $s = (\ell, x)$*;*
- *a polyhedral constraint* $\mathsf{Init}(\ell) \subseteq \mathsf{Inv}(\ell)$ *called initial condition, which determines the set of initial values for the variables at location* $\ell \in \mathsf{Loc}$*; every behavior of* H *must start in one of the initial conditions;*
- *a polyhedral constraint over dotted variables* $\mathsf{Flow}(\ell) \subseteq \mathbb{R}^{\dot{X}}$ *called flow condition, which gives for each location* $\ell \in \mathsf{Loc}$ *the set of possible derivatives a trajectory can possibly take using a differential inclusion such as* $\dot{x} \in \mathsf{Flow}(\ell)$*;*
- *a polyhedral constraint* $\mathsf{Jump}(e) \subseteq \mathbb{R}^X \times \mathbb{R}^{X'}$ *over unprimed and primed variables called jump relation, which defines the set of possible successors* x' *of* x *when transition* $e \in \mathsf{Edg}$ *is taken; jump relations are typically given by a polyhedral guard constraint* $\mathcal{G}_e \subseteq \mathbb{R}^X$ *and an affine assignment (or reset)* $x' = r_e(x)$ *as* $\mathsf{Jump}(e) = \{(x, x') \mid x \in \mathcal{G}_e \wedge x' = r_e(x)\}$*; also, under certain definitions, every location* ℓ *is associated with an uncontrolled transition* $\bar{e} \in \mathsf{Edg}$ *from* l *to itself and jump relation defined as* $\mathsf{Jump}(\bar{e}) = \{(x, x') \mid x' \downarrow_Y = x \downarrow_Y \wedge x' \in \mathsf{Inv}(\ell)\}$*, which represents arbitrary assignments that the environment might perform on the uncontrolled variables* $U = X \setminus Y$*;*
- *an event function* $\mathsf{Event}(e) \in \mathsf{Lab} \cup \{\tau\}$ *that maps every edge* $e \in \mathsf{Edg}$ *to either a synchronization label or the internal event* τ*; uncontrolled transitions are always mapped to* τ*.*

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.

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

$$(\ell_0, x_0) \xrightarrow{\delta_0} (\ell_0, y_0) \xrightarrow[e_1]{\alpha_1} (\ell_1, x_1) \xrightarrow{\delta_1} (\ell_1, y_1) \xrightarrow[e_2]{\alpha_2} \dots \xrightarrow{\delta_k} (\ell_k, y_k),$$

that satisfies for the three conditions listed below.

1. Initialisation: *the first state satisfies an initial condition, i.e.,* $x_0 \in \mathsf{Init}(\ell_0)$.
2. Continuous flow: *for every* $i = 0, \dots, k$*, there exist a trajectory* $\xi : [0, \delta_i] \to \mathbb{R}^X$ *from* x_i *to* y_i *and with dwell time* $\delta_i \in \mathbb{R}_{\geq 0}$ *over location* l_i*, that is,* ξ *is a continuously differentiable function such that* $\xi(0) = x_i$*,* $\xi(\delta_i) = y_i$*, and it holds true that* $\dot{\xi}(t) \in \mathsf{Flow}(\ell_i)$ *and* $\xi(t) \in \mathsf{Inv}(\ell_i)$ *for all* $t \in [0, \delta_i]$.
3. Discontinuous jumps: *for every* $i = 1, \dots, k$ *we have that* $e_i \in \mathsf{Edg}$ *has source* ℓ_{i-1} *and destination* ℓ_i*,* $\alpha_i = \mathsf{Event}(e_i)$*, and* $(y_{i-1}, x_i) \in \mathsf{Jump}(e_i)$.

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 [27] and in Counter Example Guided Abstraction Refinement (CEGAR) frameworks [38], which we

data: lists of symbolic states W and R, initially empty

foreach $\ell \in \mathsf{Loc}$ *s.t.* $\mathsf{Init}(\ell) \neq \emptyset$ **do**
  $\mathcal{P} \leftarrow \mathsf{post}_C(l, \mathsf{Init}(\ell))$;
  push $(\ell, \mathcal{P})$ into the waiting list W;
while $W \neq \emptyset$ **do**
  pop $(\ell, \mathcal{P})$ from W;
  if $\mathcal{P} \subseteq \mathcal{P}''$ *and* $\ell = \ell''$ *for some* $(\ell'', \mathcal{P}'') \in R \cup W$ **then**
    continue;
  foreach $e \in \mathsf{Edg}$ *with source* l *and destination* l' **do**
    $\mathcal{P}' \leftarrow \mathsf{post}_C(\ell', \mathsf{post}_D(e, \mathcal{P}))$;
    push $(\ell', \mathcal{P}')$ into W;
  add $(\ell, \mathcal{P})$ to the passed list R;

Algorithm 1: Symbolic analysis procedure.

will discuss in more detail in Sect. 4. It also follows from these semantics that with a simple model transformation,[1] a LHA can be verified by model checkers able to handle linear constraints over the rationals, see [43].

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 variables valuations $S \subseteq \mathbb{R}^n$, let $\mathsf{post}_C(\ell, S)$ be the set of valuations reachable by letting time elapse from any valuation in S over location $\ell \in \mathsf{Loc}$,

$$\mathsf{post}_C(\ell, S) = \left\{ y \;\middle|\; \exists x \in S, \delta \in \mathbb{R}_{\geq 0} \colon (\ell, x) \xrightarrow{\delta} (\ell, y) \right\}. \tag{1}$$

Let $\mathsf{post}_D(e, S)$ be the set of valuations resulting from transition $e \in \mathsf{Edg}$ from any valuation in S

$$\mathsf{post}_D(e, S) = \left\{ x' \;\middle|\; \exists x \in S : (\ell, x) \xrightarrow[e]{\alpha} (\ell', x') \right\}, \tag{2}$$

where ℓ and ℓ' are source and target locations of e and $\alpha = \mathsf{Event}(e)$.

Starting from the initial states, post_C and post_D are applied in alternation along the structure of the control graph. In model checkers such as HyTech [33], PHAVer [22] and SpaceEx [28], the symbolic analysis of a linear hybrid automaton is performed using *symbolic states* $s = (\ell, \mathcal{P})$, where $\ell \in \mathsf{Loc}$ and $\mathcal{P}$ is 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, 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 according to the steps below.

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

1. Initialization: compute the continuous successors of the initial states, put them on the waiting list, and proceed to step 2.
2. Containment checking: pop a symbolic state $s = (\ell, \mathcal{P})$ from the waiting list and check whether it has been encountered before, i.e., it is subsumed by some symbolic state in passed or waiting list. Repeat step 2 until either the waiting list is empty or a never encountered state is found. If the waiting list is empty, terminate and return. If a new state is found, proceed to step 3.
3. Post computation: compute all one-step successors $\mathsf{post}_C(\ell', \mathsf{post}_D(e, \mathcal{P}))$ of s along all transition e that are outgoing from ℓ (and have destination ℓ') and push them to the waiting list. Add s to the passed list and repeat step 2.

Upon termination of the procedure (which happens empirically, and is not guaranteed in general), the passed list R represents the whole set of reachable states.

An important aspect of the fixed point algorithm outlined above is that it attempts to reduce redundant exploration. This is taken care of by step 2, resp. line 6 in Algorithm 1, which implicitly performs the following operations:

2.1 it discards states that are contained in any symbolic state on the passed list,
2.2 it discards states that are contained in any remaining symbolic state on the waiting list—this is known as *waiting list filtering*.

Note that filtering the waiting list is a heuristic, which may or may not lead to an efficiency improvement with respect to a containment check over the passed list only; however, in practice, it leads to savings in computational time which compensates for the overhead of comparing a large number of symbolic states.

3 Implementing Symbolic Analysis Using Polyhedra

In this section we briefly describe how to implement the symbolic analysis outlined in Sect. 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,[2] the classical approach [20] is based on the Double Description (DD) method [42], where the constraint form is paired with a *generator form* and conversion algorithms [16] can compute 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 [50], NewPolka in Apron [37], PPL (Parma Polyhedra Library) [6], and PPLite [10]; the last three also support strict linear constraints.

[2] For instance, VPL (Verified Polyhedron Library) [12].

Generator Form and $\mathcal{V}$-Polyhedra. The classical definition of generators for closed polyhedra has been extended in [6] 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 [30,31] and further developed in [5], 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 [7,8] 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. Converting between $\mathcal{H}$ and $\mathcal{V}$ polyhedra has a worst case complexity that is exponential in the size of the input representation; it is therefore essential to keep representations small, e.g., by removing redundancies. Cartesian factoring [29] can greatly reduce the space needed to represent a $\mathcal{V}$-polyhedron. The space dimensions $X = \{x_1, \ldots, x_n\}$ are *partitioned* into a sequence of blocks $(B_1, \ldots, B_k)$ so that each linear constraint

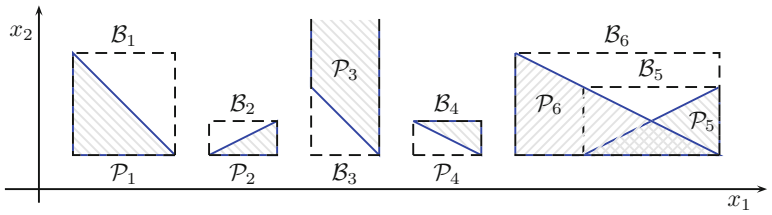

Fig. 1. Incomplete decision procedures speed up the containment checks.

in the $\mathcal{H}$-representation mentions the dimensions of a single block B_i; then, the $\mathcal{H}$-polyhedron $\mathcal{P}$ is factored into k polyhedra $(\mathcal{P}_1, \ldots, \mathcal{P}_k)$; if needed, these $\mathcal{H}$-polyhedra are converted to a sequence of $\mathcal{V}$-polyhedra. This can be much more efficient in time and space compared to a direct conversion. The ELINA library [50] uses 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 [9], 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 [9] 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.[3]

Figure 1 shows a few examples, where for each polyhedron $\mathcal{P}_i$ (solid blue) we draw the corresponding bounding box $\mathcal{B}_i$ (dashed black). Intuitively, we know that $\mathcal{P}_1 \not\subseteq \mathcal{P}_2$ because $\text{vol}(\mathcal{B}_1) > \text{vol}(\mathcal{B}_2)$; we know that $\mathcal{P}_3 \not\subseteq \mathcal{P}_1$ because $\text{num\_rays}(\mathcal{B}_3) = 1 > 0 = \text{num\_rays}(\mathcal{B}_1)$; we know that $\mathcal{P}_2 \not\subseteq \mathcal{P}_4$ 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 $\mathcal{B}_5 \subseteq \mathcal{B}_6$ no semi-decision procedure applies and we need to resort to the more expensive polyhedra containment check.

[3] 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.

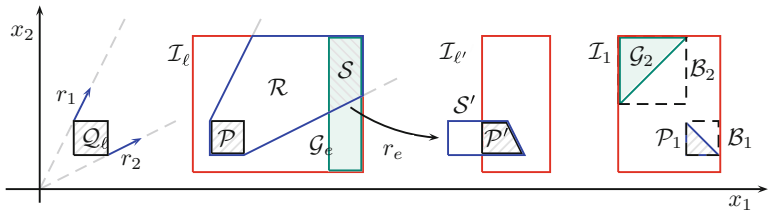

Fig. 2. Continuous and discrete post operators.

Implementing the Continuous Post Operator

For a fixed location ℓ, the flow relation of an LHA is specified by a polyhedron $\mathcal{Q}_\ell = \mathsf{Flow}(\ell)$ 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:

$$\mathcal{P} \nearrow \mathcal{Q}_\ell = \{\, p + t \cdot q \mid p \in \mathcal{P}, q \in \mathcal{Q}_\ell, t \in \mathbb{R}, t \geq 0 \,\}. \tag{3}$$

Assuming that $\mathcal{Q}_\ell$ is a closed $\mathcal{V}$-polyhedron described by generators (P, C, R), where $C = \emptyset$, the set $\mathcal{P} \nearrow \mathcal{Q}_\ell$ is a convex polyhedron that can be computed by (incrementally) adding to $\mathcal{P}$ the finite set of *rays* $R' = P \cup R$ [30].[4]

An example of applying the continuous post operator to a symbolic state $(\ell, \mathcal{P})$ is shown on the left hand side of Fig. 2. Suppose that $\mathcal{I}_\ell = \mathsf{Inv}(\ell)$ and $\mathcal{Q}_\ell = \mathsf{Flow}(\ell)$ are the polyhedra representing the invariant and the flow condition for location ℓ. Then, $\mathsf{post}_C(\ell, \mathcal{P}) = (\mathcal{P} \nearrow \mathcal{Q}_\ell) \cap \mathcal{I}_\ell = \mathcal{R}$,[5] is computed by first adding rays r_1 and r_2 to $\mathcal{P}$ and then computing set intersection to restore the invariant $\mathcal{I}_\ell$.

Implementing the Discrete Post Operator

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

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.[6]

[4] 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.

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

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

Guarded Assignments. Let $e \in \mathsf{Edg}$ be a transition composed by a polyhedral guard $\mathcal{G}$ and a reset $x' = r_e(x)$ only containing affine assignments. Then, the image of relation $\mathsf{Jump}(e)$ on input $\mathcal{P}$ can be computed as $\mathcal{P}' = r_e[\mathcal{P} \cap \mathcal{G}_e]$, where $r_e[X] = \{r_e(x) \colon x \in X\}$ denotes the image of set $X \subseteq \mathbb{R}^n$ though the linear transformation r_e. To avoid inefficiencies, particular care has to be taken when implementing the image of linear transformation $r_e[\cdot]$. Most polyhedra libraries implement a *sequential assignment* operator, which can be directly used when the (parallel) reset operator r_e 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 r_e^{-1} and r_e, respectively. In the non-invertible case (e.g., $x_1' = x_2 + 3$), only the generator form is updated (using r_e), 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 $x_1' = \text{rhs}$ can be computed as[7]

$$\big((\mathcal{P} \uparrow_{\{x_1'\}}) \cap \{x_1' - \text{rhs} \leq 0, \text{rhs} - x_1' \leq 0\}\big) \downarrow_{X'}, \tag{4}$$

followed by a renaming of x_1' into x_1. This approach has been extended in [9] 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 Fig. 2. Suppose that there exists a single transition e exiting from source location ℓ to target location ℓ', having the polyhedron $\mathcal{G}_e$ as guard component and a reset component modeled by affine transformation r_e (which combines a rotation and a translation); let also $\mathcal{I}_{\ell'}$ be the invariant for the target location ℓ'. Then, starting from $\mathcal{R}$, we obtain $\mathsf{post}_D(e, \mathcal{R}) = r_e[\mathcal{R} \cap \mathcal{G}_e] \cap \mathcal{I}_{\ell'} = r_e[\mathcal{S}] \cap \mathcal{I}_{\ell'} = \mathcal{S}' \cap \mathcal{I}_{\ell'} = \mathcal{P}'$. On the right hand side of Fig. 2 we also show an example where, by using the boxed polyhedra proposal of [9], it is sometimes possible to efficiently detect *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).

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

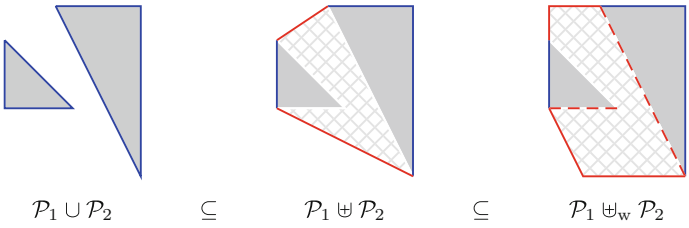

Fig. 3. Set union overapproximations: polyhedral hull vs. constraint hull.

One possibility is to choose a less precise symbolic domain, such as octagons [41] or template polyhedra [48]. 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 [31] 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* [47]): 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. 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

An exact reachability analysis using polyhedra provides strong soundness guarantees in the sense that, if a counterexample to a safety property is identified over the symbolic representation, then corresponding a trajectory must exist in the system. Also, if the analysis finds a fixed point without identifying any counterexample, then the system is safe. However, an exact symbolic analysis is computationally costly. Computing the image of a post operator amounts to a projection of a system of linear inequalities over the output variables of the operator. Methods for computing projections include quantifier elimination methods as, e.g., the Fourier-Motzkin algorithm [52], or double-description methods (see Sect. 3), which suffer from exponential complexity blow-ups in the worst case. Moreover, tight representations of the reachable states may in some cases prevent the reachability algorithm from identifying a fixed point and, therefore, produce an answer at all; conversely, coarser abstractions can help reaching a fixed point. A method that tackles the above shortcomings is abstraction.

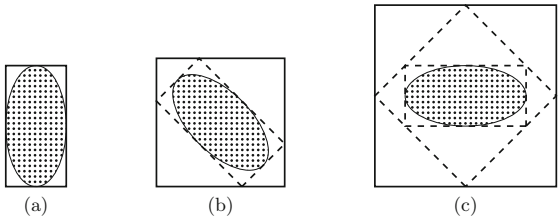

Fig. 4. The wrapping effect.

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° 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 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.

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) [17]. It consists of two phases, one that abstracts the system and another which

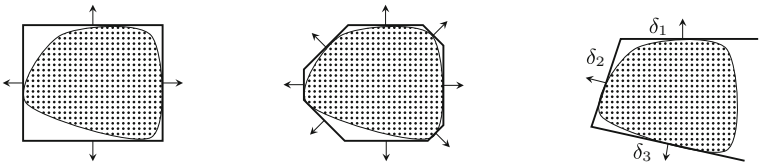

Fig. 5. Template polyhedra.

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 example of parameterised abstraction is that of *template polyhedra*, which has been successfully applied not only to the verification of LHA [21], but also computer programs [49], hybrid automata with linear ODEs [39,48], and more recently neural network control [4]. Formally, template polyhedra are defined in terms of supporting halfspaces, which are in turn defined in terms of support functions. Given a convex set of states $\mathcal{X} \subseteq \mathbb{R}^n$, the support function of $\mathcal{X}$ in a direction $\delta \in \mathbb{R}^n$ is

$$\rho_{\mathcal{X}}(\delta) = \sup\{\langle x, \delta \rangle \colon x \in \mathcal{X}\}, \tag{5}$$

namely, the supremum of the inner product of δ with all elements of $\mathcal{X}$ [45]. This gives the offset with respect to the origin of the tightest halfspace contaning $\mathcal{X}$—the supporting halfspace—that is orthogonal to δ. Finally, a template polyhedron of $\mathcal{X}$ is a finite intersections of supporting halfspaces of $\mathcal{X}$ that are orthogonal to a finite set of directions $\Delta \subset \mathbb{R}^n$. We call Δ a template and define the Δ-polyhedron of $\mathcal{X}$ as the following set:

$$\cap \{\underbrace{\{x \colon \langle x, \delta \rangle \leq \rho_{\mathcal{X}}(\delta)\}}_{\text{supporting halfspace}} \colon \delta \in \Delta\}. \tag{6}$$

Rectangular and octagonal abstractions are special cases of templates polyhedra, as Fig. 5a and b exemplify; an arbitrary template parameterizes the shape of the polyhedron as shown in Fig. 5c.

In this section, abstract safety analysis can be seen as the procedure in Algorithm 1, but where $\mathcal{P}$ and $\mathcal{P}'$ are over-approximated as template polyhedra at respectively lines 2 and 9. Similarly to Sect. 3, we interpret the post operators as operators over sets of states in $\mathbb{R}^n$ as follows:

$$\mathsf{post}_C(\ell, \mathcal{P}) = ((\mathcal{P} \cap \mathcal{I}_\ell) + \operatorname{coni} \mathcal{Q}_\ell) \cap \mathcal{I}_\ell \tag{7}$$

$$\mathsf{post}_D(e, \mathcal{R}) = A_e[\mathcal{R} \cap \mathcal{G}_e] + \{b_e\}, \tag{8}$$

where $\mathcal{I}_\ell = \mathsf{Inv}(\ell)$ are invariant and $\mathcal{Q}_\ell = \mathsf{Flow}(\ell)$ and flow constraint of location ℓ, and transition $e \in \mathsf{Edg}$ is modeled as a guarded assignment with guard $\mathcal{G}_e$ and affine reset function $r_e(x) = A_e x + b_e$, with $A_e \in \mathbb{R}^{n \times n}$ and $b_e \in \mathbb{R}^n$

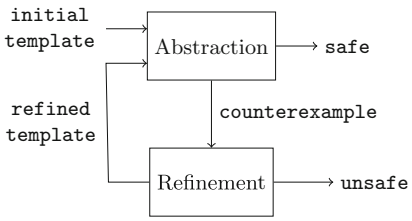

Fig. 6. Architecture of a CEGAR loop.

(a post operator for general jump conditions is described in [11]). Operator $\mathsf{post}_C(\ell, \mathcal{P})$ in Eq. 7 represents time elapse over a mode ℓ, and is equivalent to $((\mathcal{P} \cap \mathcal{I}_\ell) \nearrow \mathcal{Q}_\ell) \cap \mathcal{I}_\ell$ as defined in Eq. 3; operator $\mathsf{post}_D(e, \mathcal{R})$ in Eq. 8 is a rewriting of $r_e[\mathcal{R} \cap \mathcal{G}]$. Here, post operators are defined as combinations of four basic operations over sets:

$$\begin{aligned} &\mathcal{X} \cap \mathcal{Y} && \text{(intersection)} \\ &\operatorname{coni} \mathcal{X} = \{t \cdot x \colon x \in \mathcal{X}, t \geq 0\} && \text{(conical hull)} \\ &\mathcal{X} + \mathcal{Y} = \{x + y \colon x \in \mathcal{X}, y \in \mathcal{Y}\} && \text{(Minkowski sum)} \\ &A[\mathcal{X}] = \{Ax \colon x \in \mathcal{X}\} && \text{(linear map)} \end{aligned}$$

where $\mathcal{X}$ and $\mathcal{Y}$ are sets in $\mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$. Computing template polyhedra for our post operators consists of building the respective support function—inductively—over these operations over sets; a method to inductively construct support functions was introduced in [39], and later extended in [11] with exact intersection and conical hull operations. Abstract safety analysis computes an abstraction of the reach set by Algorithm 1 as a union of template polyhedra, until either it terminates or a counterexample is found; in the latter case, we refine the template polyhedra in a CEGAR loop.

Refining a template polyhedron amounts to adding directions to the template [15,25]. 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 [11,26]. 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

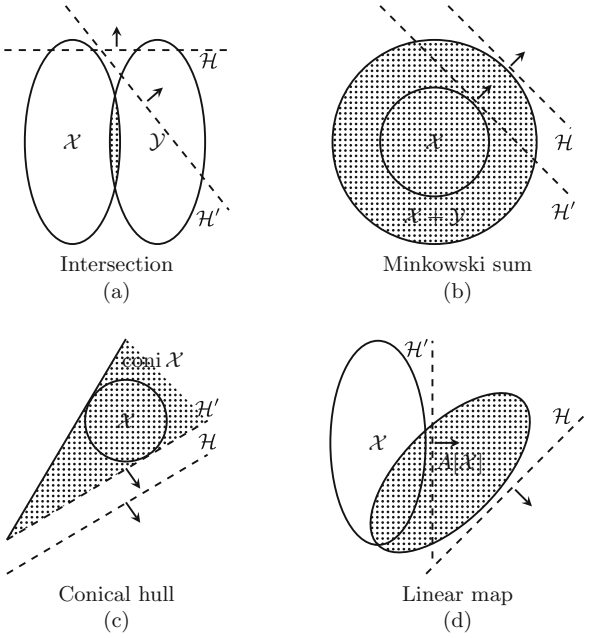

Fig. 7. Halfspace interpolants.

found. While this loop may in general not terminate, the same counterexample can never be encountered twice; this ensures progress.

A counterexample is a finite path $\ell_0, e_1, \ell_1, \ldots, e_k, \ell_k$ over the control graph of the hybrid automaton for which the respective sequence of abstract template polyhedra encounters a bad state, that is,

$$\mathcal{P}_0 = \Delta\text{-polyhedron of } \mathsf{post}_0(\mathcal{X}_0), \ldots, \mathcal{P}_k = \Delta\text{-polyhedron of } \mathsf{post}_k(\mathcal{P}_{k-1}),$$

and $\mathcal{P}_k \cap \mathcal{B} \neq \emptyset$, where $\mathcal{X}_0 = \mathsf{Init}(\ell_0)$ denotes the initial condition, $\mathcal{B} \subseteq \mathbb{R}^n$ denotes the bad region, $\mathsf{post}_0(\mathcal{X}) = \mathsf{post}_C(\ell_0, \mathcal{X})$, and $\mathsf{post}_i(\mathcal{X}) = \mathsf{post}_C(\ell_i, \mathsf{post}_D(e_i, \mathcal{X}))$ for $i = 1, \ldots, k$. The region that results from this path can be seen as a concatenation 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 interpolants* along these steps such that the last halfspace separates the result from the bad region. More precisely, we want to construct a sequence of halfspaces $\mathcal{H}_0, \ldots, \mathcal{H}_k \subseteq \mathbb{R}^n$ such that

$$\mathsf{post}_0(\mathcal{X}_0) \subseteq \mathcal{H}_0, \mathsf{post}_1(\mathcal{H}_0) \subseteq \mathcal{H}_1 \ldots, \mathsf{post}_k(\mathcal{H}_{k-1}) \subseteq \mathcal{H}_k, \mathcal{H}_k \cap \mathcal{B} = \emptyset. \quad (9)$$

To compute these halfspaces we break down these post operators into combinations of basic operations over sets. As indicated above, four operations are sufficient for the symbolic analysis of LHA: intersection between sets $\mathcal{X} \cap \mathcal{Y}$,

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

Edition	Tool	DISC2	DISC3	DISC4	DISC5
		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

Minkowski sum $\mathcal{X}+\mathcal{Y}$, conical hull $\operatorname{coni}\mathcal{X}$, and linear map $A[\mathcal{X}]$. For this reason, we can construct these halfspaces inductively over every operation. Specifically, for any halfspace $\mathcal{H}$ that contains the result of an operation, i.e., $\mathcal{X}\cap\mathcal{Y}\subseteq\mathcal{H}$, $\mathcal{X}+\mathcal{Y}\subseteq\mathcal{H}$, $\operatorname{coni}\mathcal{X}\subseteq\mathcal{H}$, or $A\mathcal{X}\subseteq\mathcal{H}$, we compute a second halfspace $\mathcal{H}'$ that includes one operand, i.e., $\mathcal{X}\subseteq\mathcal{H}'$, and abstracts it so as to preserve inclusion of the result into H, i.e., $\mathcal{H}'\cap\mathcal{Y}\subseteq\mathcal{H}$, $\mathcal{H}'+\mathcal{Y}\subseteq\mathcal{H}$, $\operatorname{coni}\mathcal{H}'\subseteq\mathcal{H}$, or $A[\mathcal{H}']\subseteq\mathcal{H}$ respectively. The intuition behind halfspace interpolants is 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; the details of the method for LHA are described in [11], and extended to hybrid automata with linear ODEs in [26]. The performance of this CEGAR approach is illustrated by some experiments in the next section.

5 Experiments

We provide some experiments to illustrate the above approaches for the analysis of Linear Hybrid Automata. They are based on results collected for the ARCH-COMP friendly verification competitions since 2017, in the category of hybrid system with piecewise constant dynamics of the [13], where different implementation techniques have been evaluated over the years. The following implementations are used:

- PHAVer [22] uses the fixed-point algorithm in Sect. 2.3 and calls the Parma Polyhedra Library [5] for the polyhedral computations in Sect. 3. PHAVer/SX is a subset of PHAVer, included as a plugin in the tool SpaceEx [28].
- PHAVerLite is a variant of PHAVer using the polyhedra library PPLite [8], which employs a novel representation and conversion algorithm [7] for NNC (Not Necessarily Closed) polyhedra. PHAVer-lite is an earlier version, implemented as a SpaceEx plugin.
- Lyse [24] is a tool for the reachability analysis of convex hybrid automata, whose constraints can be linear or non-linear but are required to be convex, 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 [34], 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. In 2017 and 2018, the tools computed the exact reachable states, so that instances with $n > 3$ were timing out. The time improvement in 2018 is due to replacing the PPL library with PPLite. In 2019, PHAVerLite-0.1 overapproximated set unions using the constraint hull operator, thereby also solving the instance with $n = 4$; finally, in 2020 the adoption of Cartesian factoring solved the instance with $n = 5$.

In Table 2, we present a more detailed evaluation of the techniques from in Sect. 3, focussing on the instance DISC3. The first four columns show the tool configuration: (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 columns 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.

When the *exact* reachable set needs to be computed, the filtering and boxing techniques are quite effective in improving efficiency. The constraint hull approximation provides another significant efficiency improvement; note that precision is degraded (78 reachable locations instead of 67 in the exact case), but the overapproximation is precise enough to prove safety. While here the Cartesian factoring technique only yields a marginal improvement, its effects become more relevant when considering bigger instances, as shown in Table 1, where for $n = 4$ the time drops from 77.51 to 2.59 s.

The above-mentioned progress is not specific to the considered benchmark: In the 2017 edition the corresponding tool verified 13 out of 20 tasks in 4:40 h, the 2020 edition solved 27 out of 28 tasks in less than 3 min.

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 [11]. For n cars, the number of discrete states is 2^n and the number of continuous variables is n. Each variable

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

Table 3. Computation times of the adaptive cruise controller [23,24].

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

x_i encodes the relative position of the i-th car and the relative velocities are subject to drift. 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 led 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 performance of the set propagation approach depends on how efficiently the required operations can be realized on the chosen set representation. 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 led to progressively more efficient libraries. In

particular, a novel representation for polyhedra with strict as well as nonstrict inequalities has led to gains on this fundamental level. Further gains have been achieved through heuristics that 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 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. Alur, R.: Formal verification of hybrid systems. In: Chakraborty, S., Jerraya, A., Baruah, S.K., Fischmeister, S. (eds.) EMSOFT, pp. 273–278. ACM (2011)
2. Alur, R., et al.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. **138**(1), 3–34 (1995)
3. Alur, R., Giacobbe, M., Henzinger, T.A., Larsen, K.G., Mikučionis, M.: Continuous-time models for system design and analysis. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 452–477. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_22
4. Bacci, E., Giacobbe, M., Parker, D.: Verifying reinforcement learning up to infinity. In: IJCAI, pp. 2154–2160. ijcai.org (2021)
5. Bagnara, R., Hill, P.M., Zaffanella, E.: Not necessarily closed convex polyhedra and the double description method. Formal Aspects Comput. **17**(2), 222–257 (2005)
6. 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)
7. Becchi, A., Zaffanella, E.: A direct encoding for NNC polyhedra. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, Part I. LNCS, vol. 10981, pp. 230–248. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_13
8. Becchi, A., Zaffanella, E.: An efficient abstract domain for not necessarily closed polyhedra. In: Podelski, A. (ed.) SAS 2018. LNCS, vol. 11002, pp. 146–165. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99725-4_11
9. Becchi, A., Zaffanella, E.: Revisiting polyhedral analysis for hybrid systems. In: Chang, B.-Y.E. (ed.) SAS 2019. LNCS, vol. 11822, pp. 183–202. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32304-2_10
10. Becchi, A., Zaffanella, E.: PPLite: zero-overhead encoding of NNC polyhedra. Inf. Comput. **275**, 104620 (2020)

11. Bogomolov, S., Frehse, G., Giacobbe, M., Henzinger, T.A.: Counterexample-guided refinement of template polyhedra. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 589–606. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_34
12. Boulmé, S., Maréchal, A., Monniaux, D., Périn, M., Yu, H.: The verified polyhedron library: an overview. In: 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2018, Timisoara, Romania, 20–23 September 2018, pp. 9–17. IEEE (2018)
13. Bu, L., et al.: 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, 12 July 2020. EPiC Series in Computing, vol. 74, pp. 1–15. EasyChair (2020)
14. Bu, L., Li, Y., Wang, L., Chen, X., Li, X.: BACH 2 : Bounded reachability checker for compositional linear hybrid systems. In: Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, 8–12 March 2010, pp. 1512–1517 (2010)
15. Chen, X., Ábrahám, E.: Choice of directions for the approximation of reachable sets for hybrid systems. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) EUROCAST 2011. LNCS, vol. 6927, pp. 535–542. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27549-4_69
16. Chernikova, N.V.: 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)
17. 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). https://doi.org/10.1007/10722167_15
18. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
19. Cousot, P., Cousot, R.: 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 1977, New York, NY, USA, pp. 238–252. Association for Computing Machinery (1977)
20. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978, pp. 84–96. ACM Press (1978)
21. Dang, T., Gawlitza, T.M.: Template-based unbounded time verification of affine hybrid automata. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 34–49. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25318-8_6
22. Frehse, G.: PHAVer: algorithmic verification of hybrid systems past HyTech. STTT **10**(3), 263–279 (2008)
23. Frehse, G., et al.: ARCH-COMP19 category report: hybrid systems with piecewise constant dynamics. In: Frehse, G., Althoff, M. (eds.) ARCH19 6th International Workshop on Applied Verification of Continuous and Hybrid Systems. EPiC Series in Computing, vol. 61, pp. 1–13. EasyChair (2019)

24. Frehse, G., et al.: ARCH-COMP18 category report: hybrid systems with piecewise constant dynamics. In: Frehse, G. (eds.) ARCH18 5th International Workshop on Applied Verification of Continuous and Hybrid Systems. EPiC Series in Computing, vol. 54, pp. 1–13. EasyChair (2018)
25. Frehse, G., Bogomolov, S., Greitschus, M., Strump, T., Podelski, A.: Eliminating spurious transitions in reachability with support functions. In: Girard, A., Sankaranarayanan, S. (eds.) Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC 2015, Seattle, WA, USA, 14–16 April 2015, pp. 149–158. ACM (2015)
26. Frehse, G., Giacobbe, M., Henzinger, T.A.: Space-time interpolants. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 468–486. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_25
27. Frehse, G., Jha, S.K., Krogh, B.H.: A counterexample-guided approach to parameter synthesis for linear hybrid automata. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 187–200. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78929-1_14
28. 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). https://doi.org/10.1007/978-3-642-22110-1_30
29. Halbwachs, N., Merchat, D., Gonnord, L.: Some ways to reduce the space dimension in polyhedra computations. Formal Methods Syst. Des. **29**(1), 79–95 (2006)
30. Halbwachs, N., Proy, Y.-E., Raymond, P.: Verification of linear hybrid systems by means of convex approximations. In: Le Charlier, B. (ed.) SAS 1994. LNCS, vol. 864, pp. 223–237. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58485-4_43
31. Halbwachs, N., Proy, Y., Roumanoff, P.: Verification of real-time systems using linear relation analysis. Formal Methods Syst. Des. **11**(2), 157–185 (1997)
32. Henzinger, T.: The theory of hybrid automata. In: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, pp. 278 (1996)
33. Henzinger, T., Ho, P.-H., Wong-Toi, H.: HyTech: a model checker for hybrid systems. Softw. Tools Technol. Transf. **1**, 110–122 (1997)
34. Henzinger, T.A., Ho, P.-H.: HyTech: the Cornell hybrid technology tool. In: Antsaklis, P., Kohn, W., Nerode, A., Sastry, S. (eds.) HS 1994. LNCS, vol. 999, pp. 265–293. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60472-3_14
35. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: Algorithmic analysis of nonlinear hybrid systems. IEEE Trans. Autom. Control **43**, 540–554 (1998)
36. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? J. Comput. Syst. Sci. **57**, 94–124 (1998)
37. Jeannet, B., Miné, A.: Apron: a library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
38. Jha, S.K., Krogh, B.H., Weimer, J.E., Clarke, E.M.: Reachability for linear hybrid automata using iterative relaxation abstraction. In: HSCC, pp. 287–300 (2007)
39. Le Guernic, C., Girard, A.: Reachability analysis of linear systems using support functions. Nonlinear Anal. Hybrid Syst. **4**(2), 250–262 (2010). IFAC World Congress 2008
40. Maler, O.: Algorithmic verification of continuous and hybrid systems. In: International Workshop on Verification of Infinite-State System (Infinity) (2013)
41. Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19**(1), 31–100 (2006)

42. Motzkin, T.S., Raiffa, H., Thompson, G.L., Thrall, R.M.: The double description method. In: Kuhn, H.W., Tucker, A.W. (eds.) Contributions to the Theory of Games – Volume II, no. 28. Annals of Mathematics Studies, pp. 51–73. Princeton University Press, Princeton, New Jersey (1953)
43. Podelski, A., Rybalchenko, A.: ARMC: the logical choice for software model checking with abstraction refinement. In: PADL, pp. 245–259 (2007)
44. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982). https://doi.org/10.1007/3-540-11494-7_22
45. Rockafellar, R.T. : Convex Analysis. Princeton University Press, Princeton (1970)
46. 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). https://doi.org/10.1007/978-3-662-49674-9_48
47. Sankaranarayanan, S., Colón, M.A., Sipma, H., Manna, Z.: Efficient strongly relational polyhedral analysis. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 111–125. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_8
48. Sankaranarayanan, S., Dang, T., Ivančić, F.: Symbolic model checking of hybrid systems using template polyhedra. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 188–202. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_14
49. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_2
50. Singh, G., Püschel, M., Vechev, M.T.: Fast polyhedra abstract domain. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 46–59. ACM (2017)
51. Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach. Springer, New York (2009). https://doi.org/10.1007/978-1-4419-0224-5
52. Williams, H.P.: Fourier's method of linear programming and its dual. Am. Math. Mon. **93**(9), 681–695 (1986)