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 construction of symbolic which helps to resolve the issue of so-called curse of dimensionality while modelling 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 abstraction.

The tool is intended to be used and extended by researches in the area of formal methods for cyber-physical 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 of 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 $t > 0$. For the sake of completeness, we recall the definition of the incremental input-to-state stability [1]:
The nonlinear control systems (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:

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

2.2. **A Unified Framework for all Systems.** We present a notion of systems which serve as unified modelling framework for nonlinear control system (1) and its symbolic abstractions. A system is a tuple

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

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

The transition of S can be denoted as $x \xrightarrow{u} x'$ for a transition $(x, u, x') \in \longrightarrow$, where state x' is a u-successor (or simply successor) of state x, for some input $u \in U$.

2.3. **Approximate Bisimulation Relation.** We introduce the notion of approximate bisimulation [4] which is further used to provide symbolic abstraction which approximate bisimilar to the original system (1).
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) $\forall (x_1, x_2) \in \mathcal{R}$, $x_1 \xrightarrow[1]{u_1} x_1'$ in S_1 implies $x_2 \xrightarrow[2]{u_2} x_2'$ in S_2 satisfying $(x_1', x_2') \in \mathcal{R}$;
(iii) $\forall (x_1, x_2) \in \mathcal{R}$, $x_2 \xrightarrow[2]{u_2} x_2'$ in S_2 implies $x_1 \xrightarrow[1]{u_1} x_1'$ in S_1 satisfying $(x_1', x_2') \in \mathcal{R}$.

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

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

with $X_1 = \mathbb{R}^n$, $X_{10} \subseteq X_1$, $U_1 = \mathsf{U}$, the transition $x_1 \xrightarrow[1]{u} x_1'$ iff there exists a solution ξ of (1) under input $u \in \mathsf{U}$ satisfying $\xi(0) = x_1$ and $x_1' = \xi_{x_1,u}(\tau)$, $Y_1 = X_1$, $H = 1_x$.
QUEST computes symbolic models that are related via approximate bisimulation relation with S_1 as discussed in Subsection 2.3. For constructing symbolic abstraction of S_1 we use state-space quantization-free approach as discussed in [5, 3]. We assume that the set of inputs is finite (this can be easily achieved by quantization of input space) 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 finite input set with cardinality P, 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

- $X_2 = \overline{U}^N$, $X_{20} = X_2$, $U_2 = \overline{U}$, $Y_2 = Y_1$;
- $x_2 \xrightarrow[\rho]{u} x_2'$, 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 $u \in U_2$;
- $H_2(x_2) = \xi_{x_s,x_2}(N\tau)$.

Further, we synthesize a controller C that ensure the output of symbolic model S_2 within given specifications over S_1. Under the property of approximate bisimulation relation, one can compose the controller C designed for S_2 with S_1 to ensure given specifications.
For more details on construction of abstraction using state-space quantization-free approach, the interested readers may refer the results in [5], [3], and [8].

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 customize specifications such as reach and stay by cascading these two algorithms. The implementation of controller synthesis using fixed point computation is similar to the one used in SCOTS[6]. For more details on implementation of fixed point algorithms, we refer 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 guide 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 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 include 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

   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/
   ```

   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`

   ```
   #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
   ```

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 to the CUDD library [7]. It accepts the parameters temporal horizon N and cardinality of finite input space P as inputs. The class `SymbolicSetSpace` directly constructs the transition relations as

Algorithm 1 Computation of transition relation

```
Require: $N$, $P$
1: Let $x_2 = (u_1, u_2, \dots, u_N) \in \overline{U}^N$, $x_2' = (u_1', u_2', \dots, u_N') \in \overline{U}^N$ and $u \in \overline{U}$
2: for all $x_2$ and $u$ do
3:     for i=1 to N-1 do
4:         $u_i' = u_{i+1}$
5:     $u_N' = u$
```

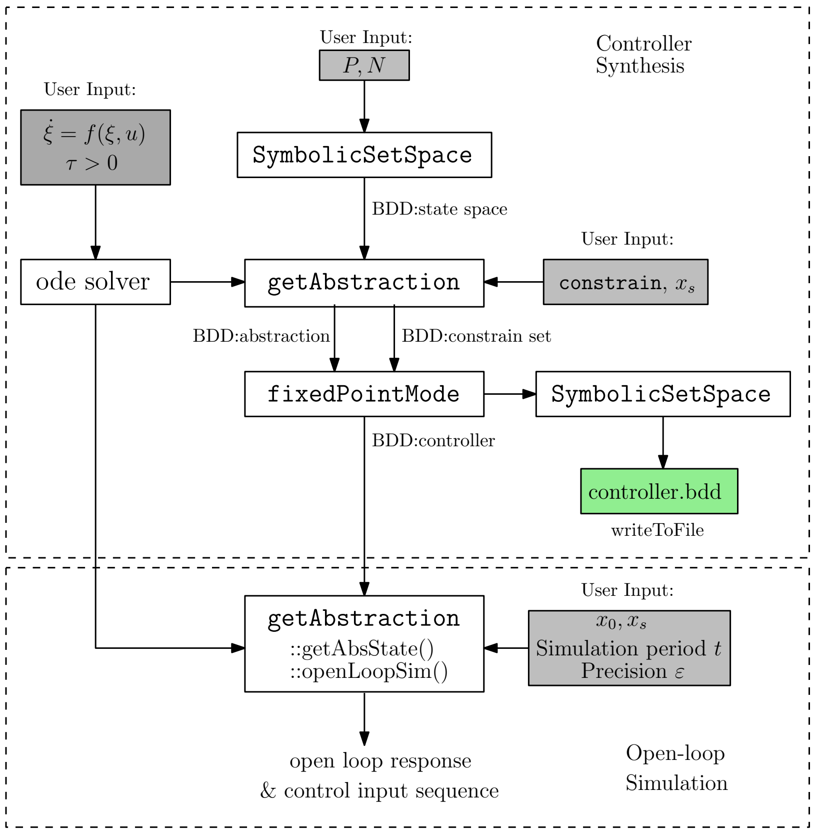

FIGURE 1. Workflow

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 required for overall operation of `QUEST`. Some of important functions are listed below:

```
getAbstraction::getConstrainSet() /* extracts set of states whose output map satisfies constrins */
getAbstraction::getOutput() /* get output map H_2(x_2) corresponding to state x_2 in abstraction*/
getAbstraction::getAbsState() /* get abstract state correponding to state in original system*/
getAbstraction::openLoopSim() /* open-loop simulation and printing output response */
```

6.3. `fixedPointMode`. The class implements fixed point computation for synthesis of controller. In particular, we use the methods `fixedPointMode::reach()`, `fixedPointMode::safe()` and `fixedPointMode::reachStay()` to synthesize controller 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 the definition of the dynamics of the system along with the mapping of abstract input (which is the one from finite input set of cardinality P) to corresponding system inputs. 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:

```
typedef std::array<double,6> state_type; /* state type */
auto system_post = [](state_type &x, int u) -> void {
/* Mapping abstract input to system inputs */
double ub[2]={0};
if(u==0)
{us[0]=0;us[1]=0;}
if(u==1)
{us[0]=0;us[1]=1;}
if(u==2)
{us[0]=1;us[1]=0;}

/* ode describing 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 an sampling time) */
ode_solver(rhs,x,nint,h); /* Runga Kutte solver */
}
```

Subsequently, we define constrain function which is used to find states in abstraction S_2 whose output map $H_2(x_2)$ are within the user defined constrains given as the boundaries over states of S_1. This constrains explicitly represents safety region for safety specification and target region for reachability/reach&stay specification, respectively. The example used in thermalmodel6R example is given as:

```
/* defining constrains for the controller
ul[i] : upper limit for the temperature in ith dimension
ll[i] : lower limit for the temperature in ith dimension */
auto constrain = [](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;
}
```

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

```
SymbolicSetSpace ss(ddmgr,P,N);
ss.addAbsStates();
```

Further, we obtain constrain set as using method `getConstrainSet` in class `getAbstraction` to implement following algorithm:

Algorithm 2 Constrain set computation

```
Require: $x_s$, system_post, constrain
1: for all $x_2 \xrightarrow{u} x_2'$ do
2:     if $\xi_{x_s,x_2}(NT)$ obtained using system_post $\models$ constrain 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 constrain set */
BDD set = ab.getConstrainSet(system_post,constrain,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 synthesis using

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

To run open-loop simulation with synthesized controller, we need to find initial state in abstraction whose output map $H_2(x_2)$ is in the ε precision to actual user defined initial state x_0. We use following code to compute initial point and execute open-loop simulation.

```
/* finding inital stae in abstraction*/
BDD w0 = ab.getAbsState(system_post,x0,xs,epsilon,sDIM);
/* open-loop simulation */
ab.openLoopSim(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, "Approximately bisimilar abstractions of incrementally stable finite or infinite dimensional systems," in *53rd Annual Conference on Decision and Control (CDC)*. IEEE, 2014, pp. 824–829.

[4] 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.

[5] 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.

[6] 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.

[7] F. Somenzi, “Cudd: Cu decision diagram package-release 2.4. 0,” *University of Colorado at Boulder*, 2004.

[8] 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.