Synthesizing SystemC Code from Delay Hybrid CSP

Gaogao Yan[1,2], Li Jiao[1], Shuling Wang[1(✉)], and Naijun Zhan[1,2(✉)]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
{yangg,ljiao,wangsl,znj}@ios.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China

Abstract. Delay is omnipresent in modern control systems, which can prompt oscillations and may cause deterioration of control performance, invalidate both stability and safety properties. This implies that safety or stability certificates obtained on idealized, delay-free models of systems prone to delayed coupling may be erratic, and further the incorrectness of the executable code generated from these models. However, automated methods for system verification and code generation that ought to address models of system dynamics reflecting delays have not been paid enough attention yet in the computer science community. In our previous work, on one hand, we investigated the verification of delay dynamical and hybrid systems; on the other hand, we also addressed how to synthesize SystemC code from a verified hybrid system modelled by Hybrid CSP (HCSP) without delay. In this paper, we give a first attempt to synthesize SystemC code from a verified delay hybrid system modelled by Delay HCSP (dHCSP), which is an extension of HCSP by replacing ordinary differential equations (ODEs) with delay differential equations (DDEs). We implement a tool to support the automatic translation from dHCSP to SystemC.

Keywords: Delay dynamical systems · Approximate bisimulation · Code generation · Delay hybrid CSP · SystemC

1 Introduction

Model-Driven Design (MDD) is considered as an effective way of developing reliable complex embedded systems (ESs), and has been successfully applied in industry [17,20], therefore drawn increasing attentions recently. A challenging problem in MDD is to transform a verified abstract model at high-level step by step to more concrete models at lower levels, and to executable code at the end. To make sure that the final code generated in MDD is correct and reliable,

This work is partially supported by "973 Program" under grant No. 2014CB340701, by NSFC under grants 61625206, 61732001 and 91418204, by CDZ project CAP (GZ 1023), and by the CAS/SAFEA International Partnership Program for Creative Research Teams.

B.-Y.E. Chang (Ed.): APLAS 2017, LNCS 10695, pp. 21–41, 2017.
https://doi.org/10.1007/978-3-319-71237-6_2

the transformation process must be guaranteed to preserve consistency between observational behaviors of the models at different levels in a rigorous way. However, this is difficult, due to the inherent complexity of most ESs, especially for hybrid systems, which contain complicated behaviour, like continuous and discrete dynamics, and the complex interactions between them, time-delay, and so on, while code only contains discrete actions. Obviously, the exact equivalence between them can never be achieved, due to the unavoidable error of discretization of continuous dynamics of hybrid systems.

As an effective way for analyzing hybrid systems and their discretization, approximate bisimulation [14] can address the above problem. Instead of requiring observational behaviors of two systems to be exactly identical, it allows errors but requires the distance between two systems remains bounded by some precisions. In our previous work [24], we used Hybrid CSP (HCSP), an extension of CSP by introducing ordinary differential equations (ODEs) for modelling continuous evolutions and interrupts for modelling interaction between continuous and discrete dynamics, as the modelling language for hybrid systems; and then, we extended the notion of approximate bisimulation to general hybrid systems modelled as HCSP processes; lastly, we presented an algorithm to discretize an HCSP process (a control model) by a discrete HCSP process (an algorithm model), and proved that they are approximately bisimilar if the original HCSP process satisfies the globally asymptotical stability (GAS) condition. Here the GAS condition requires the ODEs starting from any initial state can always infinitely approach to its equilibrium point as time proceeds [8]. Recently, in [26], we further considered how to discretize an HCSP process without GAS, and refine the discretized HCSP process to SystemC code, which is approximately bisimilar to the original HCSP process in a given bounded time.

On the other hand, in practice, delay is omnipresent in modern control systems. For instance, in a distributed real-time control system, control commands may depend on communication with sensors and actuators over a communication network introducing latency. This implies that safety or stability certificates obtained on idealized, delay-free models of systems prone to delayed coupling may be erratic, and further the incorrectness of the code generated from these models. However, automated methods for system verification and code generation that ought to address models of system dynamics reflecting delays have not been paid enough attention yet in the computer science community.

Zou *et al.* proposed in [28] a safe enclosure method to automatic stability analysis and verification of delay differential equations by using interval-based Taylor over-approximation to enclose a set of functions by a parametric Taylor series with parameters in interval form. Prajna *et al.* extended the barrier certificate method for ODEs to the polynomial time-delay differential equations setting, in which the safety verification problem is formulated as a problem of solving sum-of-square programs [23]. Huang *et al.* presents a technique for simulation based time-bounded invariant verification of nonlinear networked dynamical systems with delayed interconnections by computing bounds on the sensitivity of trajectories (or solutions) to changes in initial states and inputs of

the system [18]. A similar simulation method integrating error analysis of the numeric solving and the sensitivity-related state bloating algorithms was proposed in [11] to obtain safe enclosures of time-bounded reach sets for systems modelled by DDEs.

However, in the literature, there is few work on how to refine a verified ES model with delay to executable code in MDD. In this paper, we address this issue, and the main contributions can be summarized as follows:

- First of all, we extend HCSP by allowing delay, called Delay HCSP (dHCSP), which is achieved by replacing ODEs with DDEs in HCSP. Obviously, HCSP is a proper subset of dHCSP as all ODEs can be seen as specific DDEs in which time delay is zero. Then, we propose the notion of *approximately bisimilar* over dHCSP processes.
- In [11], the authors presented an approach to discretizing a DDE by a sequence of states corresponding to discrete time-stamps and meanwhile the error bound that defines the distance from the trajectory is computed automatically on-the-fly. As a result, by adjusting step size of the discretization, the given precision can be guaranteed. Inspired by their work, we consider how to discretize a dHCSP process S such that the discretized dHCSP process is approximately bisimilar to S. This is done by defining a set of rules and proving that any dHCSP process S and its discretization are approximately bisimilar within bounded time with respect to the given precision.
- Finally, we present a set of code generation rules from discrete dHCSP to executable SystemC code and prove the equivalence between them.

We implement a prototypical tool to automatically transform a dHCSP process to SystemC code and provide some case studies to illustrate the above approach. Due to space limitation, the proofs of theorems in this paper are available in [25].

1.1 Related Work

Generating reliable code from control models is a dream of embedded engineering but difficult. For some popular models such as Esterel [10], Statecharts [16], and Lustre [15], code generation is supported. However, they do not take continuous behavior into consideration. Code generation is also supported in some commercial tools such as Simulink [2], Rational Rose [1], and TargetLink [3], but the correctness between the model and the code generated from it is not formally guaranteed, as they mainly focus on the numerical errors. The same issue exists in SHIFT [12], a modelling language for hybrid automata. Generating code from a special hybrid model, CHARON [5], was studied in [6,7,19]. Particularly, in order to ensure the correctness between a CHARON model and its generated code, a formal criteria *faithful implementation* is proposed in [7], but it can only guarantee the code model is under-approximate to the original hybrid model. The main difference between the above works and ours lies in that the delayed dynamics is considered for the code generation from hybrid models in our work.

For the discretization of DDEs, we can refer to some existing works which focus on the verification of systems containing delayed differential dynamics. In [28], a method for analyzing the stability and safety of a special class of DDEs was proposed, which cannot deal with the mixed ODE-DDE form. In [22], the authors proposed a method for constructing a symbolic model from an incrementally input-to-state stable (δ-ISS) nonlinear time-delay system, and moreover proved the symbolic model and the original model are approximately bisimilar. After that, they proved the same result for the incrementally input-delay-to-state stable (δ-IDSS) nonlinear time-delay system with unknown and time-varying delays in [21]. Unfortunately, the δ-ISS and δ-IDSS condition are difficult to check in practice. A simulation-based method is proposed in [18] for computing an over-approximate reachable set of a time-delayed nonlinear networked dynamical system. Within this approach, a significant function (i.e., the IS discrepancy function), used for bounding the distance between two trajectories, is difficult to find for general dynamical systems. In [11], a further extension of [18] that can handle any kind of DDEs with constant time delays is introduced, which can be appropriately used for the discretization of DDEs in dHCSP. But no work is available on how to generate executable code from a verified model with delay.

The rest of this paper is organized as: Some preliminary notions on DDEs and SystemC are introduced in Sect. 2. Section 3 extends HCSP to dHCSP and defines the approximate bisimulation on dHCSP. In Sect. 4, the discretization of dHCSP processes is presented and the correctness of the discretization is proved. The translation from discrete dHCSP to SystemC code is presented in Sect. 5. In Sect. 6, a case study is provided to illustrate our approach. Section 7 concludes the paper and discusses the future work.

2 Preliminaries

In this section, we introduce some preliminary knowledge that will be used later.

2.1 Delay Dynamical Systems

For a vector $\mathbf{x} \in \mathbb{R}^n$, $\|\mathbf{x}\|$ denotes its L^2 norm, i.e., $\|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$. Given a vector $\mathbf{x} \in \mathbb{R}^n$ and $\epsilon \in \mathbb{R}_0^+$, $N(\mathbf{x}, \epsilon)$ is defined as the ϵ-neighbourhood of $\mathbf{x}$, i.e., $N(\mathbf{x}, \epsilon) = \{\mathbf{y} \in \mathbb{R}^n | \|\mathbf{x} - \mathbf{y}\| \leq \epsilon\}$. Then, for a set $S \subseteq \mathbb{R}^n$, $N(S, \epsilon)$ is defined as $N(S, \epsilon) = \bigcup_{\mathbf{x} \in S} \{\mathbf{y} \in \mathbb{R}^n | \|\mathbf{x} - \mathbf{y}\| \leq \epsilon\}$, and $conv(S)$ is denoted as the convex hull of S. If S is compact, $dia(S) = sup_{\mathbf{x}, \mathbf{x}' \in S} \|\mathbf{x} - \mathbf{x}'\|$ denotes its diameter.

In this paper, we consider delay dynamical systems governed by the form:

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{x}(t - r_1), ..., \mathbf{x}(t - r_k)), & t \in [0, \infty) \\ \mathbf{x}(t) = \mathbf{g}(t), & t \in [-r_k, 0] \end{cases} \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the state, $\dot{\mathbf{x}}(t)$ denotes the derivative of $\mathbf{x}$ with respect to t, and $\mathbf{x}(t) = \mathbf{g}(t)$ is the *initial condition*, where $\mathbf{g}$ is assumed to be $\mathrm{C}^0[-r_k, 0]$. Without loss of generality, we assume the delay terms are ordered as $r_k > ... > r_1 > 0$.

A function $X(\cdot) : [-r_k, \nu) \to \mathbb{R}^n$ is said to be a *trajectory* (solution) of (1) on $[-r_k, \nu)$, if $X(t) = \mathbf{g}(t)$ for all $t \in [-r_k, 0]$ and $\dot{X}(t) = \mathbf{f}(X(t), X(t - r_1), ..., X(t-r_k))$ for all $t \in [0, \nu)$. In order to ensure the existence and uniqueness of the maximal trajectory from a continuous initial condition $\mathbf{g}(t)$, we assume $\mathbf{f}$ is continuous for all arguments, and moreover $\mathbf{f}$ is continuously differentiable in the first argument (i.e., $\mathbf{x}(t)$). Then, we write $X(t, \mathbf{g}(t_0))$ with $t_0 \in [-r_k, 0]$ to denote the point reached at time t from the initial state $\mathbf{g}(t_0)$, which should be uniquely determined. Moreover, if $\mathbf{f}$ is Lipschitz, i.e., there exists a constant $L > 0$ s.t. $\|\mathbf{f}(\mathbf{x}) - \mathbf{f}(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\|$ holds for all $\mathbf{x}, \mathbf{y}$, we can conclude $X(\cdot)$ is unique over $[-r_k, \infty)$. Please refer to [9] for the theories of *delay differential equations*.

2.2 SystemC

SystemC is a system-level modelling language supporting both system architecture and software development. It provides a uniform platform for the modelling of complex embedded systems. Essentially it is a set of C++ classes and macros. According to the naming convention of SystemC, most identifiers are prefixed with *SC_* or *sc_*, such as *SC_THREAD*, *SC_METHOD*, *sc_inout*, *sc_signal*, *sc_event*, etc.

Modules, denoted by *SC_MODULE*, are the basic blocks of a SystemC model. A model usually contains several modules, within which sub-designs, constructors, processes, ports, channels, events and other elements may be included. Each module is defined as a class. The constructor of a module is denoted as *SC_CTOR()*, in which some initialization operations carry out. Processes are member functions of the module, describing the actual functionality, and multiple processes execute concurrently in nature. A process has a list of sensitive events, by whose notifications its execution is controlled. Two major types of processes, *SC_METHOD* and *SC_THREAD*, are supported in SystemC. Generally, an *SC_METHOD* can be invoked multiple times, whereas an *SC_THREAD* can only be invoked once.

Ports in SystemC are components using for communicating with each other between modules. They are divided into three kinds by the data direction, i.e., *sc_in*, *sc_out* and *sc_inout* ports. Only ports with the same data type can be connected (via channels). *Channels* are used for connecting different sub-designs, based on which the communication is realized (by calling corresponding methods in channels, i.e., *read()* and *write()*). Channels are declared by $sc\_signal\langle\rangle$. Another important element using for synchronization is *event*, which has no value and no duration. Once an event occurs, the processes waiting for it will be resumed. Generally, an event can be notified immediately, one delta-cycle (defined in the execution phase below) later, or some constant time later.

The simulation of a SystemC model starts from the entrance of a method named *sc_main*(), in which three phases are generally involved: elaboration, execution and post-processing. During the elaboration and the post-processing phase, some initialization and result processing are carried out, respectively. We mainly illustrate the execution phase in the next.

The execution of SystemC models is event-based and it can be divided into four steps: (1) Initialization, executing all concurrent processes in an unspecified order until they are completed or suspended by a *wait()*; (2) Evaluation, running all the processes that are ready in an unspecified order until there are no more ready process; (3) Updating, copying the value of containers (e.g., channels) to the current location, then after that, if any event occurs, go back to step 2. Here, the cycle from evaluation to updating and then go back to evaluation is known as the delta-cycle; (4) Time advancing, if no more processes get ready currently, time advances to the nearest point where some processes will be ready. If no such point exists or the time is greater than a given time bound, the execution will terminate. Otherwise, go back to Step 2.

3 Delay Hybrid CSP (*d*HCSP)

In this section, we first extend HCSP with delay, and then discuss the notion of approximate bisimulation over *d*HCSP processes by extending the corresponding notion of HCSP defined in [24].

3.1 Syntax of *d*HCSP

*d*HCSP is an extension of HCSP by introducing DDEs to model continuous evolution with delay behavior. The syntax of *d*HCSP is given below:

$$
\begin{aligned}
P ::= {} & \text{skip} \mid x := e \mid \text{wait } d \mid ch?x \mid ch!e \mid P; Q \mid B \to P \mid \\
& P \sqcap Q \mid P^* \mid [\!]_{i\in I}(io_i \to Q_i) \mid \langle F(\dot{\mathbf{s}}(t), \mathbf{s}(t), \mathbf{s}(t-r_1), ..., \mathbf{s}(t-r_k)) = 0 \& B\rangle \mid \\
& \langle F(\dot{\mathbf{s}}(t), \mathbf{s}(t), \mathbf{s}(t-r_1), ..., \mathbf{s}(t-r_k)) = 0 \& B\rangle \trianglerighteq [\!]_{i\in I}(io_i \to Q_i) \\
S ::= {} & P_1 \| P_2 \| \ldots \| P_n \text{ for some} n \geq 1
\end{aligned}
$$

where $x, \mathbf{s}$ stand for variables and vectors of variables, respectively, B and e are Boolean and arithmetic expressions, d is a non-negative real constant, ch is a channel name, io_i stands for a communication event (i.e., either $ch_i?x$ or $ch_i!e$ for some x, e), $k \geq 0$ is an index and for each r_i, $r_i \in \mathbb{R}_0^+$, P, Q, P_i, Q_i are sequential process terms, and S stands for a *d*HCSP process term, that may be parallel. The informal meaning of the individual constructors is as follows:

- skip, $x := e$, wait d, $ch?x$, $ch!e$, $P; Q$, $[\!]_{i\in I}(io_i \to Q_i)$, $B \to P$, $P \sqcap Q$ and P^* are defined the same as in HCSP.
- $\langle F(\dot{\mathbf{s}}(t), \mathbf{s}(t), \mathbf{s}(t-r_1), ..., \mathbf{s}(t-r_k)) = 0 \& B\rangle$ is the time-delay continuous evolution statement. It forces the vector $\mathbf{s}$ of real variables to obey the DDE F as long as B, which defines the domain of $\mathbf{s}$, holds, and terminates when B turns false. Without loss of generality, we assume that the set of B is open, thus the escaping point will be at the boundary of B. The special case when $k = 0$ corresponds to an ODE that models continuous evolution without delay. The communication interrupt $\langle F(\dot{\mathbf{s}}(t), \mathbf{s}(t), \mathbf{s}(t-r_1), ..., \mathbf{s}(t-r_k)) = 0 \& B\rangle \trianglerighteq [\!]_{i\in I}(io_i \to Q_i)$ behaves like $\langle F(\dot{\mathbf{s}}(t), \mathbf{s}(t), \mathbf{s}(t-r_1), ..., \mathbf{s}(t-r_k)) = 0 \& B\rangle$, except that the continuous evolution is preempted as soon as one of the communications io_i takes place, which is followed by the respective Q_i. These two statements are the essential extensions of *d*HCSP from HCSP.

– For $n \geq 1$, $P_1 \| P_2 \| \ldots \| P_n$ builds a system in which n concurrent processes run independently and communicate with each other along the common channels connecting them.

To better understand dHCSP, we introduce delay behavior to the water tank system considered in [4,24].

Example 1. The system is a parallel composition of two components *Watertank* and *Controller*, modelled by *WTS* as follows:

$$\begin{aligned} WTS &\stackrel{\text{def}}{=} Watertank \| Controller \\ Watertank &\stackrel{\text{def}}{=} v := v_0; d := d_0; (v = 1 \rightarrow \\ &\quad \langle \dot{d}(t) = Q_{max} - \pi s^2 \sqrt{g(d(t) + d(t-r))} \rangle \trianglerighteq (wl!d \rightarrow cv?v); \\ &\quad v = 0 \rightarrow \langle \dot{d}(t) = -\pi s^2 \sqrt{g(d(t) + d(t-r))} \rangle \trianglerighteq (wl!d \rightarrow cv?v)^* \\ Controller &\stackrel{\text{def}}{=} y := v_0; x := d_0; (\text{wait } p; wl?x; \\ &\quad x \geq ub \rightarrow y := 0; x \leq lb \rightarrow y := 1; cv!y)^* \end{aligned}$$

where Q_{max}, π, s and g are system parameters, the control variable v can take two values, 1 or 0, which indicate the watering valve on the top of the tank is open or closed, respectively, d is the water level of the *Watertank* and its dynamics depends on the value of v. For each case, the evolution of d follows a DDE that is governed by both the current state and the past state r time ago. The time delay r accounts for time involved in communication between the watertank and the controller.

The system is initialized by an initial state, i.e., v_0 and d_0 for the controller variable and water level, respectively. wl and cv are channels connecting *Watertank* and *Controller* for transferring information (water level and control variable respectively) between them. In the *Controller*, the control variable y is updated with a period of p, and its value is decided by the water level read from the *Watertank* (x in *Controller*). If $x \geq ub$ holds, where ub is an upper bound, y is set to 0 (valve closed), else if $x \leq lb$ holds, where lb is a lower bound, y is set to 1 (valve open), otherwise, y keeps unchanged. Basically, starting from the initial state, *Watertank* and *Controller* run independently for p time, then *Watertank* sends the current water level to *Controller*, according to which the value of the control variable is updated and then sent back to *Watertank*, after that, a new period repeats. The goal of the system is to maintain the water level within a desired scope.

3.2 Semantics of dHCSP

In order to define an operational semantics of dHCSP, we use non-negative reals $\mathbb{R}^+$ to model time, and introduce a global clock *now* to record the time in the execution of a process. Different from ODE, the solution of a DDE at a given time is not a single value, but a time function. Thus, to interpret a process S, we first define a state ρ as the following mapping:

$$\rho : (Var(S) \rightarrow (Intv \rightarrow \mathbb{R}^n)) \cup (\{now\} \rightarrow \mathbb{R}^+)$$

where $Var(S)$ represents the set of state variables of S, and $Intv$ is a timed interval. The semantics of each state variable with respect to a state is defined as a mapping from a timed interval to the value set. We denote by $\mathcal{D}$ the set of such states. In addition, we introduce a flow H as a mapping from a timed interval to a state set, i.e. $H : Intv \to \mathcal{D}$ called *flow*, to represent the continuous flow of process S over the timed interval $Intv$.

A structural operational semantics of dHCSP is defined by a set of transition rules. Each transition rule has the form of $(P, \rho) \xrightarrow{\alpha} (P', \rho', H)$, where P and P' are dHCSP processes, α is an event, ρ, ρ' are states, H is a *flow*. It expresses that, starting from initial state ρ, by performing event α, P evolves into P', ends in state ρ', and produces the execution flow H. The label α represents events, which can be a discrete non-communication event, e.g. skip, assignment, or the evaluation of Boolean expressions, uniformly denoted by τ, or an external communication event $ch!c$ or $ch?c$, or an internal communication $ch.c$, or a time delay d, where $c \in \mathbb{R}, d \in \mathbb{R}^+$. When both $ch!c$ and $ch?c$ occur, a communication $ch.c$ occurs.

Before defining the semantics, we introduce an abbreviation for manipulating states. Given a state ρ, $d \in \mathbb{R}^+$, and a set of variables V, $\rho[V \Downarrow_d]$ means the clock takes progress for d time units, and the values of the variables in V at time $\rho(now)+d$ is defined as a constant function over timed interval $[\rho(now), \rho(now)+d]$. Precisely, for any t in the domain,

$$\rho[V \Downarrow_d](x)(t) \stackrel{\text{def}}{=} \begin{cases} \rho(x)(t) & \text{if } x \notin V \\ \rho(x)(\rho(now)) & \text{otherwise} \end{cases}$$

For space of limitation, we only present the transition rules for the time-delayed continuous evolution statement here, the rules for other constructors can be defined similarly to the ones in HCSP, see [27]. The first rule represents that the DDE evolves for d time units, while B always preserves true throughout the extended interval.

$$\frac{\begin{array}{c}\text{Assume } X : [0,\infty) \to ([-r,\infty) \to \mathbb{R}^{d(\mathbf{s})}) \text{ is the solution of } \langle F(\dot{\mathbf{s}}(t), ..., \mathbf{s}(t-r_k))=0 \& B\rangle \\ \text{with initial value } \mathbf{s}(t) = H(t)(\mathbf{s})(t) \text{ for } t \in [\rho(now)-r, \rho(now)] \text{ and} \\ \forall d > 0. \forall t \in [0,d), [\![B]\!]_L^{\rho[now \mapsto now+t, \mathbf{s} \mapsto X_t]} = \mathbf{True}\end{array}}{(\langle F(\dot{\mathbf{s}}(t), ..., \mathbf{s}(t-r_k))=0 \& B\rangle, \rho) \xrightarrow{d} \begin{pmatrix} \langle F(\dot{\mathbf{s}}(t), ..., \mathbf{s}(t-r_k))=0 \& B\rangle, \\ \rho[V \backslash \{\mathbf{s}\} \Downarrow_d][now \mapsto now+d, \mathbf{s} \mapsto X_d], H_d^{\rho,\mathbf{s},X} \end{pmatrix}}$$

where H is the initial history before executing the DDE (recording the past state of $\mathbf{s}$); and for any t, X_t is defined as a function over timed interval $[\rho(now), \rho(now)+t]$ such that $X_t(a) = X(t)(a-\rho(now))$ for each a in the domain; and the produced flow $H_d^{\rho,\mathbf{s},X}$ is defined as: $\forall t \in [\rho(now), \rho(now)+d]. H_d^{\rho,\mathbf{s},X}(t) = \rho[now \mapsto t, \mathbf{s} \mapsto X_{t-\rho(now)}]$.

The second rule represents that, when the negation $\neg B$ is true at the initial state, the DDE terminates.

$$\frac{[\![\neg B]\!]_L^\rho = \mathbf{True}}{(\langle F(\dot{\mathbf{s}}(t), ..., \mathbf{s}(t-r_k))=0 \& B\rangle, \rho) \xrightarrow{\tau} (\epsilon, \rho)}$$

3.3 Approximate Bisimulation on *d*HCSP

First of all, as a convention, we use $\overset{\alpha}{\twoheadrightarrow}$ to denote the τ transition closure of transition α, i.e., there is a sequence of τ actions before and/or after α. Given a state ρ defined over interval $[t_1, t_2]$, for each $t \in [t_1, t_2]$, we define $\rho \downarrow_t$ of type $Var(S) \cup \{now\} \to Val$ to restrict the value of each variable to the result of the corresponding function at time t:

$$\rho \downarrow_t (x) = \begin{cases} \rho(x)(t) & \text{for all } x \in Var(S) \\ \rho(x) & \text{for} x = now \end{cases}$$

With this function, we can reduce the operations manipulating a state with function values to the ones manipulating states with point values. Meanwhile, we assume $(S, \rho) \overset{0}{\twoheadrightarrow} (S, \rho)$ always holds for any process S and state ρ.

Definition 1 (Approximate bisimulation). *Suppose $\mathcal{B}$ is a symmetric binary relation on* d*HCSP processes such that S_1 and S_2 share the same set of state variables for $(S_1, S_2) \in \mathcal{B}$, and $\boldsymbol{d}$ is the metric of L^2 norm, and $h \in \mathbb{R}^+$ and $\varepsilon \in \mathbb{R}^+$ are the given time and value precision, respectively. Then, we say $\mathcal{B}$ is an approximately bisimulation w.r.t. h and ε, denoted by $\mathcal{B}_{h,\varepsilon}$, if for any $(S_1, S_2) \in \mathcal{B}_{h,\varepsilon}$, and (ρ_1, ρ_2) with $\boldsymbol{d}(\rho_1 \downarrow_{\rho_1(now)}, \rho_2 \downarrow_{\rho_2(now)}) \leq \varepsilon$, the following conditions are satisfied:*

1. *if $(S_1, \rho_1) \overset{\alpha}{\twoheadrightarrow} (S_1', \rho_1')$ and $\alpha \notin \mathbb{R}^+$, then there exists (S_2', ρ_2') such that $(S_2, \rho_2) \overset{\alpha}{\twoheadrightarrow} (S_2', \rho_2')$, $(S_1', S_2') \in \mathcal{B}_{h,\varepsilon}$ and $\boldsymbol{d}(\rho_1' \downarrow_{\rho_1'(now)}, \rho_2' \downarrow_{\rho_2'(now)}) \leq \varepsilon$, or there exist (S_2^*, ρ_2^*), (S_2', ρ_2') and $0 < t \leq h$ such that $(S_2, \rho_2) \overset{t}{\twoheadrightarrow} (S_2^*, \rho_2^*, H_2^*)$, $(S_2^*, \rho_2^*) \overset{\alpha}{\twoheadrightarrow} (S_2', \rho_2')$, $(S_1, S_2^*) \in \mathcal{B}_{h,\varepsilon}$, $\boldsymbol{d}(\rho_1 \downarrow_{\rho_1(now)}, \rho_2^* \downarrow_{\rho_2^*(now)} \leq \varepsilon$; $(S_1', S_2') \in \mathcal{B}_{h,\varepsilon}$ and $\boldsymbol{d}(\rho_1' \downarrow_{\rho_1'(now)}, \rho_2' \downarrow_{\rho_2'(now)}) \leq \varepsilon$.*
2. *if $(S_1, \rho_1) \overset{t}{\twoheadrightarrow} (S_1', \rho_1', H_1)$ for some $t > 0$, then there exist (S_2', ρ_2') and $t' \geq 0$ such that $|t - t'| \leq h$, $(S_2, \rho_2) \overset{t'}{\twoheadrightarrow} (S_2', \rho_2', H_2)$, $(S_1', S_2') \in \mathcal{B}_{h,\varepsilon}$, and for any $o \in [\rho_1(now), \rho_1(now) + \min(t, t')]$, $\boldsymbol{d}(\rho_1' \downarrow_o, \rho_2' \downarrow_o) \leq \varepsilon$, and $\boldsymbol{d}(\rho_1' \downarrow_{\rho_1'(now)}, \rho_2' \downarrow_{\rho_2'(now)}) \leq \varepsilon$.*

Definition 2. *Two* d*HCSP processes S_1 and S_2 are approximately bisimilar with respect to precision h and ε, denoted by $S_1 \cong_{h,\varepsilon} S_2$, if there exists an (h, ε)-approximate bisimulation relation $\mathcal{B}_{h,\varepsilon}$ s.t. $(S_1, S_2) \in \mathcal{B}_{h,\varepsilon}$.*

Theorem 1. *Given two* d*HCSP processes, it is decidable whether they are approximately bisimilar on $[0, T]$ for a given $T \in \mathbb{R}^+$.*

4 Discretization of *d*HCSP

The process on generating code from *d*HCSP is similar to that from HCSP [24], consisting of two phases: (1) discretization of the *d*HCSP model; (2) code generation from the discretized *d*HCSP model to SystemC.

Benefiting from its compositionality, dHCSP can be discretized by defining rules for all the constructors, in which the discretization of delay continuous dynamics (i.e., DDE) is the most critical. Let S be a dHCSP process, $T \in \mathbb{R}^+$ be a time bound, h and ε be the given precisions for time and value, respectively. Our goal is to construct a discrete dHCSP process $D_{h,\varepsilon}(S)$ from S, s.t. S is (h,ε)-approximately bisimilar to $D_{h,\varepsilon}(S)$ on $[0,T]$, i.e., $S \cong_{h,\varepsilon} D_{h,\varepsilon}(S)$ on $[0,T]$. To achieve this, we firstly introduce a simulation-based method (inspired by [11]) for discretizing a single DDE and then extend it for multiple DDEs to be executed in sequence; afterwards, we present the discretization of dHCSP in bounded time.

4.1 Discretization of DDE (DDEs) in Bounded Time

To solve DDEs is much more difficult than to solve ODEs, as DDEs are history dependent, therefore, non-Markovian, in contrast, ODEs are history independent and Markovian. So, in most cases, explicit solutions to DDEs are impossible, therefore, DDEs are normally solved by using approximation based techniques [9]. In [11], the authors propose a novel method for safety verification of delayed differential dynamics, in which a validated simulator for a DDE is presented. The simulator produces a sequence of discrete states for approximating the trajectory of a DDE and meanwhile calculates the corresponding local error bounds. Based on this work, we can obtain a validated discretization of a DDE w.r.t. the given precisions h and ε. Furthermore, we can easily extend the simulator to deal with systems containing multiple DDEs in sequence.

Next we first consider the discretization of a DDE within bounded time $T_d \in \mathbb{R}^+$, for some $T_d \leq T$. The purpose is to find a discrete step size h s.t. the DDE and its discretization are (h,ξ)-approximately bisimilar within $[0,T_d]$, for a given precision ξ that is less than the global error ε. For simplifying the notations, we consider a special case of DDE in which only one delay term, $r > 0$, exists, as in

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{x}(t-r)), & t \in [0,\infty) \\ \mathbf{x}(t) = \mathbf{g}(t), & t \in [-r,0] \end{cases} \tag{2}$$

where we use $\mathbf{f}(\mathbf{x},\mathbf{x}_r)$ to denote the dynamics, $\mathbf{x}$ for current state and $\mathbf{x}_r$ for the past state at $t-r$. In fact, the method for this special case can be easily extended to the general case as in (1), by recording the past states between $t-r_k$ and t, the detailed discussion can be found in [11].

For a DDE $\mathbf{f}(\mathbf{x},\mathbf{x}_r)$ with initial condition $\mathbf{g}(t)$ which is continuous on $[-r,0]$, delay term r, step size h, and time bound T_d, the validated simulator in [11] can produce three *lists* (denoted as $[\![\cdot]\!]$) with the same length, namely, (1) $\mathbf{t} = [\![t_{-m},...,t_0,t_1,...,t_n]\!]$, storing a sequence of time stamps on which the approximations are computed ($t_{-m},...,t_0$ for the time before 0, i.e., $[-r,0]$, with $m = r/h$), satisfying $t_{-m},...,t_{-1} < 0 = t_0 < t_1 < ... < t_n = T_d$ and $t_i - t_{i-1} = h$ for all $i \in [-m+1,n]$, (2) $\mathbf{y} = [\![\mathbf{x}_{-m},...,\mathbf{x}_0,\mathbf{x}_1,...,\mathbf{x}_n]\!]$, recording a sequence of approximate states of $\mathbf{x}$ starting from $\mathbf{x}_{-m}$, corresponding to time stamps in $\mathbf{t}$, (3) $\mathbf{d} = [\![d_{-m},...,d_0,d_1,...,d_n]\!]$, recording the corresponding sequence of local

error bounds. The implementation of the simulator is based on the well-known *forward Euler method*, i.e., $\mathbf{x} := \mathbf{x} + h\mathbf{f}(\mathbf{x}, \mathbf{x}_r)$. In addition, we usually require the delay term r be an integral multiple of the step size h, i.e., $m \in \mathbb{N}^+$, in order to ensure the past state $\mathbf{x}_r$ could be found in $\mathbf{y}$.

A remarkable property of the simulator

$$X(t, \mathbf{g}(0)) \in conv(N(\mathbf{x}_i, d_i) \cup N(\mathbf{x}_{i+1}, d_{i+1}))$$

holds for each $t \in [t_i, t_{i+1}]$ with $i = 0, 1, ..., n-1$, where $X(\cdot)$ is the trajectory of $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{x}_r)$, and $N(\mathbf{x}_i, d_i)$ is the d_i-neighbourhood of $\mathbf{x}_i$ ($\mathbf{x}_i$ and d_i are elements of $\mathbf{y}$ and $\mathbf{d}$, respectively). Based on this fact, we can use $\mathbf{x}_{i+1}$ as the approximation of $X(t, \mathbf{g}(0))$ for all $t \in [t_i, t_{i+1}]$ for any $i \in [0, n-1]$, s.t. the DDE (2) and the sequence $[\![\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_n]\!]$ are (h, ξ)-approximately bisimilar on $[0, T_d]$, if the diameter of every $conv(N(\mathbf{x}_i, d_i) \cup N(\mathbf{x}_{i+1}, d_{i+1}))$ is less than the precision ξ, i.e., $dia(conv(N(\mathbf{x}_i, d_i) \cup N(\mathbf{x}_{i+1}, d_{i+1}))) < \xi$ for all $i \in [0, n-1]$.

Theorem 2 (Approximation of a DDE). *Let Γ be a DDE as in (2), and $\mathbf{f}$ in (2) is continuously differentiable on $[0, T_d]$, and $\mathbf{x}_0 \in \mathbb{R}^n$ with $\|\mathbf{x}_0 - \mathbf{g}(0)\| \leq d_0$. Then for any precision $\xi > 0$ and $0 < d_0 < \xi$, there exists a step size $h > 0$ such that Γ and*

$$\mathbf{x} := \mathbf{x}_0; (wait\ h; \mathbf{x} := \mathbf{x} + h\mathbf{f}(\mathbf{x}, \mathbf{x}_r))^{\frac{T_d}{h}};$$

are (h, ξ)-approximately bisimilar on $[0, T_d]$.

Based on the simulation algorithm given in [11], we design a method for automatically computing a step size h s.t. the DDE as in (2) and its discretization are (h, ξ)-approximately bisimilar on $[0, T_d]$, as presented in Algorithms 1 and 2.

Algorithm 1. ComStepsize_oneDDE: computing the step size h for the one DDE

Input: The dynamics $\mathbf{f}(\mathbf{x}, \mathbf{x}_r)$, initial state $\mathbf{x}_0$, delay term r, precision ξ, and time bound T_d;

```
1: h = r; v = true; t = [[−h, 0]]; y = [[x0, x0]]; d = [[0, 0]];
2: while true do
3:    CheckStepsize(f(x, xr), r, h, ξ, [0, Td], t, y, d, v);
4:    if v = false then
5:       h = h/2; v = true;
6:       t = [[−h, 0]];
7:    else
8:       break;
9:    end if
10: end while
11: return h;
```

Algorithm 1 is designed for computing a valid step size h for a given DDE. It first initializes the value of h to r and Boolean variable v, which indicates whether

the current h is a valid step size, to *true*, and the lists for simulating the DDE, i.e., $\mathbf{t}, \mathbf{y}$, and $\mathbf{d}$ (line 1). Here, we assume the initial condition is a constant function, i.e., $\mathbf{x}_t = \mathbf{x}_0$, on $[-r, 0]$, therefore, states before time 0 are represented as one state at $-h$. Then, it iteratively checks whether the current value of h can make Theorem 2 hold, by calling the function *CheckStepsize* that is defined in Algorithm 2 (lines 2–10). If current h is not valid (v is set to *false* for this case), h is set to a smaller value, i.e., $h/2$, and v is reset to *true*, and $\mathbf{t}$ is reinitialized according to the new h (lines 4–6). Otherwise, a valid h is found, then the while loop exits (lines 7–9). The termination of the algorithm can be guaranteed by Theorem 2, thus a valid h can always be found and returned (line 11).

Algorithm 2. CheckStepsize: checking whether the step size h is valid for precision ξ

Input: The dynamics $\mathbf{f}(\mathbf{x}, \mathbf{x}_r)$, delay term r, step size h, precision ξ, time span $[T_1, T_2]$, boolean variable v, and simulation history $\langle \mathbf{t}, \mathbf{y}, \mathbf{d} \rangle$ before T_1;

1: $n = length(\mathbf{t})$; $m = r/h$;
2: **while** $\mathbf{t}(n) < T_2$ **do**
3: $\quad \mathbf{t}(n+1) = \mathbf{t}(n) + h$;
4: $\quad \mathbf{y}(n+1) = \mathbf{y}(n) + \mathbf{f}(\mathbf{y}(n), \mathbf{y}(n-m)) * h$;
5: $\quad \mathbf{e}(n) =$ **Find** minimum e s.t.

$$\begin{cases} \|\mathbf{f}(\mathbf{x} + t * \mathbf{f}, \mathbf{x}_r + t * \mathbf{g}) - \mathbf{f}(\mathbf{y}(n), \mathbf{y}(n-m))\| \leq e - \sigma, for \\ \forall t \in [0, h] \\ \forall \mathbf{x} \in N(\mathbf{y}(n), \mathbf{d}(n)) \\ \forall \mathbf{x}_r \in N(\mathbf{y}(n-m), \mathbf{d}(n-m)) \\ \forall \mathbf{f} \in N(\mathbf{f}(\mathbf{y}(n), \mathbf{y}(n-m)), e) \\ \forall \mathbf{g} \in N(\mathbf{f}(\mathbf{y}(n-m), \mathbf{y}(n-2m)), \mathbf{e}(n-m)); \end{cases}$$

6: $\quad \mathbf{d}(n+1) = \mathbf{d}(n) + h * \mathbf{e}(n)$;
7: $\quad$ **if** $\max(\mathbf{y}(n)+\mathbf{d}(n), \mathbf{y}(n+1)+\mathbf{d}(n+1)) - \min(\mathbf{y}(n)-\mathbf{d}(n), \mathbf{y}(n+1)-\mathbf{d}(n+1)) > \xi$ **then**
8: $\quad\quad v = false$;
9: $\quad\quad$ break;
10: $\quad$ **else**
11: $\quad\quad \mathbf{t} = [\![\mathbf{t}, \mathbf{t}(n+1)]\!]$; $\mathbf{y} = [\![\mathbf{y}, \mathbf{y}(n+1)]\!]$; $\mathbf{d} = [\![\mathbf{d}, \mathbf{d}(n+1)]\!]$;
12: $\quad\quad n = n + 1$;
13: $\quad$ **end if**
14: **end while**
15: return $\langle v, \mathbf{t}, \mathbf{y}, \mathbf{d} \rangle$;

Algorithm 2 implements function *CheckStepsize*, which is slightly different from the simulation algorithm given in [11]. The history of $\langle \mathbf{t}, \mathbf{y}, \mathbf{d} \rangle$ is added to the inputs, for simulating multiple DDEs in sequence. At the beginning, the variable n that stores the last recent simulation step is initialized as the length of current $\mathbf{t}$, and an offset m is set to r/h thus $\mathbf{y}(n-m)$, i.e., the $(n-m)$th

element of list $\mathbf{y}$, locates the delayed approximation at time $\mathbf{t}(n) - r$ (line 1). When current time (i.e., $\mathbf{t}(n)$) is less than the end of the time span (i.e., T_2), the lists $\mathbf{t}$, $\mathbf{y}$ and $\mathbf{d}$ are iteratively updated by adding new elements, until T_2 is reached (lines 2–14). In each iteration, firstly, the time stamp is added by the step size h and the approximate state at this time is computed by the *forward Euler method* (line 4), and then the local error bound $\mathbf{d}(n+1)$ is derived based on the local error slope $\mathbf{e}(n)$ (line 6), which is reduced to a constrained optimization problem (line 5) that can be solved by some solvers in Matlab or by some SMT solvers like iSAT [13] which can return a validated result, please refer to [11] for the details. After these values are computed, whether the diameter of the convex hull of the two adjacent approximate points at the time stamps $\mathbf{t}(n)$ and $\mathbf{t}(n+1)$ by taking their local error bounds into account is greater than the given error ξ is checked (lines 7–13). If the diameter is greater than ξ, the while loop is broken and v is set to *false* (lines 8–9), which means h will be reset to $h/2$ in Algorithm 1. Otherwise, h is valid for this simulation step and the new values of $\mathbf{t}$, $\mathbf{y}$ and $\mathbf{d}$ are added into the corresponding lists (lines 10–12), then a new iteration restarts until T_2 is reached. At last, the new values of v, $\mathbf{t}$, $\mathbf{y}$ and $\mathbf{d}$ are returned (line 15).

A *d*HCSP may contain multiple DDEs, especially for those to be executed in sequence in which the initial states of following DDEs may depend on the flows of previous DDEs. In order to handle such cases, we present Algorithm 3 for computing the global step size that meets the required precision ξ within bounded time T_d. Suppose a sequence of DDEs $\mathbf{f}_1(\mathbf{x}, \mathbf{x}_r), \mathbf{f}_2(\mathbf{x}, \mathbf{x}_r), \cdots, \mathbf{f}_k(\mathbf{x}, \mathbf{x}_r)$ is to be executed in sequence. For simplicity, assume all DDEs share the same delay term r, and the execution sequence of the DDEs is decided by a scheduler (*Schedule* in line 6). At the beginning, h and v are initialized as the delay term r and *true* respectively (line 1). Then, before the current time (i.e., $\mathbf{t}(end)$) reaches the end of the time span (i.e., T_d), a while loop is executed to check whether h satisfies the precision ξ, in which *ComStepsize_oneDDE* and *CheckStepsize* are called (lines 2–13). In each iteration, the three lists $\mathbf{t}$, $\mathbf{y}$ and $\mathbf{d}$ are initialised as before (line 3), then the valid h for the first DDE $\mathbf{f}_1(\mathbf{x}, \mathbf{x}_r)$ is computed by calling *ComStepsize_oneDDE* (line 4), where t_1 denotes the length of the execution time of $\mathbf{f}_1(\mathbf{x}, \mathbf{x}_r)$. Afterwards, for the following DDEs, an inner while loop to check whether the calculated h is within the error bound ξ is executed (lines 5–12). Thereof, which DDE should be executed is determined by *Schedule* (one DDE may be executed for multiple times), and the corresponding span of execution time is represented as $[t_{i-1}, t_i]$ for the i-th DDE (lines 6–7). If h is not valid for some DDE, i.e., $v = false$ (line 8), depending on the return value of *CheckStepsize* function, a new smaller h (i.e., $h/2$) is chosen and v is reset to *true*, then the inner *while* loop is broken (lines 8–11) and a new iteration restarts from time 0 with the new h (line 3); Otherwise, a valid h is found (line 13). Since we can always find small enough step size to make all DDEs meet the precision within $[0, T_d]$ by Theorem 2, Algorithm 3 is ensured to terminate (line 14).

Algorithm 3. ComStepsize_multiDDEs: computing the step size h for multiple DDEs

Input: A sequence of dynamics $\mathbf{f}_1(\mathbf{x}, \mathbf{x}_r), \mathbf{f}_2(\mathbf{x}, \mathbf{x}_r), ..., \mathbf{f}_k(\mathbf{x}, \mathbf{x}_r)$, initial state $\mathbf{x}_0$, delay term r, precision ξ, and time bound T_d (assume running from $\mathbf{f}_1(\mathbf{x}, \mathbf{x}_r)$);

$h = r$; $v = true$;
while $\mathbf{t}(end) < T_d$ **do**
  $\mathbf{t} = [\![-h, 0]\!]$; $\mathbf{y} = [\![\mathbf{x}_0, \mathbf{x}_0]\!]$; $\mathbf{d} = [\![0, 0]\!]$;
  $h = ComStepsize\_oneDDE(\mathbf{f}_1(\mathbf{x}, \mathbf{x}_r), \mathbf{x}_0, r, \xi, t_1)$;
  while $\mathbf{t}(end) < T_d$ **do**
    $i = Schedule(\mathbf{f}_1(\mathbf{x}, \mathbf{x}_r), \mathbf{f}_2(\mathbf{x}, \mathbf{x}_r), ..., \mathbf{f}_k(\mathbf{x}, \mathbf{x}_r))$;
    $CheckStepsize(\mathbf{f}_i(\mathbf{x}, \mathbf{x}_r), r, h, \xi, [t_{i-1}, t_i], \mathbf{t}, \mathbf{y}, \mathbf{d}, v)$;
    if $v = false$ **then**
      $h = h/2$; $v = true$;
      break;
    end if
  end while
end while
return h;

4.2 Discretization of *d*HCSP in Bounded Time

Now we can define the set of rules to discretize a given *d*HCSP process S and obtain a discrete *d*HCSP process $D_{h,\varepsilon}(S)$ such that they are (h, ε)-approximately bisimilar on $[0, T]$, for given h, ε and T. The rule for the discretization of DDE is given below, and other rules are the same as the ones for HCSP presented in [24].

$$\frac{\langle \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{x}_r) \& B \rangle}{\begin{array}{c}(N(B,\varepsilon) \wedge N'(B,\varepsilon) \to (\text{wait } h; \mathbf{x} := \mathbf{x} + h\mathbf{f}(\mathbf{x}, \mathbf{x}_r)))^{\frac{T}{h}}; \\ N(B,\varepsilon) \wedge N'(B,\varepsilon) \to \textbf{stop}\end{array}}$$

For a Boolean expression B, $N(B, \varepsilon)$ is defined as its ε-neighbourhood. For instance, $N(B, \varepsilon) = \{x | x > 2 - \varepsilon\}$ for $B = \{x | x > 2\}$. Then, $\langle \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{x}_r) \& B \rangle$ is discretized as follows: first, execute a sequence of assignments (T/h times) to $\mathbf{x}$ according to *Euler method*, i.e., $\mathbf{x} := \mathbf{x} + h\mathbf{f}(\mathbf{x}, \mathbf{x}_r)$, whenever $N(B, \varepsilon) \wedge N'(B, \varepsilon)$ holds, where $N'(B, \varepsilon) = N(B, \varepsilon)[\mathbf{x} \mapsto \mathbf{x} + h\mathbf{f}(\mathbf{x}, \mathbf{x}_r)]$, i.e., the value of $N(B, \varepsilon)$ at the next discretized step; then, if both $N(B, \varepsilon)$ and $N'(B, \varepsilon)$ still hold, but the time has already reached the upper bound T, the process behaves like **stop**, which indicates that the behavior after T will not be concerned.

4.3 Correctness of the Discretization

In order to ensure $D_{h,\varepsilon}(S)$ defined in Sect. 4.2 is approximately bisimilar to S, we need to put some extra conditions on S, i.e., requiring it to be robustly safe. The condition is similar to that in [24]. We define the $(-\epsilon)$-neighbourhood like the ϵ-neighbourhood, i.e., for a set $\phi \subseteq \mathbb{R}^n$ and $\epsilon \geq 0$, $N(\phi, -\epsilon) = \{\mathbf{x} | \mathbf{x} \in \phi \wedge \forall \mathbf{y} \in \neg\phi. \|\mathbf{x} - \mathbf{y}\| > \epsilon\}$. Intuitively, $\mathbf{x} \in N(\phi, -\epsilon)$ means $\mathbf{x}$ is inside ϕ and moreover

the distance between it and the boundary of ϕ is greater than ϵ. To distinguish the states of process S from those of dynamical systems, we use ρ (ρ_0 for initial state) to denote the states of S here. Below, the notion of a robustly safe system is given.

Definition 3 ((δ,ϵ)-robustly safe). *Let $\delta > 0$ and $\epsilon > 0$ be the given time and value precisions, respectively. A* d*HCSP process S is (δ,ϵ)-robustly safe with respect to a given initial state ρ_0, if the following two conditions hold:*

- *for every continuous evolution $\langle \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x},\mathbf{x}_r)\&B\rangle$ occurring in S, when S executes up to $\langle \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x},\mathbf{x}_r)\&B\rangle$ at time t with state ρ, if $\rho(B) = false$, and there exists $\widehat{t} > t$ with $\widehat{t} - t < \delta$ and $\mathbf{d}(\rho, \rho_0[\mathbf{x} \mapsto X(\widehat{t}, \rho_0(\mathbf{x})])) < \epsilon$, then $\rho \in N(\neg B, -\epsilon)$;*
- *for every alternative process $B \to P$ occurring in S, if B depends on continuous variables of S, then when S executes up to $B \to P$ at state ρ, $\rho \in N(B,-\epsilon)$ or $\rho \in N(\neg B,-\epsilon)$.*

Intuitively, the (δ,ϵ)-robustly safe condition ensures the difference, between the violation time of the same Boolean condition B in S and $D_{h,\varepsilon}(S)$, is bounded. As a result, we can choose appropriate values for δ, ϵ, h and ε s.t. S and $D_{h,\varepsilon}(S)$ can be guaranteed to have the same control flows, and furthermore the distance between their "jump" time (the moment when Boolean condition associated with them becomes false) can be bounded by h. Finally the "approximation" between the behavior of S and $D_{h,\varepsilon}(S)$ can be guaranteed. The range of both δ and ϵ can be estimated by simulation.

Based on the above facts, we have the main theorem as below.

Theorem 3 (Correctness). *Let S be a* d*HCSP process and ρ_0 the initial state at time 0. Assume S is (δ,ϵ)-robustly safe with respect to ρ_0. Let $0 < \varepsilon < \epsilon$ be a precision and $T \in \mathbb{R}^+$ a time bound. If for any DDE $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x},\mathbf{x}_r)$ occurring in S, $\mathbf{f}$ is continuously differentiable on $[0,T]$, and there exists h satisfying $h < \delta < 2h$ if $\delta > 0$ such that Theorem 2 holds for all $\mathbf{f}$ in S, then $S \cong_{h,\varepsilon} D_{h,\varepsilon}(S)$ on $[0,T]$.*

Notice that for a given precision ε, there may not exist an h satisfying the conditions in Theorem 3. It happens when the DDE fails to leave far enough away from the boundary of its domain B in a limited time. However, for the special case that $\delta = 0$, we can always find a sufficiently small h such that $S \cong_{h,\varepsilon} D_{h,\varepsilon}(S)$ on $[0,T]$.

5 From Discretized *d*HCSP to SystemC

For a dHCSP process S, its discretization $D_{h,\varepsilon}(S)$ is a model without continuous dynamics and therefore can be implemented with an algorithm model. In this section, we illustrate the procedure for automatically generating a piece of SystemC code, denoted as $SC(D_{h,\varepsilon}(S))$, from a discretized dHCSP process $D_{h,\varepsilon}(S)$, and moreover ensure they are "equivalent", i.e., bisimilar. As a result,

Table 1. Part of rules for code generation of *d*HCSP

$x := e$	$\rightarrow$	$x = e; wait(SC\_ZERO\_TIME);$
wait d	$\rightarrow$	$wait(d, SC\_TU);$
$D_{h,\varepsilon}(P); D_{h,\varepsilon}(Q)$	$\rightarrow$	$SC(D_{h,\varepsilon}(P)); SC(D_{h,\varepsilon}(Q));$
$B \rightarrow D_{h,\varepsilon}(P)$	$\rightarrow$	$if(B)\{SC(D_{h,\varepsilon}(P));\}$
$D_{h,\varepsilon}(P) \sqcap D_{h,\varepsilon}(Q)$	$\rightarrow$	$if(rand()\%2)\{SC(D_{h,\varepsilon}(P));\}\ else\{SC(D_{h,\varepsilon}(Q));\}$
$(D_{h,\varepsilon}(P))^*$	$\rightarrow$	$while(i <= num(P^*))\{\ SC(D_{h,\varepsilon}(P)); i++;\}$

for a given precision ε and time bound T, if there exists h such that Theorem 3 holds, i.e., $S \cong_{h,\varepsilon} D_{h,\varepsilon}(S)$ on $[0, T]$, we can conclude that the generated SystemC code $SC(D_{h,\varepsilon}(S))$ and the original *d*HCSP process S are (h, ε)-approximately bisimilar on $[0, T]$.

Based on its semantics, a *d*HCSP model that contains multiple parallel processes is mapped into an *SC_MODULE* in SystemC, and each parallel component is implemented as a thread, e.g., $D_{h,\varepsilon}(P_1) \| D_{h,\varepsilon}(P_2)$ is mapped into two concurrent threads, $SC\_THREAD(SC(D_{h,\varepsilon}(P_1)))$ and $SC\_THREAD(SC(D_{h,\varepsilon}(P_2)))$, respectively. For each sequential process, i.e., $D_{h,\varepsilon}(P_i)$, we define the corresponding rule for transforming it into a piece of SystemC code, according to the type of $D_{h,\varepsilon}(P_i)$.

In Table 1, parts of generation rules are shown for different types of the sequential process $D_{h,\varepsilon}(P_i)$. For $x := e$, it is mapped into an equivalent assignment statement (i.e., $x = e$), followed by a statement $wait(SC\_ZERO\_TIME)$ for making the update valid. For wait d, it is straightforward mapped into a statement wait(d, SC_TU), where SC_TU is the time unit of d, such as *SC_SEC* (second), *SC_MS* (millisecond), *SC_US* (microsecond), etc. The sequential composition and alternative statements are defined inductively. Nondeterminism is implemented as an *if-else* statement, in which $rand()\%2$ returns 0 or 1 randomly. A *while* statement is used for implementing the repetition constructor, where $num(P^*)$ returns the upper bound of the repeat times for P.

In order to represent the communication statement, additional channels in SystemC (i.e., *sc_signal*) and events (i.e., *sc_event*) are introduced to ensure the synchronization between the input side and output side. Consider the discretized input statement, i.e., $ch? := 1; ch?x; ch? := 0$, Boolean variable $ch?$ is represented as an *sc_signal* (i.e., *ch_r*) with Boolean type, and moreover additional *sc_event* (i.e., *ch_r_done*) is imported to represent the completion of the action that reads values from channel *ch*. As a result, the SystemC code generated from it is defined as: first, Boolean signal *ch_r* is initialized as 1, which means channel *ch* is ready for reading (lines 2–3); then, the reading process waits for the writing of the same channel from another process until it has done (lines 4–6); after that, it gets the latest value from the channel and assigns it to variable x (lines 7–8); at last, it informs the termination of its reading to other processes and resets *ch_r* to 0 (lines 9–11). Here, there are two sub-phases within the second phase (lines 4–6): first, deciding whether the corresponding writing side is ready (line 4), if not (i.e., $ch\_w = 0$), the reading side keep waiting until

the writing side gets ready, i.e., $ch\_w = 1$ (line 5); afterwards, the reading side will wait for another event which indicates that the writing side has written a new value into the channel ch (line 6), for ensuring the synchronization.

```
1  // code for input statement
2  ch_r=1;
3  wait(SC_ZERO_TIME);
4  if(!ch_w)
5       wait(ch_w.posedge_event());
6  wait(ch_w_done);
7  x=ch.read();
8  wait(SC_ZERO_TIME);
9  ch_r_done.notify();
10 ch_r=0;
11 wait(SC_ZERO_TIME);
```

The discretized continuous statement is mapped into two sequential parts in SystemC. For the first part, i.e., $(N(B,\varepsilon) \wedge N'(B,\varepsilon) \rightarrow (\text{wait } h; \mathbf{x} := \mathbf{x}+h\mathbf{f}(\mathbf{x},\mathbf{x}_r)))^{\frac{T}{h}}$, a **for** loop block is generated (lines 2–8), in which a sequence of *if* statements, corresponding to Boolean condition $(N(B,\varepsilon) \wedge N'(B,\varepsilon)$, are executed (lines 3–7). Within every conditional statement, a **wait** statement and an assignment statement (based on *Euler method*) are sequentially performed (lines 4–6). Here, $N(B,e)$, $N\_p(B,e)$ and $f(x,x\_r)$ are helper functions (implemented by individual functions) that are generated from $N(B,\varepsilon)$, $N'(B,\varepsilon)$ $(e = \varepsilon$ here) and $\mathbf{f}(\mathbf{x},\mathbf{x}_r)$, respectively. For the second part, i.e., $N(B,\varepsilon) \wedge N'(B,\varepsilon) \rightarrow$ **stop**, it is mapped into a **return** statement guarded by a condition that is identical with that in line 3 (lines 9–10).

```
1  // code for delayed continuous statement
2  for(int i=0;i<T/h;i++){
3       if(N(B,e)&&N_p(B,e)){
4            wait(h,SC_TU);
5            x=x+h*f(x,x_r);
6            wait(SC_ZERO_TIME);
7       }
8  }
9  if(N(B,e)&&N_p(B,e)){
10      return;
11 }
```

For space limitation, the rest of the code generation rules can be found in [25]. Thus now, for a given discretized dHCSP process $D_{h,\varepsilon}(S)$, we can generate its corresponding SystemC implementation $SC(D_{h,\varepsilon}(S))$. Furthermore, their "equivalence" can be guaranteed by the following theorem.

Theorem 4. *For a* d*HCSP process* S*,* $D_{h,\varepsilon}(S)$ *and* $SC(D_{h,\varepsilon}(S))$ *are bisimilar.*

6 Case Study

In this section, we illustrate how to generate SystemC code from dHCSP through the example of water tank in Exmaple 1. As discussed above, for a given dHCSP process, the procedure of code generation is divided into two steps: (1) compute the value of step size h that can ensure the original dHCSP process and its discretization are approximately bisimilar with respect to the given precisions; (2) generate SystemC code from the discretized dHCSP process. We have implemented a tool that can generate code from both HCSP and dHCSP processes[1].

Continue to consider Exmaple 1. For given h, ε and T, by using the discretized rules, a discretization system $WTS_{h,\varepsilon}$ is obtained as follows:

$$
\begin{aligned}
WTS_{h,\varepsilon} &\stackrel{\text{def}}{=} Watertank_{h,\varepsilon} \,\|\, Controller_{h,\varepsilon} \\
Watertank_{h,\varepsilon} &\stackrel{\text{def}}{=} v := v_0; d := d_0; (v = 1 \to (wl! := 1; (wl! \wedge \neg wl? \to \\
&\quad (wait\ h; d(t+h) = d(t) + h(Q_{max} - \pi s^2 \sqrt{g(d(t) + d(t-r))}))^{\frac{T}{h}}; \\
&\quad wl! \wedge wl? \to (wl!d; wl! := 0; cv? := 1; cv?v; \\
&\quad cv? := 0); wl! \wedge \neg wl? \to \textbf{stop}); \\
&\quad v = 0 \to (wl! := 1; (wl! \wedge \neg wl? \to (wait\ h; \\
&\quad d(t+h) = d(t) + h(-\pi s^2 \sqrt{g(d(t) + d(t-r))}))^{\frac{T}{h}}; \\
&\quad wl! \wedge wl? \to (wl!d; wl! := 0; cv? := 1; cv?v; \\
&\quad cv? := 0); wl! \wedge \neg wl? \to \textbf{stop}))^* \\
Controller_{h,\varepsilon} &\stackrel{\text{def}}{=} y := v_0; x := d_0; (wait\ p; wl? := 1; wl?x; \\
&\quad wl? := 0; x \geq ub \to y := 0; x \leq lb \to y := 1; \\
&\quad cv! := 1; cv!y; cv! := 0)^*
\end{aligned}
$$

Given $Q_{max} = 2.0$, $\pi = 3.14$, $s = 0.18$, $g = 9.8$, $p = 1$, $r = 0.1$, $lb = 4.1$, ub=5.9, $v_0 = 1$ and $d_0 = 4.5$, we first build an instance of WTS (the Watertank_delay.hcsp file). Then, according to the simulation result, we can estimate that the valid scope of δ and ϵ for WTS is $\delta = 0$ and $\epsilon \leq 0.217$, respectively. By Theorem 3, we can infer that a discretized time step h must exist s.t. WTS and $WTS_{h,\varepsilon}$ are (h, ε)-approximately bisimilar, with $\varepsilon \leq \epsilon$. For given values of ε and time bound T, e.g., $\varepsilon = 0.2$ and $T = 10$, we obtain $h = 0.025$ (by Algorithm 3 in Sect. 4.1) s.t. Theorem 3 holds, i.e., $WTS \cong_{h,\varepsilon} D_{h,\varepsilon}(WTS)$ on $[0, 10]$. After that, we can automatically generate SystemC code equivalent to $D_{h,\varepsilon}(WTS)$ (by calling HCSP2SystemC.jar).

The comparison of the results, i.e., the curves of the water level (d in the figure), which are acquired from the simulation of the original dHCSP model and the generated SystemC code respectively is shown in Fig. 1. The result on the whole time interval $[0, 10]$ is illustrated in Fig. 1(a), and the specific details around two vital points, i.e., 5 and 8, are shown in Fig. 1 (b) and (c), respectively.

[1] The tool and all examples for HCSP and dHCSP can be found at https://github.com/HCSP-CodeGeneration/HCSP2SystemC.

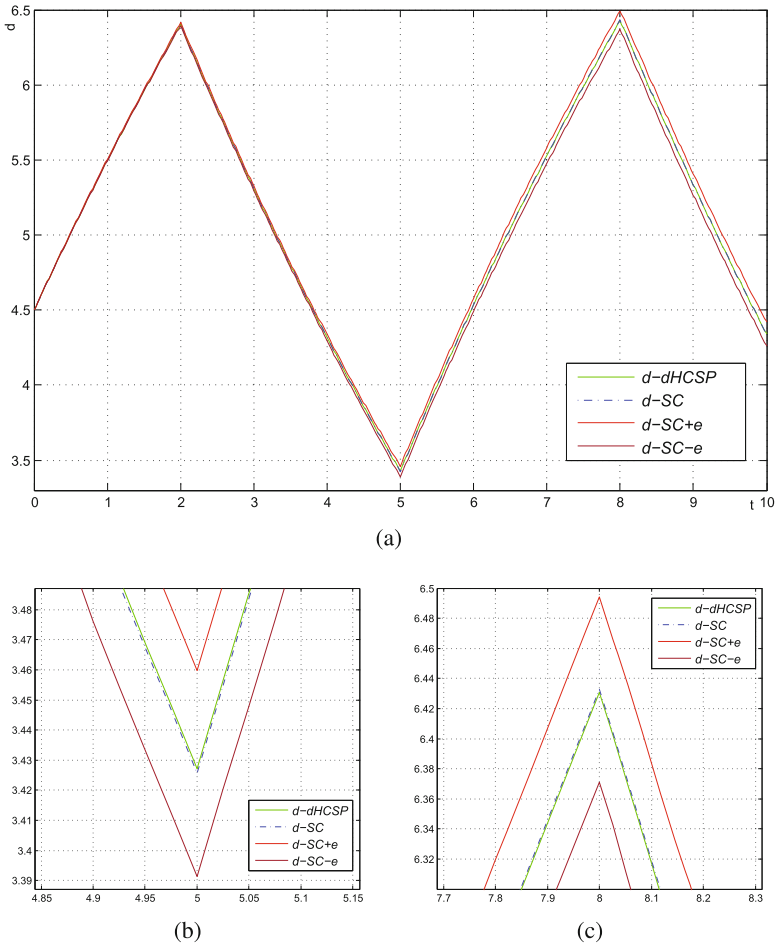

Fig. 1. The dHCSP model *vs.* the SystemC code of WTS (Color figure online).(a) The result on [0,10]; (b) Zoom in on the result around 5; (c) Zoom in on the result around 8.

In the figures, the simulation result (by calling the DDE solver *dde*23 in Matlab) is represented by green solid (i.e., d-d$HCSP$), and the result obtained by running the generated SystemC code is represented by blue dashed (i.e., d-SC). The upper bound (lower bound) of the SystemC result, by adding (subtracting) the local error bounds computed in Algorithm 3, is represented by red solid (dark red solid), i.e., d-SC+e (d-SC-e). As Fig. 1 shows, the results of simulation and SystemC code both always fall into the interval determined by the upper and lower error bounds, which indicates the correctness of the discretization. Moreover, the distance between the state of the simulation and the state of SystemC code is less than the required precision (i.e., $\varepsilon = 0.2$), in every interval of h length.

7 Conclusion

In this paper, we present an automatic translation from abstract *d*HCSP models to executable SystemC code, while preserving approximate equivalence between them within given precisions. As a modelling language for hybrid systems, *d*HCSP includes continuous dynamics in the form of DDEs and ODEs, discrete dynamics, and interactions between them based on communication, parallel composition and so on. In the discretization of *d*HCSP within bounded time, on one hand, based on our previous work, we discretize a DDE by a sequence of approximate discrete states and control the distance from the trajectory within a given precision, by choosing a proper discretized time step to make the error bound less than the precision; and on the other hand, by requiring the original *d*HCSP models to be robustly safe, we guarantee the consistency between the execution flows of the source model and its discretization in the sense of approximate bisimulation with respect to the given error tolerance.

As a future work, we will continue to transform from SystemC code into other practical programming languages, such as C, C++, java, etc. In addition, we also consider to apply our approach to more complicated real-world case studies.

References

1. Rational Rose. http://www-03.ibm.com/software/products/en/rosemod
2. Simulink. https://cn.mathworks.com/products/simulink.html
3. TargetLink. https://www.dspace.com/en/inc/home/products/sw/pcgs/targetli.cfm
4. Ahmad, E., Dong, Y., Wang, S., Zhan, N., Zou, L.: Adding formal meanings to AADL with hybrid annex. In: Lanese, I., Madelaine, E. (eds.) FACS 2014. LNCS, vol. 8997, pp. 228–247. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15317-9_15
5. Alur, R., Grosu, R., Hur, Y., Kumar, V., Lee, I.: Modular specification of hybrid systems in charon. In: Lynch, N., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 6–19. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46430-1_5
6. Alur, R., Ivancic, F., Kim, J., Lee, I., Sokolsky, O.: Generating embedded software from hierarchical hybrid models. In: LCTES 2003, pp. 171–182 (2003)
7. Anand, M., Fischmeister, S., Hur, Y., Kim, J., Lee, I.: Generating reliable code from hybrid-systems models. IEEE Trans. Comput. **59**(9), 1281–1294 (2010)
8. Angeli, D., et al.: A Lyapunov approach to incremental stability properties. IEEE Trans. Autom. Control **47**(3), 410–421 (2002)
9. Bellen, A., Zennaro, M.: Numerical Methods for Delay Differential Equations. Oxford University Press, Oxford (2013)
10. Berry, G.: The foundations of esterel. In: Proof, Language, and Interaction, Essays in Honour of Robin Milner, pp. 425–454 (2000)
11. Chen, M., Fränzle, M., Li, Y., Mosaad, P.N., Zhan, N.: Validated simulation-based verification of delayed differential dynamics. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 137–154. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_9

12. Deshpande, A., Göllü, A., Varaiya, P.: SHIFT: a formalism and a programming language for dynamic networks of hybrid automata. In: Antsaklis, P., Kohn, W., Nerode, A., Sastry, S. (eds.) HS 1996. LNCS, vol. 1273, pp. 113–133. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0031558
13. Fränzle, M., Herde, C., Ratschan, S., Schubert, T., Teige, T.: Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. J. Satisf. Boolean Model. Comput. **1**, 209–236 (2007)
14. Girard, A., Pappas, G.: Approximation metrics for discrete and continuous systems. IEEE Trans. Autom. Control **52**(5), 782–798 (2007)
15. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language lustre. In: Proceedings of the IEEE, pp. 1305–1320 (1991)
16. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
17. Henzinger, T.A., Sifakis, J.: The embedded systems design challenge. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 1–15. Springer, Heidelberg (2006). https://doi.org/10.1007/11813040_1
18. Huang, Z., Fan, C., Mitra, S.: Bounded invariant verification for time-delayed nonlinear networked dynamical systems. Nonlinear Anal. Hybrid Syst. **23**, 211–229 (2017)
19. Hur, Y., Kim, J., Lee, I., Choi, J.-Y.: Sound code generation from communicating hybrid models. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 432–447. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_29
20. Lee, E.: What's ahead for embedded software? Computer **33**(9), 18–26 (2000)
21. Pola, G., Pepe, P., Di Benedetto, M.: Symbolic models for nonlinear time-varying time-delay systems via alternating approximate bisimulation. Int. J. Robust Nonlinear Control **25**(14), 2328–2347 (2015)
22. Pola, G., Pepe, P., Di Benedetto, M., Tabuada, P.: Symbolic models for nonlinear time-delay systems using approximate bisimulations. Syst. Control Lett. **59**(6), 365–373 (2010)
23. Prajna, S., Jadbabaie, A.: Methods for safety verification of time-delay systems. In: CDC 2005, pp. 4348–4353 (2005)
24. Yan, G., Jiao, L., Li, Y., Wang, S., Zhan, N.: Approximate bisimulation and discretization of hybrid CSP. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 702–720. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_43
25. Yan, G., Jiao, L., Wang, S., Zhan, N.: Synthesizing SystemC code from delay hybrid CSP (full version). https://www.dropbox.com/s/rxbrib49bx1yv60/APLAS2017-FULL.pdf?dl=0
26. Yan, G., Jiao, L., Wang, L., Wang, S., Zhan, N.: Automatically generating SystemC code from HCSP formal models (Submitted)
27. Zhan, N., Wang, S., Zhao, H.: Formal Verification of Simulink/Stateflow Diagrams: A Deductive Way. Springer, New York (2016)
28. Zou, L., Fränzle, M., Zhan, N., Mosaad, P.N.: Automatic verification of stability and safety for delay differential equations. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 338–355. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_20