A Two-Way Path Between Formal and Informal Design of Embedded Systems

Mingshuai Chen[1], Anders P. Ravn[2], Shuling Wang[1], Mengfei Yang[3], and Naijun Zhan[1](✉)

[1] State Key Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing, China
{chenms,wangsl,znj}@ios.ac.cn
[2] Department of Computer Science, Aalborg University, Aalborg, Denmark
[3] Chinese Academy of Space Technology, Beijing, China

Abstract. It is well known that informal simulation-based design of embedded systems has a low initial cost and delivers early results; yet it cannot guarantee the correctness and reliability of the system to be developed. In contrast, the correctness and reliability of the system can be thoroughly investigated with formal design, but it requires a larger effort, which increases the development cost. Therefore, it is desirable for a designer to move between formal and informal design. This paper describes how to translate Hybrid CSP (HCSP) formal models into Simulink graphical models, so that the models can be simulated and tested using a MATLAB platform, thus avoiding expensive formal verification if the development is at a stage where it is considered unnecessary. Together with our previous work on encoding Simulink/Stateflow diagrams into HCSP, it provides a two-way path in the design of embedded systems, so that the designer can flexibly shift between formal and informal models. The translation from HCSP into Simulink diagrams is implemented as a fully automatic tool, and the correctness of the translation is justified using Unifying Theories of Programming (UTP).

Keywords: Simulink · HCSP · UTP · Verification · Hybrid systems

1 Introduction

Correct and efficient design of complex embedded systems is a grand challenge for computer science and control theory. Model-based design (MBD) is thought to be an effective solution. This approach begins with an abstract model of the system to be developed. Extensive analysis and verification of the abstract model are then performed so that errors can be identified and corrected at a very early stage. Then the higher-level abstract model is refined to step by step to more detailed models till a level where the system can be built with existing components or a few newly developed ones.

Many MBD approaches targeting embedded systems have been proposed and used in industry and academia. These approaches can be simulation-based informal ones such as Simulink/Stateflow [1,2], Modelica [3], SysML [4],

J.P. Bowen and H. Zhu (Eds.): UTP 2016, LNCS 10134, pp. 65–92, 2017.
DOI: 10.1007/978-3-319-52228-9_4

MARTE [5], or they can be verification based like Metropolis [6], Ptolemy [7], hybrid automata [8], CHARON [9], HCSP [10,11], Differential Dynamic Logic [12], Hybrid Hoare Logic [13]. It is evident that informal design of embedded systems has a low initial cost and is intuitively appealing, because simulations give results early on, but it cannot fully guarantee the correctness and reliability of the system to be developed; in contrast, the correctness and reliability of the system can be thoroughly investigated with formal design, but the cost is higher and it requires specialized skills. Therefore, it is desirable to provide a two-way path between formal and informal approaches for a designer.

The first contribution of this paper is to provide one lane of this path. It takes a formal model and translates it automatically to a Simulink model. The other lane has been developed in previous work [14,15]. The translation from the formal to informal model presented here, is implemented as a fully automatic tool. Another contribution of this paper is to provide a justification of the correctness of the translation. To this end, we define a UTP semantics for Simulink and a UTP semantics for HCSP, and then establish a correspondence between the two. Due to lack of space, the implementation and a case study on a lunar lander have been omitted and can be found in [16].

1.1 Related Work

There has been a range of works on translating Simulink/Stateflow into modelling formalisms supported by analysis and verification tools. Mathworks itself released a tool named *Simulink Design Verifier* [17] (SDV) for formal analysis of Simulink/Stateflow models. However, currently, SDV can only be used to detect low-level errors such as integer overflow, dead logic, array access violation, division by zero, and so on, in blocks of a model, but not system-level properties of the complete model with the physical and environmental aspects taken into account.

Simulation-based verification [18] can be used to verify system-level properties in a bounded time, but cannot be applied for unbounded verification. Thus there is work on translating Simulink into other modelling formalisms, for which analysis and verification tools are developed. Tripakis *et al.* [19] presented an algorithm of translating discrete-time Simulink models to Lustre, a synchronous language developed with formal semantics and a number of tools for validation and analysis, and later extended the work by incorporating a subset of Stateflow [20]. Cavalcanti *et al.* [21] presented a semantics for discrete-time Simulink diagrams using Circus [22], a combination of Z and CSP. Meenakshi *et al.* [23] gave an algorithm that translates a subset of Simulink intothe input language of model checker NuSMV. Sifakis *et al.* proposed a translation into BIP in [24]. BIP [25] stands for Behaviour, Interaction and Priority, which is a component-based formal model for real-time concurrent systems. These works do not consider continuous time models of Simulink. This is considered in the follpwing works. Yang and Vyatkin [26] translate Simulink into Function Blocks. Zhou and Kumar investigated how to translate Simulink into *Input/Output Hybrid*

Automata [27], while the translation of both discrete and continuous time fragments of Simulink into *SpacEx Hybrid Automata* was considered in [28]. In [29], Chen *et al.* translates Simulink models to a real-time specification language Timed Interval Calculus (TIC). In this, continuous Simulink diagrams can be analyzed by a theorem prover. However, the translation is limited as it handles only continuous blocks whose outputs can be represented explicitly by a mathematical relation on inputs. In contrast, in [14] is a translation from Simulink into HCSP which handles all continuous blocks using the notion of differential equations and invariants.

Contract-based frameworks for Simulink are described in [30,31]. In [30], Simulink diagrams are represented by SDF graphs, and discrete-time blocks are specified by contracts with pre/post-conditions. Then sequential code is generated from the SDF graph, and the code is verified using traditional refinement-based techniques. In [31], Simulink blocks are annotated with rich types, then the SimCheck tool extracts verification conditions from the Simulink model and the annotations, and submits them to an SMT solver for verification. While in our approach, all Simulink/Stateflow models can be specified and verified using Hybrid Hoare Logic and its deductive verification techniques.

In [32], a compositional formal semantics built on predicate transformers was proposed for Simulink, based on which, a tool for verification of Simulink blocks was reported in [33], consisting of two components: a translator from Simulink hierarchical block diagrams into predicate transformers and an implementation of the theory of predicate transformers in Isabelle. The UTP semantics of Simulink/Stateflow defined here is quite similar to the one given in [32].

There have been several formal semantics defined for HCSP. In He's original work on HCSP [10], an algebraic semantics of HCSP was given. Subsequently, a Duration Calculus (DC) based semantics for HCSP was presented in [11]. These two original formal semantics of HCSP are very restrictive and incomplete, for example, it is unclear whether the set of algebraic rules defined in [10] is complete, and *super-dense computations* and recursion are not well handled in [11]. In [13,34–36], operational, axiomatic and DC-based denotational semantics for HCSP are proposed, and the relations among them are discussed. In this paper, we re-investigate the semantics of HCSP by defining its simulation semantics using Simulink and its UTP-based denotational semantics, and study the correspondence between the two semantics.

The rest of this paper introduces HCSP and Simulink in Sect. 2, Sect. 3 presents the translation from HCSP into Simulink. Section 4 presents a justification of the translation by proving consistency of the UTP semantics. A conclusion is in Sect. 5.

2 Preliminaries

HCSP [10,11,34] is a language for describing hybrid systems. It extends the well-known language of Communicating Sequential Processes (CSP) with timing constructs, interrupts, and differential equations for modelling continuous evolution. Data exchange among processes is confined to instantaneous synchronous communication, avoiding shared variables between parallel processes.

Simulink [1] is an interactive platform for modelling, simulating and analyzing multidomain dynamic and embedded systems. It provides a graphical block diagramming tool and a customizable set of block libraries for building executable models.

2.1 Hybrid CSP (HCSP)

The syntax of HCSP processes is given below:

$$\begin{aligned} P ::=& \,\mathrm{skip} \mid x := e \mid ch?x \mid ch!e \mid P;Q \mid B \to P \mid P \sqcup Q \mid P^* \\ &\mid \langle F(\dot{s}, s) = 0 \& B\rangle \mid \langle F(\dot{s}, s) = 0 \& B\rangle \trianglerighteq []_{i\in I}(io_i \to Q_i) \\ S ::=& \,P \mid S \| S \end{aligned}$$

Here x and s stand for variables, B and e are conventional Boolean and arithmetic expressions. P, Q, Q_i are sequential processes; and io_i stands for a communication event, which is either $ch?x$ or $ch!e$, and ch for a channel name. A system S is either a sequential process, or a parallel composition of several sequential processes.

The processes taken from CSP, skip, $x := e$ (assignment), $ch?x$ (input), $ch!e$ (output), $P;Q$ (sequential composition), $B \to P$ (conditional statement), P^* (repetition), $P \sqcup Q$ (internal choice), and $S\|S$ (parallel composition) have their standard meaning.

The evolution statement is $\langle F(\dot{s}, s) = 0 \& B\rangle$, where s represents a vector of real variables and $\dot{s}$ the first-order time derivative of s. It forces s to evolve according to the differential equations defined by the functional $\mathcal{F}$ as long as B holds, and it terminates immediately when B becomes false.

The process $\langle F(\dot{s}, s) = 0 \& B\rangle \trianglerighteq []_{i\in I}(io_i \to Q_i)$ behaves like $\langle F(\dot{s}, s) = 0 \& B\rangle$, except that the evolution is preempted as soon as one of the communications io_i occurs. That is followed by the respective Q_i. However, if the evolution statement terminates before a communication occurs, then the process terminates immediately.

2.2 Simulink

A Simulink model contains a set of blocks, subsystems, and wires, where blocks and subsystems cooperate by setting values on the wires between them. An elementary block gets input signals and computes the output signals. However, to make Simulink more useful, almost every block in Simulink contains some user-defined parameters to alter its functionality. One typical parameter is *sample time* which defines how frequently the computation is. done. Two special values, 0 and -1, may be set for sample time, where the sample time 0 indicates that the block is used for simulating the physical environment and hence computes continuously, and -1 signifies that the sample time of the block is not set, it will be determined by the sample times of the in-going wires to the block. Thus,

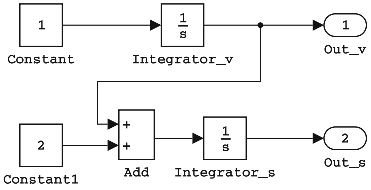


Fig. 1. $\dot{v} = 1, \dot{s} = v + 2$

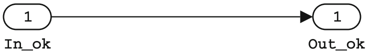


Fig. 2. skip statement

blocks are classified into two categories, i.e. *continuous* and *discrete*, according to their sample times.

Blocks and subsystems in a Simulink model receive inputs and compute outputs in parallel, and wires specify the data flow between blocks and subsystems. Computation in a block takes no time and the output is delivered immediately to its receiver.

As a convention, in the sequel, when describing Simulink diagrams, we use x to stand for the input signal on in-port `In_x`, x' for the output signal on out-port `Out_x`, possibly with a subscript to indicate which subsystem the signal belongs to. For instance, x'_P indicates an output signal on `Out_x` inside a subsystem `P`.

3 From HCSP to Simulink

The translation from HCSP processes into graphical Simulink models starts from the most basic ingredients, i.e. expressions, to primitive statements and then is followed by compositional components.

3.1 Expressions

Arithmetic expressions in HCSP are translated to a normal subsystem in Simulink. A variable x is encoded into an input block of the subsystem, a constant c into a constant block with corresponding value, and parentheses determine priority of the computation. As for the operations over reals, a sequence of $+$ and $-$ (or $*$ and $/$) is shrunk into a sum (or product) block with multiple input signals in Simulink. In the example for assignment in Fig. 3 is included the Simulink subsystem for the expression $x + y * z$. *Boolean expressions* are translated similarly.

3.2 Differential Equations

The syntax of differential equations in HCSP is $F \mathrel{\widehat{=}} \dot{s} = e \mid F, F$, where s stands for a continuous variable, $\dot{s}$ is the time derivative of s, and F, F indicates a group of differential equations that evolve simultaneously over time.

Each single differential equation is encoded into a continuous integrator block with an input signal of the value of e and an output signal of s; equations in the same group are a normal subsystem in Simulink. For example, $\dot{v} = 1, \dot{s} = v + 2$, becomes the block in Fig. 1, the integrator block of s takes the value of $v+2$ and an internal initial value s_0 to calculate the integral and then generate a signal of s, i.e. $s(t) = \int_{t_0}^{t}(v(t) + 2)dt + s_0$.

3.3 `skip` Statement

In HCSP, skip terminates immediately with no effect on the process, and thus there is intuitively no need to draw anything in Simulink diagrams. However, blocks and subsystems in a Simulink model are running inherently in parallel, but processes in HCSP can be executed sequentially, thus we need a mechanism to specify sequential execution in a Simulink diagram. Inspired by UTP [37], we introduce Boolean signals ok and ok' into each subsystem to represent initiation and termination. Whenever ok' is false, the process has not terminated and the final values of the process variables are unobservable. Similarly, when ok is false, the process has not started and even the initial values are unobservable. These conventions permit translation of sequential composition. Both ok and ok' are local to each HCSP process, and they never occur in expressions.

In a Simulink subsystem ok and ok' are given by an in-port `In_ok` and an out-port `Out_ok` respectively. Since skip does nothing and terminates instantly, the subsystem for skip in Simulink in Fig. 2 has $ok' = ok$, indicating that whenever skip starts, it terminates immediately without any effect.

3.4 Assignment

Figure 3 illustrates the subsystem in Simulink with an example of assignment $x := x + y * z$, where for ease of understanding, we unpack the subsystem of arithmetic expression e. The output signals are computed by the following equations:

$$ok' = ok \qquad x' = \begin{cases} x'_{new}, & ok \wedge \neg d(ok) \\ x, & \neg ok \wedge \neg d(ok) \\ d(x'), & d(ok) \end{cases} \qquad \mathbf{u}' = \mathbf{u}$$

Here, $\mathbf{u}$ stands for the set of signals that are not processed by the current subsystem, i.e. y and z in this example. x'_{new} represents the newly computed signal, here produced by block `Add1`. Moreover, we use $d(x)$ to denote the value of x in the previous period. It is kept through a unit delay block that holds its input for one period of the sample time.

3.5 Continuous Evolution

The Simulink diagram translated from an evolution in HCSP is shown in Fig. 4, where the group of differential equations $\mathcal{F}$ and the Boolean condition B are encapsulated into a single subsystem respectively. The enabled subsystem `F`

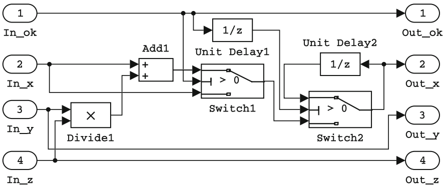


Fig. 3. $x := x + y * z$

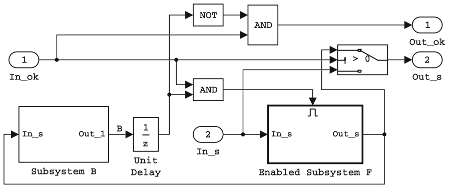


Fig. 4. Continuous evolution

contains a set of integrator blocks corresponding to the vector s of continuous variables, and executes continuously whenever the value of the input signal, abbreviated as en, on the enable-port is positive. Intuitively, subsystem `B` guards the evolution of subsystem `F` by taking the output signals of `F` as its inputs, i.e. $s_B = s'_F$, and partially controlling the enable signal of `F` via its output Boolean signal, denoted by B. As a consequence, an algebraic loop occurs between subsystem `B` and `F` which is not allowed in Simulink, the simple solution is to introduce a unit delay block with an initial value 1 inserted after subsystem `B`. Thus the boundary condition is evaluated after completion of an integrator step. Formally, given inputs, the output signals are computed by the following equations:

$$en = ok \wedge d(B) \qquad ok' = ok \wedge \neg d(B) \qquad s' = \begin{cases} s'_F, & ok \\ s, & \neg ok \end{cases}$$

3.6 Conditional Statement

Figure 5 illustrates the translation from a conditional statement of HCSP into a Simulink diagram. In most cases, subsystem `B` and `P` share the same group of input signals x, and for those distinct input signals, we add corresponding inports for `B` or `P`, which is not presented in Fig. 5. Accordingly, the output signals are computed according to

$$ok_P = ok \wedge B \qquad ok' = \begin{cases} ok'_P, & B \\ ok, & \neg B \end{cases} \qquad x' = x'_P.$$

3.7 Internal Choice

Given an internal choice $P \sqcup Q$, we use $outSigs$(P) and $outSigs$(Q) to represent the set of output signals (including ok') of subsystem `P` and `Q` respectively, and encode the random choice according to the following two situations.

- For each $x' \in outSigs(\mathrm{P}) \cap outSigs(\mathrm{Q})$, we introduce a switch block in Simulink diagrams for signal routing, which switches x' between x'_P from `P` and x'_Q from `Q` based on the value of the second input.
- For each $y' \in outSigs(\mathrm{P}) - outSigs(\mathrm{Q})$, we directly output the signal y'_P from `P` as the final value of y', because in case that P is not chosen by the system, y' stays unchanged. For each $z' \in outSigs(\mathrm{Q}) - outSigs(\mathrm{P})$, analogously.

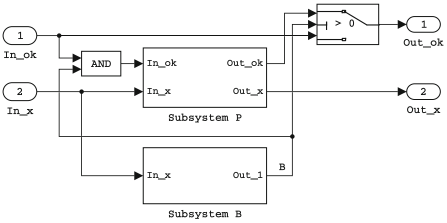


Fig. 5. Conditional statement

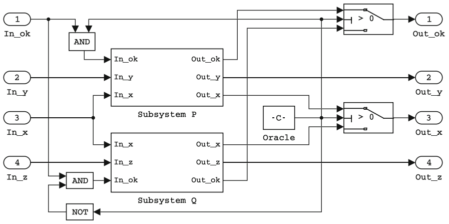


Fig. 6. Internal choice

Figure 6 illustrates a pattern to implement the above two cases. In order to guarantee that only one process in the internal choice is switched on, every switch block here needs to share exactly the same *switching condition*. As shown in Fig. 6, the two switch blocks share a common criteria (> 0) for passing first input as well as an identical second input signal, abbreviated as *Ran*, generated by an *oracle* that provides a non-deterministic signal[1]. The computation of signal ok and ok' can be formalized as

$$\begin{cases} ok_P = ok \wedge Ran \\ ok_Q = ok \wedge \neg Ran \end{cases} \qquad ok' = \begin{cases} ok'_P, \; Ran \\ ok'_Q, \; \neg Ran \end{cases}$$

3.8 Sequential Composition

An essential work in translating sequential composition into Simulink models, is to construct the initiation and termination of a process, which has already been done by introducing ok and ok' signals in connection with the *skip* process.

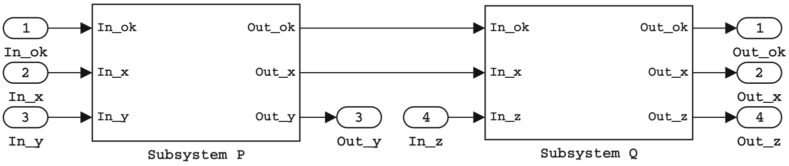


Fig. 7. Sequential composition

Figure 7 illustrates a straightforward encoding of sequential composition into Simulink diagrams. For exclusive signals y and z, we draw corresponding ports independently for subsystem P and Q. The set of common signals x processed by

[1] An oracle that interprets non-determinism is none of the blocks in Simulink library, inasmuch as the random block provided by Simulink generates pseudo random numbers, which is in itself deterministic.

both P and Q is linked sequentially from P to Q, and the same happens for *ok* and *ok*′.

$$ok_P = ok \quad ok_Q = ok'_P \quad ok' = ok'_Q \qquad x_P = x \quad x_Q = x'_P \quad x' = x'_Q$$

3.9 Repetition

The basic idea in encoding repetition is to link the outputs of subsystem P back into its in-ports, and we need to specify a finite random number N to control the number of times that P executes.

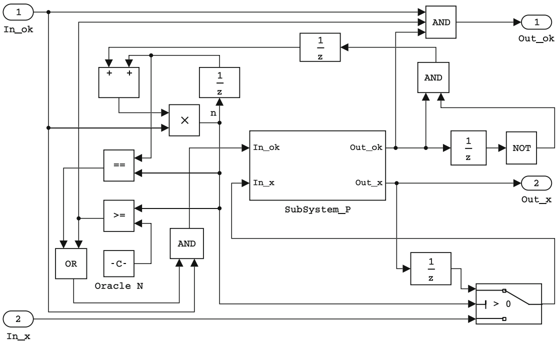


Fig. 8. Repetition

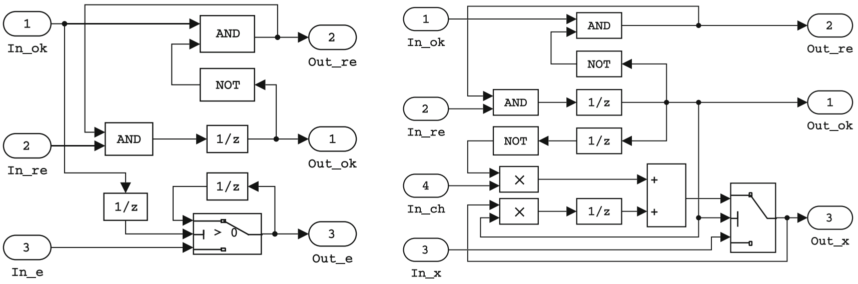
The integrated pattern to encode repetition p^* into Simulink diagrams is elaborated in Fig. 8. Here, a unit delay block with an initial value of 0 is introduced to break the algebraic loop that occurs when we link the outputs of P back. Besides, we introduce an oracle carrying a non-negative random number N to specify the number of repetitions of subsystem P. The update of variables is formulated as the following equations:

$$\begin{aligned} n &= ok \times (d(n) + d(ok'_P \wedge \neg d(ok'_P))) \qquad & ok' &= ok \wedge ok'_P \wedge (n \geq N) \\ ok_P &= ok \wedge (n == d(n) \vee n \geq N) & x_P &= \begin{cases} d(x'_P), n > 0 \\ x, \quad\; n == 0 \end{cases} \end{aligned}$$

3.10 Communication Events

For each communication event, either a *sender* (*ch*!*e*) or a *receiver* (*ch*?*x*), we construct a subsystem in Simulink to deliver the message along *ch* for the matching pair of events. In order to synchronise the interaction, we introduce another pair of Boolean signals *re* and *re*′ (*re* is short for ready) into each subsystem that corresponds to a communication event. *re* indicates whether the matching event is ready for the communication, while *re*′ indicates whether the event itself is ready for the communication.

A communication along channel ch takes place as soon as both re_{ch} and re'_{ch} are true, then both the sending and the receiving parties are ready, otherwise one or both sides must wait. Additionally, re and re' are local signals, which never occur in the process statements. Furthermore, re and re' in a Simulink subsystem are constructed as an in-port signal named `In_re` and an out-port signal named `Out_re` respectively. Figure 9 illustrates the Simulink diagrams that interpret communication events, which can be elaborated in the following two parts.



(a) $ch!e$ (b) $ch?x$

Fig. 9. Communication events

- For a sender $ch!e$, the output signals are computed according to

$$re' = ok \wedge \neg ok' \quad ok' = f(d(re \wedge re')) \quad e' = \begin{cases} e, & \neg d(ok) \\ d(e'), & d(ok) \end{cases},$$

  where the *keep* pattern $f(s(t)) = ok(t) \wedge (s(t) \vee f(s(t-1)))$ for $t > cnow$, with $f(s(t)) = 0$ for $t \in [cnow, cnow + 1)$, here $cnow$ is the current time. This is to keep ok' true since the communication is finished, i.e., since both re and re' turn true.
- For a receiver $ch?x$, the output signals are computed according to

$$re' = ok \wedge \neg ok' \quad ok' = f(d(re \wedge re'))$$
$$x' = \begin{cases} x, & \text{if } \neg ok' \\ \neg d(ok') \times ch + d(ok') \times d(x'), & \text{otherwise} \end{cases}$$

3.11 Parallel

For $P \| Q$, we consider the following two cases:

Without communications. This is a trivial case that we draw a subsystem encapsulating the two subsystems in terms of P and Q, but without any wires (except those carrying ok, ok') between the two subsystems, as shared variables are not allowed in HCSP. Specifically, we set $ok_P = ok_Q = ok$, and $ok' = ok'_P \wedge ok'_Q$.

With communications. As for a parallel process $P \| Q$ with inter-communications along a set of common channels $comChan(P, Q)$, we draw a subsystem containing the subsystems corresponding to P and Q, as well as some additional wires to bind all channels in $comChan(P, Q)$ and deliver messages along them.

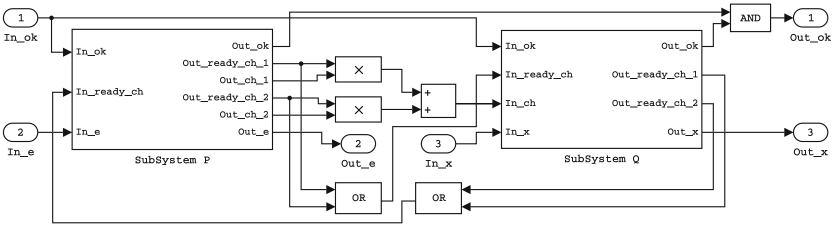


Fig. 10. Parallel $e := 0; ch!e; < \dot{e} = 1 \& e < 2 >; ch!e \| x := 3; ch?x; ch?x$

We elaborate the above idea by showing a Simulink diagram corresponding to a parallel process in Fig. 10, where the signals relevant to communications are attached with subscripts to specify the name of the common channel and the distinctive events corresponding to the same channel. Suppose that there are m and n events relevant to ch in subsystem `P` and `Q` respectively, then the computation in Fig. 10 is done by

$$ok_P = ok_Q = ok \quad ok' = ok'_P \wedge ok'_Q \quad re_{ch\_P} = \bigvee\nolimits_{i=1}^{n} re'_{ch\_i\_Q} \quad re_{ch\_Q} = \bigvee\nolimits_{j=1}^{m} re'_{ch\_j\_P}$$

indicating that the two subsystems in parallel are activated simultaneously when the system starts, and the parallel process terminates when both `P` and `Q` terminate. Furthermore, the channel ch on one side claims ready to the other side if either of its involved events is ready, which means that the communication events on different parties of a common channel are matched dynamically during the execution. Moreover, the value that `Q` receives along channel ch is computed as $ch_Q = \sum_{j=1}^{m} re'_{ch\_j\_P} \times ch'_{j\_P}$.

3.12 External Choice by Communications

As a subcomponent of interruption in HCSP, the external choice $[]_{i \in I}(io_i \rightarrow Q_i)$ waits until one of the communications io_i takes place, and then it is followed by the respective Q_i, where I is a non-empty finite set of indices, and $\{io_i\}_{i \in I}$ are communication events, i.e. either $ch!e$ or $ch?x$. In addition, if more than one among $\{io_i\}_{i \in I}$ are ready simultaneously, only one of them executes, this is determined randomly by the system. Thus, if the matching side of every io_i involved is ready for communication, then the external choice degenerates to internal choice. Besides, the syntax $(io_i \rightarrow Q_i)$ actually indicates a sequential composition $(io_i \rightarrow \text{skip}\ ; Q_i)$, to which the translation approach already has been introduced above.

Taking $P \,[\!]\, R$ as an example, where $P \widehat{=} io_1; Q_1$ and $R \widehat{=} io_2; Q_2$, then the ok signal of P can be computed by $ok_P = f(ok \wedge re_P \wedge (\neg re_R \vee (Ran < 0)))$. This means that when the external choice starts ($ok = 1$) and the matching event of io_1 is ready ($re_P = 1$), P is chosen to execute if either the matching event of io_2 is not ready ($re_R = 0$), or the random number Ran, where $Ran \in [-1, 1)$, occurs to be negative ($Ran < 0$) when both of the matching event are ready. A *keep* pattern $f(s)$ is used here to keep the signal ok_P true, otherwise it may jump back to false after that the communication terminates. The subsystem R is handled analogously. Thus, the output signals of the subsystem of $P \,[\!]\, R$ are given by $ok' = ok'_P \vee ok'_R$, $x' = \begin{cases} x'_P, \; ok'_P \\ x'_R, \; ok'_R \\ x, \quad \neg ok'_P \wedge \neg ok'_R \end{cases}$.

3.13 Interruption

Obviously, $\langle F(\dot{s}, s) = 0 \& B \rangle \trianglerighteq \,[\!]_{i \in I} (io_i \to Q_i)$ is equivalent to $\langle F(\dot{s}, s) = 0 \& (B \wedge \neg re'_R) \rangle; re'_R \to \,[\!]_{i \in I} (io_i \to Q_i)$, where $re'_R = f\left(\bigvee_{i \in I} re'_{oi_i}\right)$, and the translation rules can be composed in a semantic-preserving way (see Sect. 4). Hence, translating an interruption into a Simulink diagram becomes a composition of translating various components that have been illustrated in previous subsections.

4 Correctness of the Translation

In this section, we define UTP semantics both for HCSP constructs and the corresponding Simulink diagrams; proving the consistency of the two semantics is then a justification of the correctness of the translation from HCSP processes to Simulink diagrams.

UTP is a relational calculus based on first-order logic, which is intended for unifying different programming paradigms. In UTP, a *sequential program* (possibly nondeterministic) is represented by a *design* $D = (\alpha, P)$, where α denotes the set of state variables (called observables). Each state variable comes in an unprimed and a primed version, denoting respectively the pre- and the post-state value of the execution of the program. In addition to the program variables and their primed versions such as x and x', the set of observables includes two designated Boolean variables, ok and ok', denoting termination and stability of the program, respectively. P stands for the construct, $p(x) \vdash R(x, x')$, which is defined as

$$(ok \wedge p(x)) \Rightarrow (ok' \wedge R(x, x')).$$

It means that if the program is activated in a stable state, ok, where the *precondition* $p(x)$ holds, the execution will terminate, ok', in a state where the *postcondition* R holds; thus the post-state x' and the initial state x are related by relation R. We use *pre.D* and *post.D* to denote the pre- and post-conditions of D, respectively. If $p(x)$ is *true*, then P is shortened as $\vdash R(x, x')$.

Refinement

Let $D_1 = (\alpha, P_1)$ and $D_2 = (\alpha, P_2)$ be two designs with the same alphabet. D_2 is a *refinement* of D_1, denoted by $D_1 \sqsubseteq D_2$, if $\forall x, x', ok, ok'. (P_2 \Rightarrow P_1)$. Aslo, let $D_1 = (\alpha_1, P_1)$ and $D_2 = (\alpha_2, P_2)$ be two designs with possible different alphabets $\alpha_1 = \{x, x'\}$ and $\alpha_2 = \{y, y'\}$. D_2 is a *data refinement* of D_1 over $\alpha_1 \times \alpha_2$, denoted by $D_1 \sqsubseteq_d D_2$, if there is a relation $\rho(y, x')$ s.t. $\rho(y, x'); D_1 \sqsubseteq D_2; \rho(y, x')$.

In UTP the domain of designs forms a complete lattice with the refinement partial order, and *true* corresponding to **abort** (*false* corresponding to **miracle**) is the smallest (largest) element of the lattice. Furthermore, this lattice is closed under the classical programming constructs. These fundamental mathematical properties ensure that the domain of designs is a proper semantic domain for sequential programming languages.

Concurrent and Reactive Designs

Semantics of *concurrent and reactive programs* is defined by the notion of *reactive designs* with an additional Boolean observable *wait* that denotes suspension of a program. A design P is a *reactive design* if it is a fixed point of $\mathcal{H}'$, i.e. $\mathcal{H}'(P) = P$, where

$$\mathcal{H}'(p \vdash R) \widehat{=} (\vdash \wedge_{x \in \alpha(P)} x' = x \wedge wait' = wait) \lhd wait \rhd (p \vdash R). \tag{1}$$

$P_1 \lhd b \rhd P_2$ is a conditional statement, which means if b holds then P_1 else P_2, where b is a Boolean expression and P_1 and P_2 are designs. Informally, Eq. (1) says that if a reactive system (a reactive design) waits for a response from the environment (i.e., *wait* holds), it will keep waiting and do nothing (i.e., keep program variables unchanged), otherwise its function $(p \vdash R)$ will be executed.

Adaptation to Dynamical Systems

Obviously, hybrid systems are concurrent and reactive systems, so the UTP semantics of a hybrid system should satisfy the UTP *healthiness condition.* On the other hand, hybrid systems show some additional features, like real-time and the mixture of discrete and continuous dynamics. For specifying these additional features, we have to extend the notion of *reactive design* in UTP to admit function variables, and quantifications over functions, as in a real-time setting, program variables and channels are interpreted as functions over time. For specifying locality, higher-order quantifications are inevitable. So, UTP will become higher-order, rather than first-order. In addition, the derivative of a variable is allowed in a predicate. Therefore, strictly speaking, we extend the relational calculus of UTP to the combined theory of ordinary differential equations and timed traces with higher-order quantification.

In order to deal with real-time, a system variable *now* is introduced, which stands for the starting time. Correspondingly, *now'* stands for the ending time of a process.

Another point is that synchronization can only block discrete dynamics and keep discrete variable unchanged, it cannot block the evolution of continuous dynamics; time does not stop. So, given a hybrid system S, say $p \vdash P$, with continuous variables $\mathbf{s}$ and discrete variables $\mathbf{x}$, whose continuous dynamics is modeled as $\langle F(\dot{\mathbf{s}}, \mathbf{s}) = 0 \& B \rangle$, written S_C[2], then the healthiness condition of reactive designs should be changed to

$$\mathcal{H}(S) = S, \text{ where} \tag{2}$$

$$\mathcal{H}(S) \mathrel{\widehat{=}} (\vdash \mathbf{x}' = \mathbf{x} \wedge \mathit{wait}' = \mathit{wait} \wedge S_C) \lhd \mathit{wait} \rhd S. \tag{3}$$

A design that meets the healthiness condition (2) is called a *hybrid design*. For simplicity, we will denote the left side of *wait* in Eq. (3) by $\mathit{\Pi}_H$ in the sequel.

For convenience, for each channel *ch*, we introduce two Boolean functions over time *ch*! and *ch*?. $ch!(t)$ means that *ch* is ready for sending at time t, similarly, $ch?(t)$ means that *ch* is ready for receiving at time t. In addition, Periodic(ch^*, st) denotes $\forall n \in \mathbb{N}.\ t = n * st \Rightarrow ch^*(t)$, which means that the communication event ch^* is ready periodically with period *st*. Also, *maximal synchronization* semantics is adopted, i.e.,

$$\forall t \geq 0.\ (ch?(t) \wedge ch!(t)) \Rightarrow (\neg ch?'(t) \wedge \neg ch!'(t)), \tag{4}$$

which means that when a synchronization is ready, it takes place immediately.

4.1 UTP Semantics for Simulink

For each Simulink construct C, the observables of C include the inputs *in*, outputs *out*, the user defined parameters, and some auxiliary variables that are introduced for defining the semantics. Some output(s) may be also input(s), i.e. $out_i = in_j'$, but we will uniformly use out_i instead of in_j' as output in the semantics. Also, we use *cnow* to denote the current time. Now the semantics is a predicate denoted by $[\![C]\!]$.

Blocks. As pointed out in [38], it is natural to interpret each block of a Simulink diagram as a predicate relating its inputs to the outputs. The behavior of a block can be divided into a set of sub-behaviors, each guarded by a condition. Moreover, these guards are exclusive and complete, i.e., the conjunction of any two of them is unsatisfiable and the disjunction of all of them is valid. So, each sub-behavior can be further specified as a predicate over input and output signals. Additionally, for each discrete block (diagram), it is assumed that its input signals from outside are available at each sampling point. So, it can be represented by a UTP formula of the form:

$$\begin{aligned}&[\![B(ps, in, out)]\!]\\ &\mathrel{\widehat{=}} \mathcal{H}(Ass \vdash out(0) = ps.init \wedge \bigwedge_{k=1}^{m} (B_k(ps, in) \Rightarrow P_k(ps, in, out))),\end{aligned} \tag{5}$$

[2] We always assume time evolution is modeled in S_C, i.e., it contains $\dot{now} = 1$.

which means that in case the environment satisfies Ass (the *precondition*), the behaviour of a block is specified by the formula at the right side of $\vdash$ (the *postcondition*). We use ps to denote a family of user-set parameters that may change the functionality of the block. As explained previously, $\bigvee_{k=1}^{m} B_k(ps, in)$, and $\neg(B_i(ps, in) \wedge B_j(ps, in))$ for any $i \neq j$, always hold.

Thus the UTP semantics of a continuous block has the following form:

$$\begin{aligned}\llbracket CB(ps, in, out)\rrbracket \;\widehat{=}\; & \mathcal{H}(in! \vdash\; out(0) = ps.init \wedge \\ & (\begin{pmatrix} B_1(in, ps) \Rightarrow F_1(\dot{out}, out, in, ps) = 0 \wedge \cdots \wedge \\ B_m(in, ps) \Rightarrow F_m(\dot{out}, out, in, ps) = 0 \end{pmatrix} \wedge out!)),\end{aligned}$$

where $F_i(\dot{out}, out, in, ps) = 0$ models the continuous evolution if B_i holds. In this case, $wait \,\widehat{=}\, \neg out?$, which means that the continuous evolution will be interrupted by outputting to the environment. Thus, Eq. (2) holds with the maximal synchronization assumption in Eq. (4).

Correspondingly, the UTP semantics of a discrete block is

$$\begin{aligned}\llbracket DB(ps, in, out)\rrbracket \;\widehat{=}\; & \mathcal{H}(\text{Periodic}(in!, ps.st) \wedge \text{Periodic}(out?, ps.st) \vdash\; out(0) = ps.init \wedge \\ & \text{Periodic}(out!, ps.st) \wedge (\exists n \in \mathbb{N}.\; cnow = n * st) \Rightarrow \\ & \begin{pmatrix} B_1(in, ps) \Rightarrow \llbracket P_{comp_1}(in, out, ps)\rrbracket \wedge \cdots \wedge \\ B_m(in, ps) \Rightarrow \llbracket P_{comp_m}(in, out, ps)\rrbracket \end{pmatrix}),\end{aligned}$$

where $\llbracket P_{comp_i}(in, out, ps)\rrbracket$ stands for the UTP semantics of the i-th discrete computation, which can be obtained in a standard way (see [37]). The precondition says that the environment should periodically input to and output from the block. In this case, $wait$ is set as $\neg\exists n \in \mathbb{N}.\; cnow = n * st$ and its continuous is $\dot{cnow} = 1$, meaning that the block keeps waiting (idle) except for the periodic points at which discrete jumps happen.

Example 1. As an illustration, we show how to concretize the UTP semantics for some basic Simulink blocks including `Constant`, `Divide`, `Not`, `Or`, `Relational`, `Switch`, `Delay` and `Integrator`. We treat `Constant` and `Delay` as continuous blocks, although they can also be treated as discrete blocks in a similar way.

A `Constant` block generates a scalar constant value:

$$\llbracket \texttt{Constant}(ps.c, out)\rrbracket \;\widehat{=}\; \mathcal{H}(\vdash\; out(0) = c \wedge \dot{out} = 0 \wedge out!).$$

The design inside $\mathcal{H}$ is equivalent to $\vdash\; out = c \wedge out!$, which is $\vdash\; out(cnow) = c \wedge out!(cnow)$. Analogous remarks apply for the following.

Similarly, the UTP semantics for the `Divide` block is as follows:

$$\begin{aligned}& \llbracket \texttt{Divide}(ps.I, ps.\{sn_i\}_{i\in I}, \{in_i\}_{i\in I}, out)\rrbracket \\ \widehat{=}\; & \mathcal{H}(\wedge_{i\in I}\text{Periodic}(in_i!, ps.st) \wedge \text{Periodic}(out?, ps.st) \vdash \text{Periodic}(out!, ps.st) \wedge \\ & \quad ((\exists n \in \mathbb{N}.\; cnow = n * ps.st) \Rightarrow\; out = \prod\nolimits_{i\in I} sin_i) \wedge \\ & \qquad (sn_i = '*' \Rightarrow sin_i = in_i) \wedge (sn_i = '/' \Rightarrow sin_i = 1/in_i)).\end{aligned}$$

The logical operator blocks `Not` and `Or` respectively perform the specified logical operations on their inputs, whose UTP semantics are given by

$$\begin{aligned}[\![\texttt{Not}(in, out)]\!] \mathrel{\widehat{=}} \mathcal{H}(&\text{Periodic}(in!, ps.st) \wedge \text{Periodic}(out?, ps.st) \vdash \text{Periodic}(out!, ps.st) \wedge\\ &\exists n \in \mathbb{N}.\ cnow = n * ps.st \Rightarrow\ out = \neg in),\end{aligned}$$

$$\begin{aligned}&[\![\texttt{Or}(ps, \{in_i\}_{i \in I}, out)]\!]\\ \mathrel{\widehat{=}}\ &\mathcal{H}(\wedge_{i \in I}\text{Periodic}(in_i!, ps.st) \wedge \text{Periodic}(out?, ps.st) \vdash \text{Periodic}(out!, ps.st) \wedge\\ &\qquad \exists n \in \mathbb{N}.\ cnow = n * ps.st \Rightarrow\ out = \bigvee\nolimits_{i \in I} in_i),\end{aligned}$$

The `Relational` operator block compares two inputs using the relational operator parameter *ps.op*, and outputs either 1 (*true*) or 0 (*false*); its UTP semantics is given by

$$\begin{aligned}&[\![\texttt{Relational}(ps.op, in_1, in_2, out)]\!]\\ \mathrel{\widehat{=}}\ &\mathcal{H}(\text{Periodic}(in_1!, ps.st) \wedge \text{Periodic}(in_2!, ps.st) \wedge \text{Periodic}(out?, ps.st) \vdash\\ &\qquad \text{Periodic}(out!, ps.st) \wedge \exists n \in \mathbb{N}.\ cnow = n * ps.st \Rightarrow\ out = ps.op(in_1, in_2)).\end{aligned}$$

The `Switch` block passes through the first input or the third input based on the value of the second input, thus its UTP semantics is:

$$\begin{aligned}&[\![\texttt{Switch}(ps, in_1, in_2, in_3, out)]\!]\\ \mathrel{\widehat{=}}\ &\mathcal{H}(\wedge_{i=1}^{3}\text{Periodic}(in_i!, ps.st) \wedge \text{Periodic}(out?, ps.st) \vdash \text{Periodic}(out!, ps.st) \wedge\\ &\qquad (\exists n \in \mathbb{N}.\ cnow = n * ps.st) \Rightarrow \begin{pmatrix} ps.op(in_2, ps.c) \Rightarrow out = in_1 \wedge \\ \neg ps.op(in_2, ps.c) \Rightarrow out = in_3 \end{pmatrix}).\end{aligned}$$

A `Delay` block holds and delays its input by one sample period, therefore its UTP semantics is:

$$[\![\texttt{Delay}(ps, in, out)]\!] \mathrel{\widehat{=}} \mathcal{H}(in! \vdash \begin{pmatrix} cnow < ps.st \Rightarrow out(cnow) = ps.init \wedge \\ cnow \geq ps.st \Rightarrow out(cnow) = in(cnow - ps.st) \end{pmatrix} \wedge out!).$$

An `Integrator` block outputs the value of the integral of its input signal with respect to time, so its UTP semantics is given by

$$[\![\texttt{Integrator}(ps, in, out)]\!] \mathrel{\widehat{=}} \mathcal{H}(in! \vdash out(0) = ps.init \wedge (\dot{out} = in \wedge out!)).$$

Diagrams. A diagram is a set of blocks with connecting wires. W.l.o.g., consider a diagram D consisting of m continuous blocks and n discrete blocks, which are connected via a set of wires. The UTP semantics for blocks were defined above. Let therefore the semantics for the continuous blocks be $[\![CB_i(ps_i, \{in_i\}_{i \in I_i}, out_i)]\!]$ for $i = 1, \ldots, m$, and for the discrete blocks be $[\![DB_j(ps'_j, \{in'_i\}_{j \in J_j}, out'_j)]\!]$ for $j = 1, \ldots, n$.

Then the UTP semantics of D can be represented by

$$
\begin{aligned}
& [\![D(ps^*, \{in_i^*\}_{i\in I}, \{out_i^*\}_{i\in J})]\!] \\
\widehat{=}\ & \exists \cup_{i=1}^{m} \{in_i\}_{i\in I_i} - \{in_i^*\}_{i\in I_C}, \exists \cup_{j=1}^{n} \{in_j'\}_{j\in J_j} - \{in_i^*\}_{i\in I_D}, \\
& \exists\{out_1, \dots out_m\} - \{out_i^*\}_{i\in J_C}, \exists\{out_1', \dots, out_n'\} - \{out_i^*\}_{i\in J_D}. \\
& \mathcal{H}(\wedge_{k\in I_D} \mathrm{Periodic}(in_k^*!, ps^*.st) \wedge \wedge_{k\in I_C} in_k^*! \wedge \wedge_{k\in J_D} \mathrm{Periodic}(out_k^*?, ps_k^*.st) \\
& \vdash\ \wedge \wedge_{k\in J}\ out_k^*(0) = ps_k^*.init \wedge \wedge_{k\in J_C} out_k^*! \\[1ex]
& \wedge \wedge_{i=1}^{m} [\![CB_i(ps_i, \{in_i\}_{i\in I_i}, out_i)]\!] \wedge \wedge_{j=1}^{n}(out_j'(0) = ps_j'.init)[\sigma,\rho] \wedge \\
& \qquad (\exists n \in \mathbb{N}.\ cnow = n * \mathrm{GCD}(ps_1'.st, \cdots, ps_n'.st)) \Rightarrow (\wedge_{j=1}^{n} ps_j'.st \mid cnow \Rightarrow \\
& \qquad\qquad \begin{pmatrix} B_{j1}(\{in_i'\}_{j\in J_j}, ps_j')[\sigma,\rho] \Rightarrow [\![P_{comp_{j1}}(ps_j', \{in_i'\}_{j\in J_j}, out_j')]\!][\sigma,\rho] \wedge \cdots \wedge \\ B_{jm}(\{in_i'\}_{j\in J_j}, ps_j')[\sigma,\rho] \Rightarrow [\![P_{comp_{jm}}(ps_j', \{in_i'\}_{j\in J_j}, out_j')]\!][\sigma,\rho] \end{pmatrix})),
\end{aligned}
$$

where GCD computes the *greatest common divisor*, and

$$
\begin{aligned}
\{in_i^*\}_{i\in I_C} &= \cup_{i=1}^{m}\{in_i\}_{i\in I_i} - (\{out_1, \dots out_m\} \cup \{out_1', \dots, out_n'\}), \\
\{in_i^*\}_{i\in I_D} &= \cup_{j=1}^{n}\{in_j'\}_{j\in J_j} - (\{out_1, \dots out_m\} \cup \{out_1', \dots, out_n'\}), \\
\{out_i^*\}_{i\in J_C} &= \{out_1, \dots out_m\} - (\cup_{i=1}^{m}\{in_i\}_{i\in I_i} \cup \cup_{j=1}^{n}\{in_j'\}_{j\in J_j}), \\
\{out_i^*\}_{i\in J_D} &= \{out_1', \dots, out_n'\} - (\cup_{i=1}^{m}\{in_i\}_{i\in I_i} \cup \cup_{j=1}^{n}\{in_j'\}_{j\in J_j});
\end{aligned}
$$

I_C and I_D stand for the dangling inputs for continuous and discrete blocks after the composition, thus $I = I_C \cup I_D$ is the set of inputs of D; J_C and J_D stand for the dangling outputs for continuous and discrete blocks after the composition, thus $J = J_C \cup J_D$ is the set of outputs of D; and σ and ρ stand for the substitutions that replace the local input signals and input channels by the corresponding output signals and channels with the common names among these blocks (continuous and discrete) in each block, respectively. Furthermore, we set in this case

$$wait \widehat{=} \wedge_{i=1}^{m} \neg out_i? \wedge \neg\exists n \in \mathbb{N}.\ cnow = n * \mathrm{GCD}(ps_1'.st, \cdots, ps_n'.st).$$

Example 2. Consider the diagram `Diag` performing $out = in + c$. According to the above discussion, its UTP semantics can be given as

$$
\begin{aligned}
& [\![\texttt{Diag}(ps, in, out)]\!] \\
\widehat{=} & \exists out'.\mathcal{H}(\mathrm{Periodic}(in!, ps.st) \wedge \mathrm{Periodic}(out?, ps.st) \vdash ([\![\texttt{Constant}(ps, out')]\!] \wedge \\
& \qquad\qquad [\![\texttt{Add}(ps, \{+1, +1\}, \{in_1, in_2\}, out)]\!][in/in_1, out'/in_2]).
\end{aligned}
$$

Subsystems

Normal Subsystems. A normal subsystem has a set of blocks and wires that specify the signal connections. Actually, a normal subsystem can be seen as a diagram by flattening, i.e., connecting the external inputs of the inside blocks with the inputs of the subsystem and the external outputs of the inside blocks with the outputs of the subsystem. Suppose a normal subsystem `NSub` with a set of inputs $\{in_i\}_{i\in I}$ and a set of outputs $\{out_j\}_{j\in J}$, and its inside blocks form a diagram `Diag` with a set of external inputs $\{in_i'\}_{i\in I'}$ and $\{out_j'\}_{j\in J'}$. Let σ be the mapping to relate $\{in_i\}_{i\in I}$ with $\{in_i'\}_{i\in I'}$, and $\{out_j'\}_{j\in J'}$ with $\{out_j\}_{j\in J}$, then the UTP semantics of it can be easily defined as

$$[\![\texttt{NSub}(ps, \{in_i\}_{i\in I}, \{out_j\}_{j\in J}]\!] \mathrel{\widehat{=}} [\![\texttt{Diag}(ps, \{in_i'\}_{i\in I'}, \{out_j'\}_{j\in J'}]\!][\sigma].$$

Enabled Subsystems. A normal subsystem is enabled by adding an enabled block. It executes each simulation step when the enabling signal has a positive value, otherwise holds the states so they keep their most recent values. So, its UTP semantics can be defined as follows:

$$\begin{array}{rl} & [\![\texttt{ESub}(ps, \{in_i\}_{i\in I}, en, \{out_j\}_{j\in J})]\!] \\ \widehat{=} & en(now) > 0 \Rightarrow [\![\texttt{NSub}(ps, \{in_i\}_{i\in I}, en, \{out_j\}_{j\in J})]\!] \wedge \\ & en(now) \leq 0 \Rightarrow out(now) = out(now - ps.st). \end{array}$$

Theorem 1. *Given a Simulink diagram C, its UTP semantics $[\![C]\!]$ satisfies the healthiness condition in Eq. (2), that is*

$$\mathcal{H}([\![C]\!]) = [\![C]\!].$$

Proof. It is straightforward by the definition of $[\![C]\!]$. □

4.2 UTP Semantics for HCSP

As advocated by Hoare and He [37], a reactive system can be identified by the set of all possible observations of that system. As usual, an alphabet is attached to a system P (and its with the following constituents: $\mathcal{V}(P)$: the set of both continuous and discrete variable names, which is arranged as a vector $\mathbf{v}$, $i\Sigma(P)$: the set of input channel names, and $o\Sigma(P)$: the set of output channel names. Together the latter two form $\Sigma(P) \widehat{=} i\Sigma(P) \cup o\Sigma(P)$, which is arranged as a vector ch_P.

Given a hybrid system, its timed observation is the tuple $\langle now, \mathbf{v}, \mathbf{f_v}, re_{ch*}, msg_{ch}\rangle$. Here now is the start point and now' the end point of a time interval of an observation i. The initial values of variables are $\mathbf{v}$, and $\mathbf{v}'$ are the final values at termination. The vector $\mathbf{f_v}$ contains of real-valued functions over the time interval $[now, now']$ that record the values of $\mathbf{v}$, evidently with $\mathbf{f_v}(now) = \mathbf{v}, \mathbf{f_v}(now') = \mathbf{v}'$. The vector re_{ch*} of $\{0, 1\}$-valued Boolean functions over $[now, now']$, indicates whether communication events $ch*$ are ready for communication. The vector msg_{ch}, of real-valued functions over $[now, now']$ records the values passed along channels ch. We further define

$$\begin{array}{r} const(\mathbf{f}, \mathbf{b}, t_1, t_2) \widehat{=} \forall t \in [t_1, t_2].\ \mathbf{f}(t) = \mathbf{b}, \\ const^l(\mathbf{f}, \mathbf{b}, t_1, t_2) \widehat{=} \forall t \in [t_1, t_2).\ \mathbf{f}(t) = \mathbf{b}, \\ const^r(\mathbf{f}, \mathbf{b}, t_1, t_2) \widehat{=} \forall t \in (t_1, t_2].\ \mathbf{f}(t) = \mathbf{b}. \end{array}$$

Using the UTP timed observations, the HCSP constructs can be defined as follows.

The skip statement, which does not alter the program state, is the relational identity:

$$\begin{aligned}[\![\text{skip}]\!] \widehat{=} \mathcal{H}(\vdash &\, \mathit{now}' = \mathit{now} \wedge \mathbf{v}' = \mathbf{v} \wedge \mathit{const}(\mathbf{f}_{\mathbf{v}}, \mathbf{v}, \mathit{now}, \mathit{now}') \wedge \\ &\mathit{const}(\mathit{re}_{\mathit{ch}*}, \mathbf{0}, \mathit{now}, \mathit{now}') \wedge \mathit{const}(\mathit{msg}_{\mathit{ch}}, \mathit{msg}_{\mathit{ch}}(\mathit{now}), \mathit{now}, \mathit{now}')).\end{aligned}$$

As skip terminates immediately, *wait* is equivalent to *false* in this case. Hereafter, let

$$\mathit{RE} \mathrel{\widehat{=}} \mathit{const}(\mathit{re}_{\mathit{ch}*}, \mathbf{0}, \mathit{now}, \mathit{now}') \wedge \mathit{const}(\mathit{msg}_{\mathit{ch}}, \mathit{msg}_{\mathit{ch}}(\mathit{now}), \mathit{now}, \mathit{now}').$$

The assignment of e to a variable x is modelled as setting x to e and keeping all other variables (denoted by $\mathbf{u}$) constant:

$$\begin{aligned}[\![x := e]\!] \widehat{=} \mathcal{H}(\vdash &\, \mathit{now}' = \mathit{now} \wedge x' = e \wedge \mathbf{u}' = \mathbf{u} \wedge \mathit{const}(\mathbf{f}_x, e, \mathit{now}, \mathit{now}') \wedge \\ &\mathit{const}(\mathbf{f}_{\mathbf{u}}, \mathbf{u}, \mathit{now}, \mathit{now}') \wedge \mathit{RE}).\end{aligned}$$

As an assignment process terminates immediately, *wait* is equivalent to *false* here.

An evolution process says that the system waits, while it is evolving until the domain constraint becomes false. So, the UTP semantics is the following hybrid design

$$[\![\langle F(\dot{s}, s) = 0 \& B\rangle]\!] \mathrel{\widehat{=}} (\vdash F(\dot{s}, s = 0) \wedge \dot{t} = 1) \lhd B \rhd [\![\text{skip}]\!].$$

Obviously, in this case *wait* is equivalent to B.

The conditional statement behaves according to whether the condition holds or not: $[\![B \to P]\!] \mathrel{\widehat{=}} [\![P]\!] \lhd B \rhd [\![\text{skip}]\!]$, and internal choice is interpreted as a non-deterministic selection between two operands: $[\![P \sqcup Q]\!] \mathrel{\widehat{=}} ([\![P]\!] \vee [\![Q]\!])$.

In order to define sequential and parallel composition, we introduce two semantic operators.

Let H_1 and H_2 be two hybrid designs with

$$\begin{aligned}H_1 \widehat{=} (\vdash \wedge_{x \in \mathcal{V}(H_1)} x' = x \wedge \mathit{wait}'_{H_1} = \mathit{wait}_{H_1} \wedge S_{H_1}) \lhd \mathit{wait}_{H_1} \rhd (p_{H_1} \vdash R_{H_1}),\\ H_2 \widehat{=} (\vdash \wedge_{x \in \mathcal{V}(H_2)} x' = x \wedge \mathit{wait}'_{H_2} = \mathit{wait}_{H_2} \wedge S_{H_2}) \lhd \mathit{wait}_{H_2} \rhd (p_{H_2} \vdash R_{H_2}),\end{aligned}$$

which satisfy the healthiness condition in Eq. (2). The sequential composition of H_1 and H_2, denoted by $H_1 \mathbin{;} H_2$ is defined by

$$\begin{aligned}H_1 \mathbin{;} H_2 \ \widehat{=}\ & \exists \mathit{wait}_{H_1}, \mathit{wait}_{H_2}.\, \exists \mathbf{v}_{H_1}, \mathit{now}_{H_1}, \mathit{ok}_{H_1}.\\ & \exists \mathbf{f}_{\mathbf{v}_{H_1}}, \mathit{re}_{\mathit{ch}_{H_1}*}, \mathit{msg}_{\mathit{ch}_{H_1}}, f_{\mathbf{v}_{H_2}}, \mathit{re}_{\mathit{ch}_{H_2}*}, \mathit{msg}_{\mathit{ch}_{H_2}}.\\ & (\vdash (\mathit{wait}_{H_1} \Rightarrow \mathit{\Pi}_{H_1}) \wedge (\mathit{wait}_{H_2} \Rightarrow \mathit{\Pi}_{H_2}) \wedge \mathit{wait}' = \mathit{wait}) \lhd \mathit{wait} \rhd\end{aligned}$$

$$
\begin{aligned}
&(\neg wait_{H_1} \wedge wait_{H_2} \wedge r_{H_1} \vdash R_{H_1})\sigma_{H_1} \wedge \\
&(\neg wait_{H_1} \wedge \neg wait_{H_2} \wedge r_{H_2} \vdash R_{H_2})\sigma_{H_2} \wedge \\
&\qquad \forall t \in [now, now_{H_1}).\ wait(t) = wait_{H_1}(t) \wedge \\
&\qquad\quad \mathbf{f}_{\mathbf{v}}(t) = \mathbf{f}_{\mathbf{v}_{H_1}}(t) \wedge re_{ch*}(t) = re_{ch_{H_1}*}(t) \wedge msg_{ch}(t) = msg_{ch_{H_1}}(t) \wedge \\
&\qquad \forall t \in [now_{H_1}, now'].\ wait(t) = wait_{H_2}(t) \wedge \\
&\qquad\quad \mathbf{f}_{\mathbf{v}}(t) = \mathbf{f}_{\mathbf{v}_{H_2}}(t) \wedge re_{ch*}(t) = re_{ch_{H_2}*}(t) \wedge msg_{ch}(t) = msg_{ch_{H_2}}(t).
\end{aligned}
$$

where

$$
\begin{aligned}
\sigma_{H_1} &= [\mathbf{v}_{H_1}/\mathbf{v}', now_{H_1}/now', ok_{H_1}/ok'][\mathbf{f}_{\mathbf{v}_{H_1}}/\mathbf{f}_{\mathbf{v}}, re_{ch_{H_1}*}/re_{ch*}, msg_{ch_{H_1}}/msg_{ch}], \\
\sigma_{H_2} &= [\mathbf{v}_{H_1}/\mathbf{v}, now_{H_1}/now, ok_{H_1}/ok][\mathbf{f}_{\mathbf{v}_{H_2}}/\mathbf{f}_{\mathbf{v}}, re_{ch_{H_2}*}/re_{ch*}, msg_{ch_{H_2}}/msg_{ch}].
\end{aligned}
$$

In the above,

$$
\begin{aligned}
&\exists \mathbf{v}_{H_1}, now_{H_1}, ok_{H_1}.\ \exists \mathbf{f}_{\mathbf{v}}^{H_1}, re_{ch_{H_1}*}, msg_{ch_{H_1}}, f_{\mathbf{v}}^{H_2}, re_{ch_{H_2}*}, msg_{ch_{H_2}}. \\
&\quad (\neg wait_{H_1} \wedge wait_{H_2} \wedge r_{H_1} \vdash R_{H_1})\sigma_{H_1} \wedge (\neg wait_{H_1} \wedge \neg wait_{H_2} \wedge r_{H_2} \vdash R_{H_2})\sigma_{H_2}
\end{aligned}
$$

is essentially equivalent to the sequential composition of the two designs $(\neg wait_{H_1} \wedge wait_{H_2} \wedge r_{H_1} \vdash R_{H_1})$ and $(\neg wait_{H_1} \wedge \neg wait_{H_2} \wedge r_{H_2} \vdash R_{H_2})$ by the theory of UTP [37].

It is easy to see that if H_1 and H_2 satisfy the healthiness condition of hybrid designs, so does $H_1 \fatsemi H_2$. Hence, $H_1 \fatsemi H_2$ is still a hybrid design, which implies that hybrid designs are closed under sequential composition.

The parallel composition of H_1 and H_2, denoted by $H_1 \parallel H_2$ is defined

$$
\begin{aligned}
H_1 \| H_2 \;\widehat{=}\; &\exists now_{H_1}, now_{H_2}, ok_{H_1}, ok_{H_2}.\ H_1[ok/ok_{H_1}] \wedge H_2[ok/ok_{H_2}] \wedge \\
&now' = max\{now'_{H_1}, now'_{H_2}\} \wedge (ok' = ok'_{H_1} \wedge ok'_{H_2}) \wedge \\
&(\forall t \in (now'_{H_1}, now'].\ \mathbf{f}_{\mathbf{v}_{H_1}}(t) = \mathbf{f}_{\mathbf{v}_{H_1}}(now'_{H_1}) \wedge \\
&\quad re_{ch_{H_1}*}(t) = re_{ch_{H_1}*}(now'_{H_1}) \wedge msg_{ch_{H_2}}(t) = msg_{ch_{H_2}}(now'_{H_2})) \wedge \\
&(\forall t \in (now'_{H_2}, now'].\ \mathbf{f}_{\mathbf{v}_{H_2}}(t) = \mathbf{f}_{\mathbf{v}_{H_2}}(now'_{H_2}) \wedge \\
&\quad re_{ch_{H_2}*}(t) = re_{ch_{H_2}*}(now'_{H_2}) \wedge msg_{ch_{H_2}*}(t) = msg_{ch_{H_2}*}(now'_{H_2})).
\end{aligned}
$$

It can be further be proved that

$$
\begin{aligned}
&H_1 \| H_2 \;\Leftrightarrow\; \exists now_{H_1}, now_{H_2}, ok_{H_1}, ok_{H_2}. \\
&\quad \vdash \begin{pmatrix} wait_{H_1} \Rightarrow \mathit{\Pi}_{H_1} \wedge \\ wait_{H_2} \Rightarrow \mathit{\Pi}_{H_2} \end{pmatrix} \lhd wait_{H_1} \wedge wait_{H_2} \rhd \begin{pmatrix} (p_{H_1} \vdash R_{H_1})[ok/ok_{H_1}] \wedge \\ (p_{H_2} \vdash R_{H_2})[ok/ok_{H_2}] \end{pmatrix} \wedge \\
&\quad now' = max\{now'_{H_1}, now'_{H_2}\} \wedge (ok' = ok'_{H_1} \wedge ok'_{H_2}) \wedge \\
&\quad (\forall t \in (now'_{H_1}, now'].\ \mathbf{f}_{\mathbf{v}_{H_1}}(t) = \mathbf{f}_{\mathbf{v}_{H_1}}(now'_{H_1}) \wedge \\
&\qquad re_{ch_{H_1}*}(t) = re_{ch_{H_1}*}(now'_{H_1}) \wedge msg_{ch_{H_2}}(t) = msg_{ch_{H_2}}(now'_{H_2})) \wedge \\
&\quad (\forall t \in (now'_{H_2}, now'].\ \mathbf{f}_{\mathbf{v}_{H_2}}(t) = \mathbf{f}_{\mathbf{v}_{H_2}}(now'_{H_2}) \wedge \\
&\qquad re_{ch_{H_2}*}(t) = re_{ch_{H_2}*}(now'_{H_2}) \wedge msg_{ch_{H_2}*}(t) = msg_{ch_{H_2}*}(now'_{H_2})).
\end{aligned}
$$

Therefore,also $H_1 \parallel H_2$ satisfies the healthiness condition of hybrid designs. Hence, $H_1 \parallel H_2$ is a hybrid design, which implies that hybrid designs are closed under parallel composition.

Now, given two HCSP processes P and Q, their sequential composition is defined $[\![P;Q]\!] \mathrel{\widehat{=}} [\![P]\!] \mathbin{⨟} [\![Q]\!]$, and their parallel composition by $[\![P \parallel Q]\!] \mathrel{\widehat{=}} [\![P]\!] \parallel [\![Q]\!]$.

A process variable X is interpreted as a predicate variable. Without confusion in the context, we use X to represent the predicate variable corresponding to process variable X, i.e. $[\![X]\!] \mathrel{\widehat{=}} X$.

The semantics for recursion is defined as the least fixed point of the corresponding recursive predicate by $[\![\mathrm{rec}\, X.P]\!] \mathrel{\widehat{=}} \mu X.[\![P]\!]$. An HCSP process P^* is thus defined as $P^* \Leftrightarrow \mathrm{rec}\, X.(\mathrm{skip} \sqcup (P\,;\, X))$. As discussed above, its semantics is given by $[\![P^*]\!] \Leftrightarrow \exists N.[\![P^N]\!]$, where $P^0 \mathrel{\widehat{=}} \mathrm{skip}$.

A receiving event can be modelled by $[\![ch?x]\!] \mathrel{\widehat{=}} \vdash LHS \lhd re_{ch?} \wedge \neg re_{ch!} \rhd RHS$, where $LHS \widehat{=} \dot{now} = 1 \wedge x' = x \wedge \mathbf{u'} = \mathbf{u}$, and

$$RHS \widehat{=} now' = now + d \wedge re'_{ch?} = 0 \wedge re'_{ch!} = 0 \wedge \mathbf{u'} = \mathbf{u} \wedge x' = msg_{ch}(now') \wedge \\ const^l(re_{ch?}, 1, now, now') \wedge const^l(re_{ch!}, 0, now, now').$$

Here, $wait \mathrel{\widehat{=}} re_{ch?} \wedge \neg re_{ch!}$, i.e., the process waits until its dual event becomes ready. The sending event $[\![ch!e]\!]$ can be defined similarly.

The communication interruption can be defined as

$$[\![\langle F(\dot{s}, s) = 0 \& B\rangle \unrhd [\!]_{i \in I}(io_i \to Q_i)]\!] \mathrel{\widehat{=}} [\![\langle F(\dot{s}, s) = 0 \& (B \wedge \neg \varGamma)\rangle; \\ \varGamma \to [\!]_{i \in I}(io_i \to Q_i)]\!]$$

where $\varGamma \widehat{=} \bigvee_{i \in I} re'_{\overline{io_i}}$, and $\overline{io_i}$ stands for the dual communication event with respect to io_i, for instance $\overline{ch?} = ch!$.

To prove whether the UTP semantics of other HCSP constructs satisfies the healthiness condition is mathematically straightforward and thus omitted here. It can be further deduced that the domain of hybrid designs forms a complete lattice with a refinement partial order, on which the classical programming operations are closed.

4.3 Justification of Correctness

Having defined a UTP semantics respectively for the HCSP components and the Simulink diagrams, we justify the translation by checking the semantic equivalence of a Simulink diagram with its corresponding HCSP construct (Theorem 2). Here are several remarks to be noted during the proofs:

1. We set the sample times of all discrete blocks to -1 in the translation, that is, all the generated discrete blocks share a globally identical sample time gst, which will be configured by the user before triggering the simulation.

2. It is assumed that the *In_ok* signal in a subsystem firstly turns true at the first sample point, i.e. $min\{t|In\_ok(t) = 1\} = gst$. Similarly, we use τ to denote the earliest time at which the *Out_ok* signal becomes true, i.e. $\tau \mathrel{\widehat{=}} min\{t|\,Out\_ok(t) = 1\}$.
3. Hereafter we use $[\![\mathtt{Wires}]\!]$ to indicate implicitly the entire group of variable substitutions within a subsystem, and blocks are referred to as their abbreviated names with potential identifiers, for instance, `Swt1` in the assignment structure stands for the block `Switch1` in Fig. 3.
4. Unless otherwise stated, the parameters of a block will be elided in the semantic function for simplicity. Besides, to distinguish the input/output signals of blocks, the leading characters of input/output signals of a subsystem are capitalized.

Theorem 2. *Given an HCSP process P, denote the translated Simulink diagram by* $\mathtt{H2S}(P)$*. Suppose there is a correspondence (denoted by EA) between* $[\![P]\!]$ *and* $[\![\mathtt{H2S}(P)]\!]$*, i.e.,* $now = gst$, $now' = \tau$, $ok = In\_ok(gst) = \top$, $ok' = Out\_ok(\tau)$, $\mathbf{v} = In\_\mathbf{v}(gst)$, $\mathbf{v}' = Out\_\mathbf{v}(\tau)$, $f_{\mathbf{v}} = Out\_\mathbf{v}|_{[gst,\tau]}$, $re_{ch*} = Out\_re_{ch*}|_{[gst,\tau]}$, *and* $msg_{ch} = Out\_re_{ch}|_{[gst,\tau]}$, *then*

$$Periodic(in!, ps.gst) \wedge Periodic(out?, ps.gst) \Rightarrow \big([\![P]\!] \Leftrightarrow [\![\mathtt{H2S}(P)]\!]|_{[gst,\tau]}\big) \qquad (6)$$

as $gst \rightarrow 0$.

Proof. By induction on the structure of HCSP components. For simplicity, we use $\mathbf{ch}*$ to denote the local communication events inside of $\mathtt{H2S}(P)$ in what follows.

skip: It is easy to see that under the assumptions,

$$\text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \Rightarrow \big([\![\text{skip}]\!] \Leftrightarrow [\![\mathtt{H2S}(\text{skip})]\!]|_{[gst,\tau]}\big)\,.$$

Assignment: Without loss of generality, we use $[\![\mathtt{Diag}_e]\!]$ to denote the semantics of the diagram which computes the right-hand side of the assignment.

$$\begin{aligned}
&[\![\mathtt{H2S}(x := e)]\!] \\
\widehat{=}\;& \exists \mathbf{ch}*.[\![\mathtt{Wires}]\!] \wedge [\![\mathtt{Diag}_e]\!] \wedge [\![\mathtt{Del1}]\!] \wedge [\![\mathtt{Del2}]\!] \wedge [\![\mathtt{Swt1}]\!] \wedge [\![\mathtt{Swt2}]\!] \\
\Leftrightarrow\;& \text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \vdash \text{Periodic}(out!, ps.gst) \wedge \\
& \forall t \geq 0.\ (t < gst \Rightarrow out\_Del1(t) = 0) \wedge (t \geq gst \Rightarrow out\_Del1(t) = In\_ok(t - gst)) \wedge \\
& \qquad\quad (t < gst \Rightarrow out\_Del2(t) = 0) \wedge (t \geq gst \Rightarrow out\_Del2(t) = Out\_x(t - gst)) \wedge \\
& (\exists n \in \mathbb{N}.\ cnow = n * ps.gst \Rightarrow \\
& \qquad\qquad (In\_ok(cnow) > 0 \Rightarrow out\_Swt1(cnow) = out\_Diag_e(cnow)) \wedge \\
& \qquad\qquad (In\_ok(cnow) \leq 0 \Rightarrow out\_Swt1(cnow) = In\_x(cnow)) \wedge \\
& \qquad\qquad (out\_Del1(cnow) > 0 \Rightarrow out\_Swt2(cnow) = out\_Del2(cnow)) \wedge \\
& \qquad\qquad (out\_Del1(cnow) \leq 0 \Rightarrow out\_Swt2(cnow) = out\_Swt1(cnow)) \wedge \\
& \qquad\qquad Out\_x(cnow) = out\_Swt2(cnow) \wedge Out\_ok(cnow) = In\_ok(cnow)
\end{aligned}$$

Using the left-hand side of (6) and restricting the time interval, we get the desired result.

$$\begin{aligned}
&\text{Periodic}(\mathit{in}!, \mathit{ps.gst}) \wedge \text{Periodic}(\mathit{out}?, \mathit{ps.gst}) \wedge [\![\mathtt{H2S}(x := e)]\!]|_{[gst,\tau]}\\
\Leftrightarrow\ &(\exists n \in \mathbb{N}.\ \mathit{cnow} = n * \mathit{ps.gst} \wedge \mathit{cnow} \in [gst, \tau]) \Rightarrow (\mathit{Out\_ok}(\mathit{cnow}) = \mathit{In\_ok}(\mathit{cnow}) \wedge\\
&\qquad\qquad \mathit{Out\_x}(\mathit{cnow}) = \mathit{out\_Diag}_e(\mathit{cnow}))\\
\Leftrightarrow\ &\mathit{ok}' \wedge \tau = \mathit{now} \wedge x' = e \wedge \mathit{const}(f_x, e, \mathit{now}, \tau) \wedge\\
&\mathbf{u}' = \mathbf{u} \wedge \mathit{const}(f_{\mathbf{u}}, \mathbf{u}, \mathit{now}, \tau) \wedge \mathit{RE} \qquad\qquad (\mathit{gst} \to 0,\ \mathit{EA})
\end{aligned}$$

Evolution statement: By the defined UTP semantics, it follows

$$\begin{aligned}
&[\![\mathtt{H2S}(\langle F(\dot{s}, s) = 0 \& B\rangle)]\!] \widehat{=} \exists \mathbf{ch}*.[\![\mathtt{Wires}]\!] \wedge [\![\mathtt{NSubB}]\!] \wedge [\![\mathtt{ESubF}]\!] \wedge [\![\mathtt{Del}]\!] \wedge\\
&\qquad\qquad\qquad\qquad [\![\mathtt{Not}]\!] \wedge [\![\mathtt{And1}]\!] \wedge [\![\mathtt{And2}]\!] \wedge [\![\mathtt{Swt}]\!]\\
\Leftrightarrow\ &\text{Periodic}(\mathit{in}!, \mathit{ps.gst}) \wedge \text{Periodic}(\mathit{out}?, \mathit{ps.gst}) \vdash \text{Periodic}(\mathit{out}!, \mathit{ps.gst}) \wedge\\
&\forall t \geq 0.\ (\mathit{out\_And1}(t) > 0 \Rightarrow \mathit{out\_ESubF}(t) = S(t)) \wedge\\
&\qquad (\mathit{out\_And1}(t) \leq 0 \Rightarrow \mathit{out\_ESubF}(t) = \mathit{out\_ESubF}(t - \mathit{gst})) \wedge\\
&\qquad (t < \mathit{gst} \Rightarrow \mathit{out\_Del}(t) = 1) \wedge (t \geq \mathit{gst} \Rightarrow \mathit{out\_Del}(t) = \mathit{out\_NSubB}(t - \mathit{gst})) \wedge\\
&(\exists n \in \mathbb{N}.\ \mathit{cnow} = n * \mathit{ps.gst}) \Rightarrow\\
&\qquad \mathit{out\_NSubB}(\mathit{cnow}) = B(\mathit{cnow}) \wedge\ \mathit{out\_Not}(\mathit{cnow}) = \neg \mathit{out\_Del}(\mathit{cnow}) \wedge\\
&\qquad \mathit{out\_And1}(t) = (\mathit{In\_ok}(\mathit{cnow}) \wedge \mathit{out\_Del}(t)) \wedge \mathit{out\_And2}(t) = (\mathit{In\_ok}(\mathit{cnow}) \wedge\\
&\qquad \mathit{out\_Not}(\mathit{cnow})) \wedge (\mathit{In\_ok}(\mathit{cnow}) > 0 \Rightarrow \mathit{out\_Swt}(\mathit{cnow}) = \mathit{out\_ESubF}(\mathit{cnow})) \wedge\\
&\qquad (\mathit{In\_ok}(\mathit{cnow}) \leq 0 \Rightarrow \mathit{out\_Swt}(\mathit{cnow}) = \mathit{In\_s}(\mathit{cnow})) \wedge\\
&\qquad \mathit{Out\_s}(\mathit{cnow}) = \mathit{out\_Swt}(\mathit{cnow}) \wedge \mathit{Out\_ok}(\mathit{cnow}) = \mathit{out\_And2}(\mathit{cnow})
\end{aligned}$$

Using the left-hand side of (6) and restricting the time interval, we have

$$\text{Periodic}(\mathit{in}!, \mathit{ps.gst}) \wedge \text{Periodic}(\mathit{out}?, \mathit{ps.gst}) \wedge \mathit{ok} \wedge [\![\mathtt{H2S}(\langle F(\dot{s}, s) = 0 \& B\rangle)]\!]|_{[gst,\tau - gst]}$$

$$\begin{aligned}
\Leftrightarrow\ &\mathit{Out\_ok}(\tau) = \top \wedge \tau - \mathit{gst} = \mathit{gst} + (\tau - 2 * \mathit{gst}) \wedge \mathit{Out\_s}(\tau - \mathit{gst}) = S(\tau - 2 * \mathit{gst}) \wedge\\
&(\exists n \in \mathbb{N}.\ \mathit{cnow} = n * \mathit{ps.gst} \wedge \mathit{cnow} \in [\mathit{gst}, \tau)) \Rightarrow \mathit{out\_NSubB}(\mathit{cnow} - \mathit{gst}) \wedge\\
&\qquad\qquad\qquad f_s(\mathit{cnow}) = S(\mathit{cnow} - \mathit{gst})) \wedge \neg \mathit{out\_NSubB}(\tau - \mathit{gst}) \wedge\\
&\qquad\qquad\qquad \mathit{out\_ESubF}(\mathit{cnow}) = \mathit{out\_ESubF}(\mathit{cnow} - \mathit{gst})\\
\Leftrightarrow\ &\mathit{ok}' \wedge \mathbf{u}' = \mathbf{u} \wedge \mathit{const}(f_{\mathbf{u}}, \mathbf{u}, \mathit{now}, \tau) \wedge \mathit{RE} \wedge\\
&(B \wedge \tau = \mathit{now} + d \wedge s' = S(d) \wedge \forall t \in [\mathit{now}, \tau].\ f_s(t) = S(t - \mathit{now}) \vee\\
&\qquad \neg B \wedge s' = s) \qquad\qquad (\mathit{gst} \to 0,\ \mathit{EA})\\
\Leftrightarrow\ &([\![F(\dot{s}, s) = 0]\!] \lhd B \rhd [\![\text{skip}]\!])
\end{aligned}$$

Thereby the semantics can be proved consistent on the interval $[gst, \tau - gst]$, moreover, when the user-defined sample time $gst \to 0$, we have the desired result.

Conditional: By the definition of H2S and the UTP semantics of Simulink:

$$\begin{aligned}
&\llbracket \texttt{H2S}(B \rightarrow P) \rrbracket \mathrel{\widehat{=}} \exists \mathbf{ch}*. \llbracket \texttt{Wires} \rrbracket \wedge \llbracket \texttt{NSubB} \rrbracket \wedge \llbracket \texttt{NSubP} \rrbracket \wedge \llbracket \texttt{And} \rrbracket \wedge \llbracket \texttt{Swt} \rrbracket \\
\Leftrightarrow\ & \mathrm{Periodic}(in!, ps.gst) \wedge \mathrm{Periodic}(out?, ps.gst) \vdash \mathrm{Periodic}(out!, ps.gst) \wedge \\
& (\exists n \in \mathbb{N}.\ cnow = n * ps.gst) \Rightarrow \\
& \qquad out\_NSubB(cnow) = B(cnow) \wedge \llbracket \texttt{NSubP}(in_{ok} = out\_And(cnow)) \rrbracket \wedge \\
& \qquad out\_And(cnow) = (In\_ok(cnow) \wedge out\_NSubB(cnow)) \wedge \\
& \qquad (out\_NSubB(cnow) > 0 \Rightarrow out\_Swt(cnow) = out\_NSubP\_ok(cnow)) \wedge \\
& \qquad (out\_NSubB(cnow) \leq 0 \Rightarrow out\_Swt(cnow) = In\_ok(cnow)) \wedge \\
& \qquad Out\_x(cnow) = out\_NSubP\_x(cnow) \wedge Out\_ok(cnow) = out\_Swt(cnow)
\end{aligned}$$

It follows

$$\begin{aligned}
& \mathrm{Periodic}(in!, ps.gst) \wedge \mathrm{Periodic}(out?, ps.gst) \wedge \llbracket \texttt{H2S}(B \rightarrow P) \rrbracket|_{[gst,\tau]} \\
\Leftrightarrow\ & (B \wedge \llbracket P \rrbracket) \vee (\neg B \wedge ok' \wedge \tau = now \wedge v' = v \wedge const(f_v, v, now, \tau) \wedge RE) \wedge \\
& \mathbf{u}' = \mathbf{u} \wedge const(f_{\mathbf{u}}, \mathbf{u}, now, \tau) \wedge RE \qquad\qquad (gst \rightarrow 0,\ EA) \\
\Leftrightarrow\ & \llbracket P \rrbracket \lhd B \rhd \llbracket \mathrm{skip} \rrbracket
\end{aligned}$$

Thus we have the desired result.

Internal choice: This is proved as for the conditional process.

Sequential composition: As shown in Fig. 7, $\mathbf{x}$ are the set of common signals processed by both P and Q, while $\mathbf{y}$ and $\mathbf{z}$ are the respective exclusive signals.

$$\begin{aligned}
& \llbracket \texttt{H2S}(P;Q) \rrbracket|_{[gst,\tau]} \mathrel{\widehat{=}} \exists \mathbf{ch}*. \llbracket \texttt{Wires} \rrbracket|_{[gst,\tau]} \wedge \llbracket \texttt{NSubP} \rrbracket|_{[gst,\tau]} \wedge \llbracket \texttt{NSubQ} \rrbracket|_{[gst,\tau]} \\
\Leftrightarrow\ & \llbracket \texttt{NSubP}(in_{ok} = In\_ok(cnow), in_x = In\_x(cnow), in_y = In\_y(cnow)) \rrbracket|_{[gst,\tau]} \wedge \\
& \llbracket \texttt{NSubQ}(in_{ok} = out\_NSubP\_ok(t), in_x = out\_NSubP\_x(t), in_z = In\_z(t)) \rrbracket|_{[gst,\tau]} \wedge \\
& (\exists n \in \mathbb{N}.\ cnow = n * ps.gst \wedge cnow \in [gst, \tau]) \Rightarrow \\
& \qquad Out\_ok(cnow) = out\_NSubQ\_ok(cnow) \wedge Out\_x(cnow) = out\_NSubQ\_x(cnow) \wedge \\
& \qquad Out\_y(cnow) = out\_NSubP\_y(cnow) \wedge Out\_z(cnow) = out\_NSubQ\_z(cnow) \\
\Leftrightarrow\ & (\exists x_m, now_m, ok_m.\ (out\_NSubP\_ok(now_m) \Leftrightarrow ok) \wedge \\
& \llbracket \texttt{NSubP} \rrbracket [x_m/x', ok_m/ok'] \wedge \llbracket \texttt{NSubQ} \rrbracket [x_m/x, ok_m/ok] \wedge \\
& \forall t \geq 0. y(t) = out\_NSubP\_y(t) \wedge x(t) = out\_NSubQ\_x(t) \wedge \\
& \qquad z(t) = out\_NSubQ\_z(t)) \qquad (gst \rightarrow 0,\ EA\ \text{ and Induction Hypothesis})
\end{aligned}$$

This gives the desired result.

Recursion: We only consider tail recursion, i.e., repetition. General recursion can be proved similarly. As shown in Fig. 8, a random number N, generated by an oracle, is used in the Simulink diagram as the number of iterations of subsystem P. Since $\llbracket P^* \rrbracket$ is defined by the least fixed point, it is clear that the inverse direction holds: $\mathrm{Periodic}(in!, ps.gst) \wedge \mathrm{Periodic}(out?, ps.gst) \Rightarrow \left(\llbracket P^* \rrbracket \Leftarrow \llbracket \texttt{H2S}(P^*) \rrbracket|_{[gst,\tau]}\right)$. For the other direction, suppose $\llbracket P^* \rrbracket$ holds, then according to the semantics, there must exist N such that $\llbracket P^N \rrbracket$ holds. We then apply the oracle, to generate the same number N, to control the execution of the Simulink diagram $\mathtt{H2S}(P^*)$, to execute for N times. The fact is thus proved, i.e.

$$\text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \Rightarrow \left([\![P^*]\!] \Rightarrow [\![\mathtt{H2S}(P^*)]\!]|_{[gst,\tau]}\right).$$

Communication events: By the definition of H2S and the UTP semantics of Simulink given in Sect. 4.1, it follows

$$\begin{aligned}
&\text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \wedge [\![\mathtt{H2S}(ch?x)]\!]|_{[gst,\tau]}\\
\Leftrightarrow\ &(\exists n \in \mathbb{N}.\ cnow = n * ps.gst \wedge cnow \in [gst, \tau]) \Rightarrow\\
&\qquad Out\_re(cnow) = (In\_ok(cnow) \wedge \neg Out\_ok(cnow)) \wedge\\
&\qquad Out\_ok(cnow) = f(In\_re(cnow - gst) \wedge Out\_re(cnow - gst)) \wedge\\
&\qquad (\neg Out\_ok(cnow) \Rightarrow Out\_x(cnow) = In\_x(cnow)) \wedge\\
&\qquad (Out\_ok(cnow) \Rightarrow Out\_x(cnow) = (\neg Out\_ok(cnow - gst)) * In\_ch(cnow))\\
\Leftrightarrow\ &Out\_ok(\tau) = \top \wedge\\
&(\exists n \in \mathbb{N}.\ cnow = n * ps.gst) \Rightarrow\\
&\qquad cnow \in [gst, \tau - gst] \Rightarrow Out\_re(cnow) = 1 \wedge In\_re(t) = 0 \wedge In\_re(\tau - gst) = 1 \wedge\\
&\qquad cnow \in (\tau - gst, \tau] \Rightarrow Out\_re(cnow) = 0 \wedge In\_re(cnow) = 0 \wedge\\
&\qquad cnow \in [gst, \tau) \Rightarrow Out\_x(cnow) = In\_x(cnow) \wedge Out\_x(\tau) = In\_ch(\tau)\\
\Leftrightarrow\ &ok' \wedge now' = now + d \wedge const(re_{ch?}, 1, now, now') \wedge\\
&const^l(re_{ch!}, 0, now, now') \wedge re_{ch!}(now') = 1 \wedge re'_{ch?}(now') = 0 \wedge re'_{ch!}(now') = 0 \wedge\\
&const^l(f_x, x, now, now') \wedge f_x(now') = msg_{ch}(now') \wedge\\
&const(\mathbf{f_u}, \mathbf{u}, now, now') \wedge \mathbf{u}' = \mathbf{u} \wedge x' = msg_{ch}(now') \qquad (gst \longrightarrow 0,\ EA)\\
\Leftrightarrow\ &(LHS \lhd re_{ch?} \wedge \neg re_{ch!} \rhd RHS)
\end{aligned}$$

Therefore, we get

$$\text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \Rightarrow \left([\![ch?x]\!] \Leftrightarrow [\![\mathtt{H2S}(ch?x)]\!]|_{[gst,\tau]}\right).$$

The equivalence for sending events can be proved similarly.

Interruption: It is trivial to prove that Theorem 2 holds for communication interruption, inasmuch as it can be interpreted by the sequential composition and conditional statement, for which we have already proved validation of Theorem 2.

Parallel: As shared variables are not allowed in HCSP, we use $\mathbf{y}$ and $\mathbf{z}$ to denote the set of exclusive signals (including *re* and *msg*) respectively for P and Q.

Let $\tau_P \widehat{=} min\{t | out\_NSubP\_ok(t) = 1\}$, and $\tau_Q \widehat{=} min\{t | out\_NSubQ\_ok(t) = 1\}$. Then, according to the definitions, we have

$$
\begin{aligned}
& [\![\mathtt{H2S}(P \| Q)]\!]|_{[gst,\tau]} \mathrel{\widehat{=}} \exists \mathbf{ch}*. [\![\mathtt{Wires}]\!]|_{[gst,\tau]} \wedge [\![\mathtt{NSubP}]\!]|_{[gst,\tau]} \wedge [\![\mathtt{NSubQ}]\!]|_{[gst,\tau]} \\
\Leftrightarrow\ & [\![\mathtt{NSubP}(in_{ok} = In\_ok(cnow), in_y = In\_y(cnow))]\!]|_{[gst,\tau]} \wedge \\
& [\![\mathtt{NSubQ}(in_{ok} = In\_ok(cnow), in_z = In\_z(cnow))]\!]|_{[gst,\tau]} \wedge \\
& (\exists n \in \mathbb{N}.\ cnow = n * ps.gst \wedge cnow \in [gst, \tau]) \Rightarrow \\
& \quad (\tau_P = \tau \vee \tau_Q = \tau) \wedge (Out\_ok(cnow) = out\_NSubP\_ok(cnow) \vee out\_NSubQ\_ok(cnow)) \wedge \\
& (\exists n \in \mathbb{N}.\ cnow = n * ps.gst \wedge cnow \in [\tau_P, \tau]) \Rightarrow Out\_y(cnow) = out\_NSubP\_y(\tau_P) \wedge \\
& (\exists n \in \mathbb{N}.\ cnow = n * ps.gst \wedge cnow \in [\tau_Q, \tau]) \Rightarrow Out\_z(cnow) = out\_NSubQ\_z(\tau_Q) \wedge \\
\Leftrightarrow\ & [\![P \| Q]\!] \qquad\qquad (gst \to 0,\ EA \text{ and Induction Hypothesis})
\end{aligned}
$$

It thus follows immediately that Theorem 2 holds for the parallel composition. □

5 Conclusion

In this paper, we presented a translation from HCSP formal models into Simulink graphical models, so that the models can be simulated and tested using a MATLAB platform, thus avoiding expensive formal verification if the development is at a stage where it is considered unnecessary. Together with our previous work on encoding Simulink/Stateflow diagrams into HCSP, it provides a two-way path in the design of embedded systems. In addition, we proposed a justification of the translation, which uses UTP as a main vehicle for arguing formally for the correspondence between the two semantics.

Acknowledgements. The work is supported partly by "973 Program" under grant No. 2014CB340701, by NSFC under grants 91418204 and 91118007, by CDZ project CAP (GZ 1023), and by the CAS/SAFEA International Partnership Program for Creative Research Teams.

References

1. Simulink User's Guide. http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf
2. Stateflow User's Guide. http://www.mathworks.com/help/pdf_doc/stateflow/sf_using.pdf
3. Tiller, M.: Introduction to Physical Modeling with Modelica. Springer, New York (2001)
4. SysML V 1.4 Beta Specification (2013). http://www.omg.org/spec/SysML
5. Selic, B., Gerard, S.: Modeling and Analysis or Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. The Springer International Series in Engineering and Computer Science. The MK/OMG Press, Burlington (2013)

6. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, H., Passerone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: an integrated electronic system design environment. IEEE Comput. **36**(4), 45–52 (2003)
7. Eker, J., Janneck, J., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity - the ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003)
8. Henzinger, T.: The theory of hybrid automata. In: LICS 1996, pp. 278–292, July 1996
9. Alur, R., Henzinger, T.A.: Modularity for timed and hybrid systems. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 74–88. Springer, Berlin (1997). doi:10.1007/3-540-63141-0_6
10. He, J.: From CSP to hybrid systems. In: A Classical Mind, Essays in Honour of C.A.R. Hoare, pp. 171–189. Prentice Hall International (UK) Ltd. (1994)
11. Chaochen, Z., Ji, W., Ravn, A.P.: A formal description of hybrid systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) HS 1995. LNCS, vol. 1066, pp. 511–530. Springer, Berlin (1996). doi:10.1007/BFb0020972
12. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. J. Log. Comput. **20**(1), 309–352 (2010)
13. Liu, J., Lv, J., Quan, Z., Zhan, N., Zhao, H., Zhou, C., Zou, L.: A calculus for hybrid CSP. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 1–15. Springer, Berlin (2010). doi:10.1007/978-3-642-17164-2_1
14. Zou, L., Zhan, N., Wang, V., Fränzle, M., Qin, S.: Verifying simulink diagrams via a hybrid hoare logic prover. In: EMSOFT 2013, pp. 1–10 (2013)
15. Zou, L., Zhan, N., Wang, S., Fränzle, M.: Formal verification of simulink/stateflow diagrams. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 464–481. Springer, Cham (2015). doi:10.1007/978-3-319-24953-7_33
16. Chen, M., Ravn, A.P., Wang, S., Yang, M., Zhan, N.: A two-way path between formal and informal design of embedded systems (extended version). http://lcs.ios.ac.cn/~chenms/papers/UTP2016_FULL.pdf
17. Simulink Design Verifier User's Guide (2010). http://www.manualslib.com/manual/392930/Matlab-Simulink-Design-Verifier-1.html#manual
18. Han, Z., Mosterman, P.J.: Towards sensitivity analysis of hybrid systems using simulink. HSCC **2013**, 95–100 (2013)
19. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time simulink to lustre. ACM Trans. Embedded Comput. Syst. **4**(4), 779–818 (2005)
20. Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., Maraninchi, F.: Defining and translating a "safe" subset of simulink/stateflow into lustre. In: EMSOFT 2004, pp. 259–268. ACM (2004)
21. Cavalcanti, A., Clayton, P., O'Halloran, C.: Control law diagrams in *Circus*. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 253–268. Springer, Berlin (2005). doi:10.1007/11526841_18
22. Woodcock, J., Cavalcanti, A.: The semantics of *Circus*. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Berlin (2002). doi:10.1007/3-540-45648-1_10
23. Meenakshi, B., Bhatnagar, A., Roy, S.: Tool for translating simulink models into input language of a model checker. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 606–620. Springer, Berlin (2006). doi:10.1007/11901433_33
24. Sfyrla, V., Tsiligiannis, G., Safaka, I., Bozga, M., Sifakis, J.: Compositional translation of simulink models into synchronous BIP. In: IEEE Fifth International Symposium on Industrial Embedded Systems, SIES 2010, pp. 217–220. IEEE (2010)

25. Bliudze, S., Sifakis, J.: The algebra of connectors - structuring interaction in BIP. IEEE Trans. Comput. **57**(10), 1315–1330 (2008)
26. Yang, C., Vyatkin, V.: Transformation of simulink models to IEC 61499 Function Blocks for verification of distributed control systems. Control Eng. Pract. **20**(12), 1259–1269 (2012)
27. Zhou, C., Kumar, R.: Semantic translation of simulink diagrams to input/output extended finite automata. Discrete Event Dyn. Syst. **22**(2), 223–247 (2012)
28. Minpoli, S., Frehse, G.: SL2SX translator: from simulink to SpaceEx verification tool. In: HSCC 2016 (2016)
29. Chen, R., Dong, J.S., Sun, J.: A formal framework for modeling and validating simulink diagrams. Formal Asp. Comput. **21**(5), 451–483 (2009)
30. Boström, P.: Contract-based verification of simulink models. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 291–306. Springer, Berlin (2011). doi:10.1007/978-3-642-24559-6_21
31. Roy, P., Shankar, N.: Simcheck: a contract type system for simulink. ISSE **7**(2), 73–83 (2011)
32. Preoteasa, V., Tripakis, S.: Refinement calculus of reactive systems. In: EMSOFT 2014, pp. 2:1–2:10 (2014)
33. Dragomir, I., Preoteasa, V., Tripakis, S.: Compositional semantics and analysis of hierarchical block diagrams. In: Bošnački, D., Wijs, A. (eds.) SPIN 2016. LNCS, vol. 9641, pp. 38–56. Springer, Cham (2016). doi:10.1007/978-3-319-32582-8_3
34. Zhan, N., Wang, S., Zhao, H.: Formal modelling, analysis and verification of hybrid systems. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Unifying Theories of Programming and Formal Engineering Methods. LNCS, vol. 8050, pp. 207–281. Springer, Berlin (2013). doi:10.1007/978-3-642-39721-9_5
35. Wang, S., Zhan, N., Guelev, D.: An assume/guarantee based compositional calculus for hybrid CSP. In: Agrawal, M., Cooper, S.B., Li, A. (eds.) TAMC 2012. LNCS, vol. 7287, pp. 72–83. Springer, Berlin (2012). doi:10.1007/978-3-642-29952-0_13
36. Guelev, D., Wang, S., Zhan, N.: Hoare reasoning about HCSP in the duration calculus (2013, submitted)
37. Hoare, C., He, J.: Unifying Theories of Programming, vol. 14. Prentice Hall, Englewood Cliffs (1998)
38. Zou, L., Lv, J., Wang, S., Zhan, N., Tang, T., Yuan, L., Liu, Y.: Verifying chinese train control system under a combined scenario by theorem proving. In: Cohen, E., Rybalchenko, A. (eds.) VSTTE 2013. LNCS, vol. 8164, pp. 262–280. Springer, Berlin (2014). doi:10.1007/978-3-642-54108-7_14