SCOTS – USER MANUAL

MATTHIAS RUNGGER

CONTENTS

Part 1. INTRODUCTION

1. About SCOTS

SCOTS is an open source software tool (available at http://www.hcs.ei.tum.de) for the synthesis of symbolic controllers for possibly perturbed, nonlinear, control systems according to [1]. It is mainly implemented in C++ and it comes with a small MATLAB interface to access the synthesized controller from within MATLAB.

The tool is intended to be used and extended by researches in the area of formal methods for cyber-physical systems. SCOTS provides a baseline implementation of one of the most basic approaches to symbolic synthesis.

SCOTS provides an implementation of the various algorithms using two different data structures. One implementation is based on *binary decision diagrams* (BDD) [2] and one implementation is based on a sparse matrix data structure. For the sparse matrix implementation see SCOTSv0.2.

2. Symbolic Controller Synthesis Basics

2.1. **Control Problems.** SCOTS supports the computation of controllers for nonlinear control systems of the form

$$\dot{\xi}(t) \in f(\xi(t), u) + [\![-w, w]\!] \tag{1}$$

where f is given by $f : \mathbb{R}^n \times U \to \mathbb{R}^n$ and $U \subseteq \mathbb{R}^m$. The vector $w = [w_1, \dots, w_n] \in \mathbb{R}^n_+$ is a perturbation bound and $[\![-w, w]\!]$ denotes the hyper-interval $[-w_1, w_1] \times \dots \times [-w_n, w_n]$. Given a time horizon $\tau > 0$, we define a *solution of* (1) *on* $[0, \tau]$ *under (constant) input* $u \in U$ as an absolutely continuous function $\xi : [0, \tau] \to \mathbb{R}^n$ that satisfies (1) for almost every (a.e.) $t \in [0, \tau]$.

The desired behavior of the closed loop is defined with respect to the τ-sampled behavior of the continuous-time systems (1). To this end, the sampled behavior of (1) is casted as *simple system* [1]

$$S_1 := (X_1, U_1, F_1) \tag{2}$$

with the *state alphabet* $X_1 := \mathbb{R}^n$, *input alphabet* $U_1 := U$ and the *transition function* $F_1 : X_1 \times U_1 \rightrightarrows X_1$ defined by

$$F_1(x, u) := \{x' \mid \exists_{\xi \text{ is a solution of (1) on } [0,\tau] \text{ under } u} : \xi(0) = x \wedge \xi(\tau) = x'\}.$$

A *specification* Σ_1 for a simple system $S_1 = (X_1, U_1, F_1)$ is simply a set

$$\Sigma_1 \subseteq \bigcup_{T \in \mathbb{Z}_{\geq 0} \cup \{\infty\}} (U_1 \times X_1)^{[0;T[} =: (U_1 \times X_1)^{\infty} \tag{3}$$

of possibly finite and infinite input-state sequences. A simple system S_1 together with a specification Σ_1 constitute an *control problem* (S_1, Σ_1).

SCOTS natively supports invariance (often referred to as safety) and reachability specifications. Consider two sets $I_1 \subseteq X_1$ and $Z_1 \subseteq X_1 \times U_1$. A *reachability* specification associated with I_1, Z_1 is defined by

$$\Sigma_1 := \{(u, x) \in (U_1 \times X_1)^{\infty} \mid x(0) \in I_1 \implies \exists_{t \in [0;T[} : (x(t), u(t)) \in Z_1\}.$$

An *invariance* specification associated with I_1, Z_1 follows by

$$\Sigma_1 := \{(u, x) \in (U_1 \times X_1)^{[0;\infty[} \mid x(0) \in I_1 \implies \forall_{t \in [0;\infty[} : (x(t), u(t)) \in Z_1\}.$$

In this context, the sets I_1 and Z_1 are referred to as *atomic propositions*. SCOTS allows to define arbitrary sets as atomic propositions. In the BDD implementation, it provides customized commands to define

- polytopes $\{x \in \mathbb{R}^n \mid Hx \leq h\}$ parameterized by $H \in \mathbb{R}^{q \times n}$, $h \in \mathbb{R}^q$, and
- ellipsoids $\{x \in \mathbb{R}^n \mid |L(x - y)|_2 \leq 1\}$ parameterized by $L \in \mathbb{R}^{n \times n}$ and $y \in \mathbb{R}^n$.

It is also possible to synthesize controllers with respect more general specifications such as

- *reach-and-stay*, see the example in `./examples/bdd/dcdc2/` and Sec. 7.3.

2.2. **Auxiliary Control Problems.** Given a simple system $S_1 = (X_1, U_1, F_1)$ representing the τ-sampled behavior of (1) and a specification Σ_1 for S_1, the control problem (S_1, Σ_1) is not solved directly, but an auxiliary, finite control problem (S_2, Σ_2) is used in the synthesis process. Here, $S_2 = (X_2, U_2, F_2)$ is a *symbolic model* or (discrete) *abstraction* of S_1 and Σ_2 is an abstract specification.

The state alphabet of X_2 is a cover of X_1 and the input alphabet U_2 is a subset of U_1. The set X_2 contains a subset $\bar{X}_2$, representing the "real" quantizer symbols, while the remaining symbols $X_2 \setminus \bar{X}_2$ are interpreted as "overflow" symbols. The set of real quantizer symbols $\bar{X}_2$ are given by congruent hyper-rectangles aligned on a uniform grid

$$\eta\mathbb{Z}^n = \{c \in \mathbb{R}^n \mid \exists_{k\in\mathbb{Z}^n} \forall_{i\in[1;n]}\, c_i = k_i\eta_i\} \tag{4}$$

with *grid parameter* $\eta \in (\mathbb{R}_+ \setminus \{0\})^n$, i.e.,

$$x_2 \in \bar{X}_2 \implies \exists_{c\in\eta\mathbb{Z}^n}\, x_2 = c + [\![-\eta/2, \eta/2]\!]. \tag{5}$$

SCOTS computes symbolic models that are related via feedback refinement relations with the plant. A *feedback refinement relation* from S_1 to S_2 is a strict relation $Q \subseteq X_1 \times X_2$ that satisfies for all $(x_1, x_2) \in Q$ and for all $u \in U_2$ with $F_2(x_2, u) \neq \emptyset$ the conditions

(1) $F_1(x_1, u) \neq \emptyset$ and
(2) $Q(F_1(x_1, u)) \subseteq F_2(x_2, u)$.

In SCOTS, the feedback refinement relation Q is given by the set-membership relation

$$Q := \{(x_1, x_2) \mid x_1 \in x_2\}. \tag{6}$$

Given an invariance (reachability) specification Σ_1 for S_1 associated with (I_1, Z_1) an *abstract specification* is given by the invariance (reachability) specification for S_2 associated with

$$I_2 = \{x_2 \in X_2 \mid x_2 \cap I_1 \neq \emptyset\} \quad \text{and} \quad Z_2 = \{x_2 \in X_2 \mid x_2 \subseteq Z_1 \neq \emptyset\}. \tag{7}$$

For the solution of the auxiliary control problems (S_2, Σ_2) SCOTS provides minimal and maximal fixed point algorithms [3].

2.3. **Growth Bound.** The construction of a symbolic model S_2 of S_1 is based on the over-approximation of attainable sets. To this end, we use the notion of a growth bound introduced in [1]. A *growth bound* of (1) is a function $\beta : \mathbb{R}^n_+ \times U' \to \mathbb{R}^n_+$, which is defined with respect to a sampling time $\tau > 0$, a set $K \subseteq \mathbb{R}^n$ and a set $U' \subseteq U$. Basically, it provides an upper bound on the deviation of solutions ξ of (1) from *nominal solutions*[1] φ of (1), i.e., for every solution ξ of (1) on $[0, \tau]$ with input $u \in U'$ and $\xi(0), p \in K$, we have

$$|\xi(\tau) - \varphi(\tau, p, u)| \leq \beta(|\xi(0) - p|, u). \tag{8}$$

Here, $|x|$ for $x \in \mathbb{R}^n$, denotes the component-wise absolute value. A growth bound can be obtained essentially by bounding the Jacobian of f. Let $L : U' \to \mathbb{R}^{n\times n}$ satisfy

$$L_{i,j}(u) \geq \begin{cases} D_j f_i(x, u) & \text{if } i = j, \\ |D_j f_i(x, u)| & \text{otherwise} \end{cases} \tag{9}$$

for all $x \in K' \subseteq \mathbb{R}^n$ and $u \in U' \subseteq U$. Then

$$\beta(r, u) = e^{L(u)\tau} r + \int_0^\tau e^{L(u)s} w \, \mathrm{d}s, \tag{10}$$

is a growth bound on $[0, \tau]$, K, U' associated with (1). The domain K' on which (9) needs to hold, is assumed to be convex and contain any solution ξ on $[0, \tau]$ of (1) with $u \in U'$ and $\xi(0) \in K$, see [1, Thm. VIII.5].

In order to use SCOTS, the user needs to provide a growth bound, which for nonlinear control systems can be provided in terms of the parameterized matrix $L(u)$ whose entries satisfy (9).

[1] A nominal solution $\varphi(\cdot, p, u)$ of (1) is defined as solution of the initial value problem $\dot{x} = f(x, u)$, $x(0) = p$.

2.4. **Closed Loop.** The *solution* of a control problem (S, Σ) is a *system* $C = (X_c, X_{c,0}, U_c, V_c, Y_c, F_c, H_c)$ which is *feedback composable* with S_1, see [1, Def. III.3], so that

$$\mathscr{B}(C \times S) \subseteq \Sigma.$$

Here $\mathscr{B}(C \times S)$ denotes the *behavior* of the closed loop $C \times S_1$, see [1, Def. V.1]. The main statement enabling the symbolic synthesis approach reads as follows [1, Thm. VI.3].

Consider two control problems (S_i, Σ_i), $i \in \{1, 2\}$. *Suppose that* Q *is a feedback refinement relation from* S_1 *to* S_2 *and* Σ_2 *is an abstract specification of* Σ_1. *If* C *solves the control problem* (S_2, Σ_2), *then* $C \circ Q$ *solves the control problem* (S_1, Σ_1).

The controller $C \circ Q$ for S_1 is given by the serial composition of the quantizer $Q : X_1 \rightrightarrows X_2$ with the controller C. The closed loop resulting from a simple system Σ_1 which represents the τ-sampled behavior of (1) and a controller $C \circ Q$ is illustrated in Fig. 1. At each $k \in \mathbb{Z}_{\geq 0}$ sampling time $\tau > 0$, the plant state $x_1 = \xi(k\tau)$ is measured and fed to the quantizer Q, which is used to determine a cell $x_2 \in X_2$ that contains $x_1 \in x_2$. Then x_2 is feed to the controller C to pick the input $u \in U_2 \subseteq U_1$ which is applied to (1).

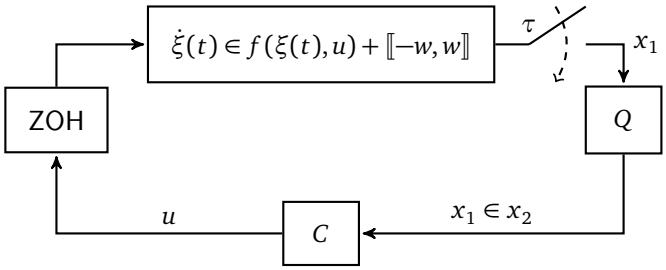

FIGURE 1. Sample-and-hold implementation of a controller synthesized with SCOTS.

Additionally to the perturbations on the right-hand-side of (1), it is possible to account for measurement errors modeled by a set-valued map $P : \mathbb{R}^n \rightrightarrows \mathbb{R}^n$ given by

$$P(x) := x + [\![-z, z]\!] \quad \text{with} \quad z \in \mathbb{R}^n_+. \tag{11}$$

Please see [1, Sec. VI.B] and [3] for some background theory. The closed loop with measurement errors is illustrated in Fig. 2.

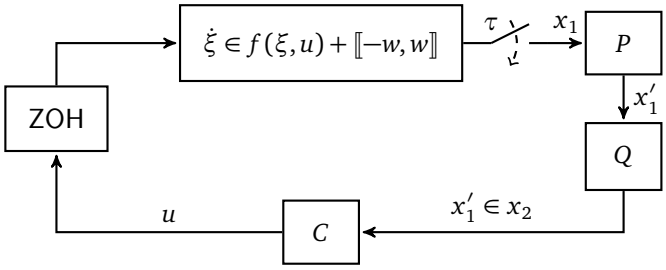

FIGURE 2. Closed loop with measurement errors modeled by the set-valued map $x_1' \in P(x_1)$.

2.5. **Synthesis via Fixed Point Computations.** For the synthesis of controllers C to enforce reachability, respectively, invariance specifications, SCOTS provides two fixed point algorithms.

Consider $S_2 = (X_2, U_2, F_2)$ with X_2 finite and $I_2 \in X_2$, $Z_2 \subseteq X_2 \times U_2$. For $Y \subseteq X_2 \times U_2$, we define the map

$$\operatorname{pre}(Y) := \{(x_2, u) \in X_2 \times U_2 \mid F_2(x_2, u) \neq \varnothing \wedge F_2(x_2, u) \subseteq \pi_{X_2}(Y)\}. \tag{12}$$

where $\pi_{X_2}(Y)$ denotes the projection of Y onto X_2.

We compute a controller to enforce a reachability specification Σ_2 associated with I_2, Z_2, by computing the minimal fixed point of the map $Y \mapsto \mathrm{pre}(Y) \cup Z_2$, which we denote by using the usual μ calculus notation [4] as

$$\mu Y.\mathrm{pre}(Y) \cup Z_2.$$

In order to extract a controller, we introduce the function $j : X_2 \to \mathbb{N} \cup \{\infty\}$ by

$$j(x) = \inf\{i \in \mathbb{N} \mid x \in \pi_{X_2}(Y_i)\}$$

where the sets Y_i are recursively given by

$$Y_0 = \varnothing$$
$$Y_{i+1} = \mathrm{pre}(Y_i) \cup Z_2.$$

Of course we have $Y_i = Y_{i+1}$ implies $Y_i = \mu Y.\mathrm{pre}(Y) \cup Z_2$. Let use define the map

$$H_c'(x_2) = \left\{u \in U \mid (x_2, u) \in Z_2 \vee (F_2(x_2, u) \neq \varnothing \wedge F_2(x_2, u) \subseteq \pi_{X_2}(Y_{j(x_2)-1}))\right\} \tag{13}$$

which is non-empty for all $x_2 \in \mu Y.\mathrm{pre}(Y) \cup Z_2$. We derive a controller as a system according to [1, Def. III.1] (ver. 2) by $C = (\{q\}, \{q\}, X_2, X_2, U_2, F_c, H_c)$ with

$$H_c(q, x_2) = \begin{cases} H_c'(x_2) \times \{x_2\} & \text{if } x_2 \in \mu Y.\mathrm{pre}(Y) \cup Z_2 \\ U_2 \times \{x_2\} & \text{otherwise} \end{cases}$$
$$F_c(q, x_2) = \begin{cases} \{q\} & \text{if } x_2 \in \mu Y.\mathrm{pre}(Y) \cup Z_2 \\ \varnothing & \text{otherwise} \end{cases}$$

Similarly, if Σ_2 is an invariance specification associated with I_2, Z_2, we compute the maximal fixed point of $Y \mapsto \mathrm{pre}(Y) \cap Z_2$, which is denoted by

$$\nu Y.\mathrm{pre}(Y) \cap Z_2.$$

Given $\nu Y.\mathrm{pre}(Y) \cap Z_2$ we define the map

$$H_c'(x_2) = \left\{u \in U \mid F_2(x_2, u) \neq \varnothing \wedge F_2(x_2, u) \subseteq \pi_{X_2}(\nu Y.\mathrm{pre}(Y) \cap Z_2)\right\}. \tag{14}$$

and the controller according to [1, Def. III.1] (ver. 2) follows again by

$$H_c(q, x_2) = \begin{cases} H_c'(x_2) \times \{x_2\} & \text{if } x_2 \in \nu Y.\mathrm{pre}(Y) \cap Z_2 \\ U_2 \times \{x_2\} & \text{otherwise} \end{cases}$$
$$F_c(q, x_2) = \begin{cases} \{q\} & \text{if } x_2 \in \nu Y.\mathrm{pre}(Y) \cap Z_2 \\ \varnothing & \text{otherwise} \end{cases}$$

In either case, it is well known [5] that C solves the control problem (S_2, Σ_2) with Σ_2 being a reachability (invariance) specification iff $I_2 \subseteq \pi_{X_2}(\mu Y.\mathrm{pre}(Y) \cup Z_2)$ ($I_2 \subseteq \pi_{X_2}(\nu Y.\mathrm{pre}(Y) \cap Z_2)$). Also for both types of specifications the controller is *memoryless* or *static*, i.e., the output is independent of the state.

3. Source Code Organization

```
./bdd                /* the source code of the BDD based implementation */
./doc                /* the C++ class documentation created with NaturalDocs */
./examples           /* directory containing various examples */
./examples/hscc16    /* examples presented at HSCC16 using the BDD impl. */
./license.txt        /* the license file */
./manual             /* directory containing this manual */
./mfiles             /* the MATLAB files */
./mfiles/mexfiles    /* mex files to access the controller from MATLAB */
./readme.txt         /* a quick description pointing to this manual */
./sparse             /* the source code of the sparse matrix implementation */
./utils              /* some helper code */
```

4. Installation

In general, SCOTS is implemented in "header-only" style and you only need a working C++ developer environment. However, the BDD based implementation uses the CUDD library by Fabio Somenzi, which can be downloaded at http://vlsi.colorado.edu/~fabio/. Moreover, for the illustration of atomic propositions as well as for the closed loop simulation a MATLAB interface and, hence, a working installation of MATLAB is needed.

The requirements are summarized as follows:

(1) A working C/C++ development environment
- Mac OS X: You should install Xcode.app including the command line tools
- Linux: Most linux OS include the necessary tools already
- Windows: We haven't tested the code with Windows

(2) A working installation of the CUDD library with
- the C++ object-oriented wrapper
- the dddmp library and
- the shared library

option enabled. The package follows the usual `configure`, `make` and `make install` installation routine. We use `cudd-3.0.0`, with the configuration

```
$ ./configure --enable-shared --enable-obj --enable-dddmp --prefix=/opt/local/
```

You should test the BDD installation by compiling a dummy programm, e.g. `test.cc`

```
#include<iostream>
#include "cuddObj.hh"
#include "dddmp.h"
int main () {
  Cudd mgr(0,0);
  BDD x = mgr.bddVar();
}
```

should be compiled by

```
$ g++ test.cc -I/opt/local/include -L/opt/local/lib -lcudd
```

On some linux machines we experienced that the header files `util.h` and `config.h` were missing in `/opt/local` and we manually copied them to `/opt/local/include`.

(3) A recent version of MATLAB. We conducted the experiments with version `R2015b (8.6.0.232648) 64-bit (maci64)`. To compile the mex files:
(a) open MATLAB and setup the mex compiler by

```
>> mex -setup C++
```

(b) edit the `Makefile` and adjust the variables
- `MATLABROOT` and `CUDDPATH`

(c) in a terminal run `$ make`

Part 2. BDD BASED IMPLEMENTATION

5. QUICKSTART

For a quickstart

(1) go to one of the examples in

```
./examples/hscc16/dcdc          /* invariance specification */
./examples/hscc16/vehicle2      /* reachability specification */
./examples/bdd/dcdc2            /* reach-and-stay specification */
./examples/bdd/boostsolver      /* using an ODE solver form the BOOST lib */
```

(2) read the readme file (if the directory contains one)
(3) edit the `Makefile`
  (a) adjust the compiler
  (b) adjust the directories of the CUDD library
(4) compile and run the executable, for example in `./examples/hscc16/dcdc` run

```
$ make
$ ./dcdc
```

(5) open MATLAB (here we assume that you have already compiled the mex files) and add `./mfiles` to the MATLAB path

```
>> addpath(genpath('../../../mfiles'))
```

(6) run the m file, for example in `./examples/hscc16/dcdc` run

```
>> dcdc
```

(7) modify the example to your needs

6. WORK FLOW OVERVIEW

A short description of the general work flow is given by

(1) Use the class `SymbolicSet` to specify
  - the real quantizer symbols $\bar{X}_2$
  - the input alphabet U_2

(2) Use the class `SymbolicModelGrowthBound` to compute the transition function F_2. You need to provide
  - the solution of the right-hand-side (1) at time τ
  - and the growth bound (8) (directly or in terms of $L(u)$ see (9))

(3) Use `SymbolicSet` to specify the atomic propositions in the specification
(4) Use `FixedPoint` to solve the auxiliary control problem (S_2, Σ_2)
(5) Use `SymbolicSet` to write the result to a file
(6) Optional: Simulate the closed loop in MATLAB by accessing the controller saved in the file

An overview of the usage is illustrated in Fig. 3.

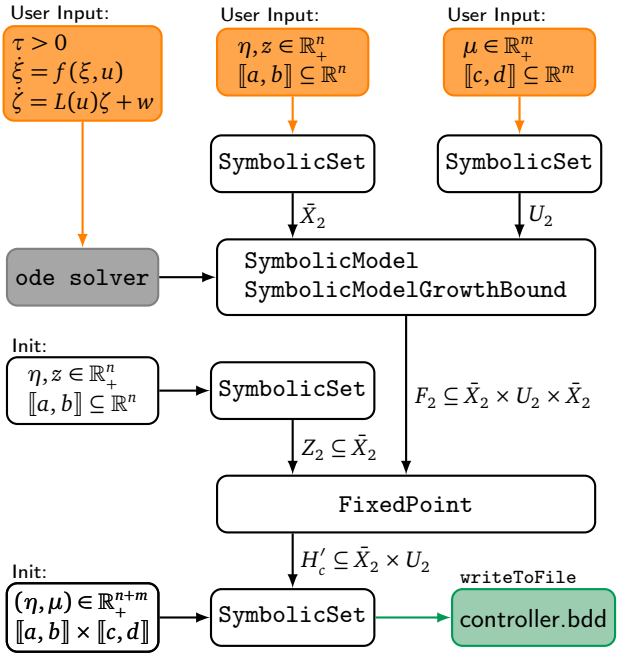

FIGURE 3. The work flow in SCOTS.

7. USAGE AND IMPLEMENTATION DETAILS

7.1. **Symbolic Set.** Each instance of the C++ class `SymbolicSet`, whose source code is located in `./bdd/SymbolicSet.hh`, is associated with a *domain* $[\![a, b]\!] \subseteq \mathbb{R}^n$ and a *grid parameter* $\eta \in \mathbb{R}^n_{>0}$. The class is used to create, manipulate and store a *subset* of the grid points contained in

$$D := ([a_1, b_1] \times \cdots \times [a_n, b_n]) \cap \eta\mathbb{Z}^n.$$

Each grid point $x \in D$ is identified with a cell $x + [\![-\eta/2, \eta/2]\!]$. Throughout the reminder of this section, let us use

$$S \subseteq D$$

to denote the subset of D which is encoded by an instance of the class `SymbolicSet`. Each grid point $x \in D$ is defined in terms of a number of binary decision (`BDD`) variables $d \in \mathbb{B}^m$ created with the `CUDD` library (see (16) for details). Each `BDD` variable obtains a unique `id`. The domain $[\![a, b]\!] \subseteq \mathbb{R}^n$, grid parameter $\eta \in \mathbb{R}^n_{>0}$ together with the `BDD` variable `ids` associated with an instance of a `SymbolicSet` make up the *abstract domain*. There exist several possibilities to instantiate an object of the class `SymbolicSet`:

(1) By using the domain $[\![a, b]\!] \subseteq \mathbb{R}^n$ and grid parameter $\eta \in \mathbb{R}^n_{>0}$, in which case new `BDD` variables are created. For example, to encode a set of grid points in $[-1, 1]^2 \cap (0.1, 0.1)\mathbb{Z}^2$, we can use

```
#include <array>
#include <iostream>
#include "cuddObj.hh"
#include "SymbolicSet.hh"
int main() {
  Cudd mgr;                                /* bdd manager to organize the BDD vars */
  double a[2]={-1,-1};                     /* lower bounds of the hyper rectangle */
  double b[2]={ 1, 1};                     /* upper bounds of the hyper rectangle */
  double eta[2]={0.1,0.1};                 /* grid node distance diameter */
  scots::SymbolicSet sset(mgr,2,a,b,eta); /* instantiate a SymbolicSet */
  return 1;
}
```

This is the standard instantiation method and should be used, for example, to create the symbolic state alphabet $\bar{X}_2$ and the symbolic input alphabet U_2, respectively.

(2) By using the copy constructor, in which case the BDD variables of original SymbolicSet instance are used for the new object.

```
scots::SymbolicSet nset(sset); /* nset and sset have the same abstract domain */
```

This option can be used for example, to create atomic propositions to formulate the specification. Suppose sset represents $\bar{X}_2$, then we use the following code to create the SymbolicSet to represent a subset of $\bar{X}_2$

```
scots::SymbolicSet target(sset); /* target and sset have the same abstract domain */
```

(3) Optionally, one can specify that the id's of the BDD variables should be newly generated

```
scots::SymbolicSet nset(sset,1); /* nset and sset have different abstract domain */
```

In this case, even though, nset and sset have the same domain $[\![a,b]\!] \subseteq \mathbb{R}^n$ and grid parameter $\eta \in \mathbb{R}^n_{>0}$, the BDD variable ids are different and therefore, both instances nset and sset have a different abstract domain. This instantiation method, is useful when it comes to the computation of the symbolic model, see Sec. 7.2.

(4) Consider two instances ss and is of the class SymbolicSet, whose domains and grid parameters are given by

$$[\![a,b]\!] \subseteq \mathbb{R}^n, \eta \in \mathbb{R}^n_{>0} \quad \text{, respectively,} \quad [\![c,d]\!] \subseteq \mathbb{R}^m, \mu \in \mathbb{R}^m_{>0}.$$

Then, we can use

```
scots::SymbolicSet prod(ss,is); /* prod is the Cartesian product of ss and is */
```

to create the object prod, whose domain and grid parameter is given by

$$[\![a,b]\!] \times [\![c,d]\!] \subseteq \mathbb{R}^{n+m}, \quad (\eta,\mu) \in \mathbb{R}^{n+m}_{>0}.$$

The BDD variable ids are taken from ss and is.

This instantiation method is useful to create a SymbolicSet to represent the controller, see Sec. 7.3.

(5) Similarly, we can create an instance of SymbolicSet as the projection of another instance of SymbolicSet. For example, consider the SymbolicSet object sset, whose domain and grid parameter are given $[\![a,b]\!] \subseteq \mathbb{R}^5$, $\eta \in \mathbb{R}^5_{>0}$. Then we use

```
std::vector<size_t> pdim={1,3,4};   /* projection dimension */
scots::SymbolicSet proj(sset,pdim); /* projection from sset onto dim {1,3,4} */
```

to create an SymbolicSet instance proj, whose domain and grid parameter are given by

$$[\![a',b']\!] \subseteq \mathbb{R}^3, \quad \eta' \in \mathbb{R}^3_{>0}$$

where $a'_1 = a_1$, $a'_2 = a_3$, $a'_3 = a_4$, $b'_1 = b_1$, $b'_2 = b_3$, $b'_3 = b_4$ and $\eta'_1 = \eta_1$, $\eta'_2 = \eta_3$, $\eta'_3 = \eta_4$.

This instantiation method is useful to extract the SymbolicSet to represent the state alphabet $\bar{X}_2$ and input alphabet U_2 from a SymbolicSet which represents $\bar{X}_2 \times U_2$, see Sec. 7.2.

Initially, when an instance of SymbolicSet is created the set S is empty, i.e., it does not contain any grid points. We can add and remove grid points associated with polytopes and ellipsoids of the form

$$P := \{x \in \mathbb{R}^n \mid Hx \leq h\} \quad \text{respectively} \quad E := \{x \in \mathbb{R}^n \mid |L(x-y)|_2 \leq 1\} \tag{15}$$

by using the methods

```
SymbolicSet::addPolytope()   /* add grid points associated with a polytope to symbolicSet_ */
SymbolicSet::remPolytope()   /* remove grid points associated with a polytope from symbolicSet_ */
SymbolicSet::addEllipsoid()  /* add grid points associated with an ellipsoid to symbolicSet_ */
SymbolicSet::remEllipsoid()  /* remove grid points associated with an ellipsoid from symbolicSet_ */
```

Each of those methods, requires an approximation parameter `scots::INNER` or `scots::OUTER`. We use those parameters to specify either to add an "outer" or "inner" approximation to the `SymbolicSet`. The inner $\check{P}$, $\check{E}$ and outer $\hat{P}$, $\hat{E}$ approximations are given by

$$\check{P} := \{x \in D \mid x + [\![-\eta/2, \eta/2]\!] \subseteq P\}, \qquad \check{E} := \{x \in D \mid |L(x-y)|_2 \leq 1 - r\},$$
$$\hat{P} := \{x \in D \mid x + [\![-\eta/2, \eta/2]\!] \cap P \neq \emptyset\}, \quad \hat{E} := \{x \in D \mid |L(x-y)|_2 \leq 1 + r\}.$$

where $r := \max_{x \in [\![-\eta/2, \eta/2]\!]} |Lx|_2$, which guarantees that the following inclusions hold

$$\bigcup_{x \in \check{E}} x + [\![-\eta/2, \eta/2]\!] \subseteq \big(E \cap [\![a, b]\!]\big) \subseteq \bigcup_{x \in \hat{E}} x + [\![-\eta/2, \eta/2]\!].$$

Further set manipulations are supported by the functions

```
SymbolicSet::clear()          /* remove all grid points from symbolicSet_ */
SymbolicSet::addGridPoints() /* add all grid points in the domain to symbolicSet_ */
SymbolicSet::addByFunction() /* add grid points according to function a to symbolicSet_ */
SymbolicSet::complement()    /* invert the grid point selection in symbolicSet_ */
```

Let us demonstrate the usage by an example. Suppose we want to store grid points associated with a polytope and an ellipsoid as defined in (15) with $H \in \mathbb{R}^{3\times 2}$, $h \in \mathbb{R}^3$, $L \in \mathbb{R}^2$ and $y \in \mathbb{R}^2$ given by

$$H := \begin{pmatrix} 1 & 0 \\ -1 & 1 \\ -1 & -1 \end{pmatrix}, \quad h := \begin{pmatrix} 0 \\ .7 \\ .7 \end{pmatrix}, \quad L := \begin{pmatrix} 2 & 0 \\ 0 & 4 \end{pmatrix}, \quad y := \begin{pmatrix} .6 \\ .25 \end{pmatrix}.$$

The sets are illustrated in Fig. 4 on the left. We use the following code to instantiate an object of

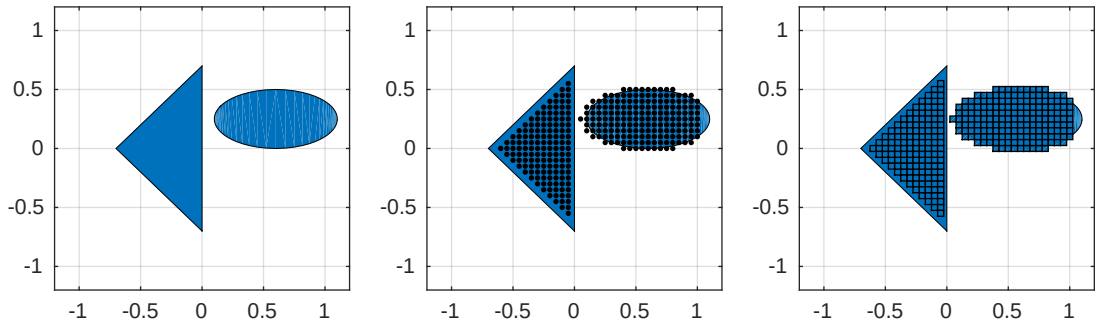

FIGURE 4. A triangle and an ellipse (displayed on the left) represented with `SymbolicSet.hh` and plotted in MATLAB. In the middle, the grid points are shown, while on the right the cells associated with the grid points are plotted.

`SymbolicSet.hh`, where we use $[\![a, b]\!] = [-1, 1] \times [-1, 1]$ and $\eta = (0.05, 0.05)$ to define the domain and grid parameter

```
#include <array>
#include <iostream>
#include "cuddObj.hh"
#include "SymbolicSet.hh"
int main() {
  Cudd mgr;                                /* bdd manager to organize the variables */
  double a[2]={-1,-1};                     /* lower bounds of the hyper rectangle */
  double b[2]={ 1, 1};                     /* upper bounds of the hyper rectangle */
  double eta[2]={0.05,0.05};               /* grid node distance diameter */
  scots::SymbolicSet sset(mgr,2,a,b,eta); /* instantiate a SymbolicSet */
```

then we use

```
  /* add inner approximation of polytope */
  double H[3*2]={1,0,-1,1,-1,-1};
  double h[3]={0,0.7,0.7};
  sset.addPolytope(3,H,h, scots::INNER);
  /* add outer approximation of ellipse */
  double L[2*2]={2,0 0,4};
  double y[2]={0.6,0.25};
  sset.addEllipsoid(L,y, scots::OUTER);
  sset.writeToFile("symset.bdd");
}
```

to add an inner approximation of P and outer approximation of E to the set `sset`. The grid points that are added to the `sset` are illustrated in Fig. 4. With the last line, we write the `BDD` representing the grid points in `sset` to the file `symset.bdd`. We use the MATLAB interface to visualize the grid points in MATLAB by

```
>> set=SymbolicSet('symset.bdd');
>> x=set.points;
>> plot(x(:,1),x(:,2),'.')
```

In C++ , the `SymbolicSet` stored in `symset.bdd` can simply be read by providing the filename in the constructore. For example,

```
  scots::SymbolicSet newset("symset.bdd");
```

reads the `SymbolicSet` from file and instantiates the object `newset` with the data stored in `symset.bdd`.

Additionally, we can use the method `SymbolicSet::addByFunction` as illustrated by the listing shown in Fig. 5 to add grid points to a `SymbolicSet`. The right side (of Fig. 5) shows the grid points that are added to `sset`.

```
  /* remove all points from the grid */
  sset.clear();
  /* define a function */
  auto f=[](double* x)->bool {
    return (x[0]*x[1]>=0.1);
  };
  /* points with f(x)=1 are added to sset */
  sset.addByFunction(f);
```

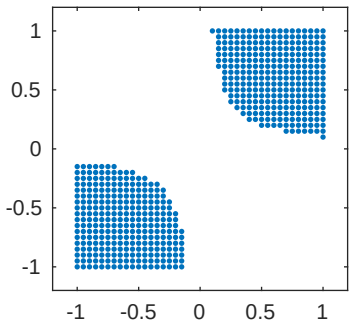

FIGURE 5. Example usage of `SymbolicSet::addByFunction`.

Internally, the grid information is stored in terms of the first grid point and the number of grid points in each dimension

```
double* SymbolicSet::firstGridPoints_
size_t* SymbolicSet::nofGridPoints_
```

This information can be access by the method `printInfo()`. In the previous example, the line

```
  sset.printInfo();
```

added at the end produces

```
First grid point: -1 -1
Grid node distance (eta)in each dimension: 0.05 0.05
Number of grid points in each dimension: 41 41
Number of grid points: 1681
Number of elements in the symbolic set: 570
```

The set S (of grid points defined by an instance of `SymbolicSet`) is encoded by an instance of a `BDD`, which (in the `CUDD` library) is a C++ class that allows one to create, store and manipulate a function that maps a number of binary variables to $\mathbb{B} := \{0, 1\}$, i.e.,

$$f : \mathbb{B}^m \to \mathbb{B}.$$

Each grid point $x \in D$ is defined in terms of binary variables $d \in \mathbb{B}^m$ created with the `CUDD` library. By using the `CUDD` library, each `BDD` variable obtains a unique `id`. The `id`'s of the `BDD` variables that are used in an instance of a `SymbolicSet` can by listed by

```
SymbolicSet::printInfo().
```

For example, if we use

```
sset.printInfo(1);
```

at the end of the code in Fig. 5, we obtain

```
Number of binary variables: 12
Bdd variable indices (=IDs) in dim 1: 0 1 2 3 4 5
Bdd variable indices (=IDs) in dim 2: 6 7 8 9 10 11
```

which tells us the number ($m = 12$) and `id`'s of the `BDD` variables that are used to encode the grid.

Each element in $d \in \mathbb{B}^m$ represents a grid point $\eta\mathbb{Z}^n$. Let us $m_i \in \mathbb{Z}_{>0}$ to denote the number of `BDD` variables that are used to encode the grid points $[a_i, b_i] \cap \eta_i\mathbb{Z}$ in dimension $i \in [1; n]$. Consider the function $\mathtt{id}_i : \mathbb{N} \to \mathbb{N}$ that maps an index (of the `BDD` variable) to the `BDD` variable `id`. Then each component $x_i \in [a_i, b_i] \cap \eta_i\mathbb{Z}$ of a grid point associated with an element $d \in \mathbb{B}^m$ is given by

$$x_i = p_i + \sum_{j=0}^{m_i-1} d_{\mathtt{id}_i(j)} 2^j \eta_j =: g_i(d) \tag{16}$$

where $p_i := \min([a_i, b_i] \cap \eta_i\mathbb{Z})$ is the first grid point for the component $i \in [1; n]$.

Consider the example listed in Fig. 5. Let us use $F : \mathbb{B}^{12} \to \mathbb{B}$ to refer to the binary function represented by `sset.symbolicSet_`, i.e., the `BDD` instance that encodes S, then, in view of the code in Fig. 5, we have

$$F(d) = 1 \iff g_1(d) \cdot g_2(d) \geq 0.1$$

where g_1 and g_2 are defined according to (16). Note that for this example $m_1 = m_2 = 6$ and the function $\mathtt{id}_2$ results in $\mathtt{id}_2(0) = 6, \ldots, \mathtt{id}_2(5) = 11$.

The individual binary variables $d \in \mathbb{B}^m$ used to represent the set of grid points in D can be listed by

```
sset.printInfo(2);
```

For the example in Fig. 5, part of the output is given by

```
000000-000-0 1
000000-00100 1
000000-01-00 1
000000-1--00 1
000001000001 1
000001000101 1
000001000110 1
...
```

The symbol - represent "don't cares", which means for example in the second line that `000000000100` and `000000100100` evaluate to 1. For the last line, we verify

$$g_1(000001000110) \cdot g_2(000001000110) = 0.6 \cdot 0.2 \geq 0.1$$

The grid point associated with a particular element $d \in \mathbb{B}^m$ is obtained by the method

```
SymbolicSet::mintermToElement()
```

Here `minterm` refers to an integer array of size m and `element` refers to a double array of size n. For example, the code

```
int minterm[12] = {0,0,0,0,0,1,0,0,0,1,1,0};
double x[2];
sset.mintermToElement(minterm,x);
std::cout << x[0] << " " << x[1] << std::endl;
```

gives

```
0.6 0.2
```

7.2. **Symbolic Model.** The classes `SymbolicModel` and `SymbolicModelGrowthBound` implemented in the files `./bdd/SymbolicModel.hh`, respectively, `./bdd/SymbolicModelGrowthBound.hh` are used to compute the transition function $F_2 : \bar{X}_2 \times U_2 \rightrightarrows \bar{X}_2$, of the symbolic model $S_2 = (X_2, U_2, F_2)$, see Sec. 2.2. The transition function can also be considered as relation $F_2 \subseteq \bar{X}_2 \times U_2 \times \bar{X}_2$. The `SymbolicModel` is the base class of `SymbolicModelGrowthBound`, which contains the information about the `SymbolicSet` representing

$$\bar{X}_2 \times U_2 \times \bar{X}_2.$$

Suppose that the instances `ss` and `is` of `SymbolicSet` encode the sets $\bar{X}_2$ and U_2, whose domain and grid parameter are given by

$$[\![a,b]\!] \subseteq \mathbb{R}^n, \eta \in \mathbb{R}^n_{>0} \quad \text{respectively} \quad [\![c,d]\!] \subseteq \mathbb{R}^m, \mu \in \mathbb{R}^m_{>0}. \tag{17}$$

In order to instantiate an object of `SymbolicModel` (or `SymbolicModelGrowthBound`), we first create a copy of `ss` to represent the post $\bar{X}_2$ with new `BDD` variable `ids` by

```
scots::SymbolicSet sspost(ss,1);
```

In this way, the post states in $\bar{X}_2$ have their own abstract domain. Afterwards we can instantiate a `SymbolicModel` by

```
scots::SymbolicModel abs(ss,is,sspost);
```

The BDD which represents $F_2 \subseteq \bar{X}_2 \times \bar{U}_2 \times \bar{X}_2$ is stored in

```
scots::SymbolicModel::transitionRelation_
```

and can be accessed via a `SymbolicSet` by

```
scots::SymbolicSet tr=abs.getTransitionRelation();
```

The domain and grid parameter of `tr` follows from (17) to

$$[\![a,b]\!] \times [\![c,d]\!] \times [\![a,b]\!] \subseteq \mathbb{R}^{n+m+n}, \quad (\eta,\mu,\eta) \in \mathbb{R}^{n+m+n}_{>0}.$$

The algorithm, which is implemented in `SymbolicModelGrowthBound` to compute the transition function of the symbolic model S_2 is listed in Alg. 1. The computation is based on a growth bound β

Algorithm 1 Computation of $F_2 : \bar{X}_2 \times U_2 \rightrightarrows \bar{X}_2$

Require: $\bar{X}_2$, U_2, β, φ, z, $r = \eta/2$, τ

1: **for all** $c + [\![-r, r]\!] \in \bar{X}_2$ and $u \in U_2$ **do**
2: $\quad r' := \beta(r + z, u)$
3: $\quad c' := \varphi(\tau, c, u)$
4: $\quad A := \{x'_2 \in X_2 \mid \left(c' + [\![-r' - z, r' + z]\!]\right) \cap x'_2 \neq \emptyset\}$
5: $\quad$ **if** $A \subseteq \bar{X}_2$ **then**
6: $\quad\quad F_2(x_2, u) := A$
7: $\quad$ **else**
8: $\quad\quad F_2(x_2, u) := \emptyset$

in line 2, and the trajectory of the nominal system φ in line 3. The mathematical definitions of β and φ are given in (10) and above (8), respectively.

The functions β and φ need to be provided by the user. For both functions, an initial value problem has to be solved. In general, explicit solutions to such problems do not exist and we are required to work with numerical approximations obtained by a numerical ODE solver. For example, in the DC-DC boost converter example in `./examples/bdd/dcdc2` a fixed step size Runge Kutta scheme of order 4 has been used. The header of the ODE solver is given by

```
/* data types for the ode solver */
typedef std::array<double,2> state_type;
typedef std::array<double,1> input_type;
/* function header of the ode solver */
template<class F>
void ode_solver(F rhs, state_type &x, input_type &u, size_t nint, double h);
```

Using the `ode_solver`, a user can define the functions φ and β in C++ as

```
std::function<void(state_type &x, input_type &u)>.
```

For example, in `./examples/bdd/dcdc2` we use

```
/* we integrate the dcdc ode by 0.5 sec (the result is stored in x) */
auto dcdc_post = [](state_type &x, input_type &u) -> void {
  size_t nint=5; /* number of intermediate step size */
  double h=0.1; /* h* nint = sampling time */
  ode_solver(system_ode,x,u,nint,h);
};
/* computation of the growth bound (the result is stored in r) */
auto radius_post = [](state_type &r, input_type &u) -> void {
  size_t nint=5; /* number of intermediate step size */
  double h=0.1; /* h* nint = sampling time */
  ode_solver(growth_bound_ode,r,u,nint,h);
};
```

to provide β and φ in line 2, respectively, in line 3 in Alg. 1. Here `system_ode` and `growth_bound_ode` are `std::function`, which implement the DC-DC boost converter ODE, respectively the ODE associated with (10), see the file `dcdc.cc` in `./examples/bdd/dcdc2`.

Alternatively, one can use a different ODE solver, for example implemented in the `odeint` library provided within the `BOOST` library. An example that uses an ODE solver provided by the `BOOST` library can be found in `./examples/bdd/boostsolver`.

Given the functions β and φ in line 2, respectively, in line 3 in Alg. 1, we are ready to call the method `SymbolicModelGrowthBound::computeTransitionRelation`. For the DC-DC boost converter example in `./examples/bdd/boostsolver` we use

```
  /* copy X and assign new BDD IDs */
  scots::SymbolicSet sspost(ss,1);
  scots::SymbolicModelGrowthBound<state_type,input_type> abs(&ss, &is, &sspost);
  /* compute the transition relation */
  abs.computeTransitionRelation(dcdc_post, radius_post);
  /* get the number of elements in the transition relation */
  std::cout << "Number of elements in the transition relation: " << abs.getSize();
```

were we instantiate an object of `SymbolicModelGrowthBound` with the `SymbolicSets` `ss`, `is` and `sspost`.

We can use the method `SymbolicSet::writeToFile` to store the transition relation to a file. For example, in `./examples/bdd/boostsolver` we access the transition relation as `SymbolicSet` and write it the file `dcdc_abs.bdd`.

```
scots::SymbolicSet tr=abs.getTransitionRelation();
/* write SymbolicSet to file */
tr.writeToFile("dcdc_abs.bdd");
```

Note that `dcdc_abs.bdd` not only contains the information of the transition relation F_2, but it also contains the `SymbolicSet` information on $\bar{X}_2 \times U_2 \times \bar{X}_2$. Hence the complete symbolic model $S_2 = (X_2, U_2, F_2)$ can be restored by

```
int main() {
  /* there is one unique manager to organize the bdd variables */
  Cudd mgr;
  /* read SymbolicSet containing the transition relation from file */
  scots::SymbolicSet trset(mgr,"dcdc_abs.bdd");
  /* the first two dimensions correspond to the state alphabet */
  std::vector<size_t> ssdim={1,2};
  scots::SymbolicSet ss(trset,ssdim);
  /* the third dimensions corresponds to the input alphabet */
  std::vector<size_t> isdim={3};
  scots::SymbolicSet is(trset,isdim);
  /* the last two dimensions corresponds to the state alphabet
   * (containing the post variables) */
  std::vector<size_t> sspostdim={4,5};
  scots::SymbolicSet sspost(trset,sspostdim);
  /* create SymbolicModel */
  scots::SymbolicModel sys(&ss,&is,&sspost);
  sys.setTransitionRelation(trset.getSymbolicSet());
  return 1;
}
```

see the file `./bdd/examples/boostsolver/dcdcReadFromFile.cc`.

7.3. **Controller Synthesis.** The fixed point computations described in Sec. 2.5 are implemented in the class `FixedPoint`, whose source code can be found in `./bdd/FixedPoint.hh`. In particular, we use the methods `FixedPoint::reach` and `FixedPoint::safe` to compute the maps given in (13) and (14), respectively. The method `FixedPoint::pre` is used to compute the map (12).

Let us demonstrate the usage of `FixedPoint` by the example in `./examples/hscc16/dcdc`. We are given an invariance problem (S_2, Σ_2), i.e., Σ_2 is an invariance specification associated with I_2, Z_2 for some $I_2 \subseteq \bar{X}_2$ and $Z_2 \subseteq \bar{X}_2 \times U_2$. In this example, we use the `SymbolicSet` object `ss` to describe $\bar{X}_2$. Since Z_2 is independent of U_2 it is sufficient to make a copy of `ss` to describe Z_2 as `SymbolicSet`. We use the code

```
double H[4*2]={-1, 0, 1, 0, 0,-1, 0, 1};
double h[4] = {-1.15,1.55,-5.45, 5.85};
/* make a copy of ss */
scots::SymbolicSet z2(ss);
z2.addPolytope(4,H,h, scots::INNER);
```

to create Z_2 and add an inner approximation of the rectangle illustrated by the red line on the left part of Fig. 6. The blue rectangles depict the cells $c + [-\eta/2, \eta/2]$ in Z_2.

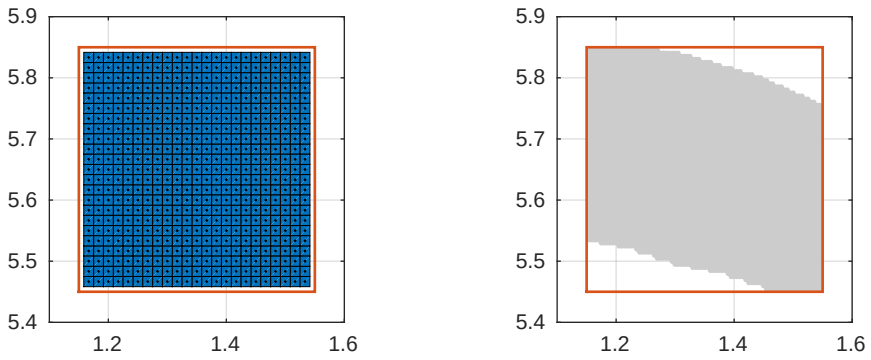

FIGURE 6. Left: The set Z_2 and its `SymbolicSet` representation plotted in MATLAB with $\eta = (1/60, 1/60)$. Right: The gray shaded region illustrates the domain of the controller for $\eta = (1/200, 1/200)$.

We instantiate an object of `FixedPoint` with the symbolic model S_2 given by a `SymbolicModel` object `abstraction`

```
scots::FixedPoint fp(&abstraction);
```

The fixed point algorithms in `FixedPoint` directly operate on the `BDD` representing the `SymbolicSet`. Hence, we first extract the `BDD` from `z2` and then use the method `FixedPoint::safe` to compute the map (14) by

```
/* the fixed point algorithm operates on the BDD directly */
BDD Z = safe.getSymbolicSet();
BDD C = fp.safe(Z);
```

In order to store the result and access the controller from within MATLAB, we interpret the map (14) as set $H_c' \subseteq \bar{X}_2 \times U_2$ and create a `SymbolicSet` object to store the controller to the file `dcdc_controller.bdd`

```
scots::SymbolicSet controller(ss,is);
controller.setSymbolicSet(C);
controller.writeToFile("dcdc_controller.bdd");
```

Now the set H_c' can be simply accessed form MATLAB using the MATLAB class `SymbolicSet`. For example, with[2]

[2]Recall that the directory `./mfiles` needs to be added to the MATLAB path, see Sec. 4.

```
>> con=SymbolicSet('dcdc_controller.bdd',[1 2]);
>> x=con.points;
>> plot(x(:,1),x(:,2),'.')
```

we load the set H_c' as `SymbolicSet` form the file `dcdc_controller.bdd` and plot its domain, see the right part of Fig. 6. By using the second argument `[1 2]` in the constructor, we only read the points in H_c' projected onto the first two dimensions, see Sec. 7.4 for more details.

The control inputs defined by the map H_c' can by accessed by

```
>> x=[1.2 5.6];
>> u=con.getInputs(x);
```

If the state x is outside the domain of H_c', i.e., $H_c'(x) = \varnothing$ we obtain

```
>> con.getInputs(x)
Error using SymbolicSet/getInputs (line 94)
The state [1.2 5.5] is not in the domain of the controller stored in:dcdc_controller.bdd
```

An example which uses `FixedPoint:reach` to solve a reachability specification can be found in `./examples/hscc16/vehicle1`.

Note that the method `FixedPoint::pre` can be used to compute the map (12). Hence, it is quite straightforward to implement more complex fixed point algorithms to compute controllers to enforce more general specifications. Let us for example consider the *reach-and-stay* specification Σ_2 associated with I_2, Z_2 given by

$$\Sigma_2 := \{(u,x) \in (U_2 \times X_2)^{[0;\infty[} \mid x(0) \in I_2 \implies \exists_{t\in[0;\infty[} \forall_{t'\in[t;\infty[} : (x(t'),u(t')) \in Z_2\}.$$

We can solve the control problem (S_2, Σ_2) by the nested fixed point algorithm

$$\mu\bar{Y}.\nu Y.(\mathrm{pre}(Y) \cap Z_2) \cup \mathrm{pre}(\bar{Y})$$

from which we derive a controller as follows. Let $Y_\infty := \mu\bar{Y}.\nu Y.(\mathrm{pre}(Y) \cap Z_2) \cup \mathrm{pre}(\bar{Y})$ and consider the sets

$$\begin{aligned} Y_0 &:= \nu Y.(\mathrm{pre}(Y) \cap Y_\infty) \\ Y_{i+1} &:= \nu Y.(\mathrm{pre}(Y) \cap Y_\infty) \cup \mathrm{pre}(Y_i) \end{aligned}$$

then a controller C, which solves the control problem (S_2, Σ_2) is given by

$$H_c(q,x_2) = \begin{cases} H_c'(x_2) \times \{x_2\} & \text{if } x_2 \in Y_\infty \\ U_2 \times \{x_2\} & \text{otherwise} \end{cases}$$

$$F_c(q,x_2) = \begin{cases} \{q\} & \text{if } x_2 \in Y_\infty \\ \varnothing & \text{otherwise} \end{cases}$$

where the map H_c' is given by

$$H_c'(x_2) = \left\{u \in U \mid F_2(x_2,u) \neq \varnothing \wedge \left((x_2,u) \in Y_0 \vee F_2(x_2,u) \subseteq \pi_{X_2}(Y_{j(x_2)-1})\right)\right\}.$$

We implemented the fixed point computation in `./examples/bdd/dcdc2` by using `FixedPoint::pre` according to the code

```
/* outer fp variables*/
BDD X=mgr.bddOne();
BDD XX=mgr.bddZero();
/* inner fp variables */
BDD Y=mgr.bddZero();
BDD YY=mgr.bddOne();
/* the controller */
BDD C=mgr.bddZero();
/* helper BDD */
BDD U=is.getCube();
/* outer iteration */
for(size_t i=1; XX != X; i++) {
  X=XX;
  BDD preX=fp.pre(X);
```

```
    /* inner iteration */
    YY = mgr.bddOne();
    for(size_t j=1; YY != Y; j++) {
      Y=YY;
      YY= ( fp.pre(Y) & T ) | preX;
    }
    XX=YY;
    /* only add (state/input) pairs whose state is not already in the controller */
    BDD N = XX & (!(C.ExistAbstract(U)));
    C=C | N;
  }
```

the controller is stored in the `BDD C`. We use the method `SymbolicSet::writeToFile` to save the result to the file `dcdc_controller.bdd` by

```
scots::SymbolicSet controller(ss,is);
controller.setSymbolicSet(C);
controller.writeToFile("dcdc_controller.bdd");
```

A closed loop trajectory of the system is illustrated in Fig. 7. The trajectory is obtained by running the MATLAB file `dcdc.m` in the example directory `./examples/bdd/dcdc2`.

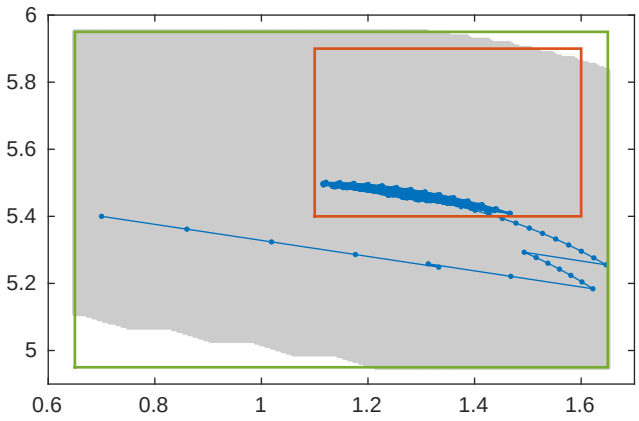

FIGURE 7. Closed loop trajectory of the DC-DC boost converter satisfying the reach-and-stay specification associated with the red rectangle.

7.4. **MATLAB Interface.** SCOTS comes with a small `mex-file` interface to access `SymbolicSets` that have been stored using the C++ code from MATLAB. The access of a `SymbolicSet` is implemented as a MATLAB class `SymbolicSet` stored in `./mfiles/SymbolicSet.m`. The MATLAB class provides an interface to the MATLAB commands implemented in the `mex-file ./mfiles/mexfiles/mexSymbolicSet.cc`. In order to be able to use the MATLAB class `SymbolicSet`, the `mex-file` needs to be compiled and added to the MATLAB path, see Sec. 4.

Suppose we stored an instance of a `SymbolicSet` with domain $[\![a,b]\!] \subseteq \mathbb{R}^n$, grid $\eta \in \mathbb{R}^n_{>0}$ and grid points

$$S \subseteq ([\![a,b]\!] \cap \eta\mathbb{Z}^n).$$

The MATLAB interface provides three functionalities:

- It allows a user to access the set S. For example, for a set S stored in `polytope.bdd` (see below) we can use

```
>> P=SymbolicSet('polytope.bdd');
>> x=P.points;
```

  to load the grid points in S into the MATLAB workspace.
- It allows a user to check if an element $x \in \mathbb{R}^n$ is covered by S, i.e.,

$$\exists_{c \in S} : \quad x \in c + [\![-\eta/2, \eta/2]\!]$$

  Usage:

```
>> P.isElement(x);
```

- For a vector $x \in \mathbb{R}^{n_1}$ with $n_1, n_2 \in [1; n[$ and $n_1 + n_2 = n$, it implements the function

$$S(x) = \{y \in \mathbb{R}^{n_2} \mid ([x]_\eta, y) \in S\} \tag{18}$$

  where the components $\hat{x}_i$ of $\hat{x} := [x]_\eta$ correspond to the nearest grid point, computed by

$$\hat{x}_i := \texttt{std} :: \texttt{round}(x_i/\eta_i)\eta_i$$

  Usage:

```
>> P.setValuedMap(x);
```

Let us consider for example the polytope given by the convex hull $P := \mathrm{co}(V)$ with

$$V = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 1 & 1 \\ -1 & 1 & 0 \\ 0 & 1 & -1 \\ -1 & 0 & -1 \end{pmatrix}$$

which is illustrated in Fig. 8. We use `SymbolicSet::addPolytope` and `SymbolicSet::writeToFile` to compute an inner approximation of P and to store the result to the file `polytope.bdd`. Subsequently, we use

```
>> P=SymbolicSet('polytope.bdd');
>> x=P.points;
>> plot3(x(:,1),x(:,2),x(:,3),'.')
```

to read and plot the grid points in MATLAB. Optionally, we can provide a list of indices `idx` $\subseteq$ `[1;n]`, in which case only the grid points projected on the specified indices are loaded. For example, with

```
>> P=SymbolicSet('polytope.bdd','projection',[1 3]);
>> x=P.points;
>> plot(x(:,1),x(:,2),'.')
```

we obtain only a two dimensional vector `x` which contains grid points

$$\left\{(x_1, x_3) \in ([a_1, b_1] \cap \eta_1\mathbb{Z}) \times ([a_3, b_3] \cap \eta_3\mathbb{Z}) \mid \exists_{(x_2 \in [a_2, b_2] \cap \eta_2\mathbb{Z})} : (x_1, x_2, x_3) \in S\right\}.$$

The outcome of the code listing above is illustrated on the right side of Fig. 8. This option might be particularly useful, if the set to be loaded contains a huge amount of grid points.

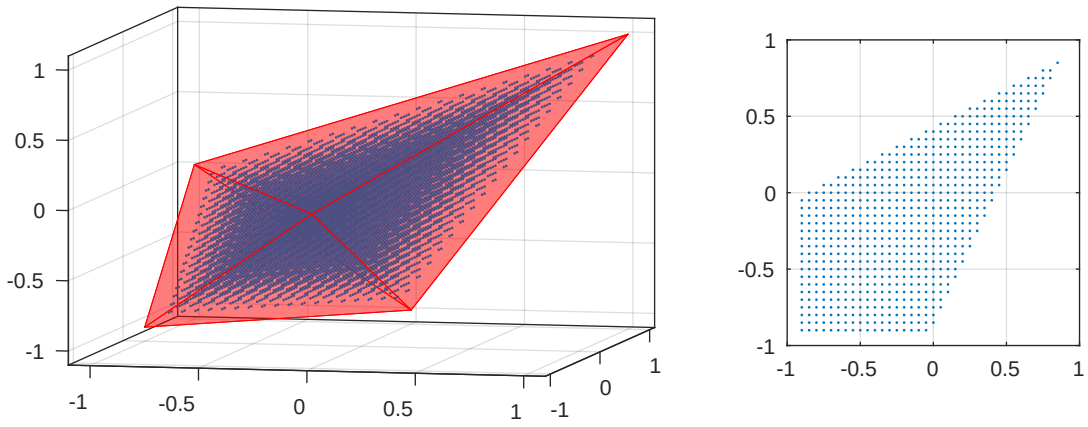

FIGURE 8.

The set defined in (18) is implemented in the MATLAB class method `SymbolicSet::setValuedMap`. In the previous example, we obtain

```
>> P.setValuedMap([0.7;.75])
ans =

    0.7000
    0.6500
```

The same functionality is implemented in the method `SymbolicSet::getInputs`. Optionally, a user can provide an index list `idx` $\subseteq [1;n]$. For example, with

```
>> P.setValuedMap([0.7;.75],[3 1]);
```

we specify that 0.7 and 0.75 correspond to the third, respectively, first coordinate of the n-dimensional state space.

Part 3. SPARSE MATRICES BASED IMPLEMENTATION

Please see SCOTSv0.2.

References

[1] G. Reißig, A. Weber, and M. Rungger. *Feedback Refinement Relations for the Synthesis of Symbolic Controllers*. 2015. arXiv: 1503.03715 [math.OC,cs.SY].

[2] R. E. Bryant. "Symbolic Boolean manipulation with ordered binary-decision diagrams". In: *ACM Computing Surveys* 24.3 (1992), pp. 293–318.

[3] M. Rungger and M. Zamani. "SCOTS: A Tool for the Synthesis of Symbolic Controllers". In: *HSCC*. ACM, 2016.

[4] A. Arnold and D. Niwinski. *Rudiments of μ-calculus*. Elsevier, 2001.

[5] P. Tabuada. *Verification and Control of Hybrid Systems – A Symbolic Approach*. Springer, 2009.