SENSE– User Manual

Mahmoud Khaled

January 13, 2017

Contents

Part I
Introduction

1 About SENSE

SENSE is an open source software tool (available at http://www.hcs.ei.tum.de) for the abstraction and controller synthesis of networked control systems (NCS). The tool is mainly implemented in C++ and it comes with a two interfaces to access the synthesized controller. The first interface is for MATLAB and allows accessing the generated controller as well as running closed-loop simulations with the NCS controlled by the synthesized controller. The second interface is for OMNeT++ and allows accessing the synthesized controller as well as visualizing the closes loop simulation.

The tool provides base for further research on the field of modeling, abstractions and controller synthesis of NCS. It provides an extensible framework for which any class of NCS can be modeled and controllers can be synthesized. It is intended to be extended by researches in the area of formal methods for cyber-physical systems (CPS). The implementation of SENSEis based on *binary decision diagrams*(BDD) [**?**]. Operations on BDDs play the major role in order to construct symbolic models of NCS from original symbolic models of the plants within.

Currently, the tool provides support for a class of NCS where delays are assumed to be prolongable at both communication channels, system-to-controller and controller-to-actuators. However, SENSE can be extended with any class of NCS as long as it is defined how to handle the transitions inside the NCS and how to handle packet dropouts.

In this part, we give a quick overview on the basic concepts which will be used during the Manual. Concepts cover constructing symbolic abstractions of NCS and synthesizing symbolic controllers base don them.

2 Symbolic Models of Networked Control Systems (NCS)

Networked control systems (NCS) combine physical components, computing devices, and communication networks all in one system forming a complex and heterogeneous class of so-called cyber-physical systems (CPS). They gained lots of interest in the past decade because of their flexibility (especially when using wireless communications) which simplifies deployment and maintenance phases. Generally, the following components exist in any NCS:

1. a control system or a plant representing a physical process to be controlled;
2. two non-ideal communication channels, one transfers state information to controller and the other transfers control inputs to the plant; and
3. a remote digital controller enforcing some complex specifications over the plant by taking into account the imperfections of the communication channel.

2.1 Control Systems

A control system is a quadruple $\Sigma = (\mathbb{R}^n, \mathsf{U}, \mathscr{U}, f)$, where $\mathbb{R}^n$ is the state space; $\mathsf{U} \subseteq \mathbb{R}^m$ is a bounded input set; $\mathscr{U}$ is a subset of the set of all functions of time from $]a,b[\subseteq \mathbb{R}$ to U with $a < 0$ and $b > 0$; and $f : \mathbb{R}^n \times \mathsf{U} \to \mathbb{R}^n$ is a continuous map satisfying the Lipschitz continuity assumption: for each compact set $Q \subseteq \mathbb{R}^n$, there exists a constant $L \in \mathbb{R}^+$ such that $\|f(x,u) - f(y,u)\| \leq L\|x - y\|$ for all $x, y \in Q$ and all $u \in \mathsf{U}$. We also define a trajectory of Σ by the locally absolutely continuous curve $\xi : \,]a,b[\to \mathbb{R}^n$ if there exist a $v \in \mathscr{U}$ that satisfies:

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

almost at any $t \in]a,b[$. We define $\xi : [0,t] \to \mathbb{R}^n$ for trajectories over closed intervals with the understanding that there exists a trajectory $\xi' :]a,b[\to \mathbb{R}^n$ for which $\xi = \xi'|_{[0,t]}$ with $a < 0$ and $b > t$. We denote by $\xi_{xv}(t)$ the point reached at time t under the input v and with the initial condition $x = \xi_{xv}(0)$. Such point is uniquely determined based on the property of f that ensures existence and uniqueness of trajectories. We say the system Σ is *forward complete* when all the trajectories are defined on intervals of the form $]a,\infty[$. From now on, we consider forward complete control systems.

2.2 Systems

We present a notion of systems which allows describing plants, NCS, their symbolic abstractions as well as controllers. 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 set of internal variables V; a transition relation $\longrightarrow \subseteq X \times U \times X$; a set of outputs Y; and an output map H which is assumed to be strict and of the form $H : X \times U \to 2^Y$. All sets are assumed to be non-empty. The system is said to be simple, if $Y = X$, $V = U$, all states are allowed to be initial states (i.e. $X_0 = X$ is acceptable) and $H = 1_X$. From now on, we consider only simple systems and we denote them by the following simplified version of system representation:

$$S = (X, X_0, U, \longrightarrow). \tag{2}$$

We denote by $\mathscr{T}(U,X)$ the set of all simple systems associated to a set of inputs U and a set of states X.

We also introduce the sampled system

$$S_\tau(\Sigma) = (X_\tau, X_{\tau,0}, U_\tau, \underset{\tau}{\longrightarrow}) \tag{3}$$

as the simple system that encapsulates all the information contained in the control system Σ at the sampling times $k\tau$, where $k \in \mathbb{N}_0$, $X_\tau = \mathbb{R}^n$, $X_{\tau,0} = \mathbb{R}^n$, and U_τ is a set of piece-wise constant curves over intervals of length τ and it is defined as:

$$U_\tau = \{v : \mathbb{R}_0^+ \to \mathsf{U} | v(t) = v((s-1)\tau), t \in [(s-1)\tau, s\tau[, s \in \mathbb{N}\}.$$

A transition $x_\tau \overset{v_\tau}{\underset{\tau}{\longrightarrow}} x'_\tau$ in the sampled system $S_\tau(\Sigma)$ is considered if and only if there exists a trajectory $\xi_{x_\tau v_\tau} : [0,\tau] \to \mathbb{R}^n$ in the system Σ such that $\xi_{x_\tau v_\tau}(\tau) = x'_\tau$. The

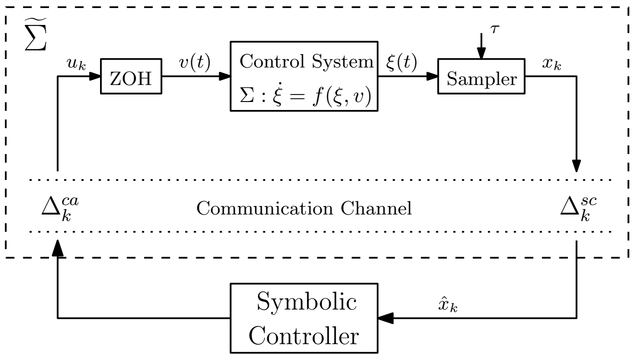

Figure 1: Diagram of the NCS $\widetilde{\Sigma}$ and the symbolic controller.

sampled system $S_\tau(\Sigma)$ is deterministic since any trajectory of the forward complete control system Σ is uniquely determined. However, its states are uncountable and therefore it is not symbolic (a system is said to be symbolic when X and U are both finite sets).

2.3 Abstraction and Controller Synthesis

Given a plant and a set of complex logic specifications, the goal is to design a controller that enforces the given specification on the plant. We need first to compute an abstraction of the plant, which is usually given in the form of a finite state machine. Then we employ some standard fixed-point algorithms that result in a finite-state controller (symbolic controller) that is able to enforce the specifications over the abstract system. Finally, the controller is refined using a suitable interface to be able to enforce the specifications against the original plant. We denote by $S_q(\Sigma)$ the abstraction of the sampled system $S_\tau(\Sigma)$ as long as there exists a feedback refinement relation (FRR) Q relating the sampled syystem to its abstraction. We refer the interested readers to [3] on how such an abstraction is derived with feedback refinement relations.

2.4 Networked Control Systems

As depicted in Fig.1, the NCS $\widetilde{\Sigma}$ includes a control system $\Sigma = (\mathbb{R}^n, \mathrm{U}, \mathcal{U}, f)$ followed by a sensor/sampler working in a time-driven fashion with a sampling-time τ (i.e. at times $s_k := k\tau$, $k \in \mathbb{N}_0$). Additionally, an event-driven (i.e. responds instantaneously to newly arriving data) zero-order-hold (ZOH) helps sustaining an inter-sampling continuous-time control input $v(t)$ extracted from the discrete-time control input u_k. We denote by $x_k := \xi(s_k)$ the state measured at time s_k.

A communication network connects the control system with the controller and introduces time-varying sensor-to-controller and controller-to-actuator delays (Δ_k^{sc}

and Δ_k^{ca}). We include packet dropouts in both channels using an emulation technique (i.e. increasing the delays) as discussed in [1] while assuming that the maximum number of subsequent dropouts is bounded. We consider the delays to be bounded and to take the form of integer multiples of the sampling time τ, meaning $\Delta_k^{sc} := N_k^{sc}\tau$ and $\Delta_k^{ca} := N_k^{ca}\tau$, where $N_k^{sc} \in [N_{\min}^{sc}; N_{\max}^{sc}]$, $N_k^{ca} \in [N_{\min}^{ca}; N_{\max}^{ca}]$, and $N_{\min}^{sc}, N_{\max}^{sc}, N_{\min}^{ca}, N_{\max}^{ca} \in \mathbb{N}$.

Based on the function of the ZOH, the discrete-time input u_k, $k \in \mathbb{N}_0$, is transformed to the continuous-time input $v(t) = u_{k^*(t)}$, where $k^*(t) := \max\{k \in \mathbb{N}_0 | s_k + \Delta_k^{ca} \leq t\}$. Within a sampling interval $[s_k, s_{k+1}[$, $v(t)$ is described explicitly by $v(t) = u_{k+j*-N_{\max}^{ca}}$, where $t \in [s_k, s_{k+1}]$ and $j* \in [0; N_{\max}^{ca} - N_{\min}^{ca}]$ is a time-index shifting used to determine the correct control input by taking care of message rejection due to out of order packet arrival. The time-shifting index $j*$ is defined as:

$$j* = N_{\max}^{ca} - \min\{i \in [N_{\min}^{ca}; N_{\max}^{ca}] \mid \widehat{N}_i \leq i\}, \tag{4}$$

where $\widehat{N}_i$ for any $i \in [N_{\min}^{ca}; N_{\max}^{ca}]$ is the delay which the packet sent i-samples beforehand had experienced (i.e. $\widehat{N}_i = N_{k-i}^{ca}$).

We consider the same situation for the channel between the sampler and the controller. We denote by $\widehat{x}_k$ the input to the controller at the sampling times $s_k := k\tau$. Then we have $\widehat{x}_k = x_{k+l*-N_{\max}^{sc}}$, where $l* \in [0; N_{\max}^{sc} - N_{\min}^{sc}]$ is defined as:

$$l* = N_{\max}^{sc} - \min\{i \in [N_{\min}^{sc}; N_{\max}^{sc}] | \widetilde{N}_i \leq i\} \tag{5}$$

where $\widetilde{N}_i$ for any $i \in [N_{\min}^{sc}; N_{\max}^{sc}]$ is the delay which the packet sent i-samples beforehand had experienced (i.e. $\widetilde{N}_i = N_{k-i}^{sc}$).

2.5 Symbolic Controllers

A formal definition for symbolic controllers is given in [3]. In this context, we can consider the symbolic controller as a finite system whose input and output are the sampled-state $x_k \in \mathbb{R}^n$ and control input $u_k \in \mathsf{U}$ respectively. The control input u_k is fed to the ZOH and then to the control system Σ in order to satisfy a given complex specification (e.g. LTL-based specifications). Such specifications may require in general dynamic controllers (i.e. controller with memory) and, therefore, we assume that the symbolic controller may be dynamic.

2.6 NCS as Systems

Based on all previously presented definitions, we describe NCS as systems. We consider bounded delays for the communication channel between the sampler and the controller represented by the $N_{\max}^{sc}\tau$ and $N_{\min}^{sc}\tau$. We assume the same for the channel between the controller and the actuator: $N_{\max}^{ca}\tau$ and $N_{\min}^{ca}\tau$.

By representing the plant Σ via the sampled system $S_\tau(\Sigma) = (X_\tau, X_{\tau,0}, U_\tau, \xrightarrow[\tau]{})$, the NCS $\widetilde{\Sigma}$, including the aforementioned non-idealities, can be represented with the system $S(\widetilde{\Sigma})$ defined by the map

$$\mathscr{L} : \mathscr{T}(U, X) \times \mathbb{N}^4 \to \mathscr{T}(U, X) \tag{6}$$

as the following:

$$\begin{aligned} S(\widetilde{\Sigma}) &= (X, X_0, U_\tau, \longrightarrow) \\ &= \mathscr{L}(S_\tau(\Sigma), N^{sc}_{\min}, N^{sc}_{\max}, N^{ca}_{\min}, N^{ca}_{\max}), \end{aligned} \tag{7}$$

where

- $X = \{X_\tau \cup q\}^{N^{sc}_{\max}} \times U_\tau^{N^{ca}_{\max}} \times [N^{sc}_{\min}; N^{sc}_{\max}]^{N^{sc}_{\max}} \times [N^{ca}_{\min}; N^{ca}_{\max}]^{N^{ca}_{\max}}$, where q is a dummy symbol;
- $X_0 = \{(x_0, q, \ldots, q, u_0, \ldots, u_0, N^{sc}_{\max}, \ldots, N^{sc}_{\max}, N^{ca}_{\max}, \ldots, N^{ca}_{\max})$, where $x_0 \in X_{\tau,0}$ and $u_0 \in U_\tau\}$;
- $(x_1, \ldots, x_{N^{sc}_{\max}}, u_1, \ldots, u_{N^{ca}_{\max}}, \widetilde{N}_1, \ldots, \widetilde{N}_{N^{sc}_{\max}}, \widehat{N}_1, \ldots, \widehat{N}_{N^{ca}_{\max}}) \xrightarrow{u}$ $(x', x_1, \ldots, x_{N^{sc}_{\max}-1}, u, u_1, \ldots, u_{N^{ca}_{\max}-1}, \widetilde{N}, \widetilde{N}_1, \ldots, \widetilde{N}_{N^{sc}_{\max}-1}, \widehat{N}, \widehat{N}_1, \ldots, \widehat{N}_{N^{ca}_{\max}-1})$ for all $\widetilde{N} \in [N^{sc}_{\min}; N^{sc}_{\max}]$ and for all $\widehat{N} \in [N^{ca}_{\max}; N^{ca}_{\max}]$ if there exist a transition $x_1 \xrightarrow[\tau]{u_{N^{ca}_{\max}-j*}} x'$, where $j*$ is defined in (4).

One can readily verify that the system $S(\widetilde{\Sigma})$ is infinite because the system $S_\tau(\Sigma)$ is infinite.

3 Symbolic Abstractions of Networked Control Systems

In this section, we provide a symbolic model for the NCS. To do so, we raise a supplementary assumption that the set U in tuple Σ is finite and $U_q = \mathsf{U}$, where U_q is the set of inputs in $S_q(\Sigma)$.

Given the symbolic abstraction $S_q(\Sigma)$ of the control system Σ, we can systematically construct the symbolic abstraction $S_*(\widetilde{\Sigma})$ for the NCS using the bounds on the different delays in the network, which is given by:

$$S_*(\widetilde{\Sigma}) = \mathscr{L}(S_q(\Sigma), N^{sc}_{\min}, N^{sc}_{\max}, N^{ca}_{\min}, N^{ca}_{\max}). \tag{8}$$

We rely on the following theorem for establishing a feedback refinement relation between the two systems $S(\widetilde{\Sigma})$ and $S_*(\widetilde{\Sigma})$ providing that the two systems $S_\tau(\Sigma)$ and $S_q(\Sigma)$ are in a feedback refinement relation.

Theorem 3.1. *[2] Consider a NCS $\widetilde{\Sigma}$ and suppose there exists a simple system $S_q(\Sigma)$ such that $S_\tau(\Sigma) \preccurlyeq_Q S_q(\Sigma)$. Then we have $S(\widetilde{\Sigma}) \preccurlyeq_{\widetilde{Q}} S_*(\widetilde{\Sigma})$, for some feedback refinement relation $\widetilde{Q}$.*

Consider the normal methodology of deriving a symbolic abstraction $S_*(\widetilde{\Sigma})$ of a NCS $\widetilde{\Sigma}$ where one should first derive a system $S(\widetilde{\Sigma})$ that captures all NCS information then construct a symbolic abstraction from it. Now, based on the proposed result, one can systematically construct the symbolic abstraction $S_*(\widetilde{\Sigma})$ directly from

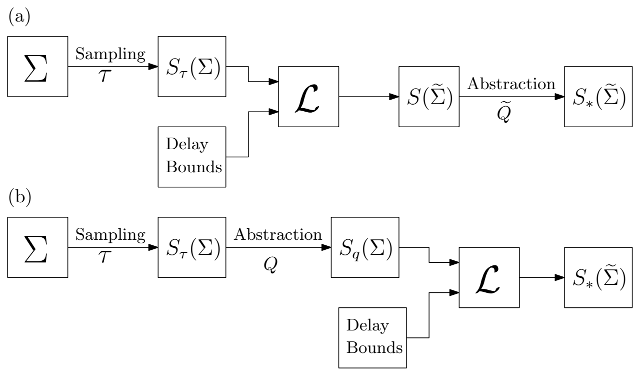

Figure 2: Two methodologies for the construction of $S_*(\widetilde{\Sigma})$ using feedback refinement relations.

the symbolic abstraction $S_q(\Sigma)$ obtained exclusively for the plant Σ. Therefore, rather than deriving an abstraction for $S(\widetilde{\Sigma})$ from scratch, we simply need to derive the abstraction $S_q(\Sigma)$ from the system $S_\tau(\Sigma)$ (which is a simpler procedure) and then apply the map $\mathscr{L}$ defined in (6) to derive the symbolic model of $S(\widetilde{\Sigma})$. Figure 2 distinguishes between the normal and proposed methodologies for the construction of the symbolic abstraction $S_*(\widetilde{\Sigma})$ for the NCS $\widetilde{\Sigma}$. The tool SENSE then relies on this concept to construct symbolic models of NCS directly from those derived for the plants within the NCS.

3.1 Specifications and Control

The desired behavior of the closed loop is defined with respect to the τ-sampled behavior of the continuous-time systems **??**. A *specification* ψ_τ for a simple system $S_\tau = (X_\tau, X_{\tau,0}, U_\tau, \xrightarrow[\tau]{})$ is simply a set

$$\psi_\tau \subseteq \bigcup_{T\in\mathbb{Z}_{\geq 0}\cup\{\infty\}} (U_\tau \times X_\tau)^{[0;T[} =: (U_\tau \times X_\tau)^\infty \tag{9}$$

of possibly finite and infinite input-state sequences. A simple system S_τ together with a specification ψ_τ constitute an *control problem* (S_τ, ψ_τ). `SENSE` supports invariance (often referred to as safety) and reachability specifications. `SENSE` expects (safe)targets sets for the (invariance)reachability as BDDs. the tool `SCOTS` can be used to define arbitrary sets as atomic propositions for this purpose. It provides customized commands to define polytopes and ellipsoids over the state space of the control system as atomic propositions. Given a simple system S_τ representing and a specification ψ_τ for S_τ, the control problem (S_τ, Σ_τ) is not solved directly, but an auxiliary, finite control problem (S_q, ψ_q) is used in the synthesis process. Here, $S_q = (X_q, X_{q,0}, U_1, \xrightarrow[q]{})$ is a *symbolic model* or (discrete) *abstraction* of S_τ and ψ_q is an abstract specification. According to the used feedback-refinement relation, the state alphabet of X_q is a cover of X_τ and the input alphabet U_q is a subset of U_τ. The set X_q contains a subset $\bar{X}_q$, representing the "real" quantizer symbols, while the remaining symbols $X_q \setminus \bar{X}_q$ are interpreted as "overflow" symbols. The tool SCOTS should be used to constrfuct the symbolic abstraction of the sampled plant inside the NCS. Such symbolic model is also given in form of a BDD. It computes symbolic models that are related via feedback refinement relations with the plant. Please refer to the manual of SCOTS for more details. SENSEshould also be provided with the delay bounds of both channels in the NCS. Together with the supplied symbolic model of the plant, SENSE can systematically construct symbolic models of NCS as follows:

$$S_*(\widetilde{\Sigma}) = \mathscr{L}(S_q(\Sigma), N^{sc}_{\min}, N^{sc}_{\max}, N^{ca}_{\min}, N^{ca}_{\max}).$$

For the solution of the auxiliary control problems $(S_*(\widetilde{\Sigma}), \psi_2)$ SENSEprovides minimal and maximal fixed point algorithms.

3.2 Controller Synthesis via Fixed Point Computations

For the synthesis of controllers C to enforce reachability, respectively, invariance specifications, SENSE provides fixed point algorithms similar to SCOTS. Please refer to the manual of SCOTS for further details about provided fixed point operations.

3.3 Prolonged-Delay Networked Control Systems

The main subtlety for symbolic control of NCS is in the refinement of the constructed discrete controller. It needs to enforce some specification over the output of the plant inside the NCS. This requires the whole state tuple x_* of $S_*(\widetilde{\Sigma})$ while only one of the elements of the tuple is available based on the packet arrived before the controller. We elaborate on the refinement of symbolic controllers in the this subsection by proposing a class of NCS in which the whole state tuple x_* of $S_*(\widetilde{\Sigma})$ can be recovered inside the controllers.

In order to refine the synthesized symbolic controllers, we target a class of NCS where the upper and lower bounds of the delays are equal at each channel. This implies that all packets suffer the same delay, i.e. $\widetilde{N} = N^{\mathrm{sc}}_{\max}$ and $\widehat{N} = N^{\mathrm{ca}}_{\max}$ for any possible sensor-to-controller delay $\widetilde{N}$ and any possible controller-to-actuator delay $\widehat{N}$. This can be readily achieved by performing extra prolongation (if needed) of the delays suffered already by the packets. For the sensor-to-controller channel, this can be readily done inside the controller. The controller needs to have a buffer to hold arriving packets and keep them in the buffer until their delays reach the maximum. For the controller-to-actuator channel, the same needs to be implemented inside the ZOH. Therefore, in this setting, state (resp. input) packets are allowed to have any delay (not necessarily integer multiples of the sampling time) between 0 and $N^{\mathrm{sc}}_{\max}$ (resp. $N^{\mathrm{ca}}_{\max}$) where $N^{\mathrm{sc}}_{\max}$ and $N^{\mathrm{ca}}_{\max}$ are integer multiples of the sampling time.

In this class of NCS, the information contained in the NCS $\widetilde{\Sigma}$ is captured by the metric system $S'(\widetilde{\Sigma}) := \mathscr{L}(S_\tau(\Sigma), N^{\mathrm{sc}}_{\max}, N^{\mathrm{sc}}_{\max}, N^{\mathrm{ca}}_{\max}, N^{\mathrm{ca}}_{\max})$. We also denote by $S'_*(\widetilde{\Sigma}) := \mathscr{L}(S_{\mathrm{q}}(\Sigma), N^{\mathrm{sc}}_{\max}, N^{\mathrm{sc}}_{\max}, N^{\mathrm{ca}}_{\max}, N^{\mathrm{ca}}_{\max})$ the corresponding symbolic model of $S'(\widetilde{\Sigma})$.

Part II

Getting Started with SENSE

4 Source Code Organization

```
./doc                  /* the documentation and manual of \SENSE */
./interface/MATLAB     /* the source code for which MATLAB can access the synthesized controllers
./interface/OMNetpp    /* the source code for which OMNet++ can access the synthesized controlle
./examples/FIFO        /* examples presented at for the prolonged delay class of NCS. */
./src                  /* the source code of the \SENSE */
./tools                /* some tools that can help analyisng symbolic models and controllers*/
./tools/bdddump        /* a tool to dump the informations inside a BDD file comming from SCOTS*/
./tools/bdd2fsm        /* a tool to export a CSV file for visualizing the states transitions mode
./contCoverage         /* a tool to print the controller coverage for 1-D and 2-D state dimensio
./utils                /* some helper code */
./license.txt          /* the license file */
./readme.txt           /* a quick description pointing to this manual */
```

5 Installation

In general, SENSEis implemented in "header-only" style and you only need a working C++ developer environment. The CUDD library by Fabio Somenzi is the base of all operations done by SENSE. It which can be downloaded at http://vlsi.colorado.edu/~fabio/. Moreover, for closed loop simulation/visualization, a working installation of MATLAB or OMNeT++ is needed.

5.1 Requirements

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: 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
```

3. Install SCOTS in the same folder with SENSE (not inside the SENSE folder). You will need SCOTS to construct the symbolic models of the plants inside NCS as well as defining atomic propositions for synthesis in SENSE.

4. A recent version of MATLAB. We conducted the experiments with versions `R2015b/R2016a/R2016b 64-bit (maci64/ubuntu/Windows10)`. Please note that for Ubuntu and Windows10, MATLAB supports gcc/g++ compilers with versions up to 4.9. Therefore, if your machine has a newer version, you have to install older versions and make them the default compilers. To compile the mex files:

   (a) open MATLAB and setup the mex compiler by

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

   (b) edit the `interface/MATLAB/Makefile` and adjust the variables: `MATLABROOT` and `CUDDPATH`

   (c) in a terminal run
   `$ make`

5. We provide interface for OMNeT++ 5.0. You can simply load the interface from inside OMNeT++ by importing the path `/interface/OMNetpp as a project from the eclipse environment.`

The folder `/examples/FIFO/` provides several examples related to the prolonged-delay class of NCS.

6 Running a sample example

For a quickstart

1. go to one of the examples in

```
./examples/FIFO/dint      /* Double-integrator example with reachability specification */
./examples/FIFO/robot     /* Robot example with infinitily-often specification */
./examples/FIFO/vehicle   /* Vehicle dynamics with reach/avoid specifications */
```

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. you will find a folder named `scots-files` containing the source codes used to generate the symbolic model of the plant inside the NCS using SCOTS. Additionally, the required atomic propositions are also generated using the source code in this director. Make sure to modify the `Makefile` like you did in the previous step if you plan to modify/compile/run the files in this directory.

5. compile and run the executable, for example in `/examples/FIFO/robot` run

```
$ make
$ ./robot_ncs
```

6. open MATLAB (here we assume that you have already compiled the mex files) and added `./interface/MATLAB` to the MATLAB path.

7. run the m file, for example in `./examples/FIFO/robot` run

```
>> robot_ncs.m
```

8. modify the example to your needs

7 Work Flow Overview

A short description of the general work flow is given by

1. Use SCOTS to construct symbolic abstraction for the plant inside the NCS. Save your constructed symbolic transition relation as a .bdd file from SCOTS.

2. Based on your targeted specification, use SCOTS to generate atomic propositions over the state space of the plant inside the NCS and save them as .bdd files.

3. You have to decide which class of NCS you will target. For example, when targeting the prolonged-delay NCS, you will use the class: `ncsFIFOTransitionRelation`.

4. Initiate your `ncsTransitionRelation` instance with the following:

   - the .bdd file from `SCOTS` representing the transition relation of the plant within the NCS.
   - the delay bounds suitable for the targeted class of NCS.
   - the dimensions of the plant (used to verify against what is stored in the provided .bdd file).

5. Once you initialize the class and the original transition relation is loaded, call the method `expand()` to construct the expanded transition relation. This might take a while as the BDD manager is working with BDD objects directly to construct such expanded relation.

6. Then, you can save the constructed abstraction of NCS for later processing (i.e. synthesizing different controllers).

7. Use the provided methods `computeReachController` and `computeSafetyController` to synthesize reachability or safety controllers respectively. You will have to provide the .bdd files of the atomic propositions according to different specification requirements. For special specifications, you can work directly by using the provided Fixed-Point operations.

8. Once the controller is synthesized correctly, save it by using the method: `writeToFile`.

9. Now, you have many options:

   - Analise the outputs using the tools from the folder `./tools`.
   - Initiate a closed-loop simulation using `MATLAB`.
   - Initiate a closed-loop simulation with visualization using `OMNeT++`.

An overview of the usage is illustrated in Fig. 7.

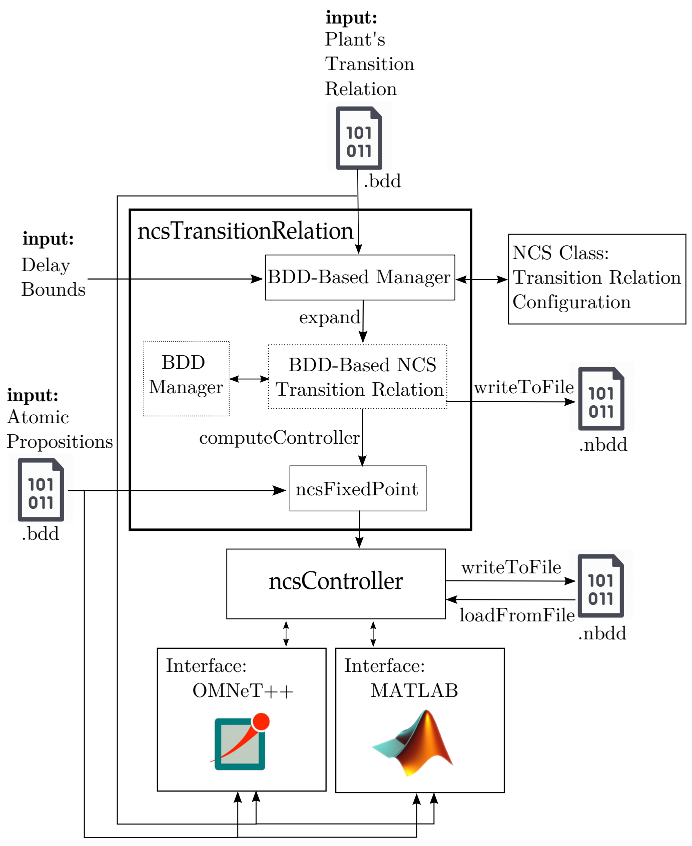

Figure 3: Diagram of the NCS $\widetilde{\Sigma}$ and the symbolic controller.

8 SENSE C++ Classes Implementation

8.1 NCS Transition Relations

The C++ class `ncsTransitionRelation` serves as a parent class to encapsulates and manage information about any transition relation of the NCS. It handles the loading/saving of the NCS-based transition relations. It also handles the construction flow of the expanded NCS transition relation. Such class need to inherited by further classes which provide further specific details about constructions of the NCS transition relation. We provide one derived class as an example which is the derived class `ncsFIFOTransitionRelation`. There are two options to initiate a `ncsTransition` object:

1. Using the transition relation of the plant inside the NCS. Here you use an existing symbolic model of the plant inside the NCS. The following is the constructor corresponding to such instantiation:

   ```
   ncsTransitionRelation::ncsTransitionRelation(cuddManager,
       BDD_INPUT_FILE, SS_DIM, IS_DIM, NSCMIN, NSCMAX, NCAMIN, NCAMAX);
   ```

   You provide your BDD manager and the SCOTS-generated .BDD file representing the transition relation of the plant inside the NCS. You have to also provide the dimensions of the state and input sets of the original plant. The latter two are used to verify against the stored information inside the .BDD file as well as for unpacking the provided BDD object. Additionally you have also to pass information about the delays inside the network.

2. Using a stored NCS transition relation. In this case, you already had a NCS-transition relation storred inside a .NBDD file and you wand to load it for further use like conbtroller synthesis. Here you can use the following constructor:

   ```
   ncsTransitionRelation::ncsTransitionRelation(Cudd_Manager,
       NBDD_INPUT_FILE);
   ```

8.2 Adding new NCS classes

The structure of SENSEis straightforward. The C++ class `ncsTransitionRelation` serves as a parent class for all types of NCS. The class handles the construction flow of the expanded NCS transition relation. Users willing to extend SENSEwith new NCS classes need to inherit this C++ class. Derived classes of this base class will just provide information about how the NCS transition relation is constructed (e.g. how packets get reordered or provide insights about how packet-dropouts happen).

Therefore, users of SENSE do not instantiate objects of `ncsTransitionRelation` directly. They use existing derived classes or develop their own. SENSE is highly extensible with additional NCS classes. All derived C++ classes shall then target their instantiations to those two options of instantiation. Such technique allows for further extensions of SENSE to easily support other classes of NCS. Any derived class should ovveride the following methods:

```
ncsState* getSourceStateTemplate(); /* which tells the parent of the structure of the NCS node*/
BDD ConstructQTransitionRules();    /* Provide rules about transitions that include dropped packets*/
BDD ConstructNormalTransRules();    /* Provide rules about normal transitions */
```

The `ncsTransitionRelation` class offers the following utility methods:

```
size_t getStateVarsCount()       /* provides the count of BDD-Vars for the NCS-state */
size_t getInpVarsCount()         /* provides the count of BDD-Vars for the NCS-input */
size_t getVarsCount()            /* provides the count of all NCS BDD-Vars */
vector<size_t> getSsVarsCount() /* provides the count of BDD-Vars for the states of original plant */
vector<size_t> getIsVarsCount() /* provides the count of BDD-Vars for the inputs to original plant */
BDD getOriginalRelation()       /* provides a BDD function of the original plant's transition relation */
BDD getTransitionRelation()     /* provides a BDD function of the NCS transition relation */
vector<size_t> getPreVars()     /* provides the indicies of BDD-Vars for pre-state of the NCS */
vector<size_t> getPostVars()    /* provides the indicies of BDD-Vars for post-state of the NCS */
vector<size_t> getOrginalVars() /* provides the indicies of BDD-Vars for input of the NCS */
double getExpandTime()          /* provides the time consumed to compute the NCs abstraction */
void WriteToFile(string)        /* saves the NCS transition relation to a .NBDD file*/
```

8.3 Prolonged-delay NCS Transition Relations

The C++ class `ncsFIFOTransitionRelation` provides specific construction for the class prolonged-NCS class of NCS. It derives the class `ncsTransitionRelation`. It need to be instantiated with sufficient information about the transition relation of the plant inside the NCS, yet the same as `ncsTransitionRelation`. When it is instansiated with the original symbolic model of th plant inside the NCS, the user should then call the inherited method `ExpandBDD()` to start the construction process of the NCS-based transition relation using the provided delay information. Here, the parent C++ class `ncsTransitionRelation` starts the construction by using the provided rules from the child C++ class. The following code shows example for constructing the NCS transition relation using the original symbolic model of the plant. The complete example can be found in the examples directory at: `/examples/FIFO/vehicle_half3`:

```
#define VERBOSEBASIC
#include "SENSE.hh"
#define ssDIM 3  /* state-space / input-space dimensions */
#define isDIM 2
#define NSCMAX 2 /* NCS Delay bounds */
#define NCAMAX 2
#define FILE_BDD_REL "scots-files/vehicle_rel.bdd"
#define FILE_NBDD_REL "vehicle_rel.nbdd"
void main() {
  Cudd cuddManager;
  ncsFIFOTransitionRelation ncsVehicle(cuddManager, FILE_BDD_REL, ssDIM, isDIM, NSCMAX, NCAMAX);
  ncsVehicle.ExpandBDD();
  cout << "NCS relation expanded in " << ncsVehicle.getExpandTime() << " seconds !" << endl;
  ncsVehicle.WriteToFile(FILE_NBDD_REL);
}
```

Users willing to extend SENSE with new NCS classes might find the implementation of the class `ncsFIFOTransitionRelation` worth reading.

8.4 NCS Controllers

The C++ class `ncsController` serves as a container for the information about any controller synthesized for NCS. It handles the loading/saving of the NCS-based controllers. It also provides an interface to access the synthesized controller. There two options to initiate a `ncsController` object:

1. Using the expanded NCS transition relation. After expanding your transition relation, you use it to synthesize a controller. The result is returned as an object of an `ncsController` class. Here is an example to load a previously expanded NCS transition relation and then synthesizing a Reachability controller for it:

```
#include "SENSE.hh"
#define FILE_BDD_TS "scots-files/vehicle_ts.bdd"
#define FILE_NBDD_REL "vehicle_rel.nbdd"
#define FILE_NBDD_CONTR "vehicle_contr.nbdd"
void main() {
  Cudd cuddManager;
  ncsFIFOTransitionRelation ncsVehicle(cuddManager, FILE_NBDD_REL);
  ncsController ncsContr = ncsVehicle.ComputeReachController(FILE_BDD_TS);
  ncsContr.WriteToFile(FILE_NBDD_CONTR);
}
```

2. Using a stored NCS controller. In this case, you already had a synthesized NCS-controller stored inside a .NBDD file and you wand to load it for further use like simulating a closed loop. Here you can use the following constructor:

```
ncsController::ncsController(Cudd_Manager, INPUT_FILE);
```

The class offers the following utility methods:

```
getControllerComputeTile() /* provides the time consumed to compute the controller */
BDD getBDD() /* provides a BDD function represinting the controller */
void WriteToFile(char*) /* saves the NCS transition relation to a .NBDD file*/
```

Part III
Simulation and Visualization Interfaces

SENSE provides two interfaces to access the synthesized controllers. The first interface is for MATLAB while the second interface is for OMNeT++. For both interfaces, we provide the following support:

- Accessing the synthesized NCS controllers from both tools.
- Software classes to simulate the dynamical plant within the NCS as well as the communication channels.
- Software classes to encapsulate the synthesized controller for closed-loop simulation including NCS state reconstruction classes required for symbolic controllers' refinement.
- Examples to simulate the closed loop behavior and visualize the results.

Closed-loop simulations can be initiated and managed using and from within both tools (MATLAB/OMNeT++). Simulations are performed in the following manner:

1. The tool simulates the plant's behavior, e.g. using an ODE solver that runs the dynamics of the plant using its current state and the input provided to it. Such ODE is already available in MATLAB. We provide identical ODE solver for OMNeT++ given inform of a Software class. Symbolic or discrete plants can be simulated directly by software.
2. The passes the generated state/output of the plant to the communication channel between the plant and the controller. For MATLAB, we provide support for a prolonged-delay communication channel in the form of FIFO channel. For OMNeT++, we provide general random-delay communication channels for more realistic simulations. We also provide a prolongation software class casting such channels to FIFO channels by performing adding extra delay to packets sothat all packets get the same delay and hence, it reduces to a FIFO channel.
3. Packets travailing from plant arrive at controller's side. Since the symbolic controller is synthesized against the NCS, it can handle only NCS states. Therefore, we provide a software class to reconstruct the arriving packets into the form of a NCS-state that the controller can handle.
4. The controller uses the reconstructed packets and access the synthesized NCS controller asking for the suggested inputs it can apply. The controller then have to select one of the provided inputs, while being sure that any input is guarenteed to achieve the specifications which the controller was designed to

force. The controller passes the control action to the second communication channel, the controller-to-actuator channel.

5. The same applies to the controller-to-actuator channel. It uses the same communication classes in `MATLAB/OMNeT++`, with the delay-prolongation classes available for use.

6. Once the control-action arrives at the plant, this is considered one closed-loop iteration. The tool might consider as many simulation loops as the user wishes. Simulation termination can also dependent on reaching a specific target or after specific time.

9 Simulation using MATLAB

9.1 Requirements and Preliminary Instructions

Consider the following before using the `MATLAB`interface:

1. Having a C/C++ compiler that is compatible with `MATLAB`'s `mex` compiler.

2. Prepare tour `mex` compiler from within `MATLAB` using the following command:
   `» mex -setup <lang>`
   replacing `<lang>` with the targeted programming language (i.e. C or C++).

3. Make sure to compile the provided C++ classes for `MATLAB` using the `mex` compiler. This can be done using any `Terminal` by navigating the the `MATLAB`'s interface folder (`./interface/MATLAB/`), editing the `Makefile` to change the `MATLAB`'s installation path and building using `make` command.

4. From inside `MATLAB`, users need to add the interface folder to the Paths of `MATLAB`.

5. Any simulation will be initiated and managed from within `MATLAB` with the help of the provided interfaces.

The provided simulation interface is an extensible base for the simulation of any class of NCS. We provide information related to the prolonged-delay class. We give instructions on how to extend it to other classes of NCS.

Here, we assume the user has already constructed an expanded NCS symbolic model and used it to synthesize a symbolic controller. The symbolic controller need to be saved as (.nbdd) file to a location known to the user. The user might require additional files that might be required for the closed-loop simulation or for visualization of results. For almost all examples, we provide a (.m) file as an example of how the closed loop simulation using `MATLAB`might be done.

9.1.1 Loading and accessing the synthesized controller

Users can load the synthesized controllers from within MATLAB using, as an example, the following object initialization command:

```
C = ncsController(<controller relative file path>);
```

Such command initiates an object that will be responsible for reading the controller from the provided location. This command might take a while to load the controller from the file to the memory. It reads all NCS information stored in the file and needed to access the synthesized controller. Users can load as many controllers as they wish and they are not limited to one controller per simulation. The object can then be called to pride inputs for a specific NCS state using the command:

```
inputs = C.getInputs(<q_values>, <x_u_values>);
```

The passed argument `q_values` is an array boolean values each corresponds to the current state in the communication channel. When the value is `true`, this implies an empty or dropped packet. When the value is `false`, this implies a normal packet. The passed argument `x_u_values` is an array representing the currenr state/control-actions in both communication channels. The challenge is always to reconstruct the NCS state (i.e. `q_values` and `x_u_values`), knowing that the symbolic controller only receives plant's states from the channel. For the prolonged-delay class, this can be easly achieved and we provide a software class to reconstruct the complete NCS-state.

9.1.2 NCS state reconstruction

Synthesized controllers of NCS accept NCS-state information and provide input to the NCS. Therefore, when it comes to controller refinement (simulations as well), the construction of NCS-state from the states of the plants within the NCS is required. The NCS-state reconstructor should be able to collect information about the NCS from an observer position (i.e. not inside the NCS) since this is the case of any NCS where the controller is remotely isolated from the plant. The reconstructor have access from the controller's side to the state arriving before the controller and the control action generated by the controller.

Based on the synthesis of the symbolic controller, the reconstructor can provide complete or partial NCS-state information. For the prolonged-delay class, the reconstructor can provide a complete NCS-state information using arriving plant's states, the input generated by the compiler and simulating the dynamics of the plant. We provide an implementation of the constructor that can be used for simulation/implementation of the synthesized controller. Users should initiate the re-constructor object:

```
T = FIFOStateReconstructor(Nscmax, Ncamax, u0, @system_de, tau);
```

It uses the NCS delay information, the initial input u_0, the differential-equation of the plant, and the sampling time to reconstruct the actual current NCS-state. Once the re-constructor is informed about the state arriving before the controller, it simulates the plants behavior backward in time using the arriving state and inputs the controller already provided. During the simulation, when the communication channel offers no output (no packets still arrived), user notify the re-constructor by calling:

```
T.pushQ(u0);
```

When the communication channel offers a state from the plant, user notify the re-constructor by calling:

```
T.pushState(<x_u_pair>);
```

The passed argument `x_u_pair` holds the current arriving state and the last input provided by the controller.

Users can at any time ask for the current NCS state by calling:

```
[q_values, x_values] = T.getState();
```

then use it to request the input from the symbolic controller.

9.1.3 Closed loop Simulation

The following pseudo-code summarizes the closed loop simulation for a prolonged-delay NCS:

```
C = ncsController(<controller relative file path>);
T =  FIFOStateReconstructor(<configuration params>);

for loop = 1:MAX_SIM_LOOPS
  buffer_sc.push(current_state);
  if(no state available)
    T.pushQ(u0);
    u = u0;
  else
    T.pushState(<current state/input pair>);
    [q_values, x_values] = T.getState();
    all_u = C.getInputs(qValues, xValues);
    u = select_one(all_u);
  end
  buffer_ca.push(u);

  current_state=simulate_the_plant(current_state, u);
end
```

It is also notable that in this NCS-class, the communication channel can be simply presented as FIFO channel and implemented as a buffer/array in `MATLAB`.

9.1.4 Extensions

Users willing to extend the class has only to provide the mechanism of NCS-state reconstruction based on their NCS class. They also need to show how the communication channel behave in such class of NCS. Other than that, they use the interface directly and do their closed-loop simulations.

10 Simulation and Visualization using OMNeT++

10.1 Requirements and Preliminary Instructions

Consider the following before using the OMNeT++interface:

1. You need to have a working installation of OMNeT++. If you plan to build OMNeT++ for Linux, you might need to use the same a C/C++ compiler used for the CUDD library.

2. To use 3D visualization support, install the development packages for `Qt4, OpenSceneGraph (3.2) and the osgEarth (2.5 or later).`

3. Add the simulation project to OMNeT++'s projects using `eclipse`. The simulation project for SENSE is given in the folder (`./interface/OMNetpp`).

4. Compile the project and make sure it builds with no errors.

5. We always assume you already synthesized a controller before doing any simulations with OMNeT++. Simulations are initiaited and managed from within OMNeT++.

6. Each simulation requires a configuration (.ini) giving the OMNeT++'s project informations about the plant, the synthesized controller and NCS configurations. Some examples come with the (.ini) provided as samples that users can follow.

The provided simulation interface is an extensible base for the simulation of any class of NCS. We provide information related to the prolonged-delay class. We give instructions on how to extend it to other classes of NCS.

Here, we assume the user has already constructed an expanded NCS symbolic model and used it to synthesize a symbolic controller. The symbolic controller need to be saved as (.nbdd) file to a location known to the user. The user might require additional files that might be required for the closed-loop simulation or for visualization of results. For many examples, we provide a (.ini) configuration file for the OMNeT++ project as a sample.

10.1.1 Structure of the NCS closed-loop simulation

Doing the closed loop simulation in NCS requires less effort for the users. Users need just to do small manipulations to the (.ini) file to provide some details about the NCS and synthesized controllers.

The (.ini) file (usually named `simulaton.ini`) is a multi-line text file. Each line describes a value to be identified. As an example, the flowing listing shows a fragement of the text file for the provided example in (./examples/FIFO/robot):

```
...
NCS_Topology.Plant.tau = 1
...
NCS_Topology.Channel_SC.max_delay = 2
```

```
...
NCS_Topology.Channel_CA.AllowDropouts = false
...
```

The assignment is done in hierarchical manner. The hierarchy of objects within the project is as follows:

```
1- NCS_Topology
  1.1- Animator
    |- UnitScale             // The scaling unit of visualization
    |- SsXLb                 // The lower limit of X-axis in 2-D Visulaization
    |- SsXUb                 // The upper limit of X-axis in 2-D Visulaization
    |- SsYLb                 // The lower limit of Y-axis in 2-D Visulaization
    |- SsYLb                 // The upper limit of Y-axis in 2-D Visulaization
    |- numTargets            // Number of target objects
    |- TargetX1              // First X-coordinates for all targets as a list
    |- TargetX2              // Last X-coordinates for all targets as a list
    |- TargetY1              // First Y-coordinates for all targets as a list
    |- TargetY2              // Last Y-coordinates for all targets as a list
    |- numObsticles          // Number of obstacle objects
    |- ObsticleX1            // First X-coordinates for all targets as a list
    |- ObsticleX2            // Last X-coordinates for all targets as a list
    |- ObsticleY1            // First Y-coordinates for all targets as a list
    |- ObsticleY2            // Last Y-coordinates for all targets as a list
    |- Icon                  // Name of the Icon file for the visualized object
    |- IconScale             // Scale of the Icon for the visualized object

  1.2- Plant
    |- tau                   // The plant's sampling period
    |- plantmodel            // Name of the plant's model
    |- initial_state         // Initial state of the plant
    |- initial_input         // Intial input present before the plant

  1.3- Channel_SC
    |- min_delay             // Minimum delay in the channel
    |- max_delay             // Maximum delay in the channel
    |- AllowDropouts         // Enable/Disable packet dropping
    |- DropoutProbability    // The dropping probability

  1.4- Channel_CA
    |- min_delay             // Minimum delay in the channel
    |- max_delay             // Maximum delay in the channel
    |- AllowDropouts         // Enable/Disable packet dropping
    |- DropoutProbability    // The dropping probability

  1.5- Symbolic_Controller
    1.5.1- prolonger
      |- isActive            // Enable/Disable the prolongation
      |- max_delay           // Upper limit of delay-prolongation

    1.5.2- controller
      |- isEventDriven       // Select whether the controller is event/time-driven
      |- plantmodel          // Name of the plant being controlled
      |- tau                 // Samling period of the plant
      |- isTargetModes       // Is the controller switches between modes upon reaching targets
      |- targetBDDs          // BDD files describint atomic propositions of targets
      |- nbdd_file           // A list synthesized symbolic controllers (one controller per mode)
      |- nsc                 // The delay in the SC channel
      |- nca                  // The delay in the CA channel
      |- initial_input        // Initial control action provided by the controller

   1.6- Smart_ZOH
     1.6.1- prolonger
      |- isActive             // Enable/Disable the prolongation
      |- max_delay            // Upper limit of delay-prolongation

     1.6.2- ZOH
```

The communication channel class is based on random delays between the minimum and maximum delay limits. The prolongation should be activated for the prolonged-delay NCS class. The simulation allows for dynamical controllers provided in from of multi-mode controllers, where each controller is placed in a separate (.nbdd) file.

10.1.2 Closed loop Simulation

Running a simulation with the provided `OMNeT++` project is straightforward. Users provide their (.ini) file and copy it to the folder (`./interface/OMNetpp/simulations`). Then, simply run the simulation from inside `OMNeT++`.

10.1.3 Extensions

Users willing to extend the interface has only to provide the mechanism of NCS-state reconstruction based on their NCS class. They also need to show how the communication channel behave in such class of NCS. Other than that, they use the interface directly and do their closed-loop simulations.

References

[1] W. Heemels and N. van de Wouw, "Stability and stabilization of networked control systems," in *Networked Control Systems*. Springer, 2010, pp. 203–253.

[2] R. Majumdar and M. Zamani, "Approximately bisimilar symbolic models for digital control systems," in *Computer Aided Verification (CAV)*, ser. LNCS, M. Parthasarathy and S. A. Seshia, Eds., vol. 7358. Springer-Verlag, July 2012, pp. 362–377.

[3] G. Reissig, A. Weber, and M. Rungger, "Feedback refinement relations for the synthesis of symbolic controllers," *IEEE Transactions on Automatic Control*, vol. PP, no. 99, pp. 1–1, 2016.