HyPro

User manual

January 25, 2019

Contents

1 Introduction

This is the documentation of the `C++` library HyPro dedicated to provide implementations for state set representations for the reachability analysis of hybrid systems.

This documentation will be updated for major releases of the library, an online version with a comprehensive API documentation which reflects the current state can be found at `https://hypro.github.io/hypro/html/`.

The source code to the library can be found at `https://github.com/hypro/hypro`.

2 Getting started

In this section we provide all information to allow for a quick installation of the library and present the first steps towards an own implementation of a reachability analysis algorithm using the library. The library is created as a CMake (version ≥ 3.0) project and can be configured (see Section 2.2.1) using the graphical interface (install `cmake-curses-gui`).

2.1 Dependencies

In a minimal setup, HyPro depends on the following third-party libraries:

- CArL (`https://github.com/smtrat/carl`),

Note that CArL itself requires and ships Eigen3 (`http://eigen.tuxfamily.org`),gmp, and boost, which is also used in HyPro but once installed by CArL can be used transitively. HyPro itself downloads glpk, so a network connection is required.

Optional third-party libraries and tools are

- SMT-RAT (`https://github.com/smtrat/smtrat`), a library for SMT-solving, which allows to use exact (rational) linear optimization.
- Z3 (`https://github.com/Z3Prover/z3`), a SMT-solver, which also allows for exact linear optimization.
- SoPlex (`http://soplex.zib.de/`), an advanced implementation of the primal and dual simplex algorithm, which also allows for exact linear optimization.
- PPL (`http://bugseng.com/products/ppl`), an external implementation of convex polytopes, which is wrapped by HyPro.

Note that SMT-RAT, SoPlex and Z3 can only be used in a mutually exclusive way.

2.2 Installation

Here we describe the basic configuration of the library and the required steps for successfully building the library. The library should be built *out of source*, e.g. in a separate build folder. For the following we assume that the folder `hypro/build` has been created.

2.2.1 Configuration

To allow creating the required files for the CMake build system, please run

```
cmake ..
```

from the created build folder. The configuration of the build files for the library can be accessed and modified using the graphical user interface for CMake from the created build folder via the command

```
ccmake ..
```

Among other, CMake specific options, HyPro provides the following configuration options:

- `HYPRO_LOGGING` allows to enable/disable logging.
- `HYPRO_LOG_LEVEL` allows to set the log level. Note: This has only an effect, when `HYPRO_LOGGING` is enabled. Possible log levels are `TRACE`, `DEBUG`, `INFO`, `WARN`, `FATAL` (sorted in decreasing order according to the verbosity).
- `HYPRO_USE_OPENMP` enables support of OpenMP for Eigen3.
- `HYPRO_USE_PPL` enables the provided PPL wrapper class for the user. Note: This requires PPL to be installed.
- `HYPRO_USE_SMTRAT` enables SMT-RAT as an additional exact linear optimizing framework. Note: This requires SMT-RAT to be installed. Cannot be used in combination with SoPlex or Z3.
- `HYPRO_USE_SOPLEX` enables SoPlex as an additional exact linear optimizing framework. Note: This requires SoPlex to be installed. Cannot be used in combination with SMT-RAT or Z3.
- `HYPRO_USE_Z3` enables Z3 as an additional exact linear optimizing framework. Note: This requires Z3 to be installed. Cannot be used in combination with SMT-RAT or SoPlex.

After having configured the build system, the library can be build.

2.2.2 Building

To build the library after successful configuration, we require to build shipped resources (currently GTest) first via

```
make resources
```

from the created build folder. After having built the shipped resources we are ready to build the library using

```
make
```

The provided test can be run using

```
make test
```

Additional information regarding installation of the current release is provided at `https://hypro.github.io/hypro/html/`.

3 Tutorial

Here we give a short tutorial on how to start off using the library. We assume the reader is familiar with basic C++.

3.1 Matrices, Vectors

HyPro uses basic Eigen3 matrix and vector types as underlying datastructures. Typetraits have been extended such that matrices and vectors can be used with the types `mpq_class` and `cl_RA` as well. For convenience, we introduce the templated typedefs `matrix_t` and `vector_t` which can be used in the same way as Eigen3 types:

```cpp
using namespace hypro;
using Number = mpq_class;

matrix_t<Number> A = matrix_t<Number>(2,2);
A << 1,2,3,4;
```

The above example allows to create a 2×2-matrix filled with the values $1, 2, 3, 4$ (row-wise). The same matrix can be obtained using cell-wise assignment:

```cpp
using namespace hypro;
using Number = mpq_class;

matrix_t<Number> A = matrix_t<Number>(2,2);
A(0,0) = 1;
A(0,1) = 2;
A(1,0) = 3;
A(1,1) = 4;
```

The same holds for the type `vector_t`, which is implicitly a $n \times 1$-matrix. The introduced typedefs `hypro::matrix_t<Number>` and `hypro::vector_t<Number>` thereby alias the Eigen3 types `Eigen::Matrix<Number, Eigen::Dynamic, Eigen::Dynamic>` and `Eigen::Matrix<Number, Eigen::Dynamic, 1>` respectively. This allows for using all Eigen3 functions on the matrix and vector types.

3.2 Points, Halfspaces

Using the previously presented matrix and vector types, we can introduce the class of a point (`hypro::Point<Number>`) and a half-space (`hypro::Halfspace <Number>`), which build the foundation for our representations.

```cpp
// vector_t and Point can be converted into each other and used for construction.
Point<Number> p1 = Point<Number>({1,2});
vector_t<Number> coordinates1 = p1.rawCoordinates();

vector_t<Number> coordinates2 = vector_t<Number>(2);
coordinates2 << 1,2;
Point<Number> p2 = Point<Number>(coordinates2);
```

A half-space in HyPro is defined by a normal vector $\vec{n} \in \mathbb{R}^d$ and a scalar type c defining the plane offset and corresponds to a constraint $\vec{n}^T \cdot x \leq c$. We can construct half-spaces using the vector type or initializer lists directly:

```
// initializer lists for construction
Halfspace<Number> hsp1 = Halfspace<Number>({1,2},3);

// using vector_t for construction
vector_t<Number> normal = vector_t<Number>(2);
normal << 1,2;
Halfspace<Number> hsp2 = Halfspace<Number>(normal, 3);
```

Halfspaces can be used to construct geometric objects such as $\mathcal{H}$-polytopes or they can be used for intersection (see next section).

3.3 Geometric objects

Geometric objects build the core of our library. In its current state HyPro provides implementations for boxes, convex polytopes ($\mathcal{H}$- and $\mathcal{V}$-representation), orthogonal polyhedra, zonotopes and support functions. A unified interface (file: `GeometricObject.h`) is defined consisting of the following functions:

- `std::size_t dimension()` returns the space dimension of the object.
- `std::pair<bool, DerivedShape> satisfiesHalfspace( const Halfspace<Number>& rhs )` computes the intersection of the current object and a given halfspace. The returned pair consists of the Boolean stating that the result object is empty and the resulting object.
- `std::pair<bool, DerivedShape> satisfiesHalfspaces( const matrix_t<Number>& _mat, const vector_t<Number>& _vec )` works similar to `satisfiesHalfspace`, except that the input is a set/conjunction of half-spaces collected in a matrix which defines the set $P = \{x | \_mat \cdot x \leq \_vec\}$.
- `DerivedShape project(const std::vector<unsigned>& dimensions )` projects the current object on the passed dimensions. Each dimension can be identified by its index starting from 0.
- `DerivedShape linearTransformation( const matrix_t<Number>& A )` returns the resulting object of a linear transformation of the current object with a provided linear map `A`.
- `DerivedShape affineTransformation( const matrix_t<Number>& A, const vector_t<Number>& b )` returns the resulting object of an affine transformation of the current object an a linear map `A` and an additional offset vector `b`.
- `DerivedShape minkowskiSum( const DerivedShape& rhs )` computes Minkowskis' sum of the current and the passed object.
- `DerivedShape intersectHalfspace( const Halfspace<Number>& rhs )` computes the intersection of the current object and the passed halfspace.
- `DerivedShape intersectHalfspaces( const matrix_t<Number>& _mat, const vector_t<Number>& _vec )` computes the intersection of the current object and a set of halfspaces given as a matrix `_mat` and a vector `_vec`.

- `bool contains( const Point<Number>& point )` tests if the passed point is contained inside the current object.
- `DerivedShape unite( const DerivedShape& rhs )` computes the union of the current object and the passed object. The result is the smallest convex shape of the same type containing the exact result which ensures closure of this operation.

We will show the usage of the interface on boxes in a short example:

```cpp
#include "representations/GeometricObject.h"
#include "datastructures/Halfspace.h"
#include "util/Plotter.h"

int main()
{
  using namespace hypro;

  // use rational arithmetic.
  typedef mpq_class Number;

  // get plotter reference.
  Plotter<Number>& plotter = Plotter<Number>::getInstance();

  // create some transformation matrix.
  matrix_t<Number> A = matrix_t<Number>::Zero(3,3);
  A(0,0) = 1;
  A(1,1) = carl::convert<double,Number>(carl::cos(45));
  A(1,2) = carl::convert<double,Number>(-carl::sin(45));
  A(2,1) = carl::convert<double,Number>(carl::sin(45));
  A(2,2) = carl::convert<double,Number>(carl::cos(45));

  // create some translation vector.
  vector_t<Number> b = vector_t<Number>::Zero(3);

  // create a box out of two given limit points.
  Box<Number> testbox(std::make_pair(Point<Number>({-2,2,-4}), Point<Number>({2,4,-2})));

  // compute all vertices of the box and output them.
  std::vector<Point<Number>> tvertices = testbox.vertices();
  for(const auto& vertex : tvertices)
    std::cout << vertex << std::endl;

  // transform the initial box with the created matrix and vector (affine transformation).
  Box<Number> res = testbox.affineTransformation(A,b);

  std::vector<Point<Number>> vertices = res.vertices();
  for(const auto& vertex : vertices)
    std::cout << vertex << std::endl;

  // create a second box which is two dimensional.
  Box<Number> testbox2(std::make_pair(Point<Number>({-2,-2}), Point<Number>({2,2})));

  // create a halfspace (cutter) from a normal vector and an offset.
  matrix_t<Number> normal = matrix_t<Number>(1,2);
  vector_t<Number> offset = vector_t<Number>(1);
  normal << 1,1;
  offset << carl::rationalize<Number>(-0.5);
  vector_t<Number> hsNormal = vector_t<Number>(2);
  hsNormal << 1,1;
  Halfspace<Number> cutter = Halfspace<Number>(hsNormal, carl::rationalize<Number>(-0.5));

  // we can also plot halfspaces
  plotter.addObject(cutter);
  unsigned original = plotter.addObject(testbox2.vertices());

  // add the intersection of the second box and the created halfspace to the plotter instance.
58   // Note: satisfiesHalfspaces returns a pair <bool, Representation>.
59   unsigned cutted = plotter.addObject(testbox2.satisfiesHalfspaces(
         normal, offset).second.vertices());
60
61   // set colors and plot (gnuplot).
62   plotter.setObjectColor(original, hypro::plotting::colors[hypro::
         plotting::green]);
63   plotter.setObjectColor(cutted, hypro::plotting::colors[hypro::plotting
         ::red]);
64   plotter.plot2d();
65
66   return 0;
67 }
```

In this example, after initialization, a box (`hypro::Box<Number>`) is created (Line 27) from its limit points (see Section 4). We collect all vertices of the created box and use the outstream operator to print their string representation to `std::cout` (Lines 30-32). Note that the outstream operator is defined for most objects in HyPro. Afterwards an affine transformation is applied (Line 35). As boxes are not closed under this operation, the result is the smallest box containing the resulting set (the interval hull of the result). After creation of a second box and a halfspace, the second box is intersected with the halfspace and the result is tested for emptiness (Line 59). Note that the function `satisfiesHalfspace(..)` returns a pair of a Boolean value and a geometric object. The Boolean value holds the result of the emptiness check and the geometric object is the resulting object.

3.4 Utility functions

Among the previously presented geometric objects, the HyPro library also holds utility functions to simplify the development of reachability analysis algorithms.

3.4.1 Linear optimization and solving

Some representations such as support functions require linear optimization to be implemented. For HyPro we provide a wrapping class `Optimizer<Number>`, which provides a unified interface for linear optimization and solving. It makes use of a two-level approach utilizing GLPK and possible secondary solvers such as SMT-RAT, SoPlex or Z3. The central idea is to use GLPK as a fast presolver and improve the result by exact optimization backends. Note that the use of the second stage solving is optional and can be selected via the configuration of the library (see Section 2). The optimizer provides the functions

- `void setMatrix (const matrix_t<Number>& _matrix)` and `void setVector (const vector_t<Number>& _vector)` which are used to set up the system of linear equations $\_matrix \cdot \vec{x} \leq \_vector$.

- `EvaluationResult<Number> evaluate (const vector_t<Number>& _direction)` calls for a linear optimization (maximize) of a previously provided system of linear constraints using the cost function `_direction`. The returned object contains the optimal solution value, a point which is optimal and an error code stating if the problem is unbounded, satisfiable or unsatisfiable.

- `bool checkConsistency()` checks, if there exists a solution to the previously defined set of constraints, i.e. if the set defined by the conjunction of linear constraints is empty, and returns `true`, if the set is not empty.

- `bool checkPoint(const Point<Number>& _point)` checks, if the passed point is a solution to the set of constraints.

- `EvaluationResult<Number> getInternalPoint()` is an extension to `checkConsistency()`, as it also returns a valid solution, in case there is one.

- `std::vector<std::size_t> redundantConstraints()` checks the set of passed constraints for redundancy. A constraint is redundant, whenever the constraint does not constrain the solution space, i.e. its removal does not add solutions to the problem.

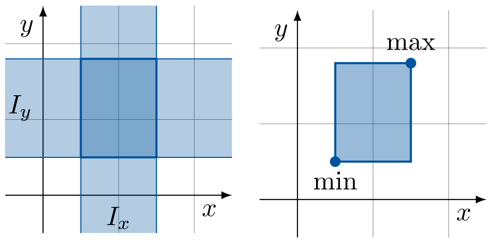
3.4.2 Output

For most types, the outstream operator (`operator<<(..)`) is overloaded allowing to obtain a simple string representation of an object. Furthermore, geometric shapes can be output to a GNUPLOT file for a visualization using the provided singleton plotter class (`hypro::Plotter<Number>`):

```
#include ”util/Plotter.h”
#include ”representations/GeometricObject.h”

int main (){
using namespace hypro;

  // get a reference to the singleton Plotter.
  Plotter<double>& plt = Plotter<double>::getInstance();

  // create some object, in this case a box.
  // we use carl::Interval to create our box here.
  std::vector<carl::Interval<double>> boxIntervals;
  boxIntervals.push_back(carl::Interval<double>(2,3));
  boxIntervals.push_back(carl::Interval<double>(1,2));
  Box<double> box = Box(boxIntervals);

  // now we can use the plotter to plot our box. The plotter returns
  // a unique id to reference the object.
  unsigned boxId = plt.addObject(box.vertices());

  // we can use the id to change the color:
  plt.setObjectColor(boxId, plotting::colors[plotting::red]);

  // when invoking the plot2d() method, the plotter creates a
  // gnuplot output file which plots the first two dimensions
  // (this can be changed in the settings of the plotter).
  plt.plot2d();

  return 0;
}
```

In its current state, the plotter creates GNUPLOT output files via the function `plot2d()` of the passed objects. The method `plotTex()` also creates a GNUPLOT output file, but with the LaTeX terminal enabled (standalone mode), such that the result will be a TeX file. In general the gnuplot files (`*.plt`) can be processed by passing them to GNUPLOT, i.e.

```
gnuplot out.plt
```



(a) A box represented as a vector of intervals.

(b) A box represented by two points.

Figure 1: Box representation.

4 State set representations

To implement a reachability analysis algorithm, we need datatypes for the representation of state sets, and certain (over-approximative) operations (union, intersection, linear transformation, Minkowski sum etc.) on them. Most flowpipe-construction-based reachability analysis approaches rely on geometric representations (e.g., boxes/hyperrectangles, oriented rectangular hulls, convex polyhedra, template polyhedra, orthogonal polyhedra, zonotopes, ellipsoids) or other symbolic representations (e.g., support functions or Taylor models) for state sets.

The variety of representations is rooted in the general problem of deciding between computational effort and precision. Generally, faster computations often come at the cost of precision loss and vice versa, more precise computations need higher computational effort. The representations might differ in their size, i.e., the required memory consumption, which has a further influence on the computational costs for operations on these representations. While some representations are able to perform certain operations very efficiently, other operations on the same representation, which are also needed for the analysis, can be computationally expensive.

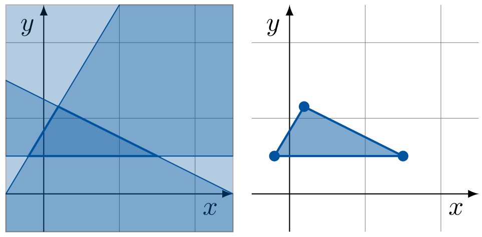
In the following we describe some of the most popular state set representations also implemented in HyPro. Let $\mathbb{I}$ be the set of all *intervals* $[a, b]$, $[a, \infty)$, $(-\infty, b]$, $(-\infty, +\infty) \subseteq \mathbb{R}$ with $a, b \in \mathbb{R}$; for simplicity, we call cross-products of intervals from $\mathbb{I}$ also intervals. An interval is *bounded* if both of its bounds are finite.

Boxes A box is defined by the cross product of intervals, one for each dimension of the state space (see Figure 1(a)).

Definition 1 (Box)**.** *A set $\mathcal{B} \subseteq \mathbb{R}^d$ is a* box *if there exist intervals $I_1, \ldots, I_d \in \mathbb{I}$ such that*

$$\mathcal{B} = I_0 \times \ldots \times I_d \, .$$

A box can be represented by the sequence $(I_1, \ldots, I_d)$ of the intervals defining it. Alternatively, we can represent a box by its minimal and its maximal point (see Figure 1(b)). Boxes are well-suited for fast computations in flowpipe construction, however, they often lead to large over-approximations. Boxes are widely used also in other fields such as in *interval constraint propagation (ICP)*, which itself is used for SMT-solving-based reachability analysis of hybrid sys-



(a) A $\mathcal{H}$-representation of a convex polytope. (b) A $\mathcal{V}$-representation of a convex polytope.

Figure 2: Convex polytope representation.

tems [FHR+07]. Implementations of boxes are also contained in most polytope libraries.

Convex polytopes A convex polyhedron can be defined by the intersection of finitely many halfspaces.

Definition 2 (Convex polyhedron)**.** *A set $\mathcal{P} \subseteq \mathbb{R}^d$ is a* convex polyhedron *if there are $n \in \mathbb{N}$ and $c_i \in \mathbb{R}^n$, $d_i \in \mathbb{R}$, $i = 1, \ldots, n$ such that*

$$\mathcal{P} = \bigcap_{i=1}^{n} h_i \textit{ where } h_i = \{x \in \mathbb{R}^d \mid c_i^T \cdot x \leq d_i\}\,.$$

In the following we restrict ourselves to closed convex polyhedra called *convex polytopes*, which have two widely used representations. An *$\mathcal{H}$-representation* (C, d) consists of a $n \times d$ matrix C with c_i being its ith row and an n-dimensional vector d with d_i being its ith components, and specifies the polytope $\mathcal{P} = \bigcap_{i=1}^{n} \{x \mid c_i^T \cdot x \leq d_i\}$ (see Figure 2(a)). Alternatively, a $\mathcal{V}$-representation consists of a finite set V of d-dimensional points and specifies a polytope as the convex hull $\mathcal{P} = \mathit{conv}(V)$ of those points (see Figure 2(b)).

Polytopes are a more complex representation as for instance boxes but allow for a more precise description of a set. The two presented representations are complementary in the complexity of the required operations for reachability analysis. Computing the convex hull of union requires little computational effort in the $\mathcal{V}$-representation but is hard in the $\mathcal{H}$-representation, whereas intersection can easily be performed with the $\mathcal{H}$-representation of a polytope but it is hard in the $\mathcal{V}$-representation. Unfortunately, conversion between the two representation requires either facet enumeration of a set of vertices or vertex enumeration of a set of hyperplanes, which are both computationally difficult. There are libraries already providing implementations of convex polytopes such as [BHZ08] [GJ00], but as they are intended to provide general purpose implementations, functionality required for hybrid systems reachability analysis is not fully optimized (e.g. PPL does currently not provide Minkowski sum implementations).

Zonotopes Zonotopes, sometimes also referred to as parallelotopes, are point-symmetric sets that can be defined as the Minkowski sum of a finite set of line segments shifted to a given centre point (see Figure 3).

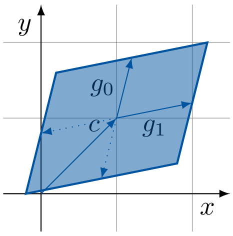


Figure 3: Zonotope representation of a set centered at c using two generators g_0 and g_1.

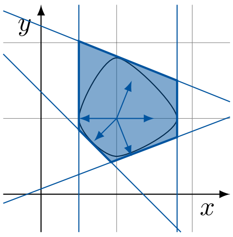


Figure 4: A set represented by a support function evaluated in 5 directions.

Definition 3 (Zonotope)**.** *A set* $\mathcal{Z} \subseteq \mathbb{R}$ *is a* zonotope *if there is a* center $c \in \mathbb{R}^n$ *and a finite set* $G = \{g_1, \ldots, g_n\}$ *of* generators $g_i \in \mathbb{R}^d$ *such that*

$$\mathcal{Z} = \left\{ x \;\middle|\; x = c + \sum_{i}^{n} \lambda_i \cdot g_i, \; -1 \leq \lambda_i \leq 1 \right\} .$$

Zonotopes can be represented by their defining vectors $(c, g_1, \ldots, g_n)$, i.e., by their center and the generators. Zonotopes are and due to their structure allow for a fast computation of the operations union and Minkowski sum. However, intersections with halfspaces or other zonotopes are hard to compute. Zonotopes are often used due to their reduced storage requirement in comparison to for example convex polytopes. Zonotope implementations are contained in the `C++` library polymake as well as in the Matlab tool collection Cora.

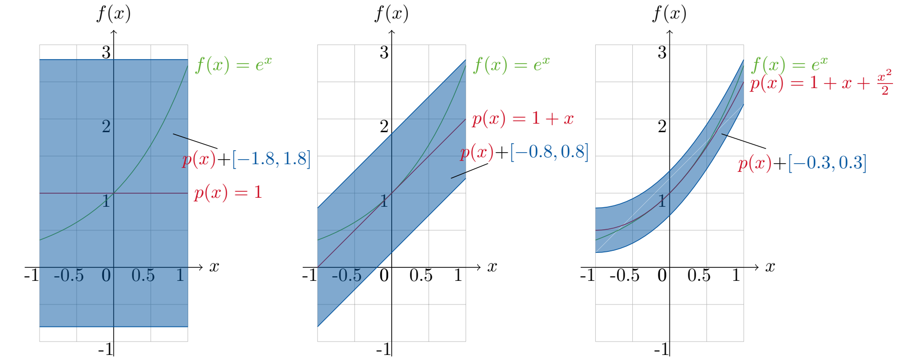
Support functions Support functions are, in contrast to the above presented representations, a symbolic representation, which allows queries for specific directions and will return a support value (see Figure 4).

Definition 4 (Support function)**.** *A* support function *is a function* $\sigma : \mathbb{R}^d \to \mathbb{R}$ *defining a set*

$$\mathcal{S} = \left\{ x \in \mathbb{R}^d \;\middle|\; r \cdot x \leq \sigma(r) \textit{ for all } r \in \mathbb{R}^d \right\} ,$$

where $\sigma(r) \in \mathbb{R}$ *is called the support value for the given direction* $r \in \mathbb{R}^d$*.*

The definition of support functions allows for an implementation, which reduces computation time during reachability analysis significantly. This is due to the fact that while other representations always maintain the explicit object, support functions only need to store the operation and its parameters.



0-order over-approximation 1-order over-approximation 2-order over-approximation

Figure 5: Taylor model approximations of different degrees.

Whenever the support value of a given direction is queried, the stored operations are applied reversed on the direction vector instead of applying them to the whole object. This implies that only directions are computed which are of interest instead of the full resulting object. The disadvantage of this representation is, that unless infinitely many directions are queried, the exact shape of the set is hidden. Support functions are used for example in the tool SpaceEx, which successfully makes use of algorithms optimized for support functions (LGG [LGG10, FGD+11], STC [FKL13, Fre15]).

Taylor models are a state set representation, which can be used for the reachability analysis of non-linear hybrid systems. The basic idea is to over-approximate the given dynamics by a polynomial of fixed degree k bloated by a suitable interval I, such that for a fixed initial set and a fixed time interval, all solutions of the ODE are contained in the area spanned by the Minkowski sum of the polynomial and the interval (see Figure 5).

Formally, assume a bounded domain $D \in \mathbb{I}^d$. A given polynomial p is a *k-order Taylor approximation* of a function $f : D \to \mathbb{R}$ iff

1. all partial derivatives of f up to order k exist and are continuous, denoted by $f \in C^k$, and
2. $f(c) = p(c)$ for the centre point c of D and for each $0 < m \leq k$, all of the order m partial derivatives of f and p coincide at c.

For any $f, g : D \to \mathbb{R}$, $f, g \in C^k$ and $k \geq 0$, we write $f \equiv_k g$ iff there is a polynomial p which is a k-order approximation of both f and g. Taylor models are based on the equivalence relation $\equiv_k$.

Definition 5 (Taylor Model)**.** *A* Taylor model *of order $k > 0$ over a bounded domain $D \in \mathbb{I}^d$ is a pair (p, I) of a polynomial p of degree at most k over d variables x and a* remainder interval *$I \in \mathbb{I}$. We say that (p, I) is a k-*order over-approximation *of a function $f : D \to \mathbb{R}$, written $f \in (p, I)$, iff (i) $p \equiv_k f$ and (ii) $\forall x \in D.\, f(x) \in p(x) + I := \{p(x) + y \,|\, y \in I\}$.*

4.1 Operations on state set representations

Flowpipe-construction-based reachability analysis algorithms need to apply certain *operations* on sets, whose complexity depends on the state set representation used. These operations include the (convex hull of) union, intersection, Minkowski sum, linear transformation as well as tests for emptiness and membership. Assume a domain D, and subsets $A, B, S \subseteq D$.

- $conv(\cdot \bigcup \cdot)$ (*union*): As convex sets are not closed under the operation union, for convex state set representations the convex hull of the union is computed; nevertheless, we often refer to this operation just as union:

  $$A \bigcup B = conv\{x \mid x \in A \ or \ x \in B\} \ .$$

  The convex hull of the union of two sets is required for the computation of the first segment of a flowpipe and in case aggregation of segments is used. Note that some state set representations are not closed under the operation convex hull either (e.g. boxes). In that case, when referring to a convex hull we mean the smallest set in that representation containing the convex hull.

- $\cdot \bigcap \cdot$ (*intersection*): The intersection of two sets is defined as

  $$A \bigcap B = \{x \mid x \in A \ and \ x \in B\} \ .$$

  Intersection of two sets is required whenever a flowpipe segment is checked against the invariant of the current location, for checking whether a guard condition holds, for checking the reachability of bad states, and for fixed-point detection. Note that all mentioned checks also imply a test for emptiness of the result of the intersection, which is why we provide a direct method performing both steps in one call for halfspaces, which are usually used to represent guards and invariants (`std::pair<bool,Representation<Number>> satisfiesHalfspace( const Halfspace <Number>& hsp)`).

- $\cdot \oplus \cdot$ (*Minkowski sum*): The Minkowski sum is the set-theoretic equivalent of addition:

  $$A \oplus B = \{x \mid x = a + b, a \in A, b \in B\} \ .$$

  In case of an autonomous system, Minkowski sum is only required for the computation of the first flowpipe segments (bloating). In case of a non-autonomous system, additional bloating for each segment is added via the application of the Minkowski sum.

- $A(\cdot)$ (*affine transformation*): The affine transformation of a given set S is defined as

  $$A(S) = \{x \mid x = A \cdot y + b, \ y \in S\} \ ,$$

  with A a $d \times d$-dimensional transformation matrix and $b \in \mathbb{R}^n$ specifies a translation. The application of an affine transformation does not increase the representation size. In flowpipe construction, the recurrence relation

	$\cdot \bigcup \cdot$	$\cdot \bigcap \cdot$	$\cdot \oplus \cdot$	$A(\cdot)$
Box			+	
$\mathcal{H}$-polytope	-	+	-	-
$\mathcal{V}$-polytope	+	-	+	+
Zonotope			+	+
Support function	+	-	+	+

Table 1: State set operations and their complexity

allows to compute the next flowpipe segment from the current one by applying an affine transformation, which makes the affine transformation in general a frequently used operation.

- *Test for emptiness* is a predicate checking whether a set is empty, i.e., whether $S = \emptyset$ holds.
- *Test for membership* is a further predicate which checks whether a given value is contained in the set, i.e., whether $x \in S$ for some input value $x \in D$.

These (and possibly further) operations must be defined and implemented for all state set representations used in the reachability analysis algorithm. We do not describe the single implementations here, but emphasize that the complexity of these operations might strongly differ for different representations.

To find the right balance between efficiency and precision, the choice of the state set representation as well as different kinds of optimisations play a crucial role in flowpipe-construction-based methods. Unfortunately, there is no optimal representation for which all necessary operations can be easily computed (see Table 1). While support functions seem to be optimal in most operations, they usually require more storage and require a lot of linear optimization calls in general. We also can observe that representations based on points, such as $\mathcal{V}$-polytopes or boxes, perform good on operations such as union, linear transformation or Minkowski sum. Representations that are based on constraints, such as $\mathcal{H}$-polytopes, naturally perform good on intersection computations.

Besides the complexity of the single operations, one has to keep in mind that not all operations are used in the same frequency. Based on the hybrid system model we want to analyse and on the approach we utilise for reachability analysis, the usage of operations varies. For example for linear autonomous systems, the operations Minkowski sum and union are performed only once per flowpipe, whereas a similar but non-autonomous system requires a more frequent application of the Minkowski sum. When enhancing standard algorithms for reduction techniques, the operation union is used more frequent, otherwise it is used again only once for each flowpipe computation (main loop iteration). The operation intersection is generally used on every computed flowpipe segment, but when the flowpipes are holding only a small number of segments and there are more locations, its significance for computation time might be reduced. There are also some results on reducing the frequency of applications for problematic operations, see e.g. [AK12] for avoiding intersection computations.

As the complexity of some operations is representation-dependent, to improve efficiency, most algorithms change the representation for certain computations using over-approximative transformations. Another efficiency-relevant issue is the reduction of the number of state sets for which successors need to be computed by clustering and aggregation: several state sets in a flowpipe or several successors for a jump can be over-approximated by a single set. Last but not least, the representation size is often reduced on the cost of an additional over-approximation error.

References

[AK12] Matthias Althoff and Bruce H. Krogh. Avoiding geometric intersection operations in reachability analysis of hybrid systems. In *Proc. of the 15th ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC'12)*, pages 45–54. ACM Press, 2012.

[BHZ08] 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.

[FGD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In *Proc. of CAV'11*, volume 6806 of *LNCS*, pages 379–395. Springer, 2011.

[FHR+07] Martin Fränzle, Christian Herde, Stefan Ratschan, Tobias Schubert, and Tino Teige. Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[FKL13] G. Frehse, R. Kateja, and C. Le Guernic. Flowpipe approximation and clustering in space-time. In *Proc. of HSCC'13*, pages 203–212. ACM, 2013.

[Fre15] Goran Frehse. Reachability of hybrid systems in space-time. In *Proc. of EMSOFT'15*, pages 41–50. IEEE Press, 2015.

[GJ00] Ewgenij Gawrilow and Michael Joswig. polymake: a framework for analyzing convex polytopes. In *Polytopes – combinatorics and computation (Oberwolfach, 1997)*, volume 29 of *DMV Sem.*, pages 43–73. Birkhäuser, Basel, 2000.

[LGG10] Colas Le Guernic and Antoine Girard. Reachability analysis of linear systems using support functions. *Nonlinear Analysis: Hybrid Systems*, 4(2):250–262, 2010.