QUEST– USER MANUAL

PUSHPAK JAGTAP

Contents

Part 1. Introduction

1. About QUEST

QUEST is an open source software tool (available at http://www.hcs.ei.tum.de) for automated controller synthesis for incrementally input-to-state stable nonlinear control systems. The tool is implemented in C++ and contains two major parts:

1. Construction of symbolic abstraction: the tool uses state-space quantization-free approach for the construction of symbolic abstraction which can be potentially more beneficial for systems with high-dimensional state spaces.
2. Symbolic controller synthesis: the synthesis of controller is implemented using fixed point computations.

The implementation of QUEST uses *binary decision diagrams* (BDD) [2] as an underlying data structure for memory-efficient storage and computation of symbolic abstraction and controller. Operations on BDDs are handled with the help of CUDD binary decision diagram library and play the major role in order to construct symbolic abstractions.

The tool is intended to be used and extended by researches in the area of formal synthesis for complex systems.

In this part, we give a quick overview on the basic concepts which will be used during the manual. We also review the theory behind constructing symbolic abstractions and synthesizing symbolic controllers based on them.

2. Symbolic Controller Synthesis Basics

2.1. Control System.

QUEST supports computation of controller synthesis for incrementally input-to-state stable nonlinear control systems of the form

$$\dot{\xi} = f(\xi, \upsilon), \tag{1}$$

where f is given by $f : \mathbb{R}^n \times \mathsf{U} \to \mathbb{R}^n$ and $\mathsf{U} \subseteq \mathbb{R}^m$. We assume that the set U is non-empty and $f(., u)$ is continuously differentiable for every $u \in \mathsf{U}$. Let $\xi_{x,\upsilon}(t)$ be the solution of (1) starting from initial condition x under input function $\upsilon \in \mathcal{U}$ at time $t > 0$, where $\mathcal{U}$ denotes the set of measurable input functions. For the sake of completeness, we recall the definition of incremental input-to-state stability [1]:

Definition 2.1. *A nonlinear control system as in (1) is said to be incrementally input-to-state stable if there exist a $\mathcal{KL}$ function β and a $\mathcal{K}_\infty$ function γ such that for any $t \in \mathbb{R}_0^+$, any two initial conditions x and $\hat{x}$, and any $\upsilon, \hat{\upsilon} \in \mathcal{U}$, the following condition is satisfied:*

$$\|\xi_{x,\upsilon}(t) - \xi_{\hat{x},\hat{\upsilon}'}(t))\| \leq \beta(\|x - \hat{x}\|, t) + \gamma(\|\upsilon - \upsilon'\|_\infty).$$

We refer the interested reader to [1] for definitions of $\mathcal{KL}$ and $\mathcal{K}_\infty$ functions.

2.2. A Unified Framework for Control Systems and Symbolic Abstractions.

We present a notion of system which serves as unified modelling framework for nonlinear control systems in (1) and their symbolic abstractions. A system is a tuple

$$S = (X, X_0, U, \longrightarrow, Y, H)$$

that consists of: a (possibly infinite) set of states X; a (possibly infinite) set of initial states $X_0 \subseteq X$; a (possibly infinite) set of inputs U; a transition relation $\longrightarrow \subseteq X \times U \times X$; a set of outputs Y; and an output map $H : X \to Y$.

We use notation $x \xrightarrow{u} x'$ for a transition $(x, u, x') \in \longrightarrow$, where state x' is a u-successor (or simply successor) of state x, for some input $u \in U$. System S is called metric if the output set Y is equipped with a metric $\mathsf{d} : Y \times Y \longleftrightarrow \mathbb{R}_0^+$.

2.3. **Approximate Bisimulation Relation.** We introduce the notion of approximate bisimulation relation [3] which is later used to relate symbolic abstractions with the original system (1).

Definition 2.2. *Let $S_1 = (X_1, X_{10}, U_1, \underset{1}{\longrightarrow}, Y_1, H_1)$ and $S_2 = (X_2, X_{20}, U_2, \underset{2}{\longrightarrow}, Y_2, H_2)$ be two metric systems having same output sets $Y_1 = Y_2$ and metric $\mathbf{d}$. For $\varepsilon \in \mathbb{R}_0^+$, a relation $\mathcal{R} \subseteq X_1 \times X_2$ is said to be an ε-approximate bisimulation relation between S_1 and S_2 if it satisfies following conditions:*

(i) $\forall (x_1, x_2) \in \mathcal{R}$, *we have* $\mathbf{d}(H_1(x_1), H_2(x_2)) \leq \varepsilon$;
(ii) $\forall (x_1, x_2) \in \mathcal{R}$, $x_1 \underset{1}{\xrightarrow{u_1}} x_1'$ *in* S_1 *implies* $x_2 \underset{2}{\xrightarrow{u_2}} x_2'$ *in* S_2 *satisfying* $(x_1', x_2') \in \mathcal{R}$;
(iii) $\forall (x_1, x_2) \in \mathcal{R}$, $x_2 \underset{2}{\xrightarrow{u_2}} x_2'$ *in* S_2 *implies* $x_1 \underset{1}{\xrightarrow{u_1}} x_1'$ *in* S_1 *satisfying* $(x_1', x_2') \in \mathcal{R}$.

2.4. **Computation of Symbolic Abstractions.** We consider the sampled behavior of (1) with sampling time $\tau > 0$. Then the corresponding system representation is given as the tuple

$$S_1 = (X_1, X_{10}, U_1, \underset{1}{\longrightarrow}, Y_1, H_1),$$

where $X_1 = \mathbb{R}^n$, $X_{10} \subseteq X_1$, $U_1 = \mathsf{U}$, there exists a transition $x_1 \underset{1}{\xrightarrow{u}} x_1'$ iff $x_1' = \xi_{x_1,u}(\tau)$, $Y_1 = X_1$, $H = 1_x$. Here, we abused notation by identifying $u \in \mathsf{U}$ with the constant input curve with domain $[0, \tau[$ and value u.
`QUEST` computes symbolic models that are related via approximate bisimulation relation to S_1. For constructing symbolic abstraction of S_1 we use a state-space quantization-free approach as discussed in [4, 7, 8]. We assume that the set of inputs is bounded and (1) is incrementally input-to-state stable. Under these assumptions, one can construct an approximate bisimilar symbolic model of S_1. Let $\overline{U}$ be the bounded quantized input space with a quantization parameter η, N be the temporal horizon, and x_s be a source state, then the symbolic model of S_1 is given by a tuple

$$S_2 = (X_2, X_{20}, U_2, \underset{2}{\longrightarrow}, Y_2, H_2),$$

where

- $X_2 = \overline{U}^N$, $X_{20} = X_2$, $U_2 = \overline{U}$, $Y_2 = Y_1$;
- $x_2 \underset{\rho}{\xrightarrow{u}} x_2'$, where $x_2 = (u_1, u_2, \ldots, u_N) \in X_2$, if and only if $x_2' = (u_2, \ldots, u_N, u)$ for some $u \in U_2$;
- $H_2(x_2) = \xi_{x_s,x_2}(N\tau)$.

For more details on the construction of symbolic abstraction using state-space quantization-free approach, the interested readers are referred to the results in [4], [8], and [7].

2.5. **Controller Synthesis via Fixed Point Computation.** `QUEST` natively supports invariance (often referred to as safety) and reachability specifications. For the synthesis of controller C to enforce these specifications, we make use of two fixed point algorithms: minimum fixed point and maximum fixed point algorithm. Moreover, `QUEST` also supports customized specifications such as reach and stay by combining these two algorithms. The implementation of controller synthesis using fixed point computation is similar to the one used in `SCOTS` [5]. For more details on implementation of fixed point algorithms, we refer the readers to material available on `https://www.hcs.ei.tum.de/en/software/scots/`.

Part 2. Getting Started with QUEST

3. Source Code Organization

```
./manual /* manual of QUEST */
./src /* the source code of the QUEST */
./examples /* directory containing various examples*/
./license.txt /* the license file */
./readme.txt /* a quick description pointing to this manual */
./installation_notes_windows/* installation guidlines for windows platform*/
```

4. Installation

In general, `QUEST` is implemented in "header-only" style and you only need a working C++ developer environment. However, `QUEST` uses the CUDD library maintained by Fabio Somenzi, which can be downloaded at http://vlsi.colorado.edu/~fabio/.

The requirements and installation instructions 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 includes the necessary tools already
   - Windows: You need to have MSYS-2 installed or use the latest update of Windows 7 providing support for Ubunto-on-windows.

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

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

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

   On Windows and linux, 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`. For further details about windows installations (wich is somehow different), please refer to the readme-win.txt file within `SCOTS`. You should also test the BDD installation by compiling a dummy programm, e.g. `test.cc` as the following

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

   which should be compiled by

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

5. Running a Sample Example

For a quickstart

(1) go to one of the examples in

```
./examples/trafficmodel /* five dimensional traffic model for safety specification */
./examples/thermalmodel10R /* ten-room thermal model for safety specification */
./examples/thermalmodel6R /* six-room thermal model for reach and stay specification */
```

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

```
$ make
$ ./trafficmodel
```

(5) for graphical visualization of output, run the m file, for example in `./examples/trafficmodel` run

```
>> trafficmodel.m
```

(6) modify the example to your needs

6. Implementation of QUEST

In this section, we describe the architecture of `QUEST`. The algorithm is mainly distributed among three C++ classes:

- `SymbolicSetSpace`
- `getAbstraction`
- `fixedPointMode`

6.1. `SymbolicSetSpace`.

The `SymbolicSetSpace` is the main class in which the transition relations as described in section 2.4 are computed with the help of binary decision diagrams (BDDs) [2] as underlying data structure. Specifically, we use the object oriented wrapper in the CUDD library [6]. It accepts temporal horizon N and quantized input set $\overline{U}$ as inputs. The class `SymbolicSetSpace` directly constructs the transition relations as

Algorithm 1 Computation of transition relation

Require: N, $\overline{U}$

1: Let $x = (u_1, u_2, \ldots, u_N) \in \overline{U}^N$, $x' = (u'_1, u'_2, \ldots, u'_N) \in \overline{U}^N$ and $u \in \overline{U}$
2: **for all** x and u **do**
3:     **for** $i = 1$ to $N - 1$ **do**
4:         $u'_i = u_{i+1}$
5:     $u'_N = u$

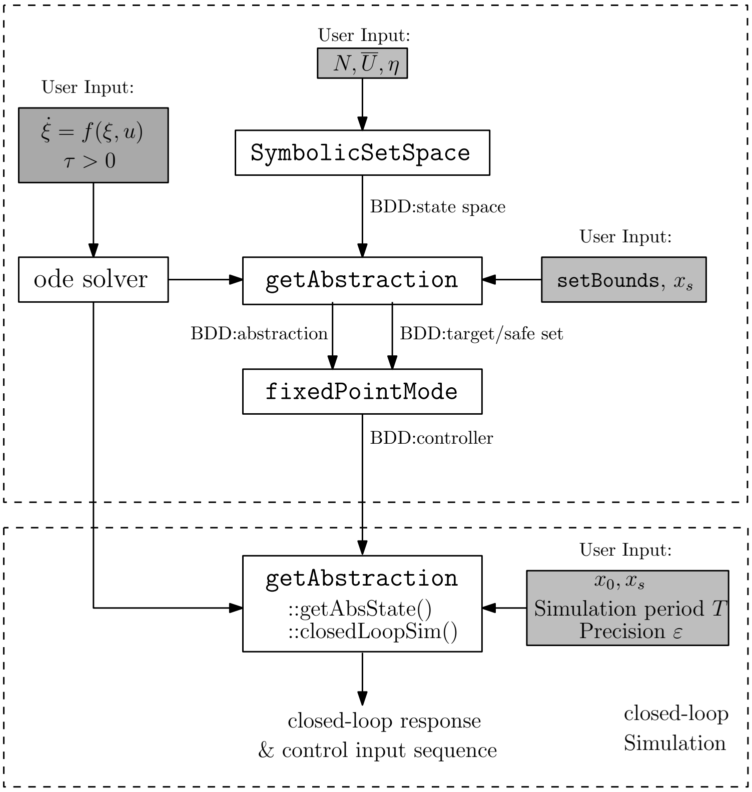

FIGURE 1. Workflow

6.2. `getAbstraction`. The `getAbstraction` is a derived class of the `abstractionMode` which manages all BDD related information, such as number and indices of variables. The `getAbstraction` class provides some supporting functions that are required for overall operation of `QUEST`. Some of important functions are listed below:

```
getAbstraction::getAbstractSet() /* get set of abstract states whose output maps are in safety/target region */
getAbstraction::getOutput() /* get output map H(x) corresponding to state x in the abstraction*/
getAbstraction::getAbsState() /* get abstract state related to concrete state in the original system*/
getAbstraction::closedLoopSim() /* closed-loop simulation and printing output response */
```

6.3. `fixedPointMode`. This class implements fixed point computation for the synthesis of controller. In particular, we use the methods `fixedPointMode::reach()`, `fixedPointMode::safe()` and `fixedPointMode::reachStay()` to synthesize controllers by solving fixed point computation for reachability, safety, and reach and stay specification, respectively.
The general work flow explaining use of classes with the different user inputs and the possible tool output is illustrated in Figure 1.

7. USAGE OF `QUEST`

In order to use `QUEST`, we create a C++ file in which we include the cudd library `cuddObj.hh` and header-only classes `SymbolicSetSpace.hh`, `abstractionMode.hh`, `getAbstraction.hh`, and `fixedPointMode.hh`. We begin with defining user inputs temporal horizon N, sampling time T, and the dynamics of the system. For synthesis of controller, the solution of (1) is needed for that we require to work with numerical approximations obtained by a numerical ODE solver. For example, in the thermalmodel example in `./examples/thermalmodel6R` a fixed step size Runge Kutte scheme of order four has been used. For implementation we use:

```
const double T = 25; /* Sampling time */
const size_t N = 6; /* Temporal Horizon */
const int sDIM = 10; /* System dimension */
const int iDIM = 2; /* Input dimension */
typedef std::array<double,sDIM> state_type; /* state type */
auto system_post = [](state_type &x, double* u) -> void {
/* ode describing six-room thermal model*/
auto rhs=[us](state_type &xx, const state_type &x) -> void {
const double a=0.05;
const double ae1=0.005;
const double ae4=0.005;
const double ae=0.0033;
const double ah=0.0036;
const double te=10;
const double th=100;
xx[0] = (-3*a-ae1-ah*us[0])*x[0]+a*x[1]+a*x[2]+a*x[4]+ae1*te+ah*th*us[0];
xx[1] = (-2*a-ae)*x[1]+a*x[0]+a*x[3]+ae*te;
xx[2] = (-2*a-ae)*x[2]+a*x[0]+a*x[3]+ae*te;
xx[3] = (-3*a-ae4-ah*us[1])*x[3]+a*x[1]+a*x[2]+a*x[5]+ae4*te+ah*th*us[1];
xx[4] = (-a-ae)*x[4]+a*x[0]+ae*te;
xx[5] = (-a-ae)*x[5]+a*x[3]+ae*te;
};
size_t nint = 5; /* no. of time step for ode solving */
double h=T/nint; /* time step for ode solving (T is the sampling time) */
ode_solver(rhs,x,nint,h); /* Runga Kutte solver */
}
```

Subsequently, we define a function `setBound` which is used to find set of abstract states whose output maps $H(x)$ are within the safety/target region defined over states of concrete system. The safe set used in thermalmodel6R example is given as:

```
/* defining safe set for the controller
ul[i] : upper bound on safe region for the temperature in the ith room
ll[i] : lower bound on safe region for the temperature in the ith room */
auto setBound = [](state_type y) -> bool {
  double ul=21.0;
  double ll=17.5;
  bool s = true;
  for(int j = 0; j < sDIM; j++){
    if( y[j] >= ul || y[j] <= ll ){
      s = false;
      break;
    }
  }
  return s;
}
```

The user also needs to provide information about system inputs. `QUEST` supports two types of input: continuous bounded input with lower and upper bound with quantization parameter and discrete set of inputs. The example thermalmodel6R uses continuous bounded inputs as shown below:

```
  /* lower bounds on inputs */
    double lb[sDIM]={0,0};
  /* upper bounds on inputs */
    double ub[sDIM]={1,1};
  /* quantization parameter */
    double eta[sDIM]={.5,1};
```

and the example thermalmodel10R considers discrete input set as

```
const size_t P = 3; /* Number of elements in input set*/
double ud[P][iDIM]={{0,0},{0,1},{1,0}};
```

Now to implement the construction of symbolic abstraction as discussed in Section 2.4, we use class `SymbolicSetSpace` which implements Algorithm 1 as

```
SymbolicSetSpace ss(ddmgr,P,N); /*for discrete input set*/
SymbolicSetSpace ss(ddmgr,iDIM,lb,ub,eta,N); /*for continuous input */
ss.addAbsStates();
```

Further, we obtain the set of abstract set whose output map satisfies ($\models$) the bounds for safe/target region by using method `getAbstractSet` in class `getAbstraction` to implement following algorithm:

Algorithm 2 Constrain set computation

Require: x_s, `system_post`, `setBounds`
1: **for all** $x \xrightarrow{u} x'$ **do**
2:     **if** $\xi_{x_s,x}(NT)$ obtained using `system_post` $\models$ `setBounds` **then**
3:         add corresponding BDD state to `BDD set`
4:     **else**
5:         discard it

and is used as

```
/* defining abstraction class */
getAbstraction<state_type> ab(&ss);
/* Computing the set of abstract state satisfying setBounds */
BDD set = ab.getAbstractSet(system_post,setBounds,xs);
```

The controller synthesis is carried out using fixed point computation using `class fixedPointMode` whose source code can be found in `./src/FixedPointMode.hh`. In particular, we use methods `FixedPointMode::safe`, `FixedPointMode::reach`, and `FixedPointMode::reachStay`, respectively. In thermalmodel6R example, We instantiate an object of `FixedPointMode` with the symbolic model S_2 given by a `SymbolicSetSpace` object `ab`

```
fixedPointMode fp(&ab);
```

and controller for reach and stay specification is synthesized using

```
BDD C;
/* controller for reach and stay specification */
C = fp.reachStay(set);
```

To run closed-loop simulation with the synthesized controller, we need to find an initial state in the abstraction whose output is ε-close to the concrete initial state x_0. We use the following code to compute initial abstract state and execute the closed-loop simulation.

```
/* finding initial abstract state in the abstraction*/
BDD w0 = ab.getAbsState(system_post,x0,xs,epsilon,sDIM);
/* closed-loop simulation */
ab.closedLoopSim(C,w0,system_post,x0,sDIM,t);
```

References

[1] D. Angeli, "A Lyapunov approach to incremental stability properties," *IEEE Transactions on Automatic Control*, vol. 47, no. 3, pp. 410–421, 2002.

[2] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 293–318, 1992.

[3] A. Girard and G. J. Pappas, "Approximation metrics for discrete and continuous systems," *IEEE Transactions on Automatic Control*, vol. 52, no. 5, pp. 782–798, 2007.

[4] E. Le Corronc, A. Girard, and G. Goessler, "Mode sequences as symbolic states in abstractions of incrementally stable switched systems," in *52nd Annual Conference on Decision and Control (CDC)*. IEEE, 2013, pp. 3225–3230.

[5] M. Rungger and M. Zamani, "SCOTS: A tool for the synthesis of symbolic controllers," in *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*. ACM, 2016, pp. 99–104.

[6] F. Somenzi, "CUDD: CU decision diagram package-release 2.4. 0," *University of Colorado at Boulder*, 2004.

[7] M. Zamani, A. Abate, and A. Girard, "Symbolic models for stochastic switched systems: A discretization and a discretization-free approach," *Automatica*, vol. 55, pp. 183–196, 2015.

[8] M. Zamani, I. Tkachev, and A. Abate, "Towards scalable synthesis of stochastic control systems," *Discrete Event Dynamic Systems*, pp. 1–29, 2016.