Task 0: Introduction + Setup

Boxes as state set representations

In this task you will get used to the structure of the project and implement your first basic state set representation: Boxes. Boxes are one of the most basic state set representations that can be used for the reachability analysis of linear hybrid systems. During this task we will implement the basic datastructure as well as the required operations to perform reachability analysis. You will use GTest to verify your implementation.

Due date: Friday, 13^{th} November 2015.

Present your implementation in a short presentation ($\sim$ 15 minutes / $\sim 10-15$ slides) along with your ideas towards the next task.

Setup

In its current state, HyPro depends on the following libraries:

- Parma Polyhedra Library (PPL)
- MPFR
- CArL
- GLPK (provided)
- Eigen3
- GTest (provided)

HyPro is a CMake project using out-of-source building and is written in C++. The basic setup is straight forward, however PPL requires some adjustments to be usable with the C++11 standard.

Installing external libraries

Libraries such as MPFR, GLPK, Eigen3 and GTest can be installed in the usual way (e.g. download and compile the source, make sure to use the -dev packages, if you use a packet manager). Note: You do not need to install the libraries, which are marked as *provided* – those come with HyPro, if required.

Installing CArL

1. Download the source from https://github.com/smtrat/carl.
2. If not already installed, install its dependencies: LIBGMP-DEV, BOOST, EIGEN3, MPFR (use -dev packages)
3. Configure via:

```
$ cd carl
carl$ mkdir build && cd build
carl/build$ cmake ..
carl/build$ ccmake ..
```

   In the GUI: Enable `USE_MPFR_FLOAT` (navigate to the entry, hit `Return`, configure (`c`), generate (`g`)).

4. Build with

```
carl/build$ make resources
carl/build$ make
```

Installing PPL

1. Download the PPL sources (http://bugseng.com/products/ppl/download), then, to make PPL C++ 11 compatible do the following:
   - In the file `src/checked_mpz_inlines.hh:55`: Change `typeof` to `decltype`
   - In the files `demos/ppl_pips/ppl_pips.cc`(lines 541 and 561) and `demos/ppl_lcdd/ppl_lcdd.cc`(lines 227 and 253): Change

```
*input_stream_p != std::cin
```

     to

```
typeid(*input_stream_p) != typeid(std::cin)
```

2. Use the provided `configure`script to configure PPL:

```
$ ./configure --with-cxxflags="-std=c++11"
```

3. Install via

```
$ sudo make install
```

Compiling HyPro

1. Get the sources from the GIT repository (see introductory slides)
2. Configure and build resources (provided libraries, if necessary):

```
$ cd hypro
hypro$ mkdir build && cd build
hypro/build$ cmake ..
hypro/build$ make resources
```

3. Compile:

```
hypro/build$ make
```

4. (optional) run tests

```
hypro/build$ make test
```

5. (optional) run specific example (see notes below)

```
hypro/build$ make example_XYZ
hypro/build$ ./bin/example_XYZ
```

If you encounter any errors during setup or elsewhere please do not hesitate to contact me (Stefan Schupp).

Tasks

As you might have noticed the tests for the representations currently fail. During this exercise sheet we will try to fix this issue by implementing the missing datastructure and its required functions:

1. Create the basic datastructure to be able to represent a d-dimensional box.

   Definition 1 (Box [LG09]) *A set $\mathcal{B}$ is a box iff it can be expressed as a product of intervals.*

   $$\mathcal{B} = [l_1; u_1] \times [l_2; u_2] \times \ldots \times [l_d; u_d]$$

   Note that you do not have to stick to the definition for your implementation but can choose other approaches. Currently we provide basic types such as points (`hypro::Point<Number>`), hyperplanes (`hypro::Hyperplane<Number>`) and intervals (`carl::Interval<Number>`). Make sure to provide convenient constructors and all required getters, setters and print methods, as outlined in the prepared file (see below). Despite from the empty constructor, you should be able to construct a box at least from

   - a vector of points (`std::vector<hypro::Point<Number> >`),
   - a vector of intervals (`std::vector<carl::Interval<Number> >`),
   - another box (`hypro::Box<Number>`),
   - by a static function `Empty(std::size_t dimension)`, which constructs an empty box of the required dimension.

   Extend the provided test file `PTermBoxTest.cpp` to verify your implementation.

2. Extend your implementation by adding implementations for all operations required for reachability analysis of hybrid systems, which includes:

   - $conv(\cdot \bigcup \cdot)$ - union. As boxes are not closed under union, it is required to compute the convex hull ($conv(..)$) of the union of two boxes.
   - $\cdot \bigcap \cdot$ - intersection. To verify a flowpipe segment against the invariant of the current location, it is required to compute the intersection of the invariant and the segment and test it for emptiness.

- *empty*$(\cdot)$ - emptiness. As stated before, this is needed to verify the segment lies inside the invariant of the current location.
- $A(\cdot)$ - linear transformation. To create a flowpipe we can encode the linear dynamics inside a location of a linear hybrid automaton into a linear transformation. Consecutive application of this linear transformation allows us to compute a flowpipe.
- *Minkowski*$(\cdot, \cdot)$ - Minkowski sum. The Minkowski sum is the set-equivalent to a sum and is needed for the over-approximation of the initial set to cover the dynamics inside the current location.

Make sure to verify your implementation against the tests in the provided test file and extend the tests where needed.

Hints and additional information

- We have prepared an empty setup in the folder `src/lib/representations/PTermBox` where you should put your implementation.
- We have provided test cases - use those to verify your implementation and extend the test cases. Your can find the test file at `src/test/representations/PTermBoxTest.cpp`
- Please implement at least the interface provided to make your implementation modular.
- If you want to use example files to test certain parts of your implementation or to discover parts of the library, you can set up a file in the `examples` folder and add its name to the `CMakeLists.txt` file in this folder. Afterwards you should be able to compile and run your example (see above).
- For matrices and vectors we make use of the library EIGEN3 and internally wrap its matrix and vector types by `hypro::matrix_t<Number>` and `hypro::vector_t<Number>` respectively.
- If you want to use plotting, make sure your datastructure allows to create a vector containing all vertices of a set (`std::vector<Point<Number»`). Note: The provided plotter implements a Graham-Scan such that the vector of points does not need to be sorted, however make sure that your points are 2-dimensional. The plotter can be found in `src/lib/util/Plotter.h`. You need to have the program `gnuplot` installed to be able to process the generated `.plt` file.

References

[LG09] Colas Le Guernic. *Reachability Analysis of Hybrid Systems with Linear Continuous Dynamics*. Theses, Université Joseph-Fourier - Grenoble I, October 2009.