SENSE: Abstraction-Based Synthesis of Networked Control Systems*

Mahmoud Khaled
Technical University of Munich
Munich, Germany
khaled.mahmoud@tum.de

Matthias Rungger
Technical University of Munich
Munich, Germany
matthias.rungger@tum.de

Majid Zamani
Technical University of Munich
Munich, Germany
zamani@tum.de

While many studies and tools target the basic stabilizability problem of networked control systems (NCS), nowadays modern systems require more sophisticated objectives such as those expressed as formulae in linear temporal logic or as automata on infinite strings. One general technique to achieve this is based on so-called symbolic models, where complex systems are approximated by finite abstractions, and then, correct-by-construction controllers are automatically synthesized for them. We present SENSE as a tool for the construction of finite abstractions for NCS and the automated synthesis of controllers, to enforce complex specifications over plants in NCS by taking into account several non-infidelities of the network.

Given a symbolic model of the plant and network delays, SENSE can efficiently construct a symbolic model of the NCS, by employing operations on binary decision diagrams (BDDs). Then, it synthesizes symbolic controllers satisfying a class of specifications. It has interfaces for the simulation and the visualization of the resulting closed-loop systems using OMNeT++ and MATLAB. Additionally, SENSE can generate ready-to-implement VHDL/Verilog or C/C++ codes from the synthesized controllers.

1 Introduction

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). NCS have attracted significant attention in the past decade due to their flexibility of deployment and maintenance (especially when using wireless communications). However, numerous technical challenges are present due to the wide range of uncertainties within NCS, introduced by their unreliable communication channels. This includes time-varying communication delays, packet dropouts, time-varying sampling/transmission intervals, and communication constraints (e.g. scheduling protocols).

Many studies deal with subsets of the previously mentioned imperfections targeting the basic stabilizability problem of NCS [3, 2, 16, 7]. However, nowadays CPS require more sophisticated specifications, including objectives and constraints given, for example, by formulae in linear temporal logic (LTL) or omega-regular languages [1].

A well-known approach to synthesize controllers enforcing such complex specifications is based on so-called symbolic models (a.k.a. discrete abstractions) [13, 5, 19, 10]. A given plant (i.e. a physical process described by a set of differential equations) is approximated by a symbolic model, i.e., a system with finite state and input sets. As the model is finite, algorithmic techniques from computer science [14, 6] are applicable to automatically synthesize discrete controllers enforcing complex logic specifications

*This work was supported in part by the German Research Foundation (DFG) through the grant ZA 873/1-1.

Submitted to:
MeTRiD 2018

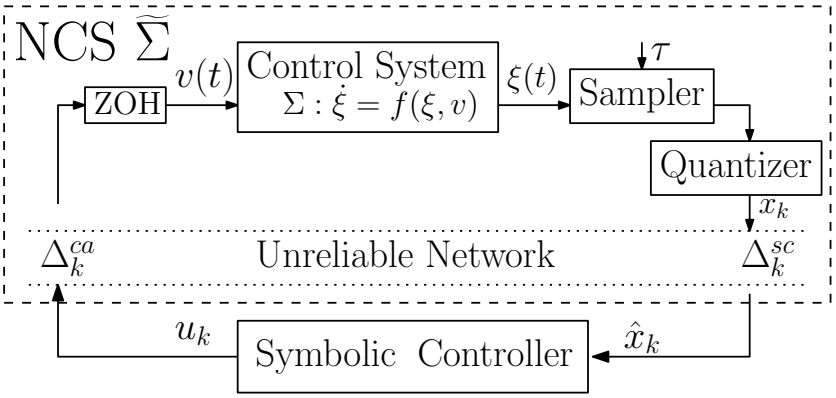


Figure 1: Structure of the NCS and the symbolic controller.

over symbolic models. Finally, those discrete controllers (a.k.a. symbolic controllers) can be refined to hybrid ones enforcing the same properties over concrete systems.

In this paper, we present SENSE, a tool for the automated synthesis of controllers for NCS. It automatizes the construction of symbolic models of NCS, given symbolic models of plants in them, and the synthesis of their controllers. Additionally, it generates automatically C/C++ or VHDL/Verilog codes that are ready for implementation and, thus, providing an end-to-end automated synthesis approach for NCS.

2 Symbolic models and controller synthesis for NCS

Figure 1 depicts the structure of NCS supported in the tool SENSE, which follows the general NCS structure reported in [16] and [17]. A NCS $\widetilde{\Sigma}$ consists of the following components: (1) a control system, often referred to as plant, representing a physical process to be controlled; (2) two non-ideal communication channels transferring state information and control inputs from/to the plant; and (3) a remote digital (symbolic) controller enforcing some complex specifications on the plant by taking into account the imperfections of the communication channels.

The general (possibly nonlinear) control systems Σ is usually modeled by a differential equation of the form $\dot{\xi}(t) = f(\xi(t), \upsilon(t))$ whose state $\xi(t) \in \mathbb{R}^n$ is measured, every sampling period τ, by a sensor/sampler, that is followed by a quantizer. The input signal υ belongs to $\mathscr{U}$, which is a subset of the set of all functions of time from $]a,b[\subseteq \mathbb{R}$ to $\mathbb{R}^m$. We denote by $\xi_{x\upsilon}(t)$ the point reached at time t under the input υ and started from the initial condition $x = \xi_{x\upsilon}(0)$. We denote by x_k the sampled and quantized state, and $k \in \mathbb{N}_0$ is an index referring to the sampling time $k\tau$. Additionally, a zero-order-hold (ZOH) helps sustaining an inter-sampling continuous-time control input $\upsilon(t)$.

A communication network connects the plant to a controller and introduces the time-varying sensor-to-controller and controller-to-actuator delays Δ_k^{sc} and Δ_k^{ca}, respectively. A formal definition of symbolic controllers is given in [10]. 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 $N_k^{sc} \in [N_{\min}^{sc}; N_{\max}^{sc}] \subset \mathbb{N}$, $N_k^{ca} \in [N_{\min}^{ca}; N_{\max}^{ca}] \subset \mathbb{N}$, and $N_{\min}^{sc}, N_{\max}^{sc}, N_{\min}^{ca}, N_{\max}^{ca} \in \mathbb{N}$.

Having a quantizer before the network as in Figure 1 and considering time-varying communication delays with upper and lower bounds, as showed in [4, 18, 17, and references therein], one can readily

consider four types of network non-idealities: (i) quantization errors; (ii) limited bandwidth; (iii) time-varying communication delays; and (iv) packet dropouts as long as the maximum number of consecutive dropouts over the network is bounded [3].

2.1 Symbolic Control of NCS

We use the symbolic approach to design controllers enforcing given logic specifications over the plants in NCS while taking the network imperfections into account. Here, we use a notion of system [13] as a unified modeling framework for both continuous systems as well as their finite abstractions. A system is a tuple $S = (X, X_0, U, F)$ that consists of a state set X; a set of initial states $X_0 \subseteq X$; an input set U; and a transition relation $F \subseteq X \times U \times X$. We also use F as a map to denote the set of post-states of x which is $F(x,u) = \{x' \in X | (x,u,x') \in F\}$.

Let $S_\tau(\Sigma) = (\mathbb{R}^n, \mathbb{R}^n, U_\tau, F_\tau)$ denote the system that captures the evolution of Σ at each sampling time τ. Set $U_\tau = \{\upsilon : [0,\tau] \to \mathbb{R}^m \mid \upsilon(0) \in \mathbb{R}^m$, and $\upsilon(t) = \upsilon(0)$, for all $0 < t \leq \tau\}$ is a an input set of constant curves over intervals of length τ. A transition $(x_\tau, \upsilon_\tau, x'_\tau)$ belongs to F_τ if and only if there exists a trajectory $\xi_{x_\tau \upsilon_\tau} : [0,\tau] \to \mathbb{R}^n$ in Σ such that $\xi_{x_\tau \upsilon_\tau}(\tau) = x'_\tau$.

Let $S_q(\Sigma) = (X_q, X_{q,0}, U_q, F_q)$, where X_q is a finite cover of X_τ and U_q is a finite subset of U_τ, denotes the symbolic model of $S_\tau(\Sigma)$ if there exists a *feedback refinement relation* (FRR) $Q \subseteq X_\tau \times X_q$ between the two systems. Interested readers can find more details about FRR and the construction of F_q in [10]. System $S_q(\Sigma)$ can then be used to synthesize controllers that enforce given logic specifications over Σ.

For NCS, one can follow the same methodology. A system $\widetilde{S}_\tau(\Sigma)$, representing $S_\tau(\Sigma)$ in the network environment with the network imperfections, is derived from $S_\tau(\Sigma)$. Then, $\widetilde{S}_q(\Sigma)$, which represents a symbolic model of the overall NCS, is constructed from $\widetilde{S}_\tau(\Sigma)$ such that there exists a FRR $\widetilde{Q}$ between the two systems. The process is rather complex since $\widetilde{S}_\tau(\Sigma)$ is high-dimensional representation of $S_\tau(\Sigma)$ that includes network imperfections. Instead, SENSE employs the results in [4, Theorem 4.1] to construct $\widetilde{S}_q(\Sigma)$ directly from $S_q(\Sigma)$ while preserving some FRR, all without going through the construction of $\widetilde{S}_\tau(\Sigma)$. This construction is achieved by applying an operator $\mathscr{L}$ as introduced in [4, 5]. More specifically, $\widetilde{S}_q(\Sigma) := (\widetilde{X}_q, \widetilde{X}_{q,0}, U_q, \widetilde{F}_q)$ is derived as $\widetilde{S}_q(\Sigma) = \mathscr{L}(S_q(\Sigma), N^{sc}_{\min}, N^{sc}_{\max}, N^{ca}_{\min}, N^{ca}_{\max})$, where

- $\widetilde{X}_q = \{X_q \cup q\}^{N^{sc}_{\max}} \times U_q^{N^{ca}_{\max}} \times [N^{sc}_{\min}; N^{sc}_{\max}]^{N^{sc}_{\max}} \times [N^{ca}_{\min}; N^{ca}_{\max}]^{N^{ca}_{\max}}$, where q is a dummy symbol;
- $\widetilde{X}_{q,0} = \{(x_0, q, \ldots, q, u_0, \ldots, u_0, N^{sc}_{\max}, \ldots, N^{sc}_{\max}, N^{ca}_{\max}, \ldots, N^{ca}_{\max})$, where $x_0 \in X_{q,0}$ and $u_0 \in U_\tau\}$;
- The transition $((x_1, \ldots, x_{N^{sc}_{\max}}, u_1, \ldots, u_{N^{ca}_{\max}}, \widetilde{N}_1, \ldots, \widetilde{N}_{N^{sc}_{\max}}, \widehat{N}_1, \ldots, \widehat{N}_{N^{ca}_{\max}}), u,$ $(x', x_1, \ldots, x_{N^{sc}_{\max}-1}, u, u_1, \ldots, u_{N^{ca}_{\max}-1}, \widetilde{N}, \widetilde{N}_1, \ldots, \widetilde{N}_{N^{sc}_{\max}-1}, \widehat{N}, \widehat{N}_1, \ldots, \widehat{N}_{N^{ca}_{\max}-1}))$ belongs to $\widetilde{F}_q$, 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 $x_0 \xrightarrow[\tau]{u_{N^{ca}_{\max}-j*}} x'$ in F_q, where $j*$ is a time-shifting index defined in [4].

Symbolic controllers are generally finite systems that accept states $\hat{x}_k$ as inputs and produce u_k as outputs to enforce a given specification over the plants in NCS. The tool SENSE natively supports safety, reachability, persistence, and recurrence specifications given as the LTL formulae $\Box\varphi_S$, $\Diamond\varphi_T$, $\Diamond\Box\varphi_S$, $\Box\Diamond\varphi_T$, respectively, where φ_T and φ_S are the predicates defining, respectively, some target and safe sets T and S. In the next section, we introduce three main modules inside SENSE:

(1) The symbolic model construction engine: it is responsible for constructing the symbolic model of the NCS from a symbolic model of the plant in it;

(2) The fixed-point operations engine: it is responsible for synthesizing correct-by-construction symbolic controllers enforcing the aforementioned LTL properties over the NCS;

(3) Various tools to analyze and simulate the closed-loop NCS, and to automatically generate code of the synthesized control software.

3 Structure of the tool SENSE

The tool SENSE is an open-source *extensible* framework that provides a base for further research on the modeling, abstraction, and controller synthesis of NCS. The research on symbolic control for NCS is ongoing [17, 4] and the tool is developed to help researchers interested in automated synthesis of NCS.

For the construction of symbolic models of NCS, SENSE has a software engine that supervises the construction procedure. The engine follows certain customizable construction rules that describe how the symbolic model of the NCS is constructed from the symbolic model of the plant. The construction rules depend on the sensor-to-controller and controller-to-actuator delays Δ_k^{sc} and Δ_k^{ca}, respectively. Construction rules (implemented in terms of C++ classes) can be readily provided for any class of NCS. However, due to the imposed restriction in the implementation phase of the synthesized controllers, following the existing theory in [4, 17], the tool SENSE is currently available with only the construction rules for a class of NCS called pronged-delay NCS (introduced later). Nevertheless, due to the modular design of SENSE, it is straightforward to define extra construction rules, e.g. taking time-varying delays into account, once the theory behind them is available.

The tool SENSE provides ready-to-use fixed-point routines that operate on the constructed symbolic models to synthesize controllers enforcing any of those specifications introduced in Section 2. It is also equipped with two interfaces to access the synthesized controllers. The first one is via MATLAB and allows accessing the synthesized controller as well as performing closed-loop simulations, in which, the plant is being remotely controlled by the synthesized controller over the network. Since the visualization capabilities of MATLAB are limited, we include a second interface using OMNeT++ [8], which provides a powerful visualization-framework. The interface of OMNeT++ allows accessing the synthesized controller to perform realistic network simulations as well as visualizing the closed-loop behavior.

Along with the core engines in SENSE, a rich set of helper tools is developed and made available for users to analyze and simulate the resulting symbolic models and the synthesized symbolic controllers. One of those tools allows for code generation of the synthesized controllers as C/C++ or VHDL/Verilog codes to cover both software and hardware implementations and provide an end-to-end solution.

3.1 The symbolic model construction engine

A straightforward implementation of $\mathscr{L}$-operator can be achieved using element-by-element expansion of the state set of the symbolic model of the plant. However, this is computationally inefficient and consumes a large amount of memory.

Symbolic models, which can be considered as labeled transition systems, are easily represented by boolean functions and encoded inside BDDs. Tool SENSE expands the BDD representing the symbolic model of the plant to construct a BDD representing the symbolic model of the NCS. Symbolic operations on BDDs play the major role in order to construct symbolic models of NCS from those of their plants. Using the CUDD library [12], SENSE employs such operations to efficiently handle computation/memory complexities when implementing the $\mathscr{L}$-operator.

Figure 2 depicts the structure of the engine for constructing symbolic models of NCS. The engine takes the symbolic model of the plant as its input. Users can construct such models using the existing tool SCOTS [11], which is provided as a library inside the tool SENSE. The constructed model is then given

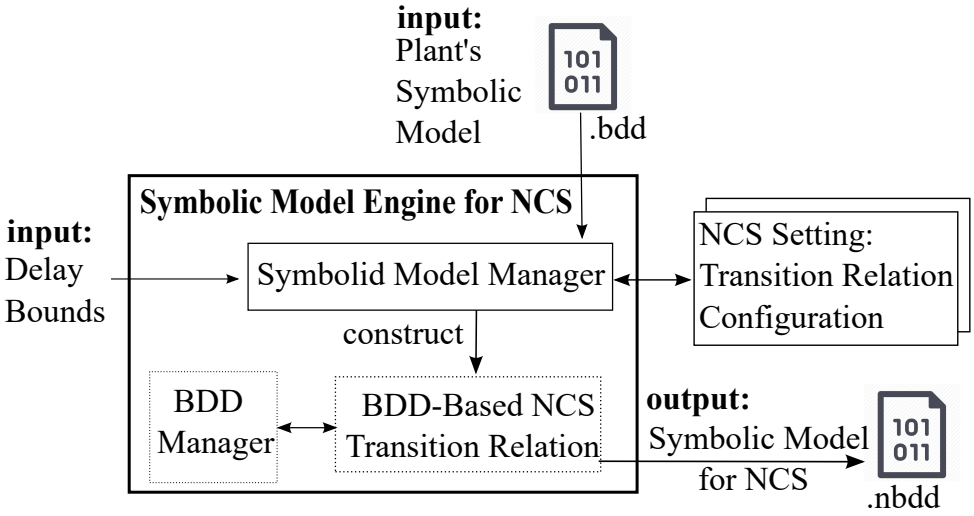


Figure 2: Structure of the engine for constructing symbolic models of NCS.

to the engine in the form of a BDD file (i.e., a file storing the BDD). The delay bounds within the NCS should also be provided. The engine then starts expanding the symbolic model of the plant. Specifically, the transition relation of the NCS is crafted by Cartesian-product-like expansion of the transition relation of the plant. This is done with the help of a BDD-Manager provided by the CUDD library as follows:

(1) adding extra binary variables to accommodate for the traveling state and input packets through the communication channels, and their delays;

(2) to construct $\widetilde{X}_q$, `SENSE` expands X_q with the dummy symbol q and then performs Cartesian products with U_q and the set of delays in both channels;

(3) to construct $\widetilde{F}_q$, `SENSE` performs BDD operations on F_q using the provided construction rules described as C++ classes.

3.1.1 The prolonged-delay NCS

Note that for constructing symbolic models of any NCS (using the operator $\mathscr{L}$) and synthesizing symbolic controllers enforcing any of the properties introduced in Section 2 over them, we only require the delays in both channels of the network to be integer multiples of the sampling period τ with some lower and upper bounds; see [4, 17] for more details.

The tool `SENSE` can handle the construction of symbolic models and the synthesis of their controllers for any class of NCS. However, in order to refine the controllers and apply them to the concrete NCS, the current theory proposed in [17] requires that the upper and lower bounds of the delays to be equal at each channel. This implies that, in each channel, all packets are delayed by the same amount of time. This can be practically achieved by performing extra prolongation (if needed) of the delays suffered by the packets. For the sensor-to-controller branch of the network, 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. Interestingly, this prolongation results in less conservativeness in terms of the existence of symbolic controllers; see [17, Lemma 6.1] for more details.

Additionally, to refine the synthesized symbolic controllers, it is also required that the symbolic model of the plant is deterministic. This can be fulfilled for a wide class of physical systems [13] having some stability properties, and it does not require the symbolic models of the NCS to be deterministic. Nevertheless, the support of controller refinement for non-deterministic symbolic models of the plants is currently under development and will be incorporated in SENSE in the near future.

3.2 Fixed-point computation for controller synthesis

The tool SENSE implements fixed-point algorithms for synthesizing controllers that enforce those specifications introduced in Section 2. Tool SENSE uses a customized version of the controller synthesis engine provided in SCOTS. In order to synthesize controllers that enforce any of the four specifications, the engine implements fixed-point algorithms that employ a Pre-operation over state-input pairs of the symbolic model of the NCS. More specifically, given a symbolic model of the NCS $\widetilde{S}_q$ and a set $Z \subseteq \widetilde{X}_q \times U_q$, the Pre-operation of $\widetilde{X}_q$ is defined by $pre(Z) = \{(x,u) \in \widetilde{X}_q \times U_q | \emptyset \neq \widetilde{F}_q(x,u) \subseteq \pi_{\widetilde{X}_q}(Z)\}$, where $\pi_{\widetilde{X}_q}$ is a projection map defined as $\pi_{\widetilde{X}_q}(Z) = \{x \in \widetilde{X}_q | \exists_{u \in U_q} (x,u) \in Z\}$. Interested readers can find more details in [11, 13, 10]. All steps in the algorithms as well as the Pre-operation are implemented by BDD operations. Users can employ the four main algorithms to describe different LTL specifications and to synthesize (possibly dynamic) symbolic controllers. We demonstrate this, with examples, for two different LTL specifications later in Section 4.

The target set T and safe set S are to be given as atomic propositions over the state space of the plant. Such sets can be generated using SCOTS and they are provided as BDD files. The tool takes care of expanding those sets to be compatible with the symbolic model of the NCS. Then, it synthesizes controllers to enforce the specifications over the original plants in NCS. Controllers are, by construction, BDD objects and they are saved as BDD files.

3.3 Simulation, analysis and code generation

The tool SENSE provides two different interfaces to access the controllers, using MATLAB and OMNeT++. For both interfaces, it provides a common C++ layer to read the BDD-based synthesized controllers.

For MATLAB, an m-script uses the C++ layer to access the controllers while providing a set of m-script functions and classes to interface the synthesized controllers. For almost all examples, m-scripts are provided to simulate the closed-loop behavior of the NCS. This includes simulating the plant's differential equation, and the network evolution.

In the interface for OMNeT++, communication channels are modeled as random-delay channels for realistic simulations. Visualizations up to 3 dimensions are supported in the visualization engine of OMNeT++.

The tool SENSE also provides the following tools that help analyzing the symbolic models as well as the synthesized controllers:

- bdd2implement: a tool to automatically generate C/C++ or VHDL/Verilog codes from the synthesized BDD-based controllers;
- bdd2fsm: a tool to generate files following the FSM data format [15] or the comma-separated format of the transition relation of symbolic models. Such representation is used by many software packages to visualize graphs (e.g., the tool StateVis[9]). Users can use the tool to analyze or understand, visually, how the symbolic model behaves;

- `bddDump`: a tool to extract the meta-data information stored inside the generated BDD files. This helps inspecting the original information about the plant such as sampling time, quantization parameters, or the binary variables;
- `contCoverage`: a tool to provide fast terminal-based ASCII-art visualization of the coverage of the synthesized controllers. It is only possible up to 2 dimensions. To visualize the coverage of controllers with 3 dimensional inputs, users can still use `MATLAB`.
- `sysExplorer`: a tool to help testing the expanded transition relation or the synthesized controllers. It handles the BDD as IN-OUT box. Users can request information about specific transitions in the transition relation by providing the initial state and a sequence of inputs, and the tool responds with the post-states. For controller testing, the tool can be used to test the controller by providing states to check the corresponding generated inputs.

3.3.1 Automated code generation of synthesized controllers

In `SENSE`, the helper tool `bdd2implement` is provided to automatically generate codes, targeting hardware and software implementations, of the synthesized controllers. It starts by determinizing the symbolic controller, in case it is not deterministic, by simply selecting the first available input, for each state of the system. Then, it converts the multi-output boolean function, representing the controller and encoded in BDD, to many single-output functions. Each single-output boolean function represents one bit of the binary-encoded output of the controller. Then,

- for C/C++ code generation, the boolean functions are dumped as C++ functions with `bool` return values along with another function that collects and constructs the output of the controller The C/C++ target-specific compiler takes care of converting such boolean functions to machine codes, implying that the controller is encoded as instructions of the target microprocessor. This, however, might violate the real-time requirements for some applications in which the time required to compute the control output, by executing the instructions, exceeds the real-time deadline. A more time-efficient (but not memory-efficient) implementation is to store the controller as a lookup-table in the memory of the target hardware and access it directly. The latter is straightforward but is not currently added to the tool.
- for VHDL/Verilog code generation, the boolean functions are directly written as register transfer level (RTL) codes. An electronic design automation (EDA) tool takes care of converting the code to technology-mapped netlist and, later, a vendor-specific software finds the suitable mapping in the target chip, in an operation called place-and-route.

3.4 The work flow inside `SENSE`

Figure 3 depicts the complete structure and the work-flow of `SENSE`. As an input, the engine takes the symbolic model of the plant, the delay bounds within the NCS, and the specification. Then, BDD files of the symbolic model of the NCS, following the construction rules, and its controller are generated. The files are used by the helper tools for analysis and code generation purposes. The steps taken by the user to use `SENSE`are summarized as follows:

1. The user computes a symbolic model of the plant as a BDD file using the included library `SCOTS`;
2. The user can extract information of the BDD file using the helper tool `bddDump`;

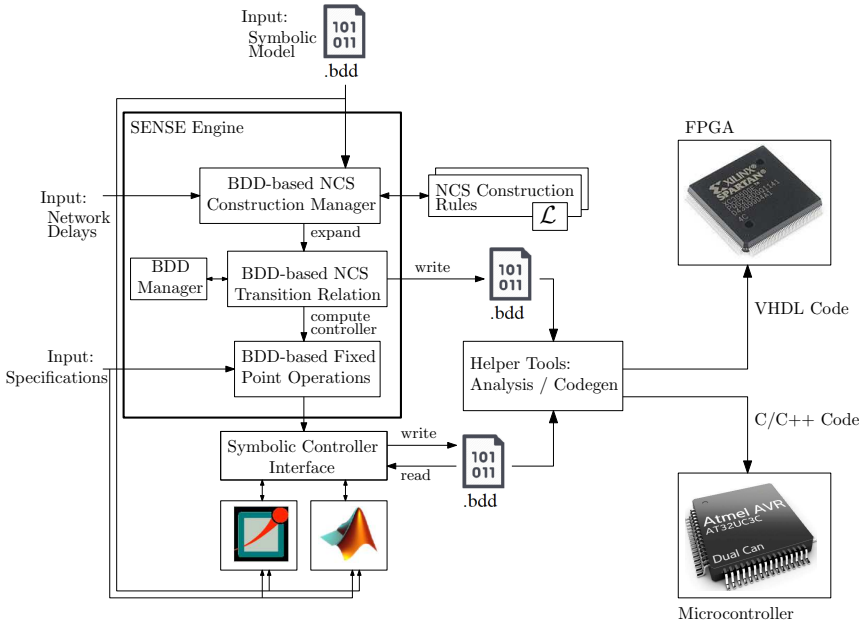


Figure 3: Work flow inside SENSE: from specifications to ready-to-implement source codes.

3. The user can also visualize the transition relation of the symbolic model via the tool `bdd2fsm` and a graph visualizing tool (e.g. StateVis);
4. The user passes the BDD file representing the plant's symbolic model to SENSE, along with the network delays and the specification;
5. The tool SENSE generates BDD files for the symbolic model of the NCS and the controller;
6. The user can benefit from the tools `sysExplorer` and `contCoverage` to test the symbolic model of the NCS and its synthesized controller;
7. The user can use the provided interfaces in SENSE, in combination with MATLAB and/or OMNeT++, to simulate and visualize the closed-loop behavior of the NCS;
8. The user can use the tool `bdd2implement` to generate the final controller for implementation.

4 SENSE in action: results and examples

We present several case studies where we construct symbolic models of NCS, namely, $\widetilde{S}_q(\Sigma)$ from the symbolic models of their plants, namely, $S_q(\Sigma)$. We also provide two examples where dynamic controllers are synthesized. The closed-loop is simulated using the interfaces with MATLAB and OMNeT++ in SENSE. Code generation is achieved using the tool `bdd2implement` in SENSE. All examples are done in a PC (Intel Xeon-E5-1620 3.5GHz 256GB RAM).

4.1 Construction of symbolic models of NCS

First, symbolic models of the plants in NCS are constructed and stored as BDD files. The BDD files are fed as inputs to the engine of SENSE along with NCS delay bounds to construct symbolic models of NCS. Table 1 summarizes the results for different case studies and different network delays given as

Table 1: Results for constructing symbolic models of prolonged-delay NCS using those of their plants.

Case Study	$\lvert S_q(\Sigma)\rvert$		(2,2)	(2,3)	(2,4)	(2,5)	(3,2)	(3,3)	(3,4)	(3,5)	(4,2)	(4,3)
DI	2039	$\lvert\widetilde{S}_q(\Sigma)\rvert$	14096	56336	225296	901136	22832	91184	364592	1.45×10^{6}	38340	152964
		Time (sec)	<1	<1	<1	<1	<1	<1	<1	<1	<1	<1
		Memory (KB)	2.0	2.4	3.1	2.9	3.0	2.9	3.1	3.2	5.2	4.3
Robot	29280	$\lvert\widetilde{S}_q(\Sigma)\rvert$	4.1×10^{6}	6.5×10^{7}	1.04×10^{9}	1.67×10^{10}	3.4×10^{7}	5.4×10^{8}	8.7×10^{9}	1.4×10^{11}	2.8×10^{8}	4.6×10^{9}
		Time (sec)	<1	<1	<1	<1	<1	<1	<1	<1	1.4	1.6
		Memory (KB)	15	14	17	16	16	21	22	19	35	33
Jet	9.0×10^{5}	$\lvert\widetilde{S}_q(\Sigma)\rvert$	1.5×10^{11}	1.0×10^{13}	6.5×10^{14}	4.2×10^{16}	7.2×10^{12}	4.6×10^{14}	2.9×10^{16}	1.9×10^{19}	3.3×10^{14}	2.1×10^{16}
		Time (sec)	1970	1637	1674	2172	3408	1772	2111	7107	4011	2854
		Memory (KB)	4323.3	2374.2	2389.1	2392.6	3683.2	4098.2	3317.2	3582.6	5894.3	4784.5
DC-DC	3.8×10^{6}	$\lvert\widetilde{S}_q(\Sigma)\rvert$	8.9×10^{7}	1.8×10^{8}	3.6×10^{8}	7.1×10^{8}	5.2×10^{8}	1.0×10^{9}	2.0×10^{9}	4.1×10^{9}	3.0×10^{9}	6.0×10^{9}
		Time (sec)	672	681	530	1131	13690	10114	9791	10084	139693	137648
		Memory (KB)	3347.2	3145.0	3176.7	2784.8	10875.2	11543.8	11572.0	11592.1	34169.2	37852.6
Vehi	1.9×10^{7}	$\lvert\widetilde{S}_q(\Sigma)\rvert$	4.2×10^{12}	2.7×10^{14}	1.7×10^{16}	1.1×10^{18}	2.3×10^{14}	1.4×10^{16}	9.3×10^{17}	5.9×10^{19}	1.3×10^{16}	7.94×10^{17}
		Time (sec)	273.3	285	238.4	173	22344	54919	27667	36467	39065	145390
		Memory (KB)	1638.4	1945.6	1843.2	1945.6	23040	40652.8	30208	22425.6	21094.4	36556.3
Inver	4.3×10^{8}	$\lvert\widetilde{S}_q(\Sigma)\rvert$	2.4×10^{14}	1.5×10^{16}	1.0×10^{18}	6.5×10^{19}	4.6×10^{16}	2.9×10^{18}	1.9×10^{20}	1.2×10^{22}	9.2×10^{16}	5.8×10^{20}
		Time (sec)	361.4	349.1	340.9	347.8	942	793	1218	910	58411	57110
		Memory (KB)	723.1	808.4	349.6	1012.7	2958	2840	3104	2898	35907	35628

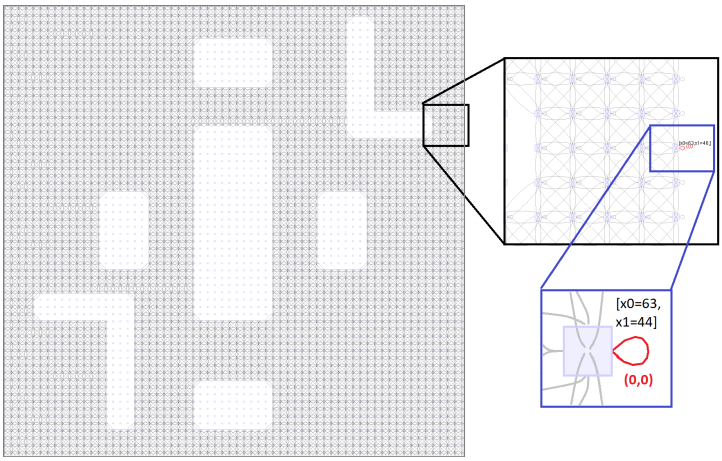
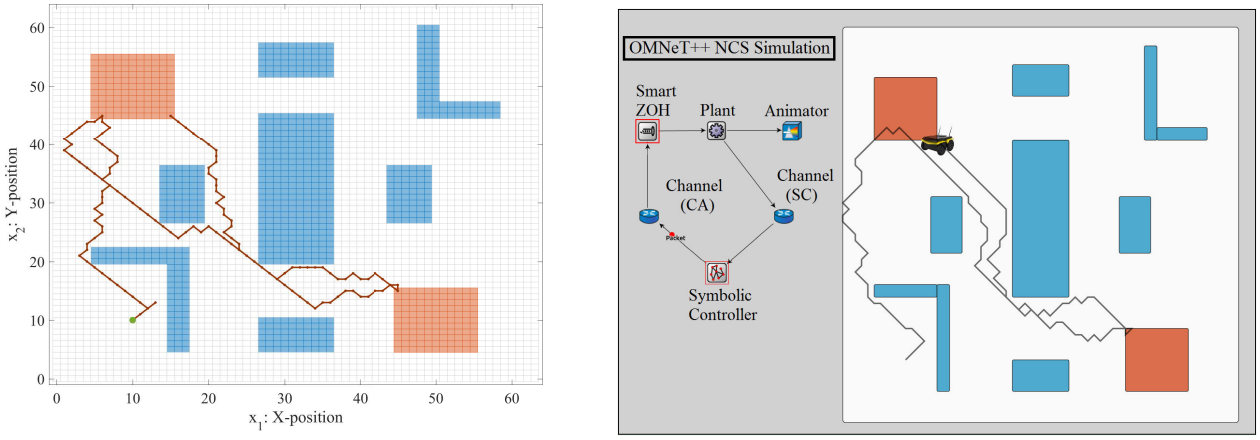


Figure 4: Visualization of the transition relation of the robot example. In this example, obstacles in the state set are considered by removing any transitions to or from them.

pairs (a,b) in the first row where the delay in the sensor-to-controller channel is upper bounded by $a\tau$ and the one in the controller-to-actuator channel is upper bounded by $b\tau$. For each case study, we show the size of the symbolic model of the plant (i.e. number of transitions and denoted by $|.|$). Then, for different network delays, we show the size of the symbolic models of NCS, the time in seconds required to construct them (from the symbolic model of the plant), and the memory in KB used to store them.

We consider several different dynamics for the plant in the NCS including a double integrator (denoted by DI), a fully actuated robot (denoted by Robot), a jet engine (denoted by Jet), a Boost DC-DC Converter (denoted by DC-DC), a vehicle dynamic (denoted by Vehi), and finally an inverted pendulum system (denoted by Inver). The details of all these plants can be found in the appendix.



(a) MATLAB (b) OMNeT++

Figure 5: Closed-loop simulation of the robot in MATLAB and in OMNeT++. Two target sets are represented by blue boxes. Nine obstacle sets are represented by red boxes.

4.2 Controller synthesis example: a remotely-controlled robot

For controller synthesis, we consider the Robot case reported in Table. 1. We consider delay parameters of the network to be $(2,2)$. The plant dynamic is described by the following differential equation:

$$\begin{bmatrix}\dot{\xi}_1\\ \dot{\xi}_2\end{bmatrix} = \begin{bmatrix}\upsilon_1\\ \upsilon_2\end{bmatrix},$$

where $(\xi_1,\xi_2) \in X \subseteq \mathbb{R}^2$ denotes the position of the robot in the bounded 2D arena X, and (υ_1,υ_2) represents a steering input. We use the following state and input sets: set of states is $[0,64]\times[0,64]$, state quantization parameters are $(1,1)$, input set is $[-1,1]\times[-1,1]$, and input quantization parameters are $(1,1)$. The control objective is described by the following LTL formula:

$$\psi = \Big(\bigwedge_{i=1}^{9} \Box(\neg \mathsf{Obstacle}_i)\Big) \wedge \Box\Diamond(\mathsf{Target1}) \wedge \Box\Diamond(\mathsf{Target2}),$$

where the atomic propositions Target1, Target2, and $\mathsf{Obstacle}_i$, $i \in \{1,\ldots,9\}$, are some hyper-intervals over X, as depicted in Figure 5.

A program that uses the library SCOTS constructs the symbolic model of the plant in 0.23 seconds. Then, the transition relation is exported using the tool bdd2fsm. Figure 4 depicts the visualized transition relation using the generated FSM file and the tool StateVis. The blank spaces in the state set represent obstacles, which is common to efficiently encode obstacles in the transition graph and to reduce the overall number of transitions.

The NCS construction engine of SENSE constructs the symbolic model of the NCS in 0.26 seconds. The fixed-point operations engine in SENSE synthesizes the controller in 8 seconds. The resulting controller is a dynamic controller with two discrete states, each corresponds to one static controller, that enforces a reachability specification for one of the target sets. Figure 5a (resp. 5b) shows the closed-loop simulation in MATLAB (resp. OMNeT++). We make use of the animation capabilities of OMNeT++ to visualize both packet transfers over the network as well as the movement of the robot in the arena. The

```
--////////////////////////////////////////////////////////////////////////////////
--
-- This file was automatically generated.
--
-- Create Date: 2018-01-15.14:14:37
-- Module Name: bddWrapper as VHDL code
-- Project Name: Symbolic Controller implementation
-- Target HW: FPGA/ASIC
--
--////////////////////////////////////////////////////////////////////////////////

library ieee;
use ieee.std_logic_1164.all;


entity BDDWrapper_entity is
port(
x0: in std_logic;
x1: in std_logic;
....
x25: in std_logic;

f0: out std_logic;
f1: out std_logic;
....
f3: out std_logic
);
end BDDWrapper_entity;


architecture BDDWrapper_behavioral of BDDWrapper_entity is
begin
f0 <= not((x0 and ((x1) or (((x2) or (((x3) or (((x4) or (((x5) or (((x6) or (((x7) ......
f1 <= not((x0 and ((x1) or (((x2) or (((x3) or (((x4) or (((x5) or (((x6) or (((x7) ......
   ....
f3 <= not((x0 and ((x1) or (((x2) or (((x3) or (((x4) or (((x5) or (((x6) or (((x7) ......
end BDDWrapper_behavioral;
```

Figure 6: Code snippet from the generated VHDL code.

target sets are indicated with red boxes and the obstacles are indicated with blue boxes. The initial state of the system is $(10, 10)$.

The tool `bdd2implement` is used to generate VHDL and C/C++ codes that are ready for implementations. Figure 6 shows a code-snippet of the generated VHDL code for one of the static reachability controllers of the Robot example.

4.3 Controller synthesis example: collision-free deployment of two robots

We use two instances of the same dynamic from the previous example to represent two different robots. The dynamics are augmented to form a system of higher dimension (i.e. 4-dimensional state set and 4-dimensional input set). The system is now described by the following differential equation:

$$\begin{bmatrix} \dot{\xi}_1 \\ \dot{\xi}_2 \\ \dot{\xi}_3 \\ \dot{\xi}_4 \end{bmatrix} = \begin{bmatrix} \upsilon_1 \\ \upsilon_2 \\ \upsilon_3 \\ \upsilon_4 \end{bmatrix}.$$

We use the following state and input sets: set of states is $[0,15] \times [0,15] \times [0,15] \times [0,15]$, state quantization parameters are $(1,1,1,1)$, input set is $[-1,1] \times [-1,1] \times [-1,1] \times [-1,1]$, and input quantization parameters are $(1,1,1,1)$. The objective of this example is to synthesize a controller that enforces the following two specifications simultaneously:

1. For the first robot:

$$\psi_1 = \Big(\bigwedge_{i=1}^{5} \Box(\neg \mathsf{Obstacle}_i^1)\Big) \wedge \Big(\bigwedge_{i=0,j=0}^{15,15} \Box(\neg \mathsf{Crash}_{i,j})\Big) \wedge \Box\Diamond(\mathsf{Target1}) \wedge \Box\Diamond(\mathsf{Target2}),$$

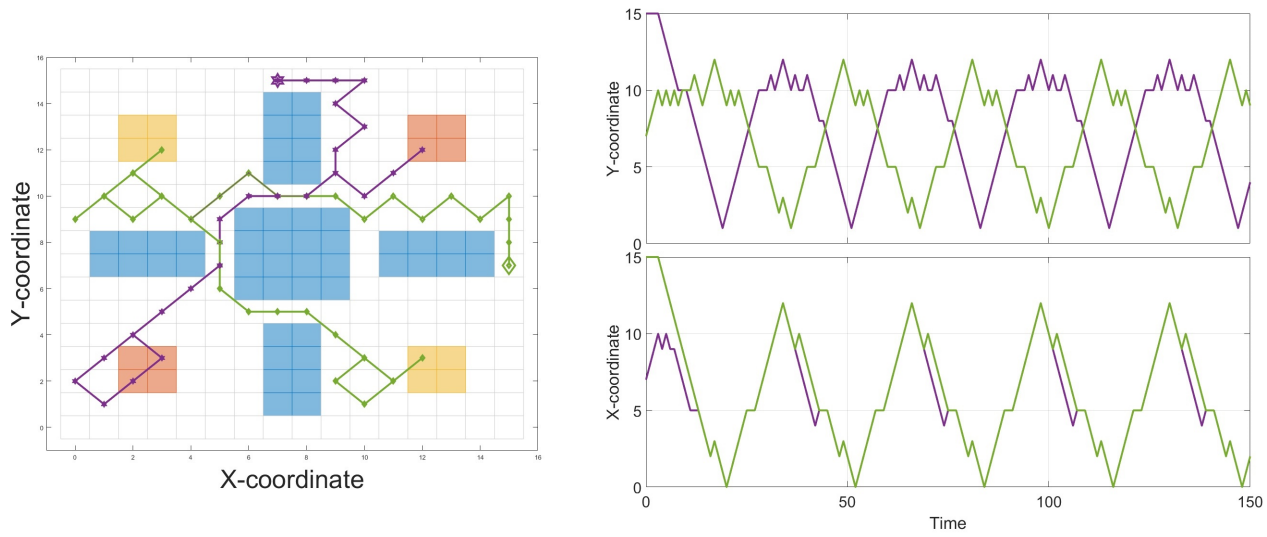


Figure 7: Closed loop simulation of the two robots example in MATLAB.

2. For the second robot:

$$\psi_2 = \Big(\bigwedge_{i=1}^{5} \Box(\neg \mathsf{Obstacle}_i^2)\Big) \wedge \Big(\bigwedge_{i=0,j=0}^{15,15} \Box(\neg \mathsf{Crash}_{i,j})\Big) \wedge \Box\Diamond(\mathsf{Target3}) \wedge \Box\Diamond(\mathsf{Target4}),$$

where Target_1, Target_2, Target_3 and Target_4 are atomic propositions over the state set and defined by the hyper-rectangles: $[2,3]\times[2,3]\times[0,15]\times[0,15]$, $[12,13]\times[12,13]\times[0,15]\times[0,15]$, $[0,15]\times[0,15]\times[2,3]\times[12,13]$, and $[0,15]\times[0,15]\times[12,13]\times[2,3]$, respectively. $\mathsf{Crash}_{i,j}$, $\forall i,j\in\{0,1,\ldots,15\}$, represent the points in the state set where the two robots collide and are defined as follows:

$$\mathsf{Crash}_{i,j} = [i\ j\ i\ j]^T.$$

$\mathsf{Obstacle}_i^1$, $\forall i\in\{0,1,\ldots,5\}$, are obstacles and are defined by the hyper-rectangle: $[6,9]\times[6,9]\times[0,15]\times[0,15]$, $[11,14]\times[7,8]\times[0,15]\times[0,15]$, $[1,4]\times[7,8]\times[0,15]\times[0,15]$, $[7,8]\times[1,4]\times[0,15]\times[0,15]$, and $[7,8]\times[11,14]\times[0,15]\times[0,15]$, respectively. $\mathsf{Obstacle}_i^2$, $\forall i\in\{0,1,\ldots,5\}$, are obstacles and are defined by the hyper-rectangle: $[0,15]\times[0,15]\times[6,9]\times[6,9]$, $[0,15]\times[0,15]\times[11,14]\times[7,8]$, $[0,15]\times[0,15]\times[1,4]\times[7,8]$, $[0,15]\times[0,15]\times[7,8]\times[1,4]$, $[0,15]\times[0,15]\times[7,8]\times[11,14]$, respectively.

The controller is synthesized and simulated as presented in the previous example. The symbolic model of the plant is constructed in 56 seconds. For delay parameters of $(2,2)$, SENSE constructs the symbolic model of the NCS in 29 seconds. Remark that the construction of the symbolic model of the plant in SCOTS is affected by the increase in the dimension more than in SENSE. This is due to the nature of SCOTS which operates element-by-element on the state space to construct an over-approximation of the reachable sets as a step while constructing the finite abstraction of the plant. Unlike SCOTS, SENSE implements the $\mathscr{L}$-operator as operations on the BDD representing the symbolic model of the plant. The tool SENSE synthesizes the controller in 75883 seconds. Figure 7 shows the closed-loop simulation in MATLAB. The initial state of the system is $(7,15,15,7)$.

References

[1] C. Baier & J. P. Katoen (April 2008): *Principles of model checking*. The MIT Press.

[2] M. B. G. Cloosterman, N. van de Wouw, W. P. M. H. Heemels & H. Nijmeijer (2009): *Stability of Networked Control Systems With Uncertain Time-Varying Delays*. IEEE Transactions on Automatic Control 54(7), pp. 1575–1580, doi:10.1109/TAC.2009.2015543.

[3] W. P. M. H. Heemels & N. van de Wouw (2010): *Stability and Stabilization of Networked Control Systems*. In Alberto Bemporad, Maurice Heemels & Mikael Johansson, editors: Networked Control Systems, Springer London, London, pp. 203–253, doi:10.1007/978-0-85729-033-5_7.

[4] M. Khaled, M. Rungger & M. Zamani (2016): *Symbolic models of networked control systems: A feedback refinement relation approach*. In: 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton), pp. 187–193, doi:10.1109/ALLERTON.2016.7852228.

[5] R. Majumdar & M. Zamani (2012): *Approximately Bisimilar Symbolic Models for Digital Control Systems*. In P. Madhusudan & Sanjit A. Seshia, editors: Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 362–377, doi:10.1007/978-3-642-31424-7_28.

[6] O. Maler, A. Pnueli & J. Sifakis (1995): *On the synthesis of discrete controllers for timed systems*. In Ernst W. Mayr & Claude Puech, editors: STACS 95, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 229–242, doi:10.1007/3-540-59042-0_76.

[7] D. Nesic & D. Liberzon (2009): *A Unified Framework for Design and Analysis of Networked and Quantized Control Systems*. IEEE Transactions on Automatic Control 54(4), pp. 732–747, doi:10.1109/TAC.2009.2014930.

[8] OpenSim-Ltd.: *OMNeT++, a Discrete Event Simulator*. Available at `https://omnetpp.org`.

[9] A. J. Pretorius & J. J. van Wijk: *StateVis, Multidimensional visualization of transition systems*. Available at `http://www.win.tue.nl/vis1/home/apretori/statevis/`.

[10] G. Reissig, A. Weber & M. Rungger (April 2017): *Feedback Refinement Relations for the Synthesis of Symbolic Controllers*. IEEE Transactions on Automatic Control 62(4), pp. 1781–1796, doi:10.1109/TAC.2016.2593947.

[11] Matthias Rungger & Majid Zamani (2016): *SCOTS: A Tool for the Synthesis of Symbolic Controllers*. In: Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC '16, ACM, New York, NY, USA, pp. 99–104, doi:10.1145/2883817.2883834.

[12] F. Somenzi (2015): *CUDD: CU Decision Diagram Package*, 3.0.0 edition. Available at `http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf`.

[13] P. Tabuada (2009): *Verification and control of hybrid systems, A symbolic approach*. Springer US, doi:10.1007/978-1-4419-0224-5.

[14] W. Thomas (1995): *On the synthesis of strategies in infinite games*. In Ernst W. Mayr & Claude Puech, editors: STACS 95, Springer Berlin Heidelberg, pp. 1–13, doi:10.1007/3-540-59042-0_57.

[15] Netherlands TU/e Technische Universiteit Eindhoven: *A short note on the FSM data format*. Available at `http://www.win.tue.nl/vis1/home/apretori/data/fsm.html`.

[16] N. van de Wouw, D. Nešić & W.P.M.H. Heemels (2012): *A discrete-time framework for stability analysis of nonlinear networked control systems*. Automatica 48(6), pp. 1144 – 1153, doi:10.1016/j.automatica.2012.03.005.

[17] M. Zamani, M. Mazo Jr, M. Khaled & A. Abate (accepted, to appear): *Symbolic Abstractions of Networked Control Systems*. IEEE Transactions on Control of Network Systems, doi:10.1109/TCNS.2017.2739645. Available at `https://arxiv.org/abs/1401.6396`.

[18] M. Zamani, M. Mazo & A. Abate (2014): *Finite abstractions of networked control systems*. In: 53rd IEEE Conference on Decision and Control, pp. 95–100, doi:10.1109/CDC.2014.7039365.

[19] M. Zamani, G. Pola, M. Mazo Jr. & P. Tabuada (2012): *Symbolic Models for Nonlinear Control Systems Without Stability Assumptions*. IEEE Transactions on Automatic Control 57(7), pp. 1804–1809, doi:10.1109/TAC.2011.2176409.

Appendix

The tool, its download links and the manuals are available through the following URL:

`https://www.hcs.ei.tum.de/en/software/sense`.

5 Instructions to reproduce the reported results

We provide instructions on how to reproduce the results in Table 1 and Figure 5 in the submitted paper. As expected from a tool paper, all experiments are **extensible**. We provide instructions on how to extend the examples or the tool itself. Table 2 expands the details of the case studies in Table 1. A state and an input of any plant are denoted by vectors $\xi = [\xi_1\ \xi_2\ \cdots\ \xi_n]^T$ and $\upsilon = [\upsilon_1\ \upsilon_2\ \cdots\ \upsilon_m]^T$, respectively, where n and m are their corresponding dimensions. Any other undefined symbols denote constants related to each case study.

5.1 Requirements and preliminaries

For the full benefits of `SENSE` and its helper tools, the following requirements are recommended:

- Any of the following C/C++ development environments:
  - Linux with GNU C/C++ build tools. We generated all results using Linux Ubuntu version 16.04.
  - Mac OS X with Xcode.app including the command line tools. We didn't run the tool on Mac OS. However, users should generally face no differences.
  - Windows with MSYS2 or with Ubuntu subsystem. Simulation and visualization are not supported.
- A working installation of the `CUDD` library. Users can download it from: `http://vlsi.colorado.edu/~fabio/CUDD/`. The package follows the usual `configure`, `make`, and `make install` installation routine. Please configure it as follows:
  `$./configure --enable-shared --enable-obj --enable-dddmp --prefix=PTH`,
  where `PTH` should be replaced by your installation path. We use `$(CUDDPATH)` to refer to the installation directory of the CUDD library.
- The tool `MATLAB` is used to simulate the closed-loop of NCS with the synthesized controllers. The `mex` compiler of `MATLAB` should be installed and ready to be used. Figure 5.a is generated in `MATLAB`.
- The tool `OMNeT++` is used to visualize and simulate the closed-loop interactions between the plant and the synthesized controller via non-ideal communication channels. Figure 5.b is generated in `OMNeT++`.

5.2 Instructions for working installation of `SENSE`

Please follow the following instructions to download, install and run one example of `SENSE`:

- Download `SENSE` from: `https://www.hcs.ei.tum.de/en/software/sense/download`.
- Navigate to `$(SENSEROOT)/examples/prolonged_ncs/dint/` and build the example DI.
- Run the command `$ make` in the terminal. Note that for a successful compilation of `dint.cc`, the variable `$(CUDDPATH)`, defined in the `Makefile`, needs to be modified to match the particular installation directory of the CUDD library.
- A successful compilation indicate, generally, the correct installation of `CUDD` and `SENSE`.

To run the example used in the installation, the BDD file of the symbolic model of the plant in the NCS needs to be available. For all provided examples, we make the required codes to generate those symbolic models available in a separate directory named `scots-files`. The directories contain programs that use

Table 2: Details of the case studies reported in Table 1

Case Study	Dynamics Equations	State set	State quantizers	Input set	Input quantizers	τ
DI	$\dot{\xi} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \xi + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \upsilon$	$[0, 3.2] \times [-1.5, 1.5]$	$(0.2, .3)$	$[-0.3, 0.3]$	0.2	0.3
Robot	$\dot{\xi} = \begin{bmatrix} \upsilon_1 \\ \upsilon_2 \end{bmatrix}$	$[0, 63] \times [0, 63]$	$(1, 1)$	$[-1, 1] \times [-1, 1]$	$(1, 1)$	1.0
Jet	$\dot{\xi} = \begin{bmatrix} -\xi_2 - \frac{3}{2}\xi_1^2 - \frac{1}{2}\xi_1^3 \\ \frac{1}{\omega^2}(\xi_1 - \upsilon_1) \end{bmatrix}$ $\omega = 1$	$[-1.9, 1.9] \times [-1.9, 1.9]$	$(0.025, 0.025)$	$[-2, 2]$	0.1	0.3
DC-DC	$\dot{\xi} = \left\{ \begin{bmatrix} \frac{-r_l}{x_l} & 0 \\ 0 & \frac{-1}{x_c(r_0+r_c)} \end{bmatrix} \xi \quad , \text{for } \upsilon = 1 \right\} + \left\{ \begin{bmatrix} \frac{-1}{x_l}(r_l + \frac{r_0 r_c}{r_0+r_c}) & \frac{-1}{x_l}\frac{r_0}{r_0+r_c} \\ \frac{1}{x_c}\frac{r_0}{r_0+r_c} & \frac{-1}{\xi_2}\frac{1}{r_0+r_c} \end{bmatrix} \xi \quad , \text{for } \upsilon = 2 \right\} + \begin{bmatrix} \frac{V_s}{x_l} \\ 0 \end{bmatrix}$ $r_0 = 1, r_l = 0.05, r_c = \frac{r_l}{10}, V_s = 1, x_l = 3, \text{and } x_c = 70$	$[1.15, 1.55] \times [5.45, 5.85]$	$(5 \times 10^{-4}, 5 \times 10^{-4})$	$[1, 2]$	1	0.5
Vehi	$\dot{\xi} = \begin{bmatrix} \upsilon_1 \cos(\alpha + \xi_3) \cos(\alpha)^{-1} \\ \upsilon_1 \sin(\alpha + \xi_3) \cos(\alpha)^{-1} \\ \upsilon_1 \tan(\upsilon_2) \end{bmatrix}$ $\alpha = \arctan(\tan(\upsilon_2)/2)$	$[0, 10] \times [0, 10] \times [-3.54, 3.54]$	$(0.2, 0.2, 0.2)$	$[-1, 1] \times [-1, 1]$	$(0.3, 0.3)$	0.3
Inver	$\dot{\xi} = \begin{bmatrix} \xi_2 \\ \frac{\upsilon_1 \cos\xi_1 - M_p L \xi_2^2 \sin\xi_1 \cos\xi_1 + (M_c+M_p) g \sin\xi_1}{-M_p L \cos(\xi_1)^2 + L(M_c+M_p)} \\ \xi_4 \\ \frac{L\xi_2 - g\sin\xi_1}{\cos(\xi_1)} \end{bmatrix}$ $M_c = 0.25, M_p = 0.5, L = 0.2, \text{and } g = 9.807$	$[-\frac{\pi}{4}, -\frac{\pi}{4}] \times [-\pi, \pi] \times [-4.5, 4.5] \times [-4.5, 4.5]$	$(0.05, 0.2, 0.3, 0.3)$	$[-1.5, 1.5]$	0.1	0.1

the provided library `SCOTS`to generate the required BDD files. Compiling and running the provided code, will generate the required symbolic abstractions of the plants in NCS.

5.3 Instructions to reproduce Table 1

We provide instructions on how to reproduce one of the case studies reported in Table 1. Other case studies follow the same instructions. We choose the Robot case study and the column corresponding to the pair of delays $(2, 2)$. Please note that `SENSE` enables a feature in the `CUDD` library named (`AutoDynamic`) leading to automatic reordering of BDD variables. Such feature enhances the performance. However, execution times have some randomness as the feature is based on a heuristic algorithm. Follow the next instructions to reproduce the results of the Robot case study in Table 1:

- Navigate to `$(SENSEROOT)/examples/prolonged_ncs/robot/sense-files`. Build and run the program `robot`. Figure 8 shows a screen-shot of how this should look like.
- Navigate to `$(SENSEROOT)/examples/prolonged_ncs/robot/`
- Using a text editor, open the file `robot_ncs.cc`. Change the defined values for the delays NSCMAX and NCAMAX to 2 and 2, respectively. Save and close the file.
- Build and run the example: `$ make; ./robot_ncs;`
  Figure 9 shows a screen-shot of how this should look like.
- Record the shown results and do the same for each case study to reproduce the complete table.

5.4 Instructions to reproduce Figure 5

5.4.1 Synthesizing a symbolic controller using `SENSE`

- Navigate to `$(SENSEROOT)/examples/prolonged_ncs/robot_infoft/`.
- Build and run the provided program and the following files will be generated:
  - `robot_rel.bdd`: contains the symbolic abstraction of the robot to be provided to `SENSE`.

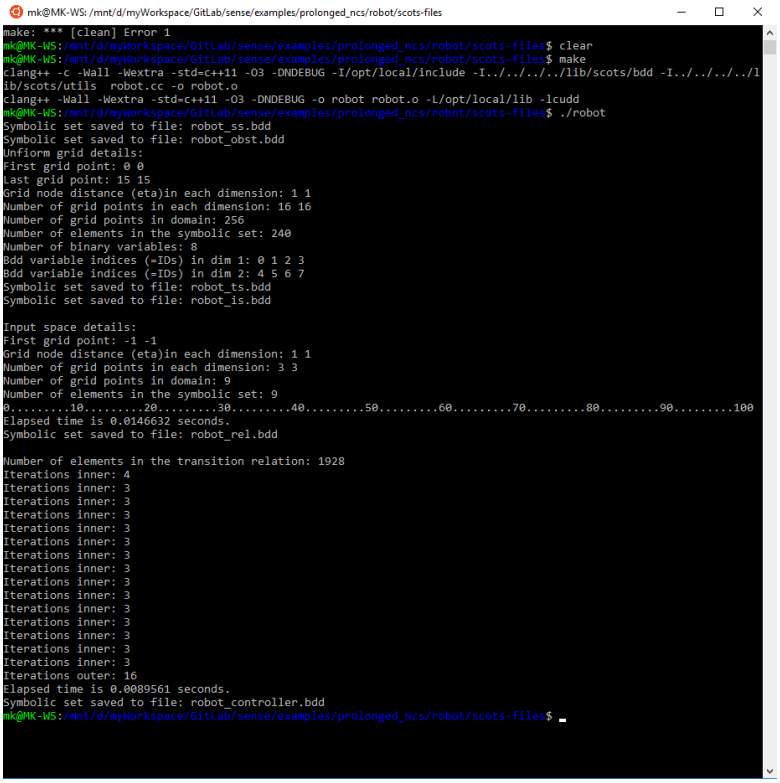


Figure 8: Screen-shot for the program provided to generate the input BDD files.

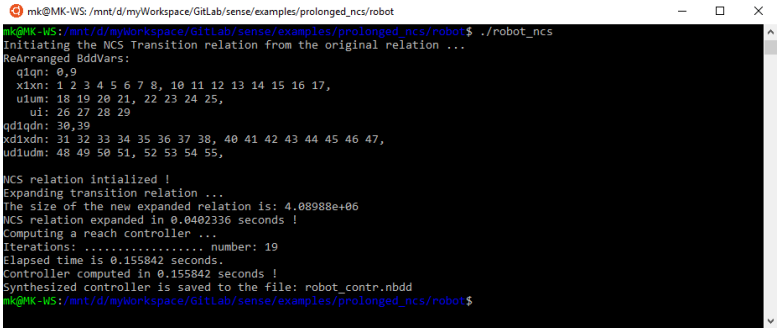


Figure 9: Screen-shot for the output of SENSE.

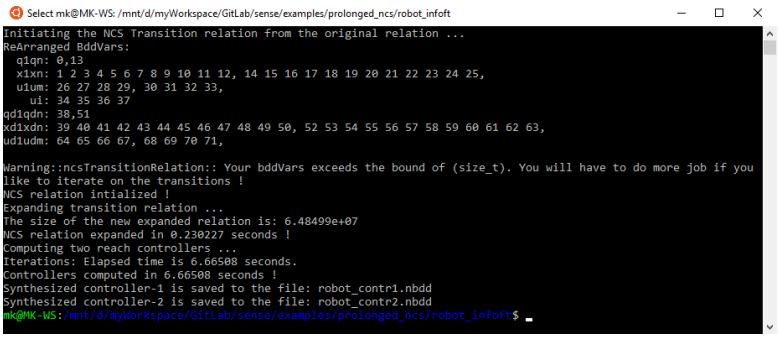


Figure 10: Screen-shot for building the C++simulation program.

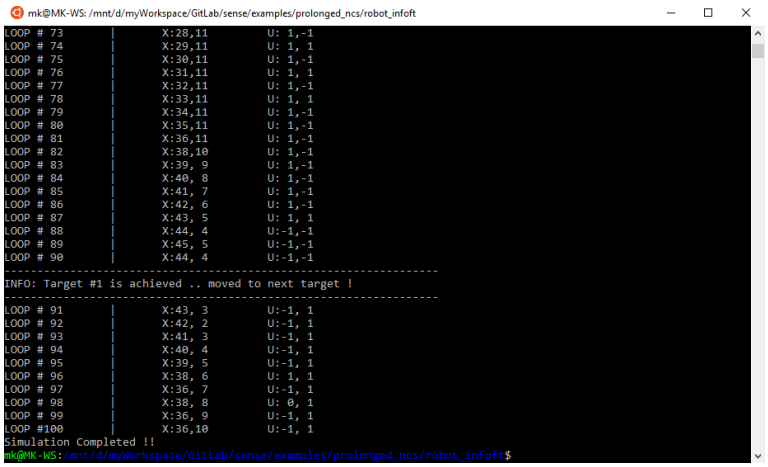


Figure 11: Screen-shot for running the C++simulation program for the Robot example.

  - `robot_ts1.bdd`: represents the atomic proposition `Target1`.
  - `robot_ts2.bdd`: represents the atomic proposition `Target2`.
- Open the file `robot_ncs.cc` and make sure the delays of the network are set to (2,2).
- We list part of the code here and give some notes:

```
...
#define NSCMAX 2
#define NCAMAX 2
#define BDD_REL "scots-files/robot_rel.bdd"
...
ncsFIFOTransitionRelation ncsRobot(cuddManger,  BDD_REL, 2, 2, NSCMAX, NCAMAX);
ncsRobot.ExpandBDD();
...
ComputeRecurrenceConjController(bdd_target_files);
...
```

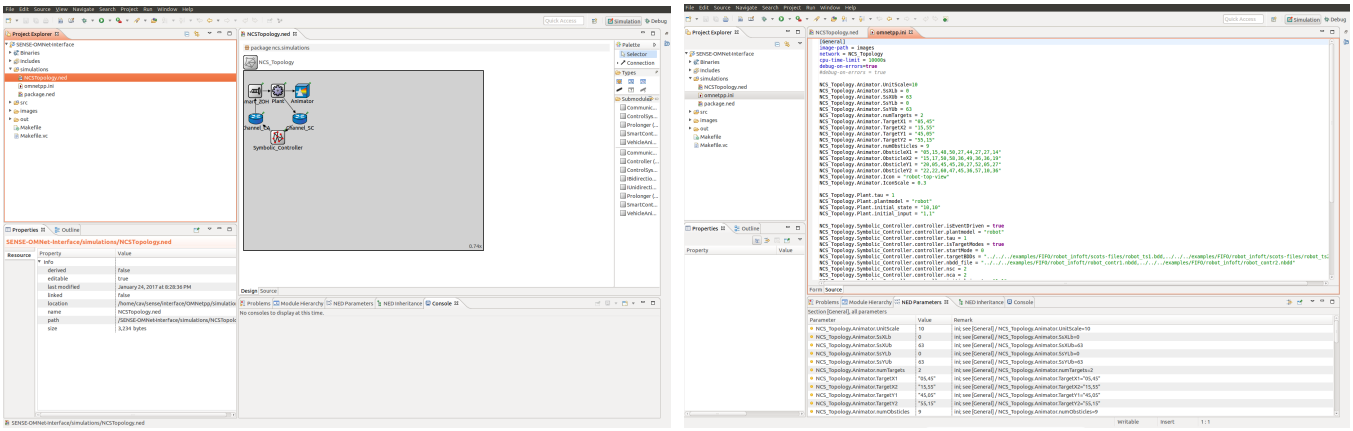
  Aside from the definitions of constants, the code starts by expanding the symbolic model `robot_rel.bdd` using the class `ncsFIFOTransitionRelation`, which is the construction-rules class reported in the paper. Then, controllers enforcing a conjunction of recurrence specifications are synthesized and saved to BDD files.
- Build and run the example: `$ make; ./robot_ncs;` . The result is two controller files: `robot_contr1.nbdd` and `robot_contr2.nbdd`, and Figure 10 depicts how this should look like.

5.4.2 Simulation using C++

For users who have no access to `MATLAB` and are not familiar with `OMNeT++`, we provide C++ simulation support. For this example, the simulation program is built automatically with the program if you already used the provided `Makefile`. Simply, run the program from the command line: `$ ./robot_ncs_sim`
The results will be shown in the terminal and should be as shown in Figure 11:

5.4.3 Simulation using `MATLAB` to reproduce Figure 5.a

Having `MATLAB` installed, follow the following steps:



(a) Screen-shot of the GUI of OMNeT++ for SENSE's project.

(b) Screen-shot of the (.ini) configuration file of the Robot example.

Figure 12: Sample of the C/C++ and VHDL codes as generated from SENSE for the robot example.

- Configure the `mex` compiler for C++ in MATLAB by executing:
  `>> mex -setup C++`
- Navigate to `$(SENSEROOT)/interface/matlab/` and compile the files `mexNcsController.cc` and `mexSymbolicSet.cc`. Edit the `Makefile` and modify the variables `$(MATLABROOT)` and `$(CUDDPATH)` to match the local installation and do: `$ make .`
- Open `MATLAB` and add the directory `$(SENSEROOT)/interface/MATLAB` to `MATLAB`'s path.
- In `MATLAB`, navigate to `$(SENSEROOT)/examples/prolonged_ncs/robot_infoft`.
- Run the provided m-file for simulation
  `>> robot_ncs`
- `MATLAB` should generate Figure 5.a.

5.4.4 Simulation using `OMNeT++` to reproduce Figure 5.b

Follow the following steps to reproduce Figure 5.b:

- Open the GUI of `OMNeT++` by executing the command: `$ omnetpp`
- From the top menu of `File`, click `Import` and select `General -> Existing Project into Workspace`. Select the provided `OMNeT++` project from the directory:
  `$(SENSEROOT)/interface/OMNETPP`. The GUI of `OMNeT++` looks as shown in Figure 12a.
- Copy the file `omnetpp.ini` from the root folder of the robot example
  `$(SENSEROOT)/examples/prolonged_ncs/robot_infoft` to the simulation folder of `OMNeT++`:
  `$(SENSEROOT)/interface/OMNETPP/simulations`.
- Open the file in `OMNeT++` to get familiar with its code-style, as depicted in Figure 12b.
  The configurations in the (.ini) file are simple hierarchical assignments in the form:
  `PARENT-MODULE.MODULE = VALUE`. The parent module is `NCS_Topology` as specified by the tool. Submodules describe the basic modules in the NCS. For example, in the configuration lines:
  `NCS_Topology.Channel_SC.min_delay = 0`
  `NCS_Topology.Channel_SC.max_delay = 0.6 ,`
  we specify the minimum and maximum delays, in seconds, for both of the communication channels.
- Make sure that the paths in the (.ini) file point, correctly, to the generated controller and the target/obstacle files.

- Build the simulation project from inside OMNeT++: Right-click on the project SENSE-OMNet-Interface and select Build Project.
- From inside OMNeT++, run the simulation: Right-click on the project SENSE-OMNet-Interface and select Run As and then OMNeT++ Simulation.
- An interactive visualization will be shown. You may adjust the screen by zooming in/out before starting the simulation using the shortcuts (CTRL-M and CTRL-N). Press F5 to start the simulation.

5.5 Instructions to generate the C/C++ or VHDL/Verilog codes

- We assume that you have already built and ran the Robot example.
- Navigate to $(SENSEROOT)/tools/bdd2implement/examples/. You will find two folders: c-sense-controller: an example to generate the C/C++code for the Robot case study and vhdl-sense-controller: an example to generate the VHDL code for the Robot case study.
- Navigate to any of the two examples and build/run it.
- A controller code is automatically generated and ready for implementation.

6 Extensibility in SENSE

6.1 Adding more examples

Writing and simulating your own example in SENSE is straightforward. Users can take a copy of existing example and build on it or define their own ones by following the steps:

- Make sure to produce the BDD files for the plant inside the NCS.
- Make a folder for your new example with at least one (.cc) to hold the code.
- You can copy the Makefile from any example to your example's directory and change the target file.
- Write your code in the (.cc) file to read the original BDD files and to synthesize the required controller.
- Build and run your code to generate your controller.
- Use either MATLAB or OMNeT++ interfaces to simulate your generated controller.

6.2 Adding new NCS setting classes

Users are encouraged to read the manual for a detailed description of adding new NCS-construction rules classes. However, we summaries the extension process in the following steps:

- In the directory $(SENSEROOT)/src, create your own (.hh) file to represent the new configuration class. You can also add the code of your class in the file ncsTransitionRelation.hh.
- Inherit the base class ncsTransitionRelation and provide implementation for the following virtual methods:
  - getSourceStateTemplate(): to help the base class understand the structure of the state of the symbolic model of new NCS class.
  - ConstructQTransitionRules(): to help the base class handle dummy in transitions due to the initialization of the NCS.
  - ConstructNonQTransitionRules(): to help the base class handle normal transitions.
- Create a new folder and examples for this class of NCS in the examples directory as shown before.