Formally Verified C Code Generation from Hybrid Communicating Sequential Processes (Full Version)

1$^{\text{st}}$ Shuling Wang
State Key Lab. of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
wangsl@ios.ac.cn

2$^{\text{nd}}$ Zekun Ji
State Key Lab. of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
jizk@ios.ac.cn

3$^{\text{rd}}$ Bohua Zhan
State Key Lab. of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
bzhan@ios.ac.cn

4$^{\text{th}}$ Xiong Xu
State Key Lab. of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
xux@ios.ac.cn

5$^{\text{th}}$ Qiang Gao
State Key Lab. of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
gaoq@ios.ac.cn

6$^{\text{th}}$ Naijun Zhan
State Key Lab. of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
znj@ios.ac.cn

***Abstract*—Hybrid Communicating Sequential Processes (HCSP) is a formal model for hybrid systems, including primitives for evolution along an ordinary differential equation (ODE), communication, and parallel composition. Code generation is needed to convert HCSP models into code that can be executed in practice, and the correctness of this conversion is essential to ensure that the generated code accurately reflects the formal model. In this paper, we propose a code generation algorithm from HCSP to C with POSIX library for concurrency. The main difficulties include how to bridge the gap between the synchronized communication model in HCSP and the use of mutexes for synchronization in C, and how to discretize evolution along ODEs and support interrupt of ODE evolution by communication. To prove the correctness of code generation, we define a formal semantics for POSIX C, and build transition system models for both HCSP and C programs. We then define an approximate bisimulation relation between traces of transition systems, and show that under certain robustness conditions for HCSP, the generated C program is approximately bisimilar to the original model. Finally, we evaluate the code generation algorithm on a detailed model for automatic cruise control, showing its utility on real-world examples.**

***Index Terms*—hybrid systems, multi-threaded C, approximate bisimulation, code generation**

I. Introduction

Cyber-Physical Systems (CPSs) can be complex, networked, systems of systems, and are often entrusted with safety-critical tasks. The efficient and verified development of safe and reliable CPSs is a priority mandated by many standards, yet a notoriously difficult and challenging field. To address design complexity under the necessity of verified CPS development, Model-Driven Development (MDD) has become a predominant approach in CPS development. MDD usually comprises different abstraction levels from top to bottom, e.g. graphical models, formal models, and code at the implementation level. There are two orthogonal principles followed by MDD:

- **Horizontal** Composition/Decomposition: $P\|Q$.
- **Vertical** Abstraction/Refinement: $P \sqsubseteq Q$.

The horizontal dimension requires a modelling mechanism that is compositional such that the complexity of modelling and verifying the system can be reduced by separately verifying its subcomponents. On the vertical dimension, specific modelling and analysis are performed at each abstraction level, and each level is refined by the more concrete level so that the behaviors are preserved. Therefore, if formal models are proved correct, the code generated from them is also guaranteed to be correct without further proof.

Especially, the MDD of hybrid systems that integrate traditional discrete models with dynamic models faces a central problem: how to transform an abstract hybrid control model to an algorithmic model at code level rigorously and automatically. The controller code determines how to sample data from the continuous plant and the entanglement between sampling data and computing control commands is intricate. An efficient approach is to discretize the continuous plant, and then, the discretized continuous plant together with the controller code constitute an embedded real-time implementation for the closed-loop hybrid system. How and according to which criterion to discretize the continuous behaviour and then generating correct C code, is addressed in this paper.

There are many industrial MDD tools targeting CPS design and development, such as Simulink/Stateflow [Mat13a], [Mat13b], SCADE [Dor08] and so on. Simulink/Stateflow and SCADE both support automatic code generation from control models, targeting at real-time applications. However, Simulink/Stateflow can only guarantee correctness of generated code by (incomplete) simulations. SCADE was founded on the synchronous dataflow language Lustre [HCRP91], and its formally verified compiler Vélus [BBD$^+$17] guarantees that the generated code faithfully implements the semantics of Lustre. However, it lacks support for continuous plants modelling. In the academic community, there are also a number of studies on formal modelling and verification of CPSs, for instance, hybrid and timed automata [Hen96], [FKL$^+$18], [WZA18], hybrid programs and dynamic differential logic [Pla18], Event-B and its hybridation [DASP21], and hybrid process algebra [CR05], [BM05]. Most of these approaches address verification only, while Event-B supports code generation with formal guarantee but only for discrete case.

Hybrid CSP (HCSP) [He94], [ZWZ17], a hybrid extension to the classic CSP (Communicating Sequential Processes) [Hoa78], is a compositional formalism in describing hybrid systems. It uses ordinary differential equations (ODEs) to model continuous evolutions and introduces interrupts to model interactions between continuous and discrete dynamics in a very flexible manner. The verification of HCSP is conducted by tools based on Hybrid Hoare Logic (HHL) [ZZW$^+$23], including verification within the interactive theorem prover Isabelle/HOL [ZZW$^+$13], [ZWZ17] as well as a more automatic tool HHLPy [SBZ23]. Graphical models for CPSs can be translated into HCSP [ZZW$^+$13], [XWZ$^+$22], and the verified HCSP can then be transformed to executable SystemC code [YJW$^+$20]. To guarantee the correctness of transformation from HCSP to SystemC, the notion of approximate bisimulation proposed by [GP07], [JDBP09] was used to measure the equivalence between hybrid and discrete systems, to allow a distance between the observations of two systems within a tolerable bound rather than exactly identical, which was proved to hold for HCSP and SystemC under some robustly-safety conditions in [YJW$^+$20].

This paper builds upon the work of Yan et al. in [YJW$^+$20]. It aims at generating code in C from HCSP models using the concurrency primitives of the POSIX pthreads library, with correctness guarantees. It adopts the notions of approximate bisimulation, robustly-safety, and some of the discretisation rules of HCSP proposed in [YJW$^+$20]. However, it is distinguished from previous work in the following aspects:

1) Although both works transform HCSP to its discretized version and then to code, in [YJW$^+$20], the discretization of the HCSP constructs related to communications, including inputs, outputs and continuous interrupt, is defined with the help of shared channel variables indicating their readiness to perform communication actions (see Table 2 and 3 of [YJW$^+$20]); and this work does not need to introduce extra shared variables at this step, thus keeps communications the same as original HCSP.
2) SystemC supports communication mechanisms similar to HCSP on its own right, so the communications in HCSP can be translated naturally to SystemC. Compared to this, communications are implemented in C using the POSIX pthreads library, in terms of mutexes and condition variables to achieve time and value synchronization. Hence, the method of translation to C is more involved. At semantic level, concurrency in HCSP follows a handshaking model while in C it follows an interleaving model controlled by mutexes and condition variables. Thus, the two concurrency mechanisms in HCSP and C are completely different, and how to prove the equivalence between them is one of the main challenges addressed in our work.

We prove the approximate bisimulation between an HCSP process and the generated C code in two steps: approximate bisimulation between the HCSP process and its discretised version, and bisimulation between the discrete HCSP process and the generated C code. Compared to the proofs of [YJW$^+$20], the first part proves the case for continuous evolution and the case for continuous interrupt based on its new discretization, and the second part is completely new: we introduce a new bisimulation relation, for which each transition step in HCSP may correspond to multiple atomic blocks of execution on the C side, but only one of them is considered as the essential step to perform the transition.

In summary, the main contributions of this paper comprise[1]:

1) We present a formal semantics for a subset of C language with POSIX threads.
2) We implement the synchronized communications of HCSP with the use of mutexes in the pthreads library, based on which we realise the transformation from any HCSP process to C code.
3) Based on the notions of approximate bisimulation, we prove the correctness of the transformation from HCSP to C. The proof uses a new bisimulation relation for verifying equivalence between two concurrency mechanisms with synchronized and interleaving settings.
4) We apply our approach on a realistic Automatic Cruise Control System, including its HCSP model, the C code generated from the model, and the comparison with original HCSP and other C implementation by simulation.

After reviewing related works, the paper is organized as follows: Sect. II introduces some preliminary knowledge of this work. Sect. III introduces the syntax and semantics of C with POSIX threads. The translation from HCSP to C is specified in Sect. IV, and the correctness of the translation is justified in Sect. V. Sect. VI illustrates our approach by a realistically-scaled case study. Sect. VII concludes.

[1] The source code and examples of the code generator are available at https://github.com/jimoc2048bits/CCodeGenerationFromHCSP.

A. Related Work

Model-based automatic code generation has been extensively studied in both academic and industrial communities [AFH$^+$10], but code generation that supports hybrid systems and provides formal correctness guarantees of generated code at the same time is not well addressed. Some examples include the aforementioned Simulink [Mat13a], SCADE [Dor08] founded on synchronous Lustre, and the formal modelling languages [Hen96], [FKL$^+$18], [WZA18]. OSATE/AADL [OSA17] provides architecture modeling and analysis of real-time systems and furthermore supports the automated code generation from AADL models including runtime behavior and scheduling to C code. However, it validates the code generation by simulation, and moreover does not support continuous time modeling. The compiler Zélus [BP13] extends Lustre [HCRP91] with ODEs and implements code generation from the extended hybrid language, which has also been implemented in SCADE 6. It supports analysis of hybrid models by type systems and semantics, and handles the detection of zero-crossing events [BP13], [BBCP12]. But it does not explicitly support constructs related to communication and concurrency. VeriPhy [BTM$^+$18] automatically transforms verified formal models of CPSs modelled in differential dynamic logic (dL) [Pla18], [FMQ$^+$15] to controller implementations that preserve safety properties of original models rather than their semantics. Thus compared to our work, it does not consider the (approximate) equivalence between the source models with ODEs and the discrete implementation, and the zero-crossing problems caused by discretization.

II. Preliminaries

This section introduces the notion of transition systems, approximate bisimulation, discretization of ODEs, and HCSP.

A. Transition Systems

Definition 1 (Transition system). *A transition system is a tuple $T = \langle Q, L, \rightarrow, Q^0, Y, H\rangle$, where Q is a set of states, $L \subseteq \mathcal{ACT} \cup \{\tau\}$ is a set of labels, $\rightarrow \subseteq Q \times L \times Q$ is a set of transitions, $Q^0 \subseteq Q$ is a set of initial states, Y is a set of observations, and $H : Q \rightarrow Y$ is an observation function. $\mathcal{ACT}$ is a set of events and τ is an internal event ($\tau \notin \mathcal{ACT}$).*

Given a transition system, for any $a \in \mathcal{ACT}$, we define the τ-closed transition $q \overset{a}{\Rightarrow} q'$ to represent that q can reach q' via action a and a sequence of τ actions, i.e.$q(\xrightarrow{\tau})^* q_i \xrightarrow{a} q_{i+1}(\xrightarrow{\tau})^* q'$. We will define the semantics of HCSP and C using transition systems. In many cases we set $\mathcal{ACT} = \emptyset$ (such as when modelling the semantics for C), then we will define $\rightarrow \subseteq Q \times Q$ instead, and $\Rightarrow$ is equivalent to $\rightarrow^*$.

B. Approximate Bisimulation

The notion of approximate bisimulation was first proposed in [GP07] to measure the equivalence between hybrid systems, to allow a limited distance between the observations of two systems. Later in [JDBP09], it was extended to allow precision not only between the observations, but also between the synchronisation labels of two systems. In [YJW$^+$20], the authors instantiate the precision parameters to be the time and value tolerances between two systems. Below we present the notion of approximate bisimulation in [YJW$^+$20]. Let $TS_i = \langle Q_i, L_i, \rightarrow_i, Q_i^0, Y_i, H_i\rangle$, $(i = 1, 2)$ be two transition systems, h and ε the time and value precisions resp.

Definition 2 (Approximate Bisimulation). *$\mathcal{B}_{h,\varepsilon} \subseteq Q_1 \times Q_2$ is called a (h,ε)-approximate bisimulation relation between TS_1 and TS_2, if it is symmetric, and for all $(q_1, q_2) \in \mathcal{B}_{h,\varepsilon}$,*

- *The distance between the observations is within the given value precision, i.e. $|H_1(q_1), H_2(q_2)| \leq \varepsilon$, where $|H_1(q_1), H_2(q_2)|$ returns the maximum of Euclidean distances of observation variables in q_1 and q_2.*
- *$\forall q_1 \xrightarrow{l_1}_1 q_1'$, $\exists q_2 \overset{l_2}{\Rightarrow}_2 q_2'$ s.t. $(q_1', q_2') \in \mathcal{B}_{h,\varepsilon}$, and $|l_1, l_2| < h$, for $l_1 \in L_1, l_2 \in L_2$. Here $|l_1, l_2|$ is 0 if $l_1 = l_2$, $|l_1 - l_2|$ if $l_1, l_2 \in R$, and ∞ otherwise.*

The (h,ε)-approximate bisimulation requires the distance between the observations under the pair (q_1, q_2) must be within ε; furthermore, if one of them is able to reach a state via an event, the other one can also reach a state via an event such that the distance between the events is within h and the pair of resulting states also satisfy the approximate bisimulation.

Definition 3. *$\mathcal{TS}_1$ and $\mathcal{TS}_2$ are approximate bisimilar with respect to h and ε, denoted $\mathcal{TS}_1 \cong_{h,\varepsilon} \mathcal{TS}_2$, if there exists a bisimulation relation $\mathcal{B}_{h,\varepsilon}$ satisfying that for all initial configurations $q_1 \in {Q_1}^0$ there exists $q_2 \in {Q_2}^0$ such that $(q_1, q_2) \in \mathcal{B}_{h,\varepsilon}$ and vice versa.*

C. Discretization of ODE

Here we present the discretization of ODEs and its correctness, which have been studied in [YJW$^+$20]. We briefly revisit the related results in [YJW$^+$20] here.

We apply the 4-stage Runge-Kutta method to discretize the continuous dynamics, which is more effective with *global discretization error* $O(h^4)$. The ODE $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ on $[t_0, t_0 + T_o]$ is discretized as

$$(\text{wait } h; \mathbf{x} := \mathbf{x} + h\mathbf{\Phi}(\mathbf{x}, h))^N; \text{wait } h'; \mathbf{x} := \mathbf{x} + h'\mathbf{\Phi}(\mathbf{x}, h')$$

where $N = \lfloor \frac{T_o}{h} \rfloor$, $h' = T_o - Nh$, and $\mathbf{\Phi}(\mathbf{x}, s) = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ with $k_1 = \mathbf{f}(\mathbf{x})$, $k_2 = \mathbf{f}(\mathbf{x} + \frac{1}{2}sk_1)$, $k_3 = \mathbf{f}(\mathbf{x} + \frac{1}{2}sk_2)$ and $k_4 = \mathbf{f}(\mathbf{x} + sk_3)$. With the initial state $\mathbf{x}_0$ at $h_0 = t_0$, the obtained sequence of approximate solutions $\{\mathbf{x}_i\}$ at time stamps $\{h_i\}$ is (below $1 \leq j \leq N$):

$$\begin{cases} \mathbf{x}_0, & h_0 = t_0, \\ \mathbf{x}_j = \mathbf{x}_{j-1} + h\mathbf{\Phi}(\mathbf{x}_{j-1}, h), & h_j = t_0 + j * h \\ \mathbf{x}_{N+1} = \mathbf{x}_N + h'\mathbf{\Phi}(\mathbf{x}_N, h'), & h_{N+1} = t_0 + T_o \end{cases}$$

Intuitively, T_o is divided into N intervals of length h and a possible residual interval of length h'. $\mathbf{\Phi}$ is computed based on the values of the vector field at the four points and used for approximating the value of $\mathbf{x}$. Below we present the global error of the discretization (see Theorem 7.2.2.3 in [SB13]).

Proposition 1 (Global Error)**.** *Assume the ODE* $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ *satisfies the local Lipschitz condition, that is, for any compact set* S *of* $\mathbb{R}^n$, $\|\mathbf{f}(\mathbf{y}_1)-\mathbf{f}(\mathbf{y}_2)\| \leq L\|\mathbf{y}_1-\mathbf{y}_2\|$ *for all* $\mathbf{y}_1, \mathbf{y}_2 \in S$. *Let* $X(t, \widetilde{\mathbf{x}}_0)$ *be the exact solution of the ODE with initial value* $\widetilde{\mathbf{x}}_0$ *on* $[0, T_0]$. *Suppose* $\mathbf{x}_0 \in \mathbb{R}^n$ *is a state with* $\|\mathbf{x}_0 - \widetilde{\mathbf{x}}_0\| \leq \xi_1$. *Then there exists a discretized step* $h_e > 0$ *s.t. for all* $0 < h \leq h_e$ *and all* $i \leq \lceil \frac{T_o}{h} \rceil$, *the global discretization error between* $X(h_i, \widetilde{\mathbf{x}}_0)$ *and* $\mathbf{x}_i$ *satisfies:*

$$\begin{aligned}
&\|X(h_i, \widetilde{\mathbf{x}}_0) - \mathbf{x}_i\| \leq M(h), \ \textit{where} \\
&M(h) = \tfrac{e^{Lh'}-1}{L} C_2(h')^4 + [1 + Lh' + \tfrac{(Lh')^2}{2} + \tfrac{(Lh')^3}{4} \\
&\qquad\quad + \tfrac{(Lh')^4}{24}] M_N(h) \\
&M_N(h) = e^{NLh}\xi_1 + \tfrac{e^{NLh}-1}{L} C_1 h^4.
\end{aligned}$$

Among them N, h', h_i *and* $\mathbf{x}_i$ *are as defined previously, and* C_1, C_2 *are positive constants depending on the local discretization error of the 4-stage Runge-Kutta method. Here given a vector* $\mathbf{x} \in \mathbb{R}^n$, $\|\mathbf{x}\|$ *denotes the infinity norm of* $\mathbf{x}$, *i.e.,* $\|\mathbf{x}\| = \max\{|x_1|, |x_2|, ..., |x_n|\}$.

From the definition of $M(h)$, the global error is monotonically increasing with respect to step size h, the Lipschitz constant L, and the two local discretization error constants C_1 and C_2. In [YJW$^+$20], the following theorem is proved for correctness of the discretization of an ODE.

Theorem 1 (Correctness of Discretization of ODEs)**.** *Suppose* L *is the Lipschitz constant of* $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ *with initial condition* $\mathbf{x}(t_0) = \widetilde{\mathbf{x}}_0$, *and* $\mathbf{x}_0$ *satisfies* $\|\mathbf{x}_0 - \widetilde{\mathbf{x}}_0\| \leq \xi_1$. *For any* $\xi > \xi_1 > 0$, *there exists* $h > 0$ *s.t.*

$$\begin{aligned}
&\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}), \ \mathbf{x}(t_0) = \widetilde{\mathbf{x}}_0. \qquad \textbf{\textit{and}} \\
&(\textit{wait } h; \mathbf{x} := \mathbf{x} + h\mathbf{\Phi}(\mathbf{x}, h))^N; \textit{wait } h'; \mathbf{x} := \mathbf{x} + h'\mathbf{\Phi}(\mathbf{x}, h')
\end{aligned}$$

are (h, ξ)*-approximately bisimilar on* $[t_0, t_0 + T_o]$.

D. Source Language HCSP

The syntax for HCSP is given as follows.

$$\begin{array}{lcl}
p & ::= & \text{skip} \mid x := e \mid ch?x \mid ch!e \mid p_1; p_2 \mid B \rightarrow p \mid p_1 \sqcup p_2 \\
 & & \mid p^* \mid \langle \dot{x} = e \& B \rangle \mid \langle \dot{x} = e \& B \rangle \unrhd [\!]_{i \in I}(ch_i* \rightarrow p_i) \\
pc & ::= & p \mid pc_1 \|_{cs} pc_2
\end{array}$$

where e represents expressions, p a sequential HCSP process, and pc the parallel composition of processes. x denotes variables, B Boolean expressions, ch, ch_i channel names. The meanings of skip, assign, sequential, conditional, internal choice and repetition are as usual. We explain the intuitive meaning of the additional constructs as follows:

- The input $ch?x$ receives a value along the channel ch and assigns it to variable x; and the output $ch!e$ sends the value of e along ch. Each of them may block waiting for the corresponding dual party to be ready.
- The repetition p^* executes p for a nondeterministic finite number of times.
- $\langle \dot{x} = e \& B \rangle$ is the continuous evolution, which evolves continuously according to the ODE $\dot{\vec{x}} = \vec{e}$ as long as the *domain* B holds, and terminates when B becomes false.
- Interrupt $\langle \dot{x} = e \& B \rangle \unrhd [\!]_{i \in I}(ch_i* \rightarrow c_i)$ behaves like $\langle \dot{x} = e \& B \rangle$, except it is preempted as soon as one of the communication events ch_i* takes place, and then is followed by the corresponding c_i.
- $pc_1 \|_{cs} pc_2$ behaves as pc_1 and pc_2 run independently except that all communications along common channels cs are synchronized between pc_1 and pc_2. We assume that variables of pc_1 and pc_2 are disjoint, and no same channel direction (e.g. $ch!$) occurs in both pc_1 and pc_2.

Some other constructs can be defined as derived. For example, wait d is an abbreviation for $t := 0; \langle \dot{t} = 1 \& t < d \rangle$.

Fig. 1 presents the small-step semantics for HCSP. Two types of transitions are introduced: $(p, s) \xrightarrow{e} (p', s')$ defines that a sequential HCSP process p executes from initial state s in one step, produces event e and results in statement p' and state s'; and $(pc, s) \xrightarrow{e}_h (pc', s')$ defines one step execution of a parallel HCSP process. Here states $s, s' \in \mathit{Vars} \rightarrow \mathit{Values}$ assign values to variables of p. τ represents an internal discrete event. A *communication event* $\langle ch\triangleright, v \rangle$, where $\triangleright$ is one of ?, !, or nothing, indicating input, output, and synchronized input/output (IO) event, respectively, where v is the transferred value. A *wait event* $\langle d, rdy \rangle$, represents an evolution of time length $d > 0$ with a set of ready channels that are waiting for communication during this period. We denote the set of the above events by *HEvts*. A *ready set* is a set of channel directions, indicating that these channel directions are waiting for communication. Two ready sets rdy_1 and rdy_2 are *compatible*, denoted by $\text{compat}(rdy_1, rdy_2)$, if there does not exist a channel ch such that $ch? \in rdy_1 \wedge ch! \in rdy_2$ or $ch! \in rdy_1 \wedge ch? \in rdy_2$.

We explain some of the transition rules below:

- For output $ch!e$, there are three cases depending on whether the communication occurs immediately: it may occur immediately, wait for some finite time or wait indefinitely (producing a wait event).
- The execution of $\langle \dot{\vec{x}} = \vec{e} \& B \rangle$ produces an execution duration of the ODE with initial state, represented as a wait event. B must become false at the end, while remaining true before that. During the evolution, the ready set is empty.
- For interruption $\langle \dot{x} = e \& B \rangle \unrhd [\!]_{i \in I}(ch_i* \rightarrow p_i)$, all the communication directions in $\{ch_i*\}_{i \in I}$ become ready at the very beginning. After that, communications have a chance to interrupt up to the time at which the ODE reaches the boundary.
- For parallel composition, without loss of generality, suppose pc is a parallel composition of sequential processes $\{p_i\}_{i \in I}$, and s is a disjoint union of the states for all p_is, i.e. $s = \bigcup_{i \in I} s_i$. $pc[p'_i/p_i]$ returns a new process by substituting p'_i for p_i in pc, and $s[s'_i/s_i]$ the same. There are three cases: if one process among pc performs a τ step, then pc can perform the same τ step (G-Tau); if two processes synchronise over a same channel, then pc performs a communication immediately (G-comm); if all processes of pc can perform a wait duration d and their ready sets are mutually compatible, then pc performs a wait duration d, joining all ready sets together (G-delay).

$$\frac{}{(x := e, s) \xrightarrow{\tau} (\text{skip}, s[x \mapsto e])} \quad \frac{(c_1, s) \xrightarrow{e} (c_1', s')}{(c_1; c_2, s) \xrightarrow{e} (c_1'; c_2, s')} \quad \frac{}{(\text{skip}; c, s) \xrightarrow{\tau} (c, s)} \quad \frac{s(B)}{(B \to c, s) \xrightarrow{\tau} (c, s)}$$

$$\frac{\neg s(B)}{(B \to c, s) \xrightarrow{\tau} (\text{skip}, s)} \quad \frac{}{(c_1 \sqcup c_2, s) \xrightarrow{\tau} (c_1, s)} \quad \frac{}{(c_1 \sqcup c_2, s) \xrightarrow{\tau} (c_2, s)} \quad \frac{}{(c^*, s) \xrightarrow{\tau} (\text{skip}, s)} \quad \frac{(c, s) \xrightarrow{e} (c', s')}{(c^*, s) \xrightarrow{e} (c'; c^*, s')}$$

$$\frac{}{(ch!e, s) \xrightarrow{\langle ch!, s(e) \rangle} (\text{skip}, s)} \quad \frac{}{(ch!e, s) \xrightarrow{\langle d, \{ch!\} \rangle} (ch!e, s)} \quad \frac{}{(ch!e, s) \xrightarrow{\langle \infty, \{ch!\} \rangle} (\text{skip}, s)}$$

$$\frac{}{(ch?x, s) \xrightarrow{\langle ch?, v \rangle} (\text{skip}, s[x \mapsto v])} \quad \frac{}{(ch?x, s) \xrightarrow{\langle d, \{ch?\} \rangle} (ch?x, s)} \quad \frac{}{(ch?x, s) \xrightarrow{\langle \infty, \{ch?\} \rangle} (\text{skip}, s)}$$

$$\frac{\begin{array}{c}\vec{p} \text{ is a solution of the ODE } \dot{\vec{x}} = \vec{e} \\ \vec{p}(0) = s(\vec{x}) \quad \forall t \in [0, d).\, s[\vec{x} \mapsto \vec{p}(t)](B)\end{array}}{(\langle \dot{\vec{x}} = \vec{e} \& B \rangle, s) \xrightarrow{\langle d, \{\} \rangle} (\langle \dot{\vec{x}} = \vec{e} \& B \rangle, s[\vec{x} \mapsto \vec{p}(d)])} \quad \frac{\neg s(B)}{(\langle \dot{\vec{x}} = \vec{e} \& B \rangle, s) \xrightarrow{\tau} (\text{skip}, s)}$$

$$\frac{\vec{p} \text{ is a solution of the ODE } \dot{\vec{x}} = \vec{e} \quad \vec{p}(0) = s(\vec{x}) \quad \forall t \in [0, d).\, s[\vec{x} \mapsto \vec{p}(t)](B)}{(\langle \dot{x} = e \& B \rangle \trianglerighteq []_{i \in I}(ch_i* \to c_i), s) \xrightarrow{\langle d, rdy(\cup_{i \in I} ch_i*) \rangle} (\langle \dot{x} = e \& B \rangle \trianglerighteq []_{i \in I}(ch_i* \to c_i), s[\vec{x} \mapsto \vec{p}(d)]}$$

$$\frac{\neg s(B)}{(\langle \dot{x} = e \& B \rangle \trianglerighteq []_{i \in I}(ch_i* \to c_i), s) \xrightarrow{\tau} (\text{skip}, s)} \quad \frac{i \in I \quad ch_i* = ch!e}{(\langle \dot{x} = e \& B \rangle \trianglerighteq []_{i \in I}(ch_i* \to c_i), s) \xrightarrow{\langle ch!, s(e) \rangle} (Q_i, s)}$$

$$\frac{i \in I \quad ch_i* = ch?x}{(\langle \dot{x} = e \& B \rangle \trianglerighteq []_{i \in I}(ch_i* \to c_i), s) \xrightarrow{\langle ch?, v \rangle} (c_i, s[x \mapsto v])}$$

$$\frac{i \in I \quad (p_i, s_i) \xrightarrow{\tau} (p_i', s_i')}{(pc, s) \xrightarrow{\tau}_h (pc[p_i'/p_i], s[s_i'/s_i])}\text{G-Tau} \quad \frac{i, j \in I \quad i \neq j \quad (p_i, s_i) \xrightarrow{\langle ch!, v \rangle} (p_i', s_i') \quad (p_j, s_j) \xrightarrow{\langle ch?, v \rangle} (p_j', s_j')}{(pc, s) \xrightarrow{\langle ch, v \rangle}_h (pc[p_i'/p_i, p_j'/p_j], s[s_i'/s_i, s_j'/s_j])}\text{G-Comm}$$

$$\frac{\forall i \in I.(p_i, s_i) \xrightarrow{\langle d, rdy_i \rangle} (p_i', s_i') \quad \forall i, j \in I. i \neq j \Rightarrow \text{compat}(rdy_i, rdy_j)}{(\|_{i \in I} p_i, s) \xrightarrow{\langle d, \cup_{i \in I} rdy_i \rangle}_h (\|_{i \in I} p_i', \uplus_{i \in I} s_i')}\text{G-delay}$$

Fig. 1. Small-step operational semantics of HCSP

- The semantics of other compound constructs is defined by structural induction.

III. Target Language C: Syntax and Semantics

We consider a subset of concurrent C with POSIX threads as the target language of code generation. The abstract syntax is defined in Fig. 2. Here d denotes constants, x variables, **retv** and **ret** introduced for semantic use to record the return value of a function and check whether a return occurs so that the remaining code of the function will not execute, $a[k]$ array elements, **op** arithmetic or Boolean operators, B Boolean expressions, tid the ID of a thread, l mutex, cv condition variables, f, g function names, $\overline{T\ x}$ an abbreviation for a sequence of variable declarations with the form $T_1\ x_1; \cdots; T_n\ x_n$.

The remaining statements define the thread APIs for achieving concurrency, provided by the POSIX thread library of C:

- **create** $tid\ F\ \overline{e}$ spawns a new thread tid, and starts execution by invoking function F with arguments $\overline{e}$.
- **lock** l locks mutex l and gains exclusive access to the data protected by l.
- **unlock** l releases mutex l and in consequence another thread is allowed to acquire l and use the shared data.
- **cwait** $cv\ l$ blocks on condition variable cv and automatically releases mutex l. As soon as cv is signaled by another thread, it is unblocked on this signal and turns to re-acquire mutex l.
- **signal** cv signals the condition variable cv to the thread that is blocked on it and in consequence the thread is released to execute.
- **join** tid waits for the thread tid to terminate.

The last $x \leftarrow$ **retv**$.Es$ represents that x is set to be the return value **retv** in local states Es. Same as **retv** and **ret**, it is introduced for defining the small-step semantics of function calls with return values. A function declaration F includes a return type T_1, a function name f, and a body consisting of a sequence of local variable declarations and a command c. At the end, a C program P is composed of a sequence of global variable declarations $decl$, a sequence of function declarations $\overline{F}$, and the `main` function as the entry point of the program.

A. Small-step Semantics

1) Notations: The semantics of C statements is defined by two judgements. They are parameterized by a static environment Γ, a thread pool T, a global state G, and a local state set Es. Γ maps a function name to its declaration, T maps each active thread to its code, G maps global variables to their values, and Es maps each thread to its local state, which in turn maps local variables of the thread to values. Below list

$$\begin{array}{lll}
\text{Expressions} & e & ::= \; d \mid x \mid \textbf{retv} \mid \textbf{ret} \mid a[k] \mid e \; \textbf{op} \; e \mid ... \\
\text{Statements} & c & ::= \; \text{skip} \mid x = e \mid x = f(\overline{e}) \mid g(\overline{e}) \mid c_1; c_2 \mid \textbf{if} \; B \; c_1 \; \textbf{else} \; c_2 \\
 & & \quad \mid \textbf{while} \; B \; c \mid \textbf{for}(c_1, e, c_2) \; c_3 \mid \textbf{return} \; e \mid \textbf{create} \; tid \; F \; \overline{e} \\
 & & \quad \mid \textbf{lock} \; l \mid \textbf{unlock} \; l \mid \textbf{cwait} \; cv \; l \mid \textbf{signal} \; cv \mid \textbf{join} \; tid \mid x \leftarrow \textbf{retv}.Es \\
\text{Variable Decls} & decl & := \; \overline{T \; x} \quad \text{Function Decls } F ::= T_1 \; f(decl_1)\{decl_2; c\} \\
\text{Programs} & P & ::= \; decl; \overline{F}; \texttt{main}
\end{array}$$

Fig. 2. Syntax of Subset of Multi-threaded C

the judgements defining the one step execution of a thread and of a thread set consisting of multiple threads resp.:

- $\Gamma, tid \vdash (c, T, G, Es) \rightarrow_c (c', T', G', Es')$, stating that under static environment Γ and thread tid, statement c executes to c' in one step, changing thread pool T (due to thread creation), global state G, local states Es to T', G' and Es' resp.
- $\Gamma \vdash (T, G, Es) \rightarrow (T', G', Es')$, stating that under static environment Γ, thread pool T executes, leading to the continuation T', changing global state G and local states Es to G' and Es' resp.

2) Semantics: We present the small-step semantics of multi-threaded C in Fig. 3. Rule (Assign) changes the value of x to e in G or Es depending on whether it is global variable (denoted by $x \in \textbf{GV}$) or not. Rules (Funcr, Ret, FunEnd) define the semantics for a function call with return value: (Funcr) first looks up the definition of function f in Γ, then builds a new local state mapping formal parameters $\overline{x}$ to values of $\overline{e}$ and local variables of f to be their default values (represented by **0** here), and executes the body statement c from this new state. At the same time, at the end of c, $x \leftarrow \textbf{retv}.Es$ is added, to be explained in (FunEnd); (Ret) sets **retv** to be the value of e and **ret** to be 1 indicating that a return value is obtained and the rest code inside the function body will not be executed any more. (FunEnd) executes when **ret** is 1: it resumes the previous state Es', assigns x by the value of **retv** recorded in Es, and meanwhile, reset **ret** to 0 indicating that this function body is jumped and thus the following code can execute as normal. In conclusion, for handling function call: Each statement is guarded by $\textbf{ret} = 0$, to mean that no return occurs thus the continuation executes; For the contrary case when $\textbf{ret} = 1$, the continuation will be dropped, indicated by rule (Ret'), until the function call ends, indicated by rule (FunEnd), which transfers the return value back and resets **ret** to 0 again.

Rule (Create) spawns a new thread tid', recording the code of tid' in the thread pool to be $F(\overline{v})$, and meanwhile setting its initial local state as the one of parent thread tid. Rule (Lock) defines that if mutex l is available, then it can be obtained by thread tid, by mapping l to be the holding thread tid in the global state G. Rule (Unlock) defines that tid releases l. The semantics of **cwait** is specified by two rules: At first, tid locks l and cv is false, t releases l (WaitF); then as soon as cv is signaled thus becomes true, **cwait** stops waiting, and then it needs to acquire mutex l again and thus is equivalent to executing **lock**, and at the same time resetting cv to be false (WaitT). **signal** cv signals cv to some thread who is waiting on it (Signal). Rule (Join) defines that when thread tid' completes the execution of its code, **join** tid' terminates directly. Rule (Threads) defines the execution of multiple threads in T, which randomly selects an available thread tid to execute and updates the thread pool, the global and local states correspondingly.

Above we have defined the semantic rules for both HCSP and C subset, based on which the transition systems for them can be built.

IV. From HCSP to C

A. Auxiliary Variables and Functions

In order to transform HCSP to C, some auxiliary variables and functions are introduced in order to achieve synchronization between multiple threads.

1) Global and Local Clocks: In order to synchronize the executions of threads in parallel, we introduce a local clock for each thread i, denoted by `localTime[i]`, to record the local execution time of i. If a thread is waiting with a time limit (and possibly for some communications), its `localTime` is set to that time limit. If the thread is only waiting for communication, its `localTime` is set to infinity (with `DBL_MAX` used in practice). A global clock `currentTime` is used to record the global execution time, and to coordinate the execution of all threads. It equals the minimum of all local clocks, thus every time a local clock makes progress, `currentTime` will be updated and in consequence threads whose time limit has reached will be woken up and be notified to execute.

2) Thread States: There are six possible forms of thread states that indicate the execution status of a thread:

- `Stopped`, representing the thread has reached the end of execution;
- `Waiting`, representing that the thread is waiting for the global clock with a time limit, specified by `localTime`. When the global clock reaches that time limit, the thread is released to run.
- `Available`, representing that the thread is waiting for a communication event. As soon as it receives the signal from another thread denoting that a compatible communication event is ready, it is woken up to run.
- `Waiting_Available`, representing that the thread is waiting both for the global clock to reach a time limit, and for a communication event. Either of them can release the thread to run.

$$\frac{\begin{array}{c}Es(tid)(\mathbf{ret}) = 0 \quad \llbracket e \rrbracket_{G,Es} = v \quad G' = (v \in \mathbf{GV})?G[x \mapsto v] : G \\ Es' = (v \in \mathbf{GV})?Es : Es[tid \mapsto Es(tid)[x \mapsto v]]\end{array}}{\Gamma, tid \vdash (x = e, T, G, Es) \rightarrow_c (\epsilon, T, G', Es')}\ \text{Assign}$$

$$\frac{Es(tid)(\mathbf{ret}) = 0 \quad \Gamma(f) = T\ f(\overline{T\ x})\{\overline{U\ y}; c\} \quad \llbracket \overline{e} \rrbracket_{G,Es} = \overline{v} \quad E = \{\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{\mathbf{0}}\}}{\Gamma, tid \vdash (x = f(\overline{e}), T, G, Es) \rightarrow_c (c; x \leftarrow \mathbf{retv}.Es, T, G, [tid \mapsto E, \mathbf{ret} \mapsto 0])}\ \text{Funcr}$$

$$\frac{Es(tid)(\mathbf{ret}) = 1}{\Gamma, tid \vdash (x \leftarrow \mathbf{retv}.Es', T, G, Es) \rightarrow_c (\epsilon, T, G, Es'[x \mapsto Es(\mathbf{retv}), \mathbf{ret} \mapsto 0])}\ \text{FunEnd}$$

$$\frac{Es(tid)(\mathbf{ret}) = 0}{\Gamma, tid \vdash (\mathbf{return}\ e, T, G, Es) \rightarrow_c (\epsilon, T, G, Es[tid \mapsto Es(tid)[\mathbf{retv} \mapsto \llbracket e \rrbracket_{G,Es}], \mathbf{ret} \mapsto 1])}\ \text{Ret}$$

$$\frac{\Gamma, tid \vdash (c_1, T, G, Es) \rightarrow_c (c_1', T', G', Es')}{\Gamma, tid \vdash (c_1; c_2, T, G, Es) \rightarrow_c (c_1'; c_2, T', G', Es')}\ \text{SeqT}$$

$$\frac{}{\Gamma, tid \vdash (\epsilon; c_2, T, G, Es) \rightarrow_c (c_2, T, G, Es)}\ \text{SeqF}$$

$$\frac{Es(tid)(\mathbf{ret}) = 0 \quad \llbracket B \rrbracket_{G,Es} = \mathbf{True} \quad \Gamma, tid \vdash (c, T, G, Es) \rightarrow_c (c', T', G', Es')}{\Gamma, tid \vdash (\mathbf{while}\ B\ c, T, G, Es) \rightarrow_c (c'; \mathbf{while}\ B\ c, T', G', Es')}\ \text{WhileT}$$

$$\frac{Es(tid)(\mathbf{ret}) = 0 \quad \llbracket B \rrbracket_{G,Es} = False}{\Gamma, tid \vdash (\mathbf{while}\ B\ c, T, G, Es) \rightarrow_c (\epsilon, T, G, Es)}\ \text{WhileF}$$

$$\frac{\neg(c \equiv x \leftarrow \mathbf{retv}.Es \vee c \equiv c_1; c_2) \quad Es(tid)(\mathbf{ret}) = 1}{\Gamma, tid \vdash (c, T, G, Es) \rightarrow_c (\epsilon, T, G, Es)}\ \text{Ret'}$$

$$\frac{Es(\mathbf{ret}) = 0 \quad \llbracket \overline{e} \rrbracket_{G,Es} = \overline{v}}{\Gamma, tid \vdash (\mathbf{create}\ tid'\ F\ \overline{e}, T, G, Es) \rightarrow_c (\epsilon, T[tid' \mapsto F(\overline{v})], G, Es[tid' \mapsto Es(tid)]}\ \text{Create}$$

$$\frac{Es(\mathbf{ret}) = 0 \quad G(l) = \bot}{\Gamma, tid \vdash (\mathbf{lock}\ l, T, G, Es) \rightarrow_c (\epsilon, T, G[l \mapsto tid], Es)}\ \text{Lock} \quad \frac{Es(\mathbf{ret}) = 0 \quad G(l) = tid}{\Gamma, tid \vdash (\mathbf{unlock}\ l, T, G, Es) \rightarrow_c (\epsilon, T, G[l \mapsto \bot], Es)}\ \text{Unlock}$$

$$\frac{Es(\mathbf{ret}) = 0 \quad G(cv) = 0 \quad G(l) = tid}{\Gamma, tid \vdash (\mathbf{cwait}\ cv\ l, T, G, Es) \rightarrow_c (\mathbf{cwait}\ cv\ l\ , T, G[l \mapsto \bot], Es)}\ \text{WaitF}$$

$$\frac{Es(\mathbf{ret}) = 0 \quad G(cv) = 1 \quad G(l) \neq tid}{\Gamma, tid \vdash (\mathbf{cwait}\ cv\ l\ , T, G, Es) \rightarrow_c (\mathbf{lock}\ l, T, G[cv \mapsto 0], Es)}\ \text{WaitT}$$

$$\frac{Es(\mathbf{ret}) = 0}{\Gamma, tid \vdash (\mathbf{signal}\ cv, T, G, Es) \rightarrow_c (\epsilon, T, G[cv \mapsto 1], Es)}\ \text{Signal} \quad \frac{Es(\mathbf{ret}) = 0 \quad T(tid') = \epsilon}{\Gamma, tid \vdash (\mathbf{join}\ tid', T, G, Es) \rightarrow_c (\epsilon, T, G, Es)}\ \text{Join}$$

$$\frac{tid \in dom(T) \quad T(tid) = c \quad Es(tid) = E \quad \Gamma, tid \vdash (c, T, G, Es) \rightarrow_c (c', T', G', Es')}{\Gamma \vdash (T, G, Es) \rightarrow_t (T'[tid \mapsto c'], G', Es')}\ \text{Threads}$$

Fig. 3. Small-step operational semantics of C

- `Running`, representing that the thread is able to execute, not waiting for the global clock or communication events;
- A non-negative number i, representing that a communication is occurring on channel i. This is an intermediate state used to synchronize communication, meaning that the thread will carry out a communication on channel i, but has not finished doing so.

The thread states are shared by all threads and used for coordinating the execution of them. In our implementation, each thread is denoted by an ID among $\{0, 1, \cdots, N-1\}$, where N is the number of threads. An array `threadState`$[N]$ is used to record the execution states of each thread.

3) Channels in C: In order to realize synchronized communication in C, we introduce global variables related to channels. In HCSP, each channel ch includes two ends: $ch?$ for input and $ch!$ for output, and they synchronize both on time and values transferred along ch. In C implementation, we introduce a structure `Channel` to define each single channel end, which is (`type, channelNo, pos`), representing its type (0 for input and 1 for output), channel ID number (represented by natural numbers 0, 1, ...), and the pointer referring to the value it holds. Especially, each pair of input and output ends of a channel have the same channel ID. We also define three arrays for achieving synchronization between inputs and outputs: for each channel i, `channelInput[i]` and `channelOutput[i]` record the thread that is ready on corresponding input and output alone channel i, and thus available to participate in the communication. The default value is -1, denoting that no thread is available for the corresponding input or output; `channelContent` acts as a buffer to save the values transmitted along channels: the output ends write to it while the input ends read from it.

4) Locks and Condition Variables: A mutex `mutex` is introduced to protect the shared resources of threads. To access these resources, a thread must acquire `mutex` first. Moreover, to achieve synchronization between threads, an array `cond`[N] of condition variables are introduced, one for each thread.

B. Transformation of HCSP

1) Continuous evolution: For continuous evolution $\langle \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) \& B \rangle$, we will first discretise it in HCSP and then transform the discretisation to C code. Before discretisation, the discretized time step h should be computed first. Given a HCSP process P, suppose the upper bound for the execution time of P is T_o, and the value precision is ε, then h is computed as follows:

- First, collect the set of ODEs of P, denoted by $\{ode_1(\overline{x_1}), \cdots, ode_k(\overline{x_k})\}$. Among them, for any $i \neq j$, $\overline{x_i}$ and $\overline{x_j}$ are continuous variables of the two ODEs resp. and they might have common variables;
- Next we compute the upper bound of the discretized step. On one hand, suppose the Lipschitz constants for the k ODEs exist and are $L^1, \cdots, L^k$, and the constants for the Runge-Kutta method are ${C_1}^1, \cdots, {C_1}^k, {C_2}^1, \cdots, {C_2}^k$, resp., as presented in Prop. 1. Then compute $M(h) \leq \frac{\varepsilon}{2}$ by assigning L, C_1, C_2 to be $\max\{L^i\}_{1\leq i\leq k}$, $\max\{{C_1}^i\}_{1\leq i\leq k}$,$\max\{{C_2}^i\}_{1\leq i\leq k}$ resp., which finally obtains an upper bound for the discretized time step, denoted by h_1. We must have $h \leq h_1$.
- On the other hand, suppose for each ODE $ode_i(\overline{x_i})$ of the form $\langle \dot{\mathbf{x}}_i = \mathbf{f}_i(\mathbf{x}_i) \& B_i \rangle$, the derivative $\mathbf{f}_i(\mathbf{x}_i)$ is bounded over the interval $[0, T_o]$, satisfying $\|\mathbf{f}_i(\mathbf{x}_i)\| \leq U_i$. Then the distance between any two states at time a and b within one time step is bounded by $U_i \cdot \|a - b\|$. Let $U_i \cdot h < \frac{\varepsilon}{2}$, then $h < \frac{\varepsilon}{2U_i}$ for any i.
- At the end, let h be $\min(h_1, \min_{1\leq i\leq k}\{\frac{\varepsilon}{2U_i}\})$.

After h is calculated, for any continuous evolution $\langle \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) \& B \rangle$ occurring in p, it is transformed to the following discrete HCSP process according to Theorem 1:

$$\begin{aligned} & j := 1; (\neg(N(B,\varepsilon) \wedge N^n(B,\varepsilon)) \rightarrow j = 0; \\ & \qquad\qquad j = 1 \rightarrow \text{wait } h; \mathbf{x} := \mathbf{x} + h\mathbf{\Phi}(\mathbf{x}, h))^N; \\ & N(B,\varepsilon) \wedge N^n(B,\varepsilon) \rightarrow \texttt{stop}; \end{aligned} \tag{1}$$

where $N = \lceil \frac{T - T_0}{h} \rceil$. Given a Boolean formula B, which can be considered as a set of states satisfying B, $N(B, \varepsilon)$ denotes the ε-neighbourhood of B, representing the set $\{a \mid \exists b. |a - b| < \varepsilon \wedge b \in B\}$ (obviously $B \subset N(B, \varepsilon)$); and $N^n(B, \varepsilon)$ is an abbreviation of $N(B[\mathbf{x} \mapsto \mathbf{x} + h\mathbf{\Phi}], \varepsilon)$, i.e. the ε-neighbourhood of B at next discretized time step. Instead of B, $N(B, \varepsilon) \wedge N^n(B, \varepsilon)$ is used to judge whether to continue evolving according to the ODE. With the help of robustly safe condition (to be explained in next section), the discretized implementation can always be guaranteed to escape within a tolerance. For the example shown in Fig. 5, the discretization escapes at next step $(n + 1)h$ when the ODE violates B between nh and $(n+1)h$. With the C implementation of wait h by time delay, the C code is a direct translation of (1).

2) Continuous Interrupt: Continuous interruption $\langle \dot{x} = e \& B \rangle \trianglerighteq \talloblong_{i\in I}(ch_i* \rightarrow p_i)$ is first transformed to a sequential composition of discrete processes in HCSP, among which $n, N, \mathbf{\Phi}, \varepsilon$ are as defined before. For any HCSP process p, *HtoD*(p) transforms p to its discretised version.

$$\begin{aligned} & j_1 := 1; j_2 := -1; \\ & (\neg(N(B,\varepsilon) \wedge N^n(B,\varepsilon)) \rightarrow j_1 := 0; \\ & \ j_1 = 1 \wedge j_2 = -1 \rightarrow c := 0; \\ & \qquad \langle \dot{c} = 1 \& c \leq h \rangle \trianglerighteq \talloblong_{i\in I}(ch_i* \rightarrow j_2 := i); \\ & \ j_1 = 1 \wedge j_2 = -1 \rightarrow \mathbf{x} := \mathbf{x} + h\mathbf{\Phi}(\mathbf{x}, h))^N; \\ & j_2 \geq 0 \rightarrow \mathbf{x} := \mathbf{x} + c\mathbf{\Phi}(\mathbf{x}, c); \mathit{HtoD}(p_{j_2}); \\ & j_1 = 1 \wedge j_2 = -1 \rightarrow \texttt{stop}; \end{aligned} \tag{2}$$

$c := 0; \langle \dot{c} = 1 \& c \leq h \rangle$ is equivalent to wait h. We use it here in order to record the communication interrupt time by variable c, used to compute the state later (guarded by $j_2 \geq 0$). The C code corresponding to wait $h \trianglerighteq \talloblong_{i\in I}(ch_i* \rightarrow j_2 := i)$ is denoted by wait_comm$(h, \{ch_i*\}_{i\in I})$, to be explained next.

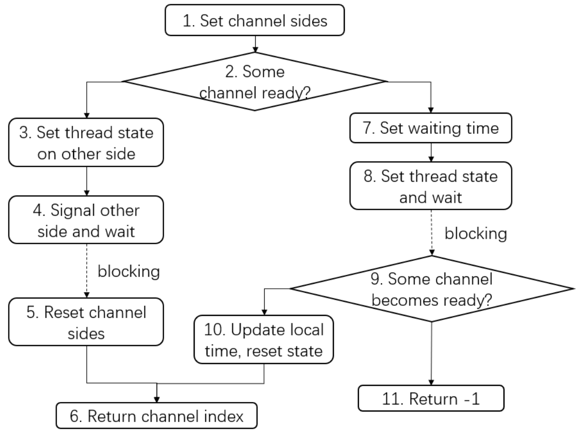
1. Set channel sides
2. Some channel ready?
3. Set thread state on other side
4. Signal other side and wait
blocking
5. Reset channel sides
6. Return channel index
7. Set waiting time
8. Set thread state and wait
blocking
9. Some channel becomes ready?
10. Update local time, reset state
11. Return -1

Fig. 4. The control flow of C code for wait_comm

3) Wait communication: Fig. 4 presents the control flow for the C code corresponding to wait_comm$(h, \{ch_i*\}_{i\in I})$, after locking the mutex in the beginning: 1, Set `channelInput` and `channelOutput` for each communication to the current thread, indicating that $\{ch_i*\}_{i\in I}$ are all available, and meanwhile write values to the channel content buffer from outputs; 2, Look through the set of compatible channel ends to see if any is ready; 3, If some channel end is ready, set the state of the other thread to that channel (reserving communication on the other side); 4, Signal to the other thread and wait; 5, On returning from wait, reset thread state to be `RUNNING` and all channel sides corresponding to $\{ch_i*\}_{i\in I}$ to be unready; 6, In case a communication is performed, the function returns the channel index for later use; 7, If no channel is ready at the beginning, set target waiting time recorded by `localtime`; 8, Set thread state to be `AVAILABLE` and wait; 9, On returning from wait, check the thread state to determine which channel among $\{ch_i\}_i$ is matched to perform the communication; 10, Update local time to be the local time of the other thread, and reset all states as in step 5; 11, If no communication occurs during time h, return -1 and terminate. Input and output can be considered as special cases of wait_comm.

Above we present the primitive functions related to communication and ODE used for transformation. By calling these primitives, each type of sequential compound processes of HCSP can be transformed inductively. At last, the top HCSP process, in the form of parallel composition, is transformed to a set of threads corresponding to all its sequential subprocesses.

In the appendix, we list the generated C code for some HCSP constructs as representatives.

V. Correctness of the Transformation

In this section, we present the correctness proof of the transformation. The proof uses the following strategy, separating the concerns of discretization and reasoning about concurrency into two steps. First, we show that the HCSP process and its discretization are approximately bisimilar. Second, we show that for the discrete fragment of HCSP (in which the discretization of any HCSP program lies), the translation to concurrent C code is (exactly) bisimilar.

We first introduce the notion of robustly safe HCSP processes (Sect. V-A). Then we state the theorem on approximate bisimulation between HCSP and C (Sect. V-B). The first part of the proof, that HCSP is approximately bisimilar to its discretization, is given partly in Sect. V-C for the case of continuous evolution and interrupt, while others are similar to [YJW$^+$20] and omitted here. Finally, we present the proof of exact bisimulation between HCSP and the generated C code in Sect. V-D.

A. Robustly Safe HCSP Processes

Due to the discretization of the ODE, the behaviors of an HCSP process and its transformed C program are not exactly equivalent but allow a difference within given precisions. An HCSP process must be robust to tolerable errors. Especially, the discretization should preserve the control behavior in alternation like $B \to p$, and moreover, it should be able to detect the change of domain boundary in continuous evolution like $\langle \dot{x} = f(x) \& B\rangle$ and locate the changing point within a tolerance (which is the so-called zero-crossing detection and location in hybrid systems [ZYM08]). Hence, we introduce the notion of *robustly safe processes* with given precisions in Def. 4. Let ϕ denote a Boolean formula and ϵ a precision, define $N(\phi, -\epsilon)$ as the set $\{v \mid v \in \phi \wedge \forall u \in \neg\phi.\, |v-u| > \epsilon\}$, which is a subset of ϕ and furthermore the distance from all states in it to the boundary of ϕ are greater than ϵ.

Definition 4 ((δ, ϵ)-robustly safe)**.** *An HCSP process P is (δ, ϵ)-robustly safe, for given time precision $\delta > 0$ and value precision $\epsilon > 0$, if the following two conditions hold:*

- *For every alternative process $B \to Q$ occurring in P with B depending on continuous variables of P, then the states v reached before the execution of $B \to Q$ satisfy $v \in N(B, -\epsilon)$ or $v \in N(\neg B, -\epsilon)$;*

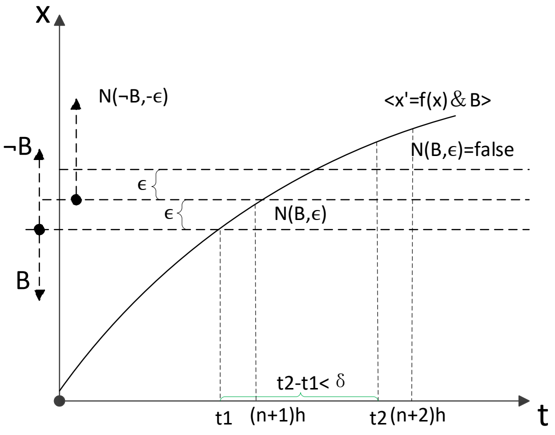

Fig. 5. The (δ, ϵ)-robustly safe continuous statement.

- *For every continuous evolution $\langle \dot{\vec{x}} = \vec{e} \& B\rangle$ occurring in P, suppose its initial state is v_0, and B turns to false at state v and time t, then there exists $\widehat{t} \in (t, t+\delta)$ s.t. $U(v[\vec{x} \mapsto X(\widehat{t}, \widetilde{v}_0)], \epsilon) \subseteq N(\neg B, -\epsilon)$, where $X(t, \widetilde{v}_0)$ is the solution of $\dot{\vec{x}} = \vec{e}$ at t with initial value $\widetilde{v}_0$.*

Def. 4 requires: (1) All reachable states before $B \to P$ are ϵ-far from the boundary of B, which is also the boundary of $\neg B$. (2) As shown by Fig. 5, when B turns false at time t_1, then before time t_2 (within δ tolerance), the ODE continues to go away from the boundary of B, at least 2ϵ further at t_2. According to our discretization (by checking $N(B, \epsilon) \wedge N^n(B, \epsilon)$), it is guaranteed to detect the change of B at next step of t_1, i.e. $(n+1)h$ in the example. Computing the robustly safe parameters (δ, ϵ) is very challenging and some methods for computing them are given in [YJW$^+$20].

B. Approximate Bisimulation between HCSP and C

We then state the main theorem for approximate bisimulation between HCSP and C. Next $\mathit{HtoD}_{h,\varepsilon}(P)$ and $\mathit{HtoC}_{h,\varepsilon}(P)$ denote the discretized version and the generated C code of HCSP process P with precisions h and ε resp.

Theorem 2. *Let P be a HCSP process and $T > 0$ is an upper bound of time. Suppose P is (δ, ϵ)-robustly safe, and for any ODE $\dot{x} = f(x)$ occurring in P, f is Lipschitz continuous. Then, for any precision $\varepsilon \in (0, \epsilon]$, there must exist $h > 0$ such that $P \cong_{h,\varepsilon} \mathit{HtoC}_{h,\varepsilon}(P)$ holds on $[0, T]$.*

We will prove Theorem 2 in two steps: $P \cong_{h,\varepsilon} \mathit{HtoD}_{h,\varepsilon}(P)$, and $\mathit{HtoD}_{h,\varepsilon}(P) \cong \mathit{HtoC}_{h,\varepsilon}(P)$.

C. Proof of approximate bisimulation between HCSP and Its Discretization

We present the proof for the approximate bisimulation between continuous HCSP processes and the discrete ones.

1) We first show the proof for continuous evolution. Denote the ODE $\langle \mathbf{x} = \mathbf{f}(\mathbf{x}) \& B\rangle$ by ode_k, suppose the exact initial value of $\mathbf{x}$ prior to its execution is $\widetilde{\mathbf{x}}_0$, and the discretized value is $\mathbf{x}_0$, satisfying $\|\widetilde{\mathbf{x}}_0 - \mathbf{x}_0\| \le \xi_1 <$

ε. Now we consider the transitions of the ODE and the discretised process (1).

- If $(\langle \mathbf{x} = \mathbf{f}(\mathbf{x})\&B\rangle, s_0) \xrightarrow{\langle d, \{\}} (\langle \mathbf{x} = \mathbf{f}(\mathbf{x})\&B\rangle, s_0[\mathbf{x} \mapsto \mathbf{p}(d)]$, where $\mathbf{p}$ is the solution of the ODE with initial state s_0, then B keeps false during the duration d, and the accumulated execution time is less than the upper bound T_o. We first prove that for any $t \in [0, d]$, $\|s_0[\mathbf{x} \mapsto \mathbf{p}(t)] - s_0'[\mathbf{x} \mapsto \mathbf{x} + \lfloor \frac{t}{h} \rfloor \cdot h\mathbf{\Phi}]\| < \varepsilon$. The left hand side, denoted by LHS, satisfies (U is the upper bound of $\mathbf{f}(\mathbf{x})$):

  $$\begin{aligned} LHS \quad & \leq \|s_0[\mathbf{x} \mapsto \mathbf{p}(t)] - s_0[\mathbf{x} \mapsto \mathbf{p}(h \cdot \lfloor \tfrac{t}{h} \rfloor)]\| + \\ & \quad \|s_0[\mathbf{x} \mapsto \mathbf{p}(h \cdot \lfloor \tfrac{t}{h} \rfloor), s_0'[\mathbf{x} \mapsto \mathbf{x} + \lfloor \tfrac{t}{h} \rfloor \cdot h\mathbf{\Phi}]\| \\ & \leq U \cdot h' + M(h) < \tfrac{\varepsilon}{2} + \tfrac{\varepsilon}{2} = \varepsilon \end{aligned}$$

  Obviously, before time $\lceil \frac{d}{h} \rceil \cdot h$, denoted by k, the discretized states are in $N(B, \varepsilon)$, i.e. all states before $\lceil d \rceil \cdot h - h$ also satisfies $N^n(B, \varepsilon)$. In consequence, process (1) at least has the following transitions:

  $$((1), s_0')(\xrightarrow{h} (\_, s_i') \rightarrow (\_, s_i'[\mathbf{x} \mapsto \mathbf{x} + h\mathbf{\Phi}]))^{k-1}$$

  where $s_i' = s_{i-1}'[\mathbf{x} \mapsto \mathbf{x} + h\mathbf{\Phi}]$ for any $i > 1$. If $N^n(B, \varepsilon)$ is still true for the k-th step, another transition proceeds and the final state is still within ε of the precise continuous value. In the other direction, the first $k - 1$ rounds are the same, but if $N^n(B, \varepsilon)$ is false for the k-th step, then at next step, B must be false, (1) terminates at previous one step, and the final values are still within ε.
- If $(\langle \mathbf{x} = \mathbf{f}(\mathbf{x})\&B\rangle, s_0) \xrightarrow{\tau} (\text{skip}, s_0]$, when $s_0(B) = false$, $t_0 \in (kh, (K + 1)h]$. From the fact that the HCSP process is (δ, ϵ)-robustly safe, then there exists $\widetilde{t} \in (t_0, t_0 + \delta)$ such that for all $s \in U(s_0[\mathbf{x} \mapsto X(\widetilde{t}, s_0)], \epsilon)$, $U(s, \epsilon)$ makes B false, and $h < \delta < 2h$. It holds that $0 < \widetilde{t} - t_0 < 2h$. Consider $\|\mathbf{x}_{k+2} - X(\widetilde{t}, s_0)\|$, where $\mathbf{x}_{k+2} = \mathbf{x} + (k + 2)h\mathbf{\Phi}$, which is reduced to:

  $$\begin{aligned} & \|\mathbf{x}_{k+2} - X((k + 2)h, s_0)\| + \\ & \quad \|X((k + 2)h, s_0) - X(\widetilde{t}, s_0)\| < \varepsilon \end{aligned}$$

  Thus, $U(\mathbf{x}_{k+2}, \epsilon) \in \neg B$, which means that $\mathbf{x}_{k+2} \in N(\neg B, -\epsilon)$. According to process (1), the condition for the $k + 2$-th repetition violates, which if considered as the escaping point for the discretised version, more than one time step distance from the exact escaping point. After that, control variable j is 0 and the rest repetition will terminate immediately.
- After the upper time bound T, the discretised process is also after the N repetition, both sides will do nothing.

2) We now prove the case for continuous interrupt, i.e. $\langle \dot{x} = e\&B\rangle \trianglerighteq [\!]_{i\in I}(ch_i* \rightarrow p_i)$, denoted by *ODEI* below, and the discrete process (2) are approximate bisimilar. For this case, the discretization of ODE is involved, but also it needs to consider the communication behavior.
   There are two cases, detailed as follows:

   - If *ODEI* performs a d duration ($d > 0$), i.e. B holds true and no communication occurs within d duration, or if it performs a τ transition because of the violation of B, then these two cases are very similar to case continuous evolution. We can prove similarly that (2) can perform a corresponding sequence of transitions including a h duration progress and the update of continuous states x such that the final states are approximate bisimilar.
   - If *ODEI* performs a communication action, e.g. $\langle ch!, v\rangle$ (indexed by k), then there must exist a process in parallel such that it can perform the compatible action $\langle ch?, v\rangle$. There are two cases, the communication occurs at the time when B is true, or when B just turns false. For the discrete process (2), j_1 is 1 for the first case and j_1 is 0 for the second case. For both, j_2 is some i. So (2) will execute to the interrupt to line 4 and then to line 6. The communication is enabled and the right continuation is chosen to execute.

D. Proof of discrete bisimulation

We present the details of part of the bisimulation proof between discretised HCSP and the generated C code. We first prove the primitives related to time delay and communications.

1) Delay Case: We first consider the case of delay only. Given HCSP program p, we write the corresponding C program as $\overline{p}$. The bisimulation relation is given as follows. For each HCSP process i, one of the following two conditions hold:

(1a) The HCSP program is p_i and the C program is $\overline{p_i}$, moreover $\mathit{localTime}(i) = \mathit{currentTime}$.

(1b) For some $d_i > 0$, the HCSP program is $\mathsf{delay}(d_i); p_i$ and the C program is $\overline{p_i}$, moreover $\mathit{localTime}(i) = \mathit{currentTime} + d_i$.

Note the above condition implies *currentTime* is less than or equal to $\mathit{localTime}(i)$ for any i. We add following invariant conditions on the C state:

(2) The global variable *currentTime* equals the minimum of $\mathit{localTime}(i)$ for all i.

(3) For each thread i, its state is WAITING if $\mathit{localTime}(i) < \mathit{currentTime}$, otherwise its state is RUNNING. The value of the condition variable corresponds to its state for each thread.

Finally, the global time on the C side and HCSP side corresponds to each other:

(4) The value of *currentTime* equals the global clock in HCSP.

We now show that each execution of C program delay corresponds to either identity or the delay transition on the HCSP program. Note the entirety of delay is within a single critical region, so it can be considered atomic.

The operation performed by the C program $\mathsf{delay}(\mathit{tid}, \mathit{secs})$ is as follows. It increments $\mathit{localTime}(\mathit{tid})$ by secs, then updates *currentTime* to be the new minimum of $\mathit{localTime}(i)$.

Any thread whose $localTime$ equals $currentTime$ afterwards is woken up.

There are two cases to consider, depending on whether $\mathsf{delay}(tid, secs)$ results in a change of $currentTime$.

- If $currentTime$ is not changed, then the transition on the C side corresponds to identity transition on the HCSP side. For thread tid, since the thread is allowed to run, by Condition 3), we have $localTime(i) = currentTime$ beforehand, so Condition 1a) holds, the HCSP program has form $wait(secs); p$, and the C program has form $delay(tid, secs); \overline{p}$. After executing $delay(tid, secs)$, the C program becomes $\overline{p}$ and $localTime(i)$ becomes $currentTime + secs$, so Condition 1b) holds afterwards. The state of thread tid is set to WAITING, so Condition 3) holds afterwards. Finally Conditions 2) and 4) are unaffected.
- If $currentTime$ is incremented by some d, then the transition on the C side corresponds to transition $\langle d, \emptyset\rangle$ on the HCSP side. The HCSP transition increments the global clock by d, and removes $\mathsf{delay}(d)$ from all processes. On the C side, for each process, since $currentTime$ is in incremented by d and $\mathsf{delay}(d)$ is removed from HCSP program at the same time, Condition 1b) is either preserved or becomes Condition 1a). In the latter case, Condition 3) is preserved since thread states are updated and signals sent in function updateCurrentTime. Condition 2) continues to hold due to the computation of $currentTime$ in function updateCurrentTime.

2) Communication Case: Next, we consider the case of input and output statements. The C program for both input and output consist of two cases, depending on whether the other side is ready on entry to the function. Each case is divided into two atomic blocks. We first review the actions taken in each atomic block. Following the convention in the code listings, we let tid be the current thread, i be the other side of communication, and ch be the channel.

- Input, other side ready: The first block sets the input thread of ch to tid, sets the state of thread i to ch, copies value from channel ch, and signals thread i. The second block updates the local time of tid if necessary, updates the global clock, sets the thread state of tid to RUNNING, and finally sets the input thread of ch to -1.
- Input, other side not ready: The first block sets the input thread of ch to tid, sets the state of thread tid to AVAILABLE, assigns the local time to be the maximum DBL_MAX, and update the current time. The second block updates the local time of tid, copies value from channel ch, sets the state of thread tid to RUNNING, sets the input thread of ch to -1, and finally signals thread i.
- Output, other side ready: The first block sets the output thread of ch to tid, copies value sent to ch, sets the state of thread i to ch, and signals thread i. The second block updates the local time of tid if necessary, updates the global clock, sets the thread state of tid to RUNNING, and finally sets the input thread of ch to -1.
- Output, other side not ready: The first block sets the output thread of ch to tid, copies value sent to ch, sets the state of thread tid to AVAILABLE, assigns the local time to be the maximum DBL_MAX, and update the current time. The second block updates the local time of tid, sets the state of thread tid to RUNNING, sets the output thread of ch to -1, and finally signals thread i.

From these descriptions, we can conclude the following pattern: If the other side is ready, the first block performs the communication and signals to the other side of this fact, the second block performs other cleanup. If the other side is not ready, the first block sets appropriate variables and then wait for the other side to become ready, the second block updates local time and performs other cleanup.

We name the second block of input when the other side is ready (resp. other side is not ready) to be $\mathsf{input_a}$ (resp. $\mathsf{input_b}$). Likewise, we name the second block of output when the other side is ready (resp. other side is not ready) to be $\mathsf{output_a}$ (resp. $\mathsf{output_b}$). These notations will be used in the examples as well as when defining the bisimulation relation below.

Intuitively, each communication executes in one of the following two patterns:

- The input side is ready first, and enters into waiting for input mode. Then the output side provides the value, the input side receives it and signals to the output side on the success of communication. This corresponds to the following four steps:

$$\begin{array}{l}\mathsf{input}, \mathsf{output} \longrightarrow \mathsf{input_b}, \mathsf{output} \longrightarrow \\ \mathsf{input_b}, \mathsf{output_a} \longrightarrow \mathsf{skip}, \mathsf{output_a} \longrightarrow \mathsf{skip}, \mathsf{skip}\end{array}$$

- The output side is ready first, and enters into waiting for output mode. Then the input side receives the value and signals to the output side, finally the output side signals back to the input side on the success of communication. This corresponds to the following four steps:

$$\begin{array}{l}\mathsf{input}, \mathsf{output} \longrightarrow \mathsf{input}, \mathsf{output_b} \longrightarrow \\ \mathsf{input_a}, \mathsf{output_b} \longrightarrow \mathsf{input_a}, \mathsf{skip} \longrightarrow \mathsf{skip}, \mathsf{skip}\end{array}$$

Note the four steps may be interrupted by other actions in between. An example of the first case is given in Example 7, where a waiting action interrupts between the first and second step.

The key idea of the proof is to associate the step where the input side receives the transmitted value to the transition on the HCSP side. In the case when the input side is ready first, this is the third of the four transitions. In the case when the output side ready first, this is the second of the four transitions.

Hence, the bisimulation relation for the case when input side is ready first is given as follows. For each HCSP process i, one of the following conditions hold.

(1a) The HCSP program is p_i and the C program is $\overline{p_i}$, moreover the state is RUNNING, and $localTime(i) = \mathsf{currentTime}$.

(1b) The HCSP program is $ch?x; p_i$ and the C program is $\mathsf{input_b}; \overline{p_i}$, moreover the state is AVAILABLE, and

$\mathit{localTime}(i)$ is $\mathsf{DBL\_MAX}$. Finally the input side of channel ch is set to the current thread.

(1c) The HCSP program is $ch!e; p_i$ and the C program is $\mathsf{output_a}; \overline{p_i}$, moreover the thread is blocked, the output side of channel ch is set to the current thread, and the value of ch is set to the value of expression e at the current state.

(1d) The HCSP program is p_i and the C program is $\mathsf{output_a}; \overline{p_i}$, moreover the thread is unblocked, the value of x is set to the value of ch.

The idea of the proof is to show the following:

(1) The atomic transition from input to $\mathsf{input_b}$ carries a local state satisfying Condition 1a) to one satisfying Condition 1b), without any changes on the HCSP side.

(2) The atomic transition from $\mathsf{input_b}$ to skip carries a local state satisfying Condition 1b) to one satisfying Condition 1a), while the communication is carried out on the HCSP side. Hence the variable x is assigned, and the output thread changes from satisfying Condition 1c) to satisfying Condition 1d).

(3) The atomic transition from output to $\mathsf{output_a}$ carries a local state satisfying Condition 1a) to one satisfying Condition 1c), without any changes on the HCSP side.

(4) The atomic transition from $\mathsf{output_a}$ to skip carries a local state satisfying Condition 1d) to one satisfying Condition 1a), without any changes on the HCSP side.

3) General case $\mathsf{wait\_comm}$ *for interrupt:* We now consider the following HCSP primitive, added for the discretization of interrupt:

$$i := \mathsf{wait\_comm}(d, \mathit{chs}).$$

This primitive waits for time d as well as communications chs. If one of the communications in chs is ready before time d, then the communication is carried out, and i is assigned its index in chs. Otherwise, i is assigned -1.

Note delay, input and output are all special cases of $\mathsf{wait\_comm}$. The $\mathsf{delay}(d)$ operation corresponds to $\mathsf{wait\_comm}(d, [])$. The input operation $ch?x$ corresponds to $\mathsf{wait\_comm}(\infty, [ch?x])$, and the output operation $ch!e$ corresponds to $\mathsf{wait\_comm}(\infty, [ch!e])$.

The translation of the above primitive to C code have the following structure.

- First, for each communication in chs, update the corresponding $\mathsf{channelInput}$ and $\mathsf{channelOutput}$ arrays. In the case of output, update the channel value as well. Then collect the list of matching communications that are ready.
- If at least one matching communication is ready, nondeterministically choose one of them and proceed to carry out the communication as in the case of *input* or *output*. At the end, reset the information in $\mathsf{channelInput}$ and $\mathsf{channelOutput}$.
- If no matching communication is ready, the local time of the thread is incremented by secs and the current time is updated. If the local time is now greater than current time, the thread enters $\mathsf{WAITING\_AVAILABLE}$ mode and blocks.
- After the thread is unblocked, if the thread is in $\mathsf{AVAILABLE}$ mode, this means the time limit is reached first, then the function returns -1.
- After the thread is unblocked, if the thread state contains a channel, this means some communication happened first, then the corresponding operations for the second part of the communication is performed. In particular, this updates local time to the time of communication, which is the local time of the other thread.

We identify the following atomic action blocks for $\mathsf{wait\_comm}$. The block $\mathsf{wait\_comm}_{\mathsf{a}i}$ is the second atomic block when the i'th communication in chs is initially ready and chosen. The block $\mathsf{wait\_comm}_{\mathsf{b}i}$ is the second atomic block when the i'th communication happened during waiting. Finally, the entire block $\mathsf{wait\_comm}$ may be executed in the case when the time limit secs is reached without communication. The possible execution orders for $\mathsf{wait\_comm}$ are the following:

(1) $\mathsf{wait\_comm} \to \mathsf{wait\_comm}_{\mathsf{a}i} \to \mathsf{skip}$ for some $0 \le i < |\mathit{chs}|$, the return value is i.

(2) $\mathsf{wait\_comm} \to \mathsf{wait\_comm}_{\mathsf{b}i} \to \mathsf{skip}$ for some $0 \le i < |\mathit{chs}|$, the return value is i.

(3) $\mathsf{wait\_comm} \to \mathsf{skip}$, and the return value is -1.

The first and third cases are very similar to the communication case and the delay case proved previously, so we will not list the proofs for them here. Below we give the proof of the second case, for which the other sides of all channels are not ready at the beginning and then one of them gets ready after waiting for some time less than the time limit h. This corresponds to the following steps (assume the corresponding channel is ch and thus we omit the channel subscript for presentation):

$$\begin{array}{l}\mathsf{wait\_comm}, (\mathsf{delay}; \mathsf{output}) \longrightarrow \\ \mathsf{wait\_comm_b}, (\mathsf{delay}; \mathsf{output}) \longrightarrow \\ \mathsf{wait\_comm_b}, \mathsf{output} \longrightarrow \mathsf{wait\_comm_b}, \mathsf{output_a} \longrightarrow \\ \mathsf{skip}, \mathsf{output_a} \longrightarrow \mathsf{skip}, \mathsf{skip}\end{array}$$

The bisimulation relation for this relation is given as follows. For each HCSP process i, one of the following conditions hold:

(1a) The HCSP program is p_i and the C program is $\overline{p_i}$, moreover the state is $\mathsf{RUNNING}$, and $\mathit{localTime}(i) = \mathsf{currentTime}$.

(1b) The HCSP program is $\mathit{wait\_comm}; p_i$ and the C program is $\mathsf{wait\_comm_b}; \overline{p_i}$, moreover the state is $\mathsf{WAITING\_AVAILABLE}$, and $\mathit{localTime}(i)$ is added by h. Finally the output side of channel ch is set to the current thread.

(1c) The HCSP program is $ch!e; p_i$ and the C program is $\mathsf{output_a}; \overline{p_i}$, moreover the thread is blocked, the output side of channel ch is set to the current thread, and the value of ch is set to the value of expression e at the current state.

(1d) The HCSP program is p_i and the C program is $\mathsf{output_a}; \overline{p_i}$, moreover the thread is unblocked, the value of x is set to the value of ch.

The idea of the proof is to show the following:

(1) The atomic transition from $\mathsf{wait\_comm}$ to $\mathsf{wait\_comm_b}$ carries a local state satisfying Condition 1a) to one satisfying Condition 1b), without any changes on the HCSP side.
(2) The atomic transition from $\mathsf{delay}; \mathsf{output}$ to output carries a local state satisfying Condition 1a) to one still satisfying Condition 1a), corresponding to a wait statement on the HCSP side.
(3) The atomic transition from output to $\mathsf{output_a}$ carries a local state satisfying Condition 1a) to one satisfying Condition 1c), without any changes on the HCSP side.
(4) The atomic transition from $\mathsf{wait\_comm_b}$ to skip carries a local state satisfying Condition 1b) to one satisfying Condition 1a), while the communication is carried out on the HCSP side. Hence the variable x is assigned, and the output thread changes from satisfying Condition 1c) to satisfying Condition 1d).
(5) The atomic transition from $\mathsf{output_a}$ to skip carries a local state satisfying Condition 1d) to one satisfying Condition 1a), without any changes on the HCSP side.

4) Other compound cases: Above we have proved the primitives related to time delay and communication. Other cases such as in (2), the expressions, and the statements such as assignments, conditional, repetition with determined number of times, sequential composition, and so on, are transformed to C directly without any essential changes and the bisimulation relation between them and the generated C code can be proved trivially based on structural induction.

E. Examples to illustrate the proof

(1) Figure 6 shows an example with delay only. The initial HCSP program is

$$\mathsf{wait}(10) \parallel \mathsf{wait}(20) \parallel \mathsf{wait}(30)$$

(2) Figure 7 shows an example with input, output and delay. The initial HCSP program is

$$ch_1?x \parallel \mathsf{wait}(10); ch_1!3$$

VI. Case Study

We adopt the realistically-scaled Automatic Cruise Control System (ACCS for short) from [XWZ$^+$22] as the case study. The HCSP model of ACCS is moderately large and covers all the concerned features of HCSP and it is robustly safe (Sect. V-A) and hence can be discretized using our approach. In this section, we translate this HCSP model to the C code and compare its execution with the simulation of the HCSP model [XWZ$^+$22] and the execution of the C code generated directly from the original AADL $\oplus$ S/S model [ZLW$^+$19].

A. The Automatic Cruise Control System

The architecture of ACCS consists of three layers. The physical layer is the physical vehicle. The software level defines control of the system and it contains three processes for obstacle detection, velocity control, and panel control, and each process is composed of several threads. These processes interact with the environment (the physical layer) through devices. The platform layer consists of a bus and a processor. The connections between processes and devices could be bound to the bus and all the threads are bound to the processor, with HPF (High-Priority-First) scheduling policy.

B. Translation from HCSP to C

In work [ZLW$^+$19], combined models of $\mathsf{AADL} \oplus \mathsf{S/S}$ are translated to C code directly. In this paper, we generate the C code of the case study from its HCSP model which has been verified in [XWZ$^+$22]. Since the correctness of the translation is guaranteed (Sect. V), the generated code, especially the code for the controller (the velocity control mentioned in Sect. VI-A), satisfies the safety requirement and therefore is reliable. We use pthreads to implement the communication between processes and the correctness of the code generation has been proved in Sect. V. The generated C code is of 3500–4000 lines, longer than the C code (about 2500 lines) generated by [ZLW$^+$19]. One major reason is that the generated C code in this paper is approximately bisimilar to the original $\mathsf{AADL} \oplus \mathsf{S/S}$ model of ACCS, which means the detailed behaviours of ACCS are reflected in the C code and vice versa, while it is not the case for [ZLW$^+$19] where no such bisimulation can be guaranteed.

C. Comparison

We consider the following scenario. At the beginning, the driver pushes the `inc` button three times with time interval 0.5s in between to set a desired speed to 3m/s. After 30s, the driver pushes the `dec` button twice in 0.5s time intervals to decrease the desired speed. On the obstacle side, we assume that the obstacle appears at time 10s and position 35m, then moves ahead with velocity 2m/s, before finally moving away at time 20s and position 55m.

The top of Fig. 8 shows the execution results of the vehicle speed, where the black line denotes the desired velocity set by the driver, and the red, green, and blue lines denote the results of our work, the work of [ZLW$^+$19], and the simulation of the HCSP model of ACCS [XWZ$^+$22], respectively. We can see that the execution result (red line) of the generated C code is almost the same with the simulation (blue line) of its HCSP model. Specifically, the average relative error (ARE) between the time series of the velocity generated from the HCSP model and its C code is 0.138% with the variance 4.686×10^{-5}. Besides, we can also observe that there is negligible difference between the results of the C code generated by [ZLW$^+$19] (green line) and the C code generated in this paper (red line): the ARE is 0.182% with the variance 4.232×10^{-3}. Readers can refer to [XWZ$^+$22] for more details about this example.

p_1	p_2	p_3	$cTime$	HCSP
$\langle \mathsf{delay}(10), 0, R\rangle$	$\langle \mathsf{delay}(20), 0, R\rangle$	$\langle \mathsf{delay}(30), 0, R\rangle$	0	$\mathsf{wait}(10) \parallel \mathsf{wait}(20) \parallel \mathsf{wait}(30)$
$\langle \mathsf{skip}, 10, W\rangle$	$\langle \mathsf{delay}(20), 0, R\rangle$	$\langle \mathsf{delay}(30), 0, R\rangle$	0	$\mathsf{wait}(10) \parallel \mathsf{wait}(20) \parallel \mathsf{wait}(30)$
$\langle \mathsf{skip}, 10, W\rangle$	$\langle \mathsf{skip}, 20, W\rangle$	$\langle \mathsf{delay}(30), 0, R\rangle$	0	$\mathsf{wait}(10) \parallel \mathsf{wait}(20) \parallel \mathsf{wait}(30)$
$\langle \mathsf{skip}, 10, R\rangle$	$\langle \mathsf{skip}, 20, W\rangle$	$\langle \mathsf{skip}, 30, W\rangle$	10	$\mathsf{skip} \parallel \mathsf{wait}(10) \parallel \mathsf{wait}(20)$
$\langle \mathsf{skip}, 10, S\rangle$	$\langle \mathsf{skip}, 20, R\rangle$	$\langle \mathsf{skip}, 30, W\rangle$	20	$\mathsf{end} \parallel \mathsf{skip} \parallel \mathsf{wait}(10)$
$\langle \mathsf{skip}, 10, S\rangle$	$\langle \mathsf{skip}, 20, S\rangle$	$\langle \mathsf{skip}, 30, R\rangle$	30	$\mathsf{end} \parallel \mathsf{end} \parallel \mathsf{skip}$
$\langle \mathsf{skip}, 10, S\rangle$	$\langle \mathsf{skip}, 20, S\rangle$	$\langle \mathsf{skip}, 30, S\rangle$	30	$\mathsf{end} \parallel \mathsf{end} \parallel \mathsf{end}$

Fig. 6. Example of bisimulation between the execution of C program and HCSP program, for the starting process $\mathsf{wait}(10) \parallel \mathsf{wait}(20) \parallel \mathsf{wait}(30)$. Each C thread is specified by the triple $\langle p, lt, st\rangle$, where p is the program that remains to be executed, lt is the local time, and st is the state.

p_1	p_2	ch_1	$cTime$	HCSP
$\langle \mathsf{input}(ch_1), 0, R, 0\rangle$	$\langle \mathsf{delay}(10); \mathsf{output}(ch_1), 0, R, 0\rangle$	$\langle -1, -1, \mathsf{null}\rangle$	0	$ch_1?x \parallel \mathsf{wait}(10); ch_1!3$
$\langle \mathsf{input_b}(ch_1), 0, A, 1\rangle$	$\langle \mathsf{delay}(10); \mathsf{output}(ch_1), 0, R, 0\rangle$	$\langle p_1, -1, \mathsf{null}\rangle$	0	$ch_1?x \parallel \mathsf{wait}(10); ch_1!3$
$\langle \mathsf{input_b}(ch_1), 0, A, 1\rangle$	$\langle \mathsf{output}(ch_1), 10, R, 0\rangle$	$\langle p_1, -1, \mathsf{null}\rangle$	10	$ch_1?x \parallel ch_1!3$
$\langle \mathsf{input_b}(ch_1), 0, ch_1, 1\rangle$	$\langle \mathsf{output_a}(ch_1), 10, R, 0\rangle$	$\langle p_1, p_2, 3\rangle$	10	$ch_1?x \parallel ch_1!3$
$\langle \mathsf{skip}, 10, R, 0\rangle$	$\langle \mathsf{output_a}(ch_1), 10, R, 0\rangle$	$\langle -1, p_2, 3\rangle$	10	$\mathsf{skip} \parallel \mathsf{skip}$
$\langle \mathsf{skip}, 10, R, 0\rangle$	$\langle \mathsf{skip}, 10, R, 0\rangle$	$\langle -1, -1, \mathsf{null}\rangle$	10	$\mathsf{skip} \parallel \mathsf{skip}$

Fig. 7. Example of bisimulation between the execution of C program and HCSP program, for the starting process $ch_1?x \parallel \mathsf{wait}(10); ch_1!3$. Each C thread is specified by the quadruple $\langle p, lt, st, pw\rangle$, where p is the program that remains to be executed, lt is the local time, st is the state, and pw is the boolean variable permWait. Each channel is specified by the triple $\langle in, out, val\rangle$, where in is the input side, out is the output side, and val is the value to be transmitted.

It should be noted that although the generated C code in this paper is longer and somewhat less efficient than the C code generated directly from the original graphical model [ZLW+19], the former is more reliable because it is translated from the (formal) HCSP model, which can be verified by tools like HHLPy [SBZ23], and moreover, the correctness of the translation can be guaranteed (Sect. V).

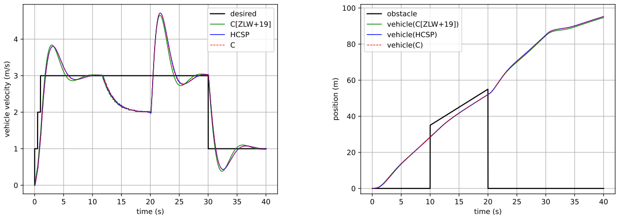

Fig. 8. Comparison of execution results

VII. Conclusions

This paper presents a formally verified C code generator from hybrid systems modelled using HCSP. In the transformation from HCSP to C, two main issues are addressed: the discretisation of continuous evolution and interrupts, and the realisation of synchronized communication of HCSP in concurrent C with the use of mutexes. For the first problem, we provide a method to compute a discretised time step such that under some robustness conditions, the continuous HCSP processes and the corresponding discrete ones are approximate bisimilar within the given precision. For the second problem, we propose a new notion of bisimulation relation: each transition step of HCSP corresponds to a sequence of atomic transitions of C, among which the substantial step to perform the equivalent transition in HCSP is determined. By transitivity of bisimulation relations, the correctness of the transformation from HCSP to C is guaranteed. Finally, we investigate a case study on Automatic Cruise Control System and its code generation using our approach. For future work, we consider to apply our approach to more practical case studies in industry.

References

[AFH+10] M. Anand, S. Fischmeister, Y. Hur, J. Kim, and I. Lee. Generating reliable code from hybrid-systems models. *IEEE Trans. Computers*, 59(9):1281–1294, 2010.

[BBCP12] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. Non-standard semantics of hybrid systems modelers. *Journal of Computer and System Sciences*, 78:877–910, May 2012.

[BBD+17] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg. A formally verified compiler for Lustre. In *PLDI 2017*, 2017.

[BM05] J. A. Bergstra and C. A. Middelburg. Process algebra for hybrid systems. *Theor. Comput. Sci.*, 335(2-3):215–280, 2005.

[BP13] T. Bourke and M. Pouzet. Zélus: a synchronous language with ODEs. In *HSCC 2013*, pages 113–118. ACM, 2013.

[BTM+18] R. Bohrer, Y. K. Tan, S. Mitsch, M. O. Myreen, and A. Platzer. VeriPhy: verified controller executables from verified cyber-physical system models. In *PLDI 2018*, pages 617–630. ACM, 2018.

[CR05] P. J. L. Cuijpers and M. A. Reniers. Hybrid process algebra. *J. Log. Algebraic Methods Program.*, 62(2):191–245, 2005.

[DASP21] G. Dupont, Y. A. Ameur, N. K. Singh, and M. Pantel. Event-B hybridation: A proof and refinement-based framework for modelling hybrid systems. *ACM Trans. Embed. Comput. Syst.*, 20(4):35:1–35:37, 2021.

[Dor08] F. X. Dormoy. SCADE 6: a model based solution for safety critical software development. In *ERTS 2008*, 2008.

[FKL+18] Y. Feng, J.-P. Katoen, H. Li, B. Xia, and N. Zhan. Monitoring CTMCs by multi-clock timed automata. In *CAV 2018, Part I*, volume 10981 of *LNCS*, pages 507–526. Springer, 2018.

[FMQ+15] N. Fulton, S. Mitsch, J.-D. Quesel, M. Völp, and A. Platzer. KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In *CADE 2015*, volume 9195 of *LNCS*, pages 527–538. Springer, 2015.

[GP07] A. Girard and G. J. Pappas. Approximation metrics for discrete and continuous systems. *IEEE Trans. Autom. Contr.*, 52(5):782–798, 2007.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proc. IEEE*, 79(9):1305–1320, 1991.

[He94] J. He. From CSP to hybrid systems. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*, pages 171–189. Prentice Hall International (UK) Ltd., 1994.

[Hen96] T. A. Henzinger. The theory of hybrid automata. In *LICS 1996*, pages 278–292. IEEE Computer Society, 1996.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[JDBP09] A. A. Julius, A. D'Innocenzo, M. D. D. Benedetto, and G. J. Pappas. Approximate equivalence and synchronization of metric transition systems. *Syst. Control. Lett.*, 58(2):94–101, 2009.

[Mat13a] MathWorks Inc. *Simulink User's Guide*, 2013. http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf.

[Mat13b] MathWorks Inc. *Stateflow User's Guide*, 2013. http://www.mathworks.com/help/pdf_doc/stateflow/sf_ug.pdf.

[OSA17] OSATE. 2017. https://osate.org.

[Pla18] A. Platzer. *Logical Foundations of Cyber-Physical Systems*. Springer, 2018.

[SB13] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. 2013.

[SBZ23] H. Sheng, A. Bentkamp, and B. Zhan. HHLPy: Practical verification of hybrid systems using Hoare logic. In *FM 2023*, volume 14000 of *LNCS*, pages 160–178. Springer, 2023.

[WZA18] L. Wang, N. Zhan, and J. An. The opacity of real-time automata. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 37(11):2845–2856, 2018.

[XWZ+22] X. Xu, S. Wang, B. Zhan, X. Jin, J.-P. Talpin, and N. Zhan. Unified graphical co-modeling, analysis and verification of cyber-physical systems by combining AADL and Simulink/Stateflow. *Theor. Comput. Sci.*, 903:1–25, 2022.

[YJW+20] G. Yan, L. Jiao, S. Wang, L. Wang, and N. Zhan. Automatically generating SystemC code from HCSP formal models. *ACM Trans. Softw. Eng. Methodol.*, 29(1):4:1–4:39, 2020.

[ZLW+19] H. Zhan, Q. Lin, S. Wang, J.-P. Talpin, X. Xu, and N. Zhan. Unified graphical co-modelling of cyber-physical systems using AADL and Simulink/Stateflow. In *UTP 2019*, volume 11885 of *LNCS*, pages 109–129. Springer, 2019.

[ZWZ17] N. Zhan, S. Wang, and H. Zhao. *Formal Verification of Simulink/Stateflow Diagrams (A Deductive Approach)*. Springer, 2017.

[ZYM08] F. Zhang, M. Yeddanapudi, and P. J. Mosterman. Zero-crossing location and detection algorithms for hybrid system simulation. *IFAC Proceedings Volumes*, 41(2):7967–7972, 2008.

[ZZW+13] L. Zou, N. Zhan, S. Wang, M. Fränzle, and S. Qin. Verifying Simulink diagrams via a hybrid Hoare logic prover. In *EMSOFT 2013*, pages 9:1–9:10. IEEE, 2013.

[ZZW+23] N. Zhan, B. Zhan, S. Wang, D. Guelev, and X. Jin. A generalized hybrid Hoare logic. In *arxiv.org/abs/2303.15020*, 2023.

Appendix

A. *Part of The Generated C code for HCSP*

1) Delay: The time delay statement, wait *sec*, is implemented by the following C function `delay(tid, secs)`, where `tid` is the ID of current thread and `secs` the delayed time duration.

```
1  void delay(int tid, double secs) {
2      lock mutex;
3      localTime[tid] += secs;
4      updateCurrentTime(tid);
5      if (localTime[tid] > currentTime) {
6          threadState[tid] = WAITING;
7          cwait cond[tid] mutex;
8      }
9      unlock mutex;
10 }
```

It first locks `mutex`, and then increases the local clock of `tid` by `secs`. As the local clock increases, the global clock (which is the minimum of all local clocks) needs to be updated, and meanwhile, releases the threads on waiting states to execute, defined by `updateCurrentTime(tid)`. After this, the state of thread `tid` is set to be `WAITING`, then waits till other threads catch up with it.

For `updateCurrentTime(tid)`, lines (2-8) assigns `minLocal` to be the minimum of the local clocks of active threads that are not waiting for communication. If the global clock is less than `minLocal`, it is progressed to be equal to `minLocal` (Line 9). Lines (12-21) checks all threads (other than current thread): if the local clock of the thread is equal to the global clock and meanwhile its state is `WAITING`, then update it to be `RUNNING` and wake it up by sending a signal; and if the state of the thread is `WAITING_AVAILABLE`, then update it to be `AVAILABLE`, meaning that the thread still needs to wait for some communication.

```
1  void updateCurrentTime(int tid) {
2      double minLocal = DBL_MAX;
3      for (int i = 0; i < numThread; i++) {
4          if (threadState[i] != STOPPED &&
5              localTime[i] < minLocal) {
6              minLocal = localTime[i];
7          }
8      }
9      if (currentTime < minLocal) {
10         currentTime = minLocal;
11     }
12     for (int i = 0; i < numThread; i++) {
13         if (i != tid && localTime[i] == currentTime &&
14             threadState[i] == WAITING) {
15             threadState[i] = RUNNING;
16             signal cond[i];
17         }
18         if (i != tid && localTime[i] == currentTime &&
19             threadState[i] ==  WAITING_AVAILABLE) {
20             threadState[i] =  AVAILABLE;
21             signal cond[i];
22         }
23     }
24 }
```

2) Input: The input $ch?x$ is transformed into `input (tid, chi)`, where `tid` is the ID of current thread and `chi` is the input channel corresponding to $ch?x$.

```
1  void input(int tid, Channel chi) {
2    lock mutex;
3    channelInput[chi.channelNo] = tid;
4    int i = channelOutput[chi.channelNo];
5    if (i != -1 & (threadState[i] == AVAILABLE ||
6       threadState[i] == WAITING_AVAILABLE)) {
7      threadState[i] = chi.channelNo;
8      copyFromChannel(chi);
9      signal cond[i];
10     cwait cond[tid] mutex;
11     channelInput[chi.channelNo] = -1;
12   }
13   else {
14     threadState[tid] = AVAILABLE;
15     localTime[tid] = DBL_MAX;
16     updateCurrentTime(tid);
17     cwait cond[tid] mutex;
18     copyFromChannel(chi);
19     threadState[tid] = RUNNING;
20     localTime[tid] = localTime[i];
21     channelInput[chi.channelNo] = -1;
22     signal cond[i];
23   }
24   unlock mutex;
25 }
```

At the beginning (lines 2-4), thread `tid` first locks `mutex`, sets current thread `tid` to be ready for the input side of channel `chi.channelNo` (i.e. the ID shared by `chi` and the compatible output channel, denoted by *cid* below); Let `i` be the thread of the compatible output, recored in `channelOutput`. If the sender is already ready on thread `i` (indicated by `i != -1`) and furthermore `i` is on a state waiting for communication, then the following sequence of actions is performed (lines 5 - 13): the state of sender thread `i` is set to be the channel ID `chi` (line 7), the input channel `chi` copies value from the channel content that is already written by the output side and assigns it to variable x (line 8); then, thread `tid` issues signal `cond[i]` to the sender `i` (line 9), releases `mutex` and waits on the reply signal from the sender (line 10). After receiving the signal from the sender, the whole communication completes, resulted in setting the thread of input `chi` to be -1 (indicating it becomes unready).

Otherwise, if the sender is not ready to execute (lines 13-224): first sets the state of thread `tid` to be `Available` to indicate that it is waiting for the compatible output (line 15), make `localtime` to be infinity (line 16); after that, releases lock `mutex` and waits for the sender (line 18); as soon as it is woken up again, meaning that the sender gets ready, the value can be obtained from the channel content array (line 19); the local time is updated, the state of `tid` is reset to `RUNNING`, and `chi` becomes unready (lines 20-22), and finally wakes up the sender thread to notify that the communication is finished.

3) Continuous Evolution: The C implementation of continuous evolution is `odec(tid)` as follows. It has almost the same control structure as the discretized ODE given in the paper, with wait statement implemented using function `delay`.

```
1  void odec(int tid) {
2      int i = 1;
3      while (i == 1) {
4          if (!N(B, eps) || !N^n(B, eps)) {i = 0;}
5          if (i == 1) {delay(tid, h); x = x + h*Phi(x,h);}
6          if (currentTime >= T) {i = 0; stop;}
7      }
8  }
```

4) Continuous Interrupt: The C implementation is `odeI (tid, nums, chs)`, followed by an if-then statement on recursive transformation of p_is depending on the return value of odeI. Here nums denotes the number of communication actions in I, chs is the array storing these actions. It is defined with the help of three auxiliary functions: interruptInit, interruptPoll, and interruptClear. After each polling operation of duration h, x is updated, followed by two checks: if it reaches the boundary, the communication interrupt terminates directly, indicated by ret=-1; if the execution time exceeds the upper bound T, the whole execution stops.

```
void odeI(int tid, int nums, Channel* chs) {
  interruptInit(tid, nums, chs);
  int i = 1, ret = -1;
  while (i == 1) {
    recordTime = localTime[tid];
    ret = interruptPoll(tid, h, nums, chs);
    if (ret >= 0) {
      i = 0;
      h = localTime[tid] - recordTime;
    }
    x = x + h*Phi(x,h);
    if (ret < 0) {
      if (!N(B, eps) || !N^n(B, eps)) {
        i = 0;
        ret = -1;
        interruptClear(tid, nums, chs);
      }
      if (currentTime >= T) {
        i = 0;
        interruptClear(tid, nums, chs);
        stop;
      }
    }
  }
}
```

Function interruptInit initializes actions in chs to be ready, and writes values to channelContent buffer from outputs by copyToChannel, and sets thread state to be AVAILABLE.

```
void interruptInit(int tid, int nums, Channel* chs) {
  lock mutex;
  int curChannel;
  for (int i = 0; i < nums; i++) {
    curChannel = chs[i].channelNo;
    if (chs[i].type == 0) {
      channelInput[curChannel] = tid;
    } else {
      channelOutput[curChannel] = tid;
      copyToChannel(chs[i]);
    }
  }
  threadState[tid] = AVAILABLE;
  return;
}
```

Function interruptPoll checks whether some communication in chs can occur during d duration.

```
int interruptPoll(int tid, double seconds, int nums,
                  Channel* chs) {
  int curChannel, av[nums];
  for (int i = 0; i < nums; i++) {
    curChannel = chs[i].channelNo;
    if (chs[i].type == 0) {
      channelInput[curChannel] = tid;
      m = channelOutput[curChannel];
      if (m != -1 && (threadState[m] == AVAILABLE ||
          threadState[m] == WAITING_AVAILABLE)) {
        av[k] = i; k++;
      }
      else {
        channelOutput[curChannel] = tid;
        copyToChannel(chs[i]);
        m = channelInput[curChannel];
        if (m != -1 && (threadState[m] == AVAILABLE ||
            threadState[m] == WAITING_AVAILABLE)) {
          av[k] = i; k++;
        }
      }
    }
  }
  if (k > 0) {
    u = nondet(0, ..., k-1);
    i = av[u];
    curChannel = chs[i].channelNo;
    if (chs[i].type == 0) {
      m = channelOutput[curChannel];
      inAFout(tid, m, chs[i]);
    }
    else {
      m = channelInput[curChannel];
      outAFin(tid, m, chs[i]);
    }
    clearchannels (chs, nums);
    unlock mutex;
    return av[u];
  }
  localTime[tid] += seconds;
  updateCurrentTime(tid);
  if (localTime[tid] > currentTime) {
    threadState[tid] = WAITING_AVAILABLE;
    cwait cond[tid] mutex;
  }
  if (threadState[tid] == AVAILABLE) {
    return -1;
  }
  curChannel = threadState[tid];
  int match_index = -1;
  for (int j = 0; j < nums; j++) {
    if (chs[j].channelNo == curChannel) {
      match_index = j;
    }
  }
  if (chs[match_index].type == 0) {
    copyFromChannel(chs[match_index]);
    m = channelOutput[curChannel];
  } else {
    m = channelInput[curChannel];
  }
  localTime[tid] = localTime[m];
  threadState[tid] = RUNNING;
  clearchannels(chs, nums);
  signal cond[m];
  unlock mutex;
  return match_index;
}
```

Function interruptClear resets the thread state and clear channels. Finally it releases mutex and returns.

```
void interruptClear(int tid, int nums, Channel* chs) {
    threadState[tid] = RUNNING;
    clearchannels(chs, nums);
    unlock mutex;
    return;
}
```