

A Two-way Path between Formal and Informal Design of Embedded Systems

Mingshuai Chen¹, Anders P. Ravn², Shuling Wang¹, Mengfei Yang³, and Naijun Zhan¹

¹ State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences
{chenms, wangsl, znj}@ios.ac.cn

² Department of Computer Science, Aalborg University
apr@cs.aau.dk

³ Chinese Academy of Space Technology

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 HCSP formal models into Simulink graphical models, so that HCSP formal 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 the desired two-way path in the design of embedded systems, so that the designer can flexibly shift between formal and informal models, according to the trade-off between efficiency and cost as opposed to correctness and reliability. The translation from HCSP into Simulink diagrams is implemented as a fully automatic tool, and the benefit of the flexibility is demonstrated by a scenario originating from the design of a spacecraft. In addition, the correctness of the translation is justified by using Unifying Theories of Programming (UTP).

Keywords: Simulink, HCSP, Validation, Verification, Embedded systems, 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 approach to meet this challenge. 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 a lower-level model step by step, until it can be built with existing components or a few newly developed ones.

Therefore, modelling, analysis and verification play a key role in MBD. Many MBD approaches targeting embedded systems have been proposed and used in industry and academia, e.g., Simulink/Stateflow [1, 2], Modelica [3], SysML [4], MARTE [5],

Metropolis [6], Ptolemy [7], hybrid automata [8], CHARON [9], HCSP [10, 11], Differential Dynamic Logic [12], Hybrid Hoare Logic [13].

These approaches can be classified into two paradigms, a simulation-based informal one such as [1–5] or a verification based formal one like [6–13]. 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], which is built by automatically translating Simulink/Stateflow (S/S) diagrams into Hybrid CSP (HCSP) [10, 11], a formal modelling language for hybrid discrete-continuous systems. As formal analysis of HCSP models is supported by an interactive Hybrid Hoare Logic (HHL) prover based on Isabelle/HOL [13, 16–18], which provides a gateway to mechanized verification of S/S models. The translation from the formal to informal model presented here, is implemented as a fully automatic tool, and the benefit of the flexibility is demonstrated by an example originating from the design of a spacecraft.

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 UTP semantics of the HCSP constructs and that of the corresponding Simulink constructs.

1.1 Related work

There has been a range of work on translating Simulink/Stateflow into modelling formalisms supported by analysis and verification tools. Mathworks itself released a tool named *Simulink Design Verifier* [19] (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 [20] can be used to verify system-level properties in a bounded time, but cannot be applied for unbounded verification.

In order to commit system-level verification and/or code generation of Simulink models, there have been a range of work on translating Simulink into other modelling formalisms, for which analysis and verification tools are developed. Tripakis *et al.* [21] 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 [22]. Cavalcanti *et al.* [23] presented a semantics for discrete-time Simulink diagrams using Circus [24], a combination of Z and CSP. Meenakshi *et al.* [25] gave an algorithm that translates a subset of Simulink into input language of model checker

NuSMV. Sifakis *et al.* proposed a translation into BIP in [26]. BIP [27] stands for Behaviour, Interaction and Priority, which is a component-based formal model for real-time concurrent systems. Among all the work mentioned above, continuous time models of Simulink are not considered. In [28], Yang and Vyatkin considered how to translate Simulink into Function Blocks. Zhou and Kumar investigated how to translate Simulink into *Input/Output Hybrid Automata* [29], while the translation of both discrete and continuous time fragments of Simulink into *SpacEx Hybrid Automata* was considered in [30]. In [31], Chen *et al.* considered how to translate Simulink models to a real-time specification language Timed Interval Calculus (TIC). Based on which continuous Simulink diagrams can be analyzed by a theorem prover. However, the translation is limited as it can only handle continuous blocks whose outputs can be represented explicitly by a mathematical relation on inputs. In contrast, in [14], we gave a translation from Simulink into HCSP. Our approach can handle all continuous blocks by using the notion of differential equations and invariants.

In addition, contract-based frameworks for Simulink are described in [32, 33]. In [32], Simulink diagrams are represented by SDF graphs, and discrete-time blocks are specified by contracts consisting of a pair of pre/post-conditions. Then sequential code is generated from the SDF graph, and the code is verified using traditional refinement-based techniques. In [33], 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 the deductive verification techniques based on that.

In [34], 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 [35], 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 [34].

There have been several formal semantics defined for HCSP. In He's original work on HCSP [10], an algebraic semantics of HCSP was given by defining a set of algebraic laws for the constructs of HCSP. Subsequently, a DC-based semantics for HCSP was presented in [11] due to Zhou *et al.* 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 computation* and recursion are not well handled in [11]. In [13, 17, 36, 37], 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 the correspondence between the two semantics, rendering HCSP more practical to engineers.

The rest of this paper is organized as follows. After introducing some preliminaries on HCSP and Simulink in Section 2, Section 3 presents the translation from HCSP into Simulink. We further provide a prototypical implementation of the translator in Section 4, followed a case study on a lunar lander in Section 5. Section 6 presents a justification

of the translation by proving consistency of the UTP semantics. A conclusion is drawn in Section 7.

2 Preliminaries

In this section, we briefly review HCSP and Simulink.

2.1 Hybrid CSP (HCSP)

HCSP [10, 11, 17] 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 different processes in parallel. A comprehensive introduction to HCSP can be found in [17].

The syntax of HCSP processes is given below:

$$\begin{aligned} P &::= \text{skip} \mid x := e \mid ch?x \mid ch!e \mid P; Q \mid B \rightarrow P \mid P \sqcup Q \mid P^* \\ &\quad \mid \langle F(\dot{s}, s) = 0 \& B \rangle \mid \langle F(\dot{s}, s) = 0 \& B \rangle \triangleright \bigsqcup_{i \in I} (io_i \rightarrow Q_i) \\ S &::= P \mid S \parallel S \end{aligned}$$

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 intended meaning of the individual constructs is as follows:

- skip , $x := e$ (assignment), $ch?x$ (input), $ch!e$ (output), $P; Q$ (sequential composition), $B \rightarrow P$ (conditional statement), $P \sqcup Q$ (internal choice), P^* (repetition) and $S \parallel S$ (parallel composition) have their standard meaning.
- $\langle F(\dot{s}, s) = 0 \& B \rangle$ is the evolution statement, 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 turns false.
- $\langle F(\dot{s}, s) = 0 \& B \rangle \triangleright \bigsqcup_{i \in I} (io_i \rightarrow Q_i)$ 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

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 of embedded systems and their environments.

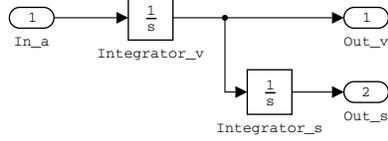
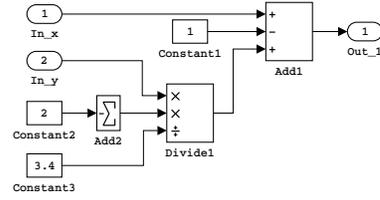


Fig. 1. The plant of a train control system

Fig. 2. $x - 1 + y * ((-2)/3.4)$

A Simulink model contains a set of blocks, subsystems, and wires, where blocks and subsystems cooperate by setting values on the wires between them. Fig. 1 gives a Simulink model of train movement, where rounded rectangles `In_a`, `Out_v` and `Out_s` are in-ports and out-ports for subsystems, and represent the acceleration, velocity, and trajectory of the train respectively. The two rectangular blocks `Integrator_v` and `Integrator_s` are *integrator* blocks of the Simulink library, each of which contains an internal parameter to represent the initial value of the output. An integrator block outputs its initial value at the beginning and the integration of the input signal afterwards. Hence, the block `Integrator_v` outputs the velocity of the train, which is the integration of the input acceleration `In_a`; and on the other hand, the block `Integrator_s` outputs the trajectory of the train, which is the integration of the input velocity.

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 functionalities. 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, 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 computed 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

In this section, we explain the details of the translation from HCSP processes as well as its subcomponents into graphical Simulink models. The translation starts from the most basic ingredients, i.e. expressions, to primitive statements and then is followed by compositional components.

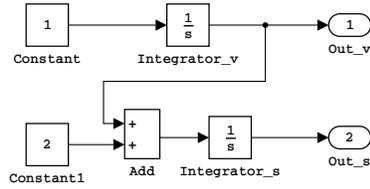
Fig. 3. $\dot{v} = 1, \dot{s} = v + 2$ 

Fig. 4. skip Statement

3.1 Expressions

Arithmetic expressions in HCSP are defined as

$$e \hat{=} x \mid c \mid -e \mid (e) \mid e + e \mid e - e \mid e * e \mid e / e$$

where x is a real variable, c stands for a real constant, and $+, -, *, /$ respectively for addition, subtraction, multiplication, and division of reals.

We construct a normal subsystem in Simulink to interpret an arithmetic expression e from HCSP, inside which 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. Fig. 2 shows the Simulink subsystem for the expression $x - 1 + y * ((-2)/3.4)$.

Boolean expressions are translated similarly.

3.2 Differential Equations

The syntax of differential equations in HCSP is $F \hat{=} \dot{s} = e \mid F, F$, 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.

In our approach, 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 , and equations in the same group are a normal subsystem in Simulink. In the translation of $\dot{v} = 1, \dot{s} = v + 2$, illustrated in Fig. 3, 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 the semantics of 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 as indicated in the previous section, but processes in HCSP can be executed sequentially, thus we need to provide a method to specify sequential execution in a Simulink diagram. Inspired by UTP [38], we introduce a pair of Boolean signals ok and ok' into each

subsystem to indicate initiation and termination. If ok' is false, the process has not terminated and the final values of the process variables are unobservable. Similarly, if ok is false, the process has never started and even the initial values are unobservable. These considerations underlie the validity of the translation of sequential composition. Additionally, ok and ok' are local variables to each subsystem corresponding to an HCSP process, and they never occur in the process statements. In a Simulink subsystem ok and ok' are constructed as an in-port signal named `In_ok` and an out-port signal named `Out_ok` respectively.

Since skip does nothing and terminates instantly, the subsystem for skip in Simulink is illustrated in Fig. 4, where $ok' = ok$ indicating that whenever the process skip starts, it terminates immediately without any effect. Also, there is no variable in the alphabet of a skip process, thus there are only ports for ok and ok' .

3.4 Assignment

Fig. 5 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:

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

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. 6, where the group of differential equations \mathcal{F} and the Boolean condition B are encapsulated into a single subsystem respectively. The enabled subsystem F 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:

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

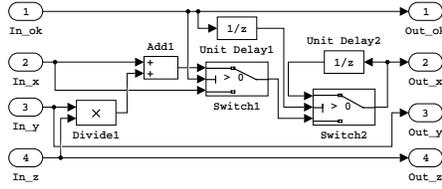
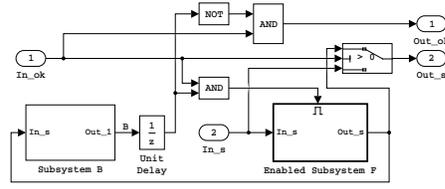
Fig. 5. $x := x + y * z$ 

Fig. 6. Continuous Evolution

3.6 Conditional Statement

Fig. 7 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 in-ports for B or P , which is not presented in Fig. 7. Accordingly, the output signals are computed according to

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

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 $x' \in outSigs(P) \cap outSigs(Q)$, 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 $y' \in outSigs(P) - outSigs(Q)$, 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 $z' \in outSigs(Q) - outSigs(P)$, analogously.

Fig. 8 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. 8, 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 signal⁴. The computation of signal ok and ok' can be formalized as

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

⁴ 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.

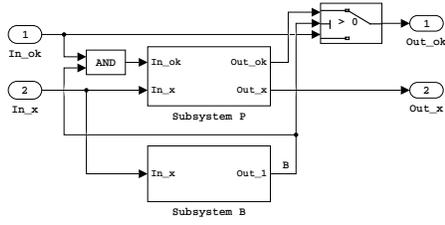


Fig. 7. Conditional Statement

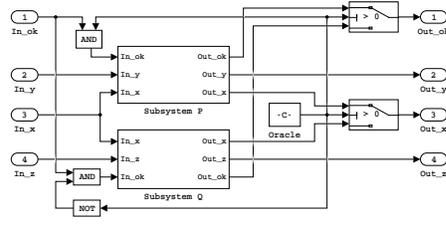


Fig. 8. Internal Choice

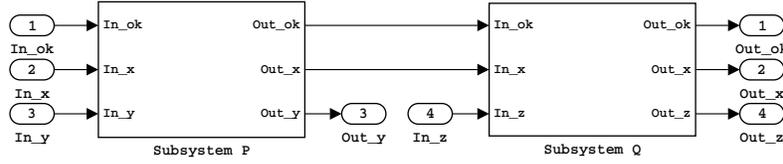


Fig. 9. Sequential Composition

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

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

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.

The integrated pattern to encode repetition p^* into Simulink diagrams is elaborated in Fig. 10. 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:

$$n = ok \times (d(n) + d(ok'_P \wedge \neg d(ok'_P))) \quad ok' = ok \wedge ok'_P \wedge (n \geq N)$$

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

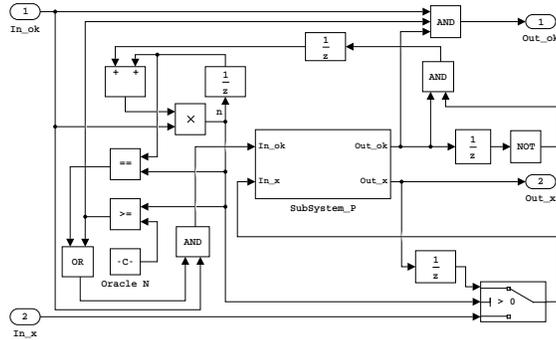


Fig. 10. Repetition

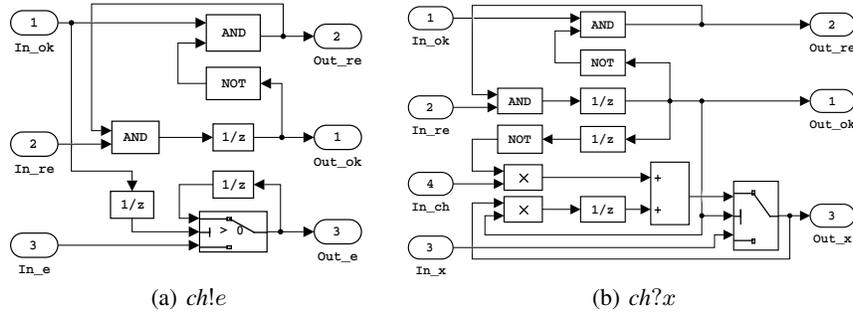


Fig. 11. Communication Events

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. Fig. 11 illustrates the Simulink diagrams that interpret communication events, which can be elaborated in the following two parts.

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

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

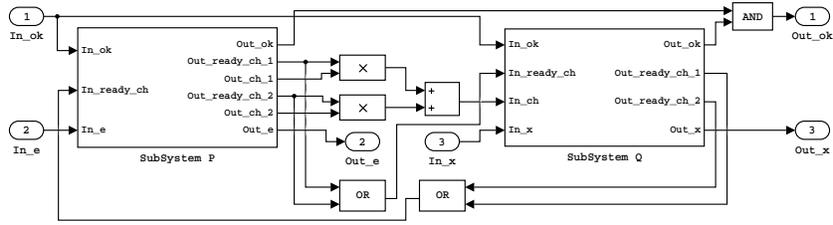


Fig. 12. Parallel $e := 0; ch!e; \langle \dot{e} = 1 \& e < 2 \rangle; ch!e || x := 3; ch?x; ch?x$

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.

We elaborate the above idea by showing a Simulink diagram corresponding to a parallel process in Fig. 12, 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. 12 is done by

$$ok_P = ok_Q = ok \quad ok' = ok'_P \wedge ok'_Q \quad re_{ch.P} = \bigvee_{i=1}^n re'_{ch.i.Q} \quad re_{ch.Q} = \bigvee_{j=1}^m re'_{ch.j.P}$$

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 $\parallel_{i \in I} (io_i \rightarrow Q_i)$ 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 $P \parallel R$ as an example, where $P \hat{=} io_1; Q_1$ and $R \hat{=} io_2; Q_2$, 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 $P \parallel R$

$$\text{are given by } ok' = ok'_P \vee ok'_R, x' = \begin{cases} x'_P, ok'_P \\ x'_R, ok'_R \\ x, \neg ok'_P \wedge \neg ok'_R \end{cases}.$$

3.13 Interruption

Obviously, $\langle F(\dot{s}, s) = 0 \& B \rangle \triangleright \parallel_{i \in I} (io_i \rightarrow Q_i)$ is equivalent to $\langle F(\dot{s}, s) = 0 \& (B \wedge \neg re'_R) \rangle; re'_R \rightarrow \parallel_{i \in I} (io_i \rightarrow Q_i)$, where $re'_R = f(\bigvee_{i \in I} re'_{oi_i})$, and the translation rules can be composed in a semantic-preserving way (see Section 6). Hence, translating an interruption into a Simulink diagram becomes a composition of translating various components that have been illustrated in previous subsections.

4 Implementation

The translation from HCSP processes into Simulink diagrams is implemented in C++ as a prototype, called *H2S*⁵, which takes an HCSP model as input and generates a Simulink graphical model encoded in a file of *mdl* format. The obtained graphical model can be simulated with configurations of several customized parameters to validate properties in the design of embedded systems. Thanks to the inherently compositional structure of HCSP processes, the translator is implemented with a set of recursive functions. *H2S* works in a fully automatic translation mode with linear complexity in both time and space. Furthermore, the generated Simulink model is compositional as well, which means the size of the Simulink model is approximately linear in the size of the original HCSP model.

⁵ Both the tool and the case study in next section are found at <http://lcs.ios.ac.cn/~chenms/tools/H2S.tar.bz2>.

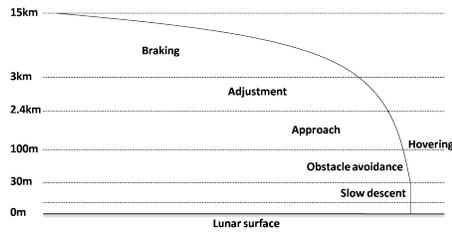


Fig. 13. Powered descent process

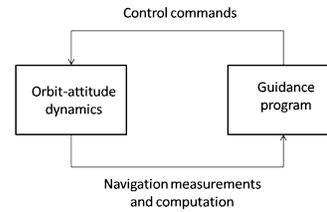


Fig. 14. A simplified configuration of GNC

For using the tool H2S, users should have a C++ compiler, Flex and Bison, as well as a library OGDF installed. Additionally, a group of customized parameters, e.g. length and stepsize of the simulation, can be specified in the configuration dialog before switching on the simulation.

5 Combining Informal and Formal Approaches

In this section, we show how to move between formal and informal design in a case study of the descent guidance control program of a lunar lander. It is a sampled-data control system composed of the physical plant and the embedded control program.

5.1 Description of the Design Problem

In the lunar lander mission, launched by China at the end of 2013 to achieve its first soft-landing and roving exploration on the moon, powered descent (see Fig. 13) is the most challenging task, because it is fully autonomous without remote control commands from earth, i.e. the lander must rely on its own guidance, navigation and control (GNC) system to, in real time, assess its current state, and generate control commands to adjust its attitude and engine thrust, to achieve soft-landing on the lunar surface. Therefore, the reliability of the GNC system is key to the success of soft-landing.

The slow descent phase is the final phase of the powered descent (see Fig. 13), which begins at an altitude of approximately 30m over the lunar surface, and terminates when the engine shut-down signal is received. The task of this phase is to ensure that the lander descends slowly and smoothly to the lunar surface, by nulling the horizontal velocity, maintaining a prescribed uniform vertical velocity, and keeping the lander in an upright position.

The integrated system is composed of the lander's dynamics and the guidance program for the present phase (see Fig. 14), where the guidance program is executed periodically with a fixed sampling period. At each sampling point, the current state of the lander is measured by an IMU (inertial measurement unit) or various sensors. Processed measurements are then input into the guidance program, which outputs control commands, e.g. the magnitude and direction of thrust, to be imposed on the lander's dynamics in the following sampling cycle. For more details of the dynamics and the guidance program, please refer to [39].

```

P ≜ PC || PD

PC ≜ v := -2; m := 1250; r := 30;
      (⟨Sys1&f > 3000⟩ ⊇ CommI;
      ⟨Sys2&f ≤ 3000⟩ ⊇ CommI)*

PD ≜ t := 0; g := 1.622; vslw := -2; f1 = 2027.5;
      (⟨chv?v1; chm?m1; f1 := m1 * aIC; chf!f1;
      temp := t; ⟨t = 1 & t < temp + 0.128⟩)*

aIC ≜ g - 0.01 * (f1/m1 - g) - 0.6 * (v1 - vslw)

Sys1 ≜ ṁ = -f/2548, v̇ = f/m - 1.622, ṙ = v

Sys2 ≜ ṁ = -f/2842, v̇ = f/m - 1.622, ṙ = v

CommI ≜ chf?f → skip || chv!v → skip || chm!m → skip

```

In the program *PC* is the continuous part of the system, i.e. the physical plant, and *PD* is the discrete part, i.e. the control program. Apparently the two process execute in parallel and interact through a group of communications, which happens periodically for every 0.128s. *aIC* stands for the commanded acceleration, namely the acceleration that should be provided by the engine thrust to maintain a smooth descent, and *g* denotes the gravitational acceleration on the moon. In addition, the lander's dynamics comprises two different forms, i.e. *Sys*₁ and *Sys*₂, depending on the magnitude of the current thrust *f*.

Fig. 15. HCSP program for the GNC control program

Design Objectives. A correct GNC control program to control the slow descent phase, with the assumption that the lunar lander enters the slow descent phase at $r = 30\text{m}$ with $v = -2\text{m/s}$, $m = 1250\text{kg}$ and $f = 2027.5\text{N}$, where r , v and m denote the altitude, vertical velocity and mass of the lunar lander, respectively, and f is the thrust imposed on the lander, satisfies the following requirements⁶:

- (R1) $|v - vslw| \leq \varepsilon$ during the slow descent phase and before touchdown, where $\varepsilon = 0.05$ is the tolerance of fluctuation of v around the target $vslw = -2\text{m/s}$;
- (R2) $|v| < vMax$ at the time of touchdown, where $vMax = 5\text{m/s}$ is the specified upper bound of $|v|$ to avoid the lander's crash when contacting the lunar surface;
- (R3) the system will finally exit the slow descent phase⁷.

5.2 From Simulink to HCSP

In [39], a Simulink model of the guidance control program of the slow descent was first built and simulated, then the Simulink model was translated into HCSP using the tool Sim2HCSP. The result is shown in Fig. 15.

Based on the translated formal model, the required properties have been proved by combining three verification tools, including iSAT-ODE [40], Flow* [41] and HHL Prover [14], which does improve the reliability of the program.

⁶ We abstract from disturbances here.

⁷ Note that if no shut-down signal is received, there exists a possibility that the lander stays in the slow descent phase after landing, which will damage the lander very much.

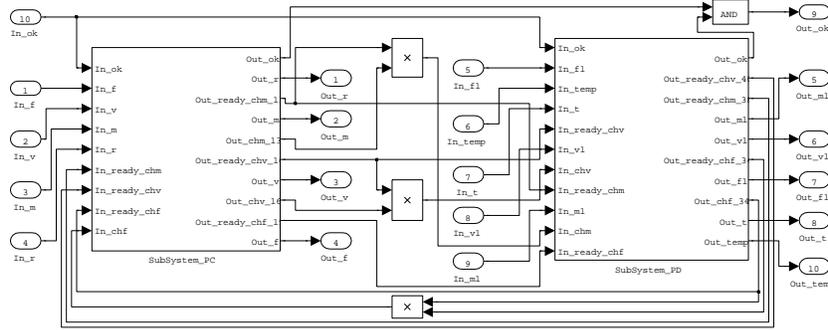


Fig. 16. The top-level view of the translated Simulink model

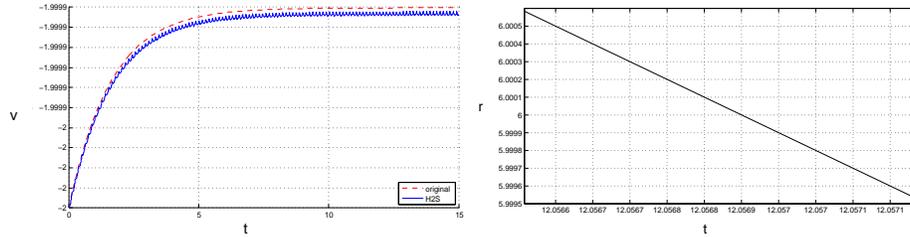


Fig. 17. The evolution in physical plant *PC*

But the problem is how to validate the translated HCSP formal model, as the correctness of the translation from Simulink/Stateflow was not addressed yet, and the domain experts or engineers cannot understand the formal model. To address this issue, we were suggested to consider translating the formal model back into Simulink, so that validation can be done by simulation. This is a part of the motivation of this work.

5.3 From HCSP to Simulink

Therefore, we translate the above HCSP model into a Simulink model using the tool H2S, which consists of 63 nested subsystems. The top-level view of the translated Simulink model is shown in Fig. 16, where a parallel pattern interprets the physical plant *PC* and the control program *PD*.

To validate the formal model, the translated Simulink model is simulated with a fixed simulation step of 0.0001s, and the evolution of the lander is shown as the quivering curve in Fig. 17. For velocity, we also illustrate the corresponding results of the original Simulink model as the dashed curve, showing that the translation loop keeps the system behaviours consistent. Moreover, the left part shows that the velocity of the lander is between -2 and -1.9999 m/s, which corresponds to (R1); the right part shows that if shut-down signal is sent out at 6m and is successfully received by the lander, then (R3) is satisfied at time 12.0569s; and then with a subsequent free fall, (R2) is guaranteed.

By combining formal and informal approaches in validation and verification of the lunar lander, the reliability was indeed improved, and the domain experts and engineers were also convinced.

6 Correctness of the Translation

In this section, we define UTP (Unifying theories of programming, [38]) semantics respectively for HCSP constructs and the corresponding Simulink diagrams, based on which proving the consistency of the two semantics essentially provide a justification of the correctness of the translation from HCSP processes to Simulink diagrams.

6.1 Extending UTP to Higher-order

UTP is, due to Hoare and He [38], 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 a predicate, denoted by $p(x) \vdash R(x, x')$, and 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 $\vdash R(x, x')$.

Definition 1. 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 $\forall x, x', ok, ok'. (P_2 \Rightarrow P_1)$. 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. $\rho(y, x'); D_1 \sqsubseteq D_2; \rho(y, x')$.

It is proved in UTP that the domain of designs forms a complete lattice with the refinement partial order, and *true* is the smallest (*worst*) element of the lattice. Furthermore, this lattice is closed under the classical programming constructs, and these constructs are *monotonic operations* on the lattice. These fundamental mathematical properties ensure that the domain of designs is a proper semantic domain for sequential programming languages. There is a nice link from the theory of designs to the theory of predicate transformers with the definition $\mathbf{wp}(p \vdash R, q) \triangleq p \wedge \neg(R; \neg q)$ that defines the *weakest precondition* of a design for a post condition q .

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

$$\mathcal{H}'(p \vdash R) \hat{=} (\vdash \wedge_{x \in \alpha(P)} x' = x \wedge \text{wait}' = \text{wait}) \triangleleft \text{wait} \triangleright (p \vdash R). \quad (1)$$

$P_1 \triangleleft b \triangleright P_2$ 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.

Obviously, hybrid systems are concurrent and reactive systems, so the UTP semantics of a hybrid system should satisfy the above *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 by underlining the following aspects:

- it allows 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 this sense. 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 quantifications.
- communication synchronization can only block discrete dynamics and keep discrete variable unchanged, yet cannot block the evolution of continuous dynamics, like time evolution. 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 ⁸, then the healthiness condition of reactive designs should be changed to

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

$$\mathcal{H}(S) \hat{=} (\vdash \mathbf{x}' = \mathbf{x} \wedge \text{wait}' = \text{wait} \wedge S_C) \triangleleft \text{wait} \triangleright S. \quad (3)$$

A design that meets the healthiness condition (2) is called a *hybrid design*, ranged over H, H_1 etc.

For simplicity, we will denote the left side of *wait* in Eq. (3) by Π_H for a given hybrid design H in the sequel.

- 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 the system.
- For a concrete hybrid system, the predicate *wait* can be defined explicitly.

For convenience, for each channel ch , we introduce two Boolean functions $ch!$ and $ch?$ over time. $ch!(t)$ means that ch is ready for sending a value at t , similarly, $ch?(t)$

⁸ We always assume time evolution is modeled in S_C , i.e., it contains $\dot{t} = 1$.

means that ch is ready for receiving a value at t . In addition, we use $\text{Periodic}(ch^*, st)$ to denote $\forall n \in \mathbb{N}. t = n * st \Rightarrow ch^*(t)$, which indicates that the communication event ch^* is ready periodically with period st . In addition, *maximal synchronization* semantics is adopted, i.e.,

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

which means that whenever the two parties of a communication are ready, the communication takes place immediately.

6.2 UTP Semantics for Simulink

In the following, we define the UTP semantics for Simulink blocks, diagrams, and subsystems respectively. For each of the Simulink constructs 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. Below we define the predicate for each Simulink construct C , denoted by $\llbracket C \rrbracket$. Also, we use $cnow$ to denote the current time of system.

Blocks As pointed out in [16], it is natural to interpret each block of a Simulink diagram as a predicate relating its inputs to the outputs using UTP. The behavior of each block can be divided into a set of sub-behaviors, each of which is 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:

$$\begin{aligned} & \llbracket B(ps, in, out) \rrbracket \\ & \hat{=} \mathcal{H}(Ass \vdash out(0) = ps.init \wedge \bigwedge_{k=1}^m (B_k(ps, in) \Rightarrow P_k(ps, in, out))), \end{aligned} \quad (5)$$

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.

Concretely, the UTP semantics of a continuous block can be given by a formula with the following form:

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

where $F_i(out, out, in, ps) = 0$ models the continuous evolution if B_i holds. In this case, $wait \hat{=} \neg out?$, 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 can be given by a formula with the following form:

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

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 [38]). 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 $cnow = 1$, meaning that the block keeps waiting (idle) except for the periodic points at which discrete jumps happen.

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

- A `Constant` block generates a scalar constant value:

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

The design inside \mathcal{H} is equivalent to $\vdash out = c \wedge out!$, should read $\vdash out(cnow) = c \wedge out!(cnow)$. Analogous remarks are applied subsequently.

- The `Add` block performs addition on its inputs, each of which is attached with a sign sn : ‘+’ or ‘-’:

$$\begin{aligned} & \llbracket \text{Add}(ps, \{sn_i\}_{i \in I}, \{in_i\}_{i \in I}, out) \rrbracket \\ \hat{=} & \mathcal{H}(\wedge_{i \in I} \text{Periodic}(in_i!, ps.st) \wedge \text{Periodic}(out?, ps.st) \vdash \text{Periodic}(out!, ps.st) \wedge \\ & (\exists n \in \mathbb{N}. cnow = n * ps.st) \Rightarrow out = \sum_{i \in I} sn_i * in_i). \end{aligned}$$

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

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

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

$$\begin{aligned} & \llbracket \text{Not}(in, out) \rrbracket \\ & \hat{=} \mathcal{H}(\text{Periodic}(in!, ps.st) \wedge \text{Periodic}(out?, ps.st) \vdash \text{Periodic}(out!, ps.st) \wedge \\ & \quad \exists n \in \mathbb{N}. cnow = n * ps.st \Rightarrow out = \neg in), \end{aligned}$$

$$\begin{aligned} & \llbracket \text{Or}(ps, \{in_i\}_{i \in I}, out) \rrbracket \\ & \hat{=} \mathcal{H}(\bigwedge_{i \in I} \text{Periodic}(in_i!, ps.st) \wedge \text{Periodic}(out?, ps.st) \vdash \text{Periodic}(out!, ps.st) \wedge \\ & \quad \exists n \in \mathbb{N}. cnow = n * ps.st \Rightarrow out = \bigvee_{i \in I} in_i), \end{aligned}$$

$$\begin{aligned} & \llbracket \text{And}(ps.I, \{in_i\}_{i \in I}, out) \rrbracket \\ & \hat{=} \mathcal{H}(\bigwedge_{i \in I} \text{Periodic}(in_i!, ps.st) \wedge \text{Periodic}(out?, ps.st) \vdash \text{Periodic}(out!, ps.st) \wedge \\ & \quad \exists n \in \mathbb{N}. cnow = n * ps.st \Rightarrow out = \bigwedge_{i \in I} in_i). \end{aligned}$$

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

$$\begin{aligned} & \llbracket \text{Relational}(ps.op, in_1, in_2, out) \rrbracket \\ & \hat{=} \mathcal{H}(\text{Periodic}(in_1!, ps.st) \wedge \text{Periodic}(in_2!, ps.st) \wedge \text{Periodic}(out?, ps.st) \vdash \\ & \quad \text{Periodic}(out!, ps.st) \wedge \exists n \in \mathbb{N}. cnow = n * ps.st \Rightarrow out = ps.op(in_1, in_2)). \end{aligned}$$

- The `Switch` block passes through the first input or the third input based on the value of the second input, thus we can define its UTP semantics as follows:

$$\begin{aligned} & \llbracket \text{Switch}(ps, in_1, in_2, in_3, out) \rrbracket \\ & \hat{=} \mathcal{H}(\bigwedge_{i=1}^3 \text{Periodic}(in_i!, ps.st) \wedge \text{Periodic}(out?, ps.st) \vdash \text{Periodic}(out!, ps.st) \wedge \\ & \quad (\exists n \in \mathbb{N}. cnow = n * ps.st) \Rightarrow \left(ps.op(in_2, ps.c) \Rightarrow out = in_1 \wedge \right. \\ & \quad \left. \neg ps.op(in_2, ps.c) \Rightarrow out = in_3 \right)). \end{aligned}$$

- A `Delay` block holds and delays its input by one sample period, therefore its UTP semantics can be define as:

$$\begin{aligned} & \llbracket \text{Delay}(ps, in, out) \rrbracket \\ & \hat{=} \mathcal{H}(in! \vdash \left(cnow < ps.st \Rightarrow out(cnow) = ps.init \wedge \right. \\ & \quad \left. cnow \geq ps.st \Rightarrow out(cnow) = in(cnow - ps.st) \right) \wedge out!). \end{aligned}$$

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

$$\llbracket \text{Integrator}(ps, in, out) \rrbracket \hat{=} \mathcal{H}(in! \vdash out(0) = ps.init \wedge (\dot{out} = in \wedge out!)).$$

Diagrams A diagram is a set of blocks connected through 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. According to the above discussion, suppose the UTP semantics for the m continuous blocks are given by

$$\begin{aligned} & \llbracket CB_i(ps_i, \{in_i\}_{i \in I_i}, out_i) \rrbracket \\ \hat{=} & \mathcal{H}(\wedge_{i \in I_i} in_i! \vdash out_i(0) = ps_i.init \wedge \\ & \left(\begin{array}{l} B_{i1}(\{in_i\}_{i \in I_i}, ps_i) \Rightarrow F_{i1}(out_i, out_i, \{in_i\}_{i \in I_i}, ps_i) = 0 \wedge \cdots \wedge \\ B_{im}(\{in_i\}_{i \in I_i}, ps_i) \Rightarrow F_{im}(out_i, out_i, \{in_i\}_{i \in I_i}, ps_i) = 0 \end{array} \right) \wedge out_i!), \end{aligned}$$

for $i = 1, \dots, m$, and the UTP semantics for the n discrete blocks are given by

$$\begin{aligned} & \llbracket DB_j(ps'_j, \{in'_j\}_{j \in J_j}, out'_j) \rrbracket \\ \hat{=} & \mathcal{H}(\wedge_{j \in J_j} \text{Periodic}(in'_j!, ps'_j.st) \wedge \text{Periodic}(out'_j?, ps'_j.st) \vdash out'_j(0) = ps'_j.init \wedge \\ & \text{Periodic}(out'_j!, ps'_j.st) \wedge (\exists n \in \mathbb{N}. cnow = n * ps'_j.st) \Rightarrow \\ & \left(\begin{array}{l} B_{j1}(\{in'_j\}_{j \in J_j}, ps'_j) \Rightarrow \llbracket P_{comp_{j1}}(ps'_j, \{in'_j\}_{j \in J_j}, out'_j) \rrbracket \wedge \cdots \wedge \\ B_{jm}(\{in'_j\}_{j \in J_j}, ps'_j) \Rightarrow \llbracket P_{comp_{jm}}(ps'_j, \{in'_j\}_{j \in J_j}, out'_j) \rrbracket \end{array} \right)), \end{aligned}$$

for $j = 1, \dots, n$. Then the UTP semantics of D can be represented by

$$\begin{aligned} & \llbracket D(ps^*, \{in_i^*\}_{i \in I}, \{out_i^*\}_{i \in J}) \rrbracket \\ \hat{=} & \exists \cup_{i=1}^m \{in_i\}_{i \in I_i} - \{in_i^*\}_{i \in I_C}, \exists \cup_{j=1}^n \{in'_j\}_{j \in J_j} - \{in_i^*\}_{i \in I_D}, \\ & \exists \{out_1, \dots, out_m\} - \{out_i^*\}_{i \in J_C}, \exists \{out'_1, \dots, out'_n\} - \{out_i^*\}_{i \in J_D}. \\ & \mathcal{H}(\wedge_{k \in I_D} \text{Periodic}(in_k^!, ps^*.st) \wedge \wedge_{k \in I_C} in_k^! \wedge \wedge_{k \in J_D} \text{Periodic}(out_k^?, ps_k^*.st) \\ & \vdash \wedge_{k \in J} out_k^*(0) = ps_k^*.init \wedge \wedge_{k \in J_C} out_k^*! \\ & \wedge \wedge_{i=1}^m \llbracket CB_i(ps_i, \{in_i\}_{i \in I_i}, out_i) \rrbracket \wedge \wedge_{j=1}^n (out'_j(0) = ps'_j.init)[\sigma, \rho] \wedge \\ & (\exists n \in \mathbb{N}. cnow = n * \text{GCD}(ps'_1.st, \dots, ps'_n.st) \Rightarrow (\wedge_{j=1}^n ps'_j.st \mid cnow \Rightarrow \\ & \left(\begin{array}{l} B_{j1}(\{in'_j\}_{j \in J_j}, ps'_j)[\sigma, \rho] \Rightarrow \llbracket P_{comp_{j1}}(ps'_j, \{in'_j\}_{j \in J_j}, out'_j) \rrbracket[\sigma, \rho] \wedge \cdots \wedge \\ B_{jm}(\{in'_j\}_{j \in J_j}, ps'_j)[\sigma, \rho] \Rightarrow \llbracket P_{comp_{jm}}(ps'_j, \{in'_j\}_{j \in J_j}, out'_j) \rrbracket[\sigma, \rho] \end{array} \right))), \end{aligned}$$

where

$$\begin{aligned} \{in_i^*\}_{i \in I_C} &= \cup_{i=1}^m \{in_i\}_{i \in I_i} - (\{out_1, \dots, out_m\} \cup \{out'_1, \dots, out'_n\}), \\ \{in_i^*\}_{i \in I_D} &= \cup_{j=1}^n \{in'_j\}_{j \in J_j} - (\{out_1, \dots, out_m\} \cup \{out'_1, \dots, out'_n\}), \\ \{out_i^*\}_{i \in J_C} &= \{out_1, \dots, out_m\} - (\cup_{i=1}^m \{in_i\}_{i \in I_i} \cup \cup_{j=1}^n \{in'_j\}_{j \in J_j}), \\ \{out_i^*\}_{i \in J_D} &= \{out'_1, \dots, out'_n\} - (\cup_{i=1}^m \{in_i\}_{i \in I_i} \cup \cup_{j=1}^n \{in'_j\}_{j \in J_j}); \end{aligned}$$

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

$$\text{wait} \hat{=} \bigwedge_{i=1}^m \neg \text{out}_i? \wedge \neg \exists n \in \mathbb{N}. \text{cnow} = n * \text{GCD}(ps'_1.st, \dots, ps'_n.st).$$

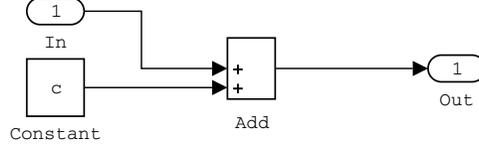


Fig. 18. A diagram *Diag* performing $out = in + c$

Example 2. Consider the diagram *Diag* performing $out = in + c$ in Fig. 18. According to the above discussion, its UTP semantics can be given as

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

Subsystems

Normal Subsystems A normal subsystem consists of 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 *NSub* can be easily defined as

$$\llbracket \text{NSub}(ps, \{in_i\}_{i \in I}, \{out_j\}_{j \in J}) \rrbracket \hat{=} \llbracket \text{Diag}(ps, \{in'_i\}_{i \in I'}, \{out'_j\}_{j \in J'}) \rrbracket[\sigma].$$

Enabled Subsystems By adding an enabled block in a normal subsystem, an enabled subsystem is created, which executes at each simulation step where the control signal has a positive value, otherwise holds the states to maintain their most recent values. So, its UTP semantics can be defined as follows:

$$\begin{aligned} & \llbracket \text{ESub}(ps, \{in_i\}_{i \in I}, en, \{out_j\}_{j \in J}) \rrbracket \\ \hat{=} & en(now) > 0 \Rightarrow \llbracket \text{NSub}(ps, \{in_i\}_{i \in I}, en, \{out_j\}_{j \in J}) \rrbracket \wedge \\ & en(now) \leq 0 \Rightarrow out(now) = out(now - ps.st). \end{aligned}$$

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

$$\mathcal{H}(\llbracket C \rrbracket) = \llbracket C \rrbracket.$$

Proof. It is straightforward by the definition of $\llbracket C \rrbracket$. □

6.3 UTP Semantics for HCSP

As advocated by Hoare and He in UTP [38], a reactive system can be identified by the set of all observations which could in any circumstances be made of that system, which is represented by a *description predicate* with the same name as the system for convenience. As usual, an alphabet is attached to a system P (and its behaviours predicate as well), including the following parts:

1. $\mathcal{V}(P)$: the set of both continuous and discrete variable names, which is arranged as a vector \mathbf{v} .
2. $i\Sigma(P)$: the set of input channel names.
3. $o\Sigma(P)$: the set of output channel names. We define $\Sigma(P) \hat{=} i\Sigma(P) \cup o\Sigma(P)$, which is arranged as a vector ch_P .

Given an alphabet of a hybrid system, its timed observation can be specified by a tuple

$$\langle now, \mathbf{v}, \mathbf{f}_{\mathbf{v}}, re_{ch^*}, msg_{ch} \rangle$$

where

- now is the start point (and now' the end point) of a time interval over which an observation is recorded.
- \mathbf{v} represent the initial values of variables \mathbf{v} , and \mathbf{v}' the final values at termination.
- $\mathbf{f}_{\mathbf{v}}$ is a vector of real-valued functions over the time interval $[now, now']$ for recording the values of \mathbf{v} , evidently with $\mathbf{f}_{\mathbf{v}}(now) = \mathbf{v}$, $\mathbf{f}_{\mathbf{v}}(now') = \mathbf{v}'$.
- re_{ch^*} is a vector of $\{0, 1\}$ -valued Boolean functions over $[now, now']$, indicating whether communication events ch^* are ready for communication.
- msg_{ch} , standing for messages, are a vector of real-valued functions over $[now, now']$ which record the values along channels ch .

We further define

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

Thereby using the UTP timed observations, the HCSP constructs can be defined respectively as follows.

- The skip statement, which does not alter the program state in any way, is modelled as the relational identity:

$$\llbracket \text{skip} \rrbracket \hat{=} \mathcal{H}(\vdash \text{now}' = \text{now} \wedge \mathbf{v}' = \mathbf{v} \wedge \text{const}(\mathbf{f}_v, \mathbf{v}, \text{now}, \text{now}') \wedge \text{const}(\text{re}_{ch^*}, \mathbf{0}, \text{now}, \text{now}') \wedge \text{const}(\text{msg}_{ch}, \text{msg}_{ch}(\text{now}), \text{now}, \text{now}')).$$

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

$$RE \hat{=} \text{const}(\text{re}_{ch^*}, \mathbf{0}, \text{now}, \text{now}') \wedge \text{const}(\text{msg}_{ch}, \text{msg}_{ch}(\text{now}), \text{now}, \text{now}').$$

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

$$\llbracket x := e \rrbracket \hat{=} \mathcal{H}(\vdash \text{now}' = \text{now} \wedge x' = e \wedge \mathbf{u}' = \mathbf{u} \wedge \text{const}(\mathbf{f}_x, e, \text{now}, \text{now}') \wedge \text{const}(\mathbf{f}_u, \mathbf{u}, \text{now}, \text{now}') \wedge RE).$$

Likewise, as assignment statement terminates immediately, *wait* is equivalent to *false* in this case.

- A continuous statement says that the system keeps waiting, meanwhile keeps continuously evolving, until the domain constraint is violated. So, the UTP semantics for continuous evolution is formulated as the following hybrid design

$$\llbracket (F(\dot{s}, s) = 0 \& B) \rrbracket \hat{=} (\vdash F(\dot{s}, s = 0) \wedge \dot{t} = 1) \triangleleft B \triangleright \llbracket \text{skip} \rrbracket.$$

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

- The conditional statement behaves according to whether the condition holds or not:

$$\llbracket B \rightarrow P \rrbracket \hat{=} \llbracket P \rrbracket \triangleleft B \triangleright \llbracket \text{skip} \rrbracket.$$

- The internal choice is interpreted as a non-deterministic selection between two operands :

$$\llbracket P \sqcup Q \rrbracket \hat{=} (\llbracket P \rrbracket \vee \llbracket Q \rrbracket).$$

- Let H_1 and H_2 be two hybrid designs with

$$H_1 \hat{=} (\vdash \wedge_{x \in \mathcal{V}(H_1)} x' = x \wedge \text{wait}'_{H_1} = \text{wait}_{H_1} \wedge S_{H_1}) \triangleleft \text{wait}_{H_1} \triangleright (p_{H_1} \vdash R_{H_1}),$$

$$H_2 \hat{=} (\vdash \wedge_{x \in \mathcal{V}(H_2)} x' = x \wedge \text{wait}'_{H_2} = \text{wait}_{H_2} \wedge S_{H_2}) \triangleleft \text{wait}_{H_2} \triangleright (p_{H_2} \vdash R_{H_2}),$$

which satisfy the healthiness condition in Eq. (2). We define the sequential composition of H_1 and H_2 , denoted by $H_1 \circledast H_2$ as

$$H_1 \circledast H_2 \hat{=} \exists \text{wait}_{H_1}, \text{wait}_{H_2}. \exists \mathbf{v}_{H_1}, \text{now}_{H_1}, \text{ok}_{H_1}.$$

$$\exists \mathbf{f}_{\mathbf{v}_{H_1}}, \text{re}_{ch_{H_1}^*}, \text{msg}_{ch_{H_1}}, \mathbf{f}_{\mathbf{v}_{H_2}}, \text{re}_{ch_{H_2}^*}, \text{msg}_{ch_{H_2}}.$$

$$(\vdash (\text{wait}_{H_1} \Rightarrow \Pi_{H_1}) \wedge (\text{wait}_{H_2} \Rightarrow \Pi_{H_2}) \wedge \text{wait}' = \text{wait}) \triangleleft \text{wait} \triangleright$$

$$(\neg \text{wait}_{H_1} \wedge \text{wait}_{H_2} \wedge r_{H_1} \vdash R_{H_1}) \sigma_{H_1} \wedge$$

$$(\neg \text{wait}_{H_1} \wedge \neg \text{wait}_{H_2} \wedge r_{H_2} \vdash R_{H_2}) \sigma_{H_2} \wedge$$

$$\forall t \in [\text{now}, \text{now}_{H_1}]. \text{wait}(t) = \text{wait}_{H_1}(t) \wedge$$

$$\mathbf{f}_v(t) = \mathbf{f}_{\mathbf{v}_{H_1}}(t) \wedge \text{re}_{ch^*}(t) = \text{re}_{ch_{H_1}^*}(t) \wedge \text{msg}_{ch}(t) = \text{msg}_{ch_{H_1}}(t) \wedge$$

$$\forall t \in [\text{now}_{H_1}, \text{now}']. \text{wait}(t) = \text{wait}_{H_2}(t) \wedge$$

$$\mathbf{f}_v(t) = \mathbf{f}_{\mathbf{v}_{H_2}}(t) \wedge \text{re}_{ch^*}(t) = \text{re}_{ch_{H_2}^*}(t) \wedge \text{msg}_{ch}(t) = \text{msg}_{ch_{H_2}}(t).$$

where

$$\begin{aligned}\sigma_{H_1} &= [\mathbf{v}_{H_1}/\mathbf{v}', \text{now}_{H_1}/\text{now}', \text{ok}_{H_1}/\text{ok}'] [\mathbf{f}_{\mathbf{v}_{H_1}}/\mathbf{f}_{\mathbf{v}}, \text{re}_{ch_{H_1}^*}/\text{re}_{ch^*}, \text{msg}_{ch_{H_1}}/\text{msg}_{ch}], \\ \sigma_{H_2} &= [\mathbf{v}_{H_1}/\mathbf{v}, \text{now}_{H_1}/\text{now}, \text{ok}_{H_1}/\text{ok}] [\mathbf{f}_{\mathbf{v}_{H_2}}/\mathbf{f}_{\mathbf{v}}, \text{re}_{ch_{H_2}^*}/\text{re}_{ch^*}, \text{msg}_{ch_{H_2}}/\text{msg}_{ch}].\end{aligned}$$

In the above,

$$\begin{aligned}\exists \mathbf{v}_{H_1}, \text{now}_{H_1}, \text{ok}_{H_1}. \exists \mathbf{f}_{\mathbf{v}}^{H_1}, \text{re}_{ch_{H_1}^*}, \text{msg}_{ch_{H_1}}, \mathbf{f}_{\mathbf{v}}^{H_2}, \text{re}_{ch_{H_2}^*}, \text{msg}_{ch_{H_2}}. \\ (\neg \text{wait}_{H_1} \wedge \text{wait}_{H_2} \wedge r_{H_1} \vdash R_{H_1}) \sigma_{H_1} \wedge (\neg \text{wait}_{H_1} \wedge \neg \text{wait}_{H_2} \wedge r_{H_2} \vdash R_{H_2}) \sigma_{H_2}\end{aligned}$$

is essentially equivalent to the sequential composition of the two designs $(\neg \text{wait}_{H_1} \wedge \text{wait}_{H_2} \wedge r_{H_1} \vdash R_{H_1})$ and $(\neg \text{wait}_{H_1} \wedge \neg \text{wait}_{H_2} \wedge r_{H_2} \vdash R_{H_2})$ by the theory of UTP [38].

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

Now, given two HCSP processes P and Q , $\llbracket P; Q \rrbracket$ can be naturally defined as

$$\llbracket P; Q \rrbracket \hat{=} \llbracket P \rrbracket \mathbin{\text{;}} \llbracket Q \rrbracket.$$

- 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.

$$\llbracket X \rrbracket \hat{=} X.$$

- The semantics for recursion is defined as the least fixed point of the corresponding recursive predicate by

$$\llbracket \text{rec } X.P \rrbracket \hat{=} \mu X. \llbracket P \rrbracket.$$

Given an HCSP process P , P^* can be defined as

$$P^* \Leftrightarrow \text{rec } X. (\text{skip} \sqcup (P; X)) \quad (6)$$

As discussed above, its semantics is given by

$$\llbracket P^* \rrbracket \Leftrightarrow \llbracket \text{rec } X. (\text{skip} \sqcup (P; X)) \rrbracket \Leftrightarrow \exists N. \llbracket P^N \rrbracket,$$

where $P^0 \hat{=} \text{skip}$.

- A receiving event can be modelled by the following hybrid design

$$\llbracket ch?x \rrbracket \hat{=} \vdash LHS \triangleleft \text{re}_{ch?} \wedge \neg \text{re}_{ch!} \triangleright RHS,$$

where

$$\begin{aligned}LHS &\hat{=} t = 1 \wedge x' = x \wedge \mathbf{u}' = \mathbf{u}, \\ RHS &\hat{=} \text{now}' = \text{now} + d \wedge \text{re}'_{ch?} = 0 \wedge \text{re}'_{ch!} = 0 \wedge \mathbf{u}' = \mathbf{u} \wedge x' = \text{msg}_{ch}(\text{now}') \wedge \\ &\quad \text{const}^l(\text{re}_{ch?}, 1, \text{now}, \text{now}') \wedge \text{const}^l(\text{re}_{ch!}, 0, \text{now}, \text{now}').\end{aligned}$$

In this case, $\text{wait} \hat{=} \text{re}_{ch?} \wedge \neg \text{re}_{ch!}$, i.e., the process keeps waiting until its dual communication event becomes ready.

- The sending event $\llbracket ch!e \rrbracket$ can be defined similarly.
- Let H_1 and H_2 be two hybrid designs with

$$H_1 \hat{=} (\vdash \wedge_{x \in \mathcal{V}(H_1)} x' = x \wedge wait'_{H_1} = wait_{H_1} \wedge S_{H_1}) \triangleleft wait_{H_1} \triangleright (p_{H_1} \vdash R_{H_1}),$$

$$H_2 \hat{=} (\vdash \wedge_{x \in \mathcal{V}(H_2)} x' = x \wedge wait'_{H_2} = wait_{H_2} \wedge S_{H_2}) \triangleleft wait_{H_2} \triangleright (p_{H_2} \vdash R_{H_2}),$$

which satisfy the healthiness condition in Eq. (2). Then, We define the parallel composition of H_1 and H_2 , denoted by $H_1 \parallel H_2$ as

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

It can be further proved that

$$\begin{aligned} H_1 \parallel H_2 \Leftrightarrow & \exists now_{H_1}, now_{H_2}, ok_{H_1}, ok_{H_2}. \\ & \vdash \left(\begin{array}{l} wait_{H_1} \Rightarrow \Pi_{H_1} \wedge \\ wait_{H_2} \Rightarrow \Pi_{H_2} \end{array} \right) \triangleleft wait_{H_1} \vee wait_{H_2} \triangleright \left(\begin{array}{l} (p_{H_1} \vdash R_{H_1})[ok/ok_{H_1}] \wedge \\ (p_{H_2} \vdash R_{H_2})[ok/ok_{H_2}] \end{array} \right) \wedge \\ & now' = max\{now'_{H_1}, now'_{H_2}\} \wedge (ok' = ok'_{H_1} \wedge ok'_{H_2}) \wedge \\ & (\forall t \in (now'_{H_1}, now']. \mathbf{f}_{\mathbf{v}_{H_1}}(t) = \mathbf{f}_{\mathbf{v}_{H_1}}(now'_{H_1}) \wedge \\ & \quad re_{ch_{H_1}^*}(t) = re_{ch_{H_1}^*}(now'_{H_1}) \wedge msg_{ch_{H_2}}(t) = msg_{ch_{H_2}}(now'_{H_2})) \wedge \\ & (\forall t \in (now'_{H_2}, now']. \mathbf{f}_{\mathbf{v}_{H_2}}(t) = \mathbf{f}_{\mathbf{v}_{H_2}}(now'_{H_2}) \wedge \\ & \quad re_{ch_{H_2}^*}(t) = re_{ch_{H_2}^*}(now'_{H_2}) \wedge msg_{ch_{H_2}^*}(t) = msg_{ch_{H_2}^*}(now'_{H_2})). \end{aligned}$$

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

Now, given two HCSP processes P and Q , $\llbracket P \parallel Q \rrbracket$ can be naturally defined as

$$\llbracket P \parallel Q \rrbracket \hat{=} \llbracket P \rrbracket \parallel \llbracket Q \rrbracket.$$

- The communication interruption can be defined as

$$\begin{aligned} \llbracket \langle F(\dot{s}, s) = 0 \& B \rangle \triangleright \llbracket_{i \in I} (io_i \rightarrow Q_i) \rrbracket \hat{=} & \llbracket \langle F(\dot{s}, s) = 0 \& (B \wedge \neg \Gamma) \rangle; \\ & \Gamma \rightarrow \llbracket_{i \in I} (io_i \rightarrow Q_i) \rrbracket \end{aligned}$$

where $\Gamma \hat{=} \bigvee_{i \in I} re'_{io_i}$, 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 the previously defined hybrid designs forms a complete lattice with a refinement partial order, on which the classical programming operations are closed.

6.4 Justification of correctness

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

1. We set for the sample times of all the discrete blocks to be -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. $\tau \hat{=} \min\{t|Out_ok(t) = 1\}$.
3. Hereafter we use $\llbracket wires \rrbracket$ 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. 5.
4. Unless otherwise stated, the parameters of a block will be abstracted away in the semantic function for simplicity. Besides, to distinguish the input/output signals of blocks, the leading characters of the input/output signals of a subsystem are capitalized.
5. The UTP semantics defined in Sect. 6.2 implicates that the signals which are not modified in the computation of a block (or subsystem) keep unchanged.

Theorem 2. *Given an HCSP process P , denote the translated Simulink diagram by $H2S(P)$. Suppose there is a correspondence (denoted by EA) between $\llbracket P \rrbracket$ and $\llbracket H2S(P) \rrbracket$, i.e., $now = gst$, $now' = \tau$, $ok = In_ok(gst) = \top$, $ok' = Out_ok(\tau)$, $v = In_v(gst)$, $v' = Out_v(\tau)$, $f_v = Out_v|_{[gst,\tau]}$, $re_{ch*} = Out_re_{ch*}|_{[gst,\tau]}$, and $msg_{ch} = Out_re_{ch}|_{[gst,\tau]}$, then we have*

$$Periodic(in!, ps.gst) \wedge Periodic(out?, ps.gst) \Rightarrow (\llbracket P \rrbracket \Leftrightarrow \llbracket H2S(P) \rrbracket)|_{[gst,\tau]} \quad (7)$$

as $gst \rightarrow 0$.

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

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

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

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

$$\begin{aligned}
& \llbracket \text{H2S}(x := e) \rrbracket \\
& \triangleq \exists \mathbf{ch} * . \llbracket \text{Wires} \rrbracket \wedge \llbracket \text{Diag}_e \rrbracket \wedge \llbracket \text{Del1} \rrbracket \wedge \llbracket \text{Del2} \rrbracket \wedge \llbracket \text{Sw1} \rrbracket \wedge \llbracket \text{Sw2} \rrbracket \\
& \Leftrightarrow \text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \vdash \text{Periodic}(out!, ps.gst) \wedge \\
& \quad \forall t \geq 0. (t < gst \Rightarrow out_Del1(t) = 0) \wedge (t \geq gst \Rightarrow out_Del1(t) = In_ok(t - gst)) \wedge \\
& \quad (t < gst \Rightarrow out_Del2(t) = 0) \wedge (t \geq gst \Rightarrow out_Del2(t) = Out_x(t - gst)) \wedge \\
& \quad (\exists n \in \mathbb{N}. cnow = n * ps.gst \Rightarrow \\
& \quad \quad (In_ok(cnow) > 0 \Rightarrow out_Sw1(cnow) = out_Diag_e(cnow)) \wedge \\
& \quad \quad (In_ok(cnow) \leq 0 \Rightarrow out_Sw1(cnow) = In_x(cnow)) \wedge \\
& \quad \quad (out_Del1(cnow) > 0 \Rightarrow out_Sw2(cnow) = out_Del2(cnow)) \wedge \\
& \quad \quad (out_Del1(cnow) \leq 0 \Rightarrow out_Sw2(cnow) = out_Sw1(cnow)) \wedge \\
& \quad \quad Out_x(cnow) = out_Sw2(cnow) \wedge Out_ok(cnow) = In_ok(cnow)
\end{aligned}$$

By providing the left-hand side of (7) and restricting the time interval of the behaviours, we get

$$\begin{aligned}
& \text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \wedge \llbracket \text{H2S}(x := e) \rrbracket_{[gst, \tau]} \\
& \Leftrightarrow (\exists n \in \mathbb{N}. cnow = n * ps.gst \wedge cnow \in [gst, \tau]) \Rightarrow (Out_ok(cnow) = In_ok(cnow)) \wedge \\
& \quad Out_x(cnow) = out_Diag_e(cnow) \\
& \Leftrightarrow ok' \wedge \tau = now \wedge x' = e \wedge const(f_x, e, now, \tau) \wedge \\
& \quad \mathbf{u}' = \mathbf{u} \wedge const(f_{\mathbf{u}}, \mathbf{u}, now, \tau) \wedge RE \qquad (gst \rightarrow 0, EA)
\end{aligned}$$

It thus follows evidently that

$$\text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \Rightarrow (\llbracket x := e \rrbracket \Leftrightarrow \llbracket \text{H2S}(x := e) \rrbracket_{[gst, \tau]}).$$

Continuous statement: By the defined UTP semantics, it follows

$$\begin{aligned}
& \llbracket \text{H2S}(\langle F(s, s) = 0 \& B \rangle) \rrbracket \triangleq \exists \mathbf{ch} * . \llbracket \text{Wires} \rrbracket \wedge \llbracket \text{NSubB} \rrbracket \wedge \llbracket \text{ESubF} \rrbracket \wedge \llbracket \text{Del} \rrbracket \wedge \\
& \quad \llbracket \text{Not} \rrbracket \wedge \llbracket \text{And1} \rrbracket \wedge \llbracket \text{And2} \rrbracket \wedge \llbracket \text{Sw} \rrbracket \\
& \Leftrightarrow \text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \vdash \text{Periodic}(out!, ps.gst) \wedge \\
& \quad \forall t \geq 0. (out_And1(t) > 0 \Rightarrow out_ESubF(t) = S(t)) \wedge \\
& \quad (out_And1(t) \leq 0 \Rightarrow out_ESubF(t) = out_ESubF(t - gst)) \wedge \\
& \quad (t < gst \Rightarrow out_Del(t) = 1) \wedge (t \geq gst \Rightarrow out_Del(t) = out_NSubB(t - gst)) \wedge \\
& \quad (\exists n \in \mathbb{N}. cnow = n * ps.gst) \Rightarrow \\
& \quad \quad out_NSubB(cnow) = B(cnow) \wedge out_Not(cnow) = \neg out_Del(cnow) \wedge \\
& \quad \quad out_And1(t) = (In_ok(cnow) \wedge out_Del(t)) \wedge out_And2(t) = (In_ok(cnow) \wedge \\
& \quad \quad out_Not(cnow)) \wedge (In_ok(cnow) > 0 \Rightarrow out_Sw(cnow) = out_ESubF(cnow)) \wedge \\
& \quad \quad (In_ok(cnow) \leq 0 \Rightarrow out_Sw(cnow) = In_s(cnow)) \wedge \\
& \quad \quad Out_s(cnow) = out_Sw(cnow) \wedge Out_ok(cnow) = out_And2(cnow)
\end{aligned}$$

By providing the left-hand side of (7) and restricting the time interval of the behaviours, we have

$$\begin{aligned}
& \text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \wedge ok \wedge \llbracket \text{H2S}(\langle F(\dot{s}, s) = 0 \& B \rangle) \rrbracket_{[gst, \tau - gst]} \\
\Leftrightarrow & \text{Out_ok}(\tau) = \top \wedge \tau - gst = gst + (\tau - 2 * gst) \wedge \text{Out_s}(\tau - gst) = S(\tau - 2 * gst) \wedge \\
& (\exists n \in \mathbb{N}. cnow = n * ps.gst \wedge cnow \in [gst, \tau]) \Rightarrow \text{out_NSubB}(cnow - gst) \wedge \\
& \quad f_s(cnow) = S(cnow - gst) \wedge \neg \text{out_NSubB}(\tau - gst) \wedge \\
& \quad \text{out_ESubF}(cnow) = \text{out_ESubF}(cnow - gst) \\
\Leftrightarrow & ok' \wedge \mathbf{u}' = \mathbf{u} \wedge \text{const}(f_{\mathbf{u}}, \mathbf{u}, now, \tau) \wedge RE \wedge \\
& (B \wedge \tau = now + d \wedge s' = S(d) \wedge \forall t \in [now, \tau]. f_s(t) = S(t - now) \vee \\
& \quad \neg B \wedge s' = s) \quad (gst \rightarrow 0, EA) \\
\Leftrightarrow & (\llbracket F(\dot{s}, s) = 0 \rrbracket \triangleleft B \triangleright \llbracket \text{skip} \rrbracket)
\end{aligned}$$

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

$$\begin{aligned}
& \text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \wedge ok \Rightarrow \\
& \quad (\llbracket \langle F(\dot{s}, s) = 0 \& B \rangle \rrbracket \Leftrightarrow \llbracket \text{H2S}(\langle F(\dot{s}, s) = 0 \& B \rangle) \rrbracket_{[gst, \tau]}).
\end{aligned}$$

Conditional statement: By the definition of H2S and the UTP semantics of Simulink given in Section 6.2, we have

$$\begin{aligned}
& \llbracket \text{H2S}(B \rightarrow P) \rrbracket \hat{=} \exists \mathbf{ch} * . \llbracket \text{wires} \rrbracket \wedge \llbracket \text{NSubB} \rrbracket \wedge \llbracket \text{NSubP} \rrbracket \wedge \llbracket \text{And} \rrbracket \wedge \llbracket \text{Swt} \rrbracket \\
\Leftrightarrow & \text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \vdash \text{Periodic}(out!, ps.gst) \wedge \\
& (\exists n \in \mathbb{N}. cnow = n * ps.gst) \Rightarrow \\
& \quad \text{out_NSubB}(cnow) = B(cnow) \wedge \llbracket \text{NSubP}(in_{ok} = \text{out_And}(cnow)) \rrbracket \wedge \\
& \quad \text{out_And}(cnow) = (\text{In_ok}(cnow) \wedge \text{out_NSubB}(cnow)) \wedge \\
& \quad (\text{out_NSubB}(cnow) > 0 \Rightarrow \text{out_Swt}(cnow) = \text{out_NSubP_ok}(cnow)) \wedge \\
& \quad (\text{out_NSubB}(cnow) \leq 0 \Rightarrow \text{out_Swt}(cnow) = \text{In_ok}(cnow)) \wedge \\
& \quad \text{Out_x}(cnow) = \text{out_NSubP_x}(cnow) \wedge \text{Out_ok}(cnow) = \text{out_Swt}(cnow)
\end{aligned}$$

It follows

$$\begin{aligned}
& \text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \wedge \llbracket \text{H2S}(B \rightarrow P) \rrbracket_{[gst, \tau]} \\
\Leftrightarrow & (B \wedge \llbracket P \rrbracket) \vee (\neg B \wedge ok' \wedge \tau = now \wedge v' = v \wedge \text{const}(f_v, v, now, \tau) \wedge RE) \wedge \\
& \quad \mathbf{u}' = \mathbf{u} \wedge \text{const}(f_{\mathbf{u}}, \mathbf{u}, now, \tau) \wedge RE \quad (gst \rightarrow 0, EA) \\
\Leftrightarrow & \llbracket P \rrbracket \triangleleft B \triangleright \llbracket \text{skip} \rrbracket
\end{aligned}$$

Thus we have

$$\text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \Rightarrow (\llbracket B \rightarrow P \rrbracket \Leftrightarrow \llbracket \text{H2S}(B \rightarrow P) \rrbracket_{[gst, \tau]}).$$

Internal choice: Similar to conditional statement, we can easily prove

$$\text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \Rightarrow (\llbracket P \sqcup Q \rrbracket \Leftrightarrow \llbracket \text{H2S}(P \sqcup Q) \rrbracket)_{[gst, \tau]}.$$

Sequential composition: As illustrates in Fig. 9, we use \mathbf{x} to denote the set of common signals processed by both P and Q , while \mathbf{y} and \mathbf{z} are exclusive signals respectively for P and Q .

$$\begin{aligned} & \llbracket \text{H2S}(P; Q) \rrbracket_{[gst, \tau]} \hat{=} \exists \mathbf{ch} * . \llbracket \text{Wires} \rrbracket_{[gst, \tau]} \wedge \llbracket \text{NSubP} \rrbracket_{[gst, \tau]} \wedge \llbracket \text{NSubQ} \rrbracket_{[gst, \tau]} \\ \Leftrightarrow & \llbracket \text{NSubP}(in_{ok} = In_{ok}(cnow), in_x = In_x(cnow), in_y = In_y(cnow)) \rrbracket_{[gst, \tau]} \wedge \\ & \llbracket \text{NSubQ}(in_{ok} = out_NSubP_{ok}(t), in_x = out_NSubP_x(t), in_z = In_z(t)) \rrbracket_{[gst, \tau]} \wedge \\ & (\exists n \in \mathbb{N}. cnow = n * ps.gst \wedge cnow \in [gst, \tau]) \Rightarrow \\ & \quad Out_{ok}(cnow) = out_NSubQ_{ok}(cnow) \wedge Out_x(cnow) = out_NSubQ_x(cnow) \wedge \\ & \quad Out_y(cnow) = out_NSubP_y(cnow) \wedge Out_z(cnow) = out_NSubP_z(cnow) \\ \Leftrightarrow & (\exists x_m, now_m, ok_m. (out_NSubP_{ok}(now_m) \Leftrightarrow ok) \wedge \\ & \llbracket \text{NSubP} \rrbracket[x_m/x', ok_m/ok'] \wedge \llbracket \text{NSubQ} \rrbracket[x_m/x, ok_m/ok] \wedge \\ & \forall t \geq 0. y(t) = out_NSubP_y(t) \wedge x(t) = out_NSubQ_x(t) \wedge \\ & \quad z(t) = out_NSubQ_z(t)) \quad (gst \rightarrow 0, EA \text{ and Induction Hypothesis}) \end{aligned}$$

It is followed immediately that

$$\text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \Rightarrow (\llbracket P; Q \rrbracket \Leftrightarrow \llbracket \text{H2S}(P; Q) \rrbracket)_{[gst, \tau]}.$$

Recursion: We only consider the tail recursion, i.e., repetition. The general recursion can be proved similarly. As shown in Fig. 10, a random number N , generated by an oracle, is introduced in the Simulink diagram to specify the number of iterations of subsystem P . Recall that $\llbracket P^* \rrbracket$ is defined by using least fix point. Thus, it is obvious that the inverse direction holds:

$$\text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \Rightarrow (\llbracket P^* \rrbracket \Leftarrow \llbracket \text{H2S}(P^*) \rrbracket)_{[gst, \tau]}.$$

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 $\text{H2S}(P^*)$, to execute for N times. The fact is thus proved, i.e.

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

Communication events: By the definition of H2S and the UTP semantics of Simulink given in Section 6.2, it follows

$$\begin{aligned}
& \text{Periodic}(in!, ps.gst) \wedge \text{Periodic}(out?, ps.gst) \wedge \llbracket \text{H2S}(ch?x) \rrbracket|_{[gst, \tau]} \\
\Leftrightarrow & (\exists n \in \mathbb{N}. cnow = n * ps.gst \wedge cnow \in [gst, \tau]) \Rightarrow \\
& \quad \text{Out}_{re}(cnow) = (\text{In}_{ok}(cnow) \wedge \neg \text{Out}_{ok}(cnow)) \wedge \\
& \quad \text{Out}_{ok}(cnow) = f(\text{In}_{re}(cnow - gst) \wedge \text{Out}_{re}(cnow - gst)) \wedge \\
& \quad (\neg \text{Out}_{ok}(cnow) \Rightarrow \text{Out}_{x}(cnow) = \text{In}_{x}(cnow)) \wedge \\
& \quad (\text{Out}_{ok}(cnow) \Rightarrow \text{Out}_{x}(cnow) = (\neg \text{Out}_{ok}(cnow - gst)) * \text{In}_{ch}(cnow)) \\
\Leftrightarrow & \text{Out}_{ok}(\tau) = \top \wedge \\
& (\exists n \in \mathbb{N}. cnow = n * ps.gst) \Rightarrow \\
& \quad cnow \in [gst, \tau - gst] \Rightarrow \text{Out}_{re}(cnow) = 1 \wedge \text{In}_{re}(t) = 0 \wedge \text{In}_{re}(\tau - gst) = 1 \wedge \\
& \quad cnow \in (\tau - gst, \tau] \Rightarrow \text{Out}_{re}(cnow) = 0 \wedge \text{In}_{re}(cnow) = 0 \wedge \\
& \quad cnow \in [gst, \tau] \Rightarrow \text{Out}_{x}(cnow) = \text{In}_{x}(cnow) \wedge \text{Out}_{x}(\tau) = \text{In}_{ch}(\tau) \\
\Leftrightarrow & ok' \wedge now' = now + d \wedge \text{const}(re_{ch?}, 1, now, now') \wedge \\
& \quad \text{const}^l(re_{ch!}, 0, now, now') \wedge re_{ch!}(now') = 1 \wedge re'_{ch?}(now') = 0 \wedge re'_{ch!}(now') = 0 \wedge \\
& \quad \text{const}^l(f_x, x, now, now') \wedge f_x(now') = msg_{ch}(now') \wedge \\
& \quad \text{const}(\mathbf{f}_u, \mathbf{u}, now, now') \wedge \mathbf{u}' = \mathbf{u} \wedge x' = msg_{ch}(now') \quad (gst \rightarrow 0, EA) \\
\Leftrightarrow & (LHS \triangleleft re_{ch?} \wedge \neg re_{ch!} \triangleright RHS)
\end{aligned}$$

Therefore, we get

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

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 \mathbb{P} and \mathbb{Q} . Let $\tau_P \hat{=} \min\{t | out_NSubP_ok(t) = 1\}$, and $\tau_Q \hat{=} \min\{t | out_NSubQ_ok(t) = 1\}$. Then, according to the definitions, we have

$$\begin{aligned}
& \llbracket \text{H2S}(P \parallel Q) \rrbracket|_{[gst, \tau]} \hat{=} \exists \mathbf{ch} * \llbracket \text{Wires} \rrbracket|_{[gst, \tau]} \wedge \llbracket \text{NSubP} \rrbracket|_{[gst, \tau]} \wedge \llbracket \text{NSubQ} \rrbracket|_{[gst, \tau]} \\
\Leftrightarrow & \llbracket \text{NSubP}(in_{ok} = \text{In}_{ok}(cnow), in_y = \text{In}_y(cnow)) \rrbracket|_{[gst, \tau]} \wedge \\
& \llbracket \text{NSubQ}(in_{ok} = \text{In}_{ok}(cnow), in_z = \text{In}_z(cnow)) \rrbracket|_{[gst, \tau]} \wedge \\
& (\exists n \in \mathbb{N}. cnow = n * ps.gst \wedge cnow \in [gst, \tau]) \Rightarrow \\
& \quad (\tau_P = \tau \vee \tau_Q = \tau) \wedge (\text{Out}_{ok}(cnow) = out_NSubP_ok(cnow) \vee out_NSubQ_ok(cnow)) \wedge \\
& \quad (\exists n \in \mathbb{N}. cnow = n * ps.gst \wedge cnow \in [\tau_P, \tau]) \Rightarrow \text{Out}_y(cnow) = out_NSubP_y(\tau_P) \wedge \\
& \quad (\exists n \in \mathbb{N}. cnow = n * ps.gst \wedge cnow \in [\tau_Q, \tau]) \Rightarrow \text{Out}_z(cnow) = out_NSubQ_z(\tau_Q) \wedge \\
\Leftrightarrow & \llbracket P \parallel Q \rrbracket \quad (gst \rightarrow 0, EA \text{ and Induction Hypothesis})
\end{aligned}$$

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

7 Conclusion

In this paper, we presented a translator from HCSP formal models into Simulink graphical models. As illustrated by the case study, the translator enables

- HCSP formal models to be simulated and tested using a MATLAB platform, thus avoiding expensive formal verification, if not necessary;
- together with our previous work on encoding S/S diagrams into HCSP, a designer of embedded systems to flexibly shift between formal and informal models, according to a desired trade-off between efficiency and cost, and correctness and reliability.

The translation from HCSP into Simulink diagrams was implemented as a fully automatic tool, which has been integrated with the translation tool from S/S into HCSP. In addition, a UTP based semantical foundation was proposed to justify that the translations from HCSP into Simulink preserve 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*, 2013, http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf.
2. *Stateflow User’s Guide*, 2013, http://www.mathworks.com/help/pdf_doc/stateflow/sf_using.pdf.
3. M. Tiller, *Introduction to Physical Modeling with Modelica*, ser. The Springer International Series in Engineering and Computer Science. Springer-Verlag, 2001.
4. *SysML V 1.4 Beta Specification*, 2013, <http://www.omg.org/spec/SysML>.
5. B. Selic and S. Gerard, *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. The MK/OMG Press, 2013.
6. F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. L. Sangiovanni-Vincentelli, “Metropolis: An integrated electronic system design environment,” *IEEE Computer*, vol. 36, no. 4, pp. 45–52, 2003.
7. J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, “Taming heterogeneity - the ptolemy approach,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.
8. T. Henzinger, “The theory of hybrid automata,” in *LICS’96*, Jul. 1996, pp. 278–292.
9. R. Alur and T. Henzinger, “Modularity for timed and hybrid systems,” in *CONCUR’97*, ser. LNCS, 1997, vol. 1243, pp. 74–88.
10. J. He, “From CSP to hybrid systems,” in *A Classical Mind, Essays in Honour of C.A.R. Hoare*. Prentice Hall International (UK) Ltd., 1994, pp. 171–189.
11. C. Zhou, J. Wang, and A. P. Ravn, “A formal description of hybrid systems,” in *Hybrid systems, LNCS 1066*, 1996, pp. 511–530.
12. A. Platzer, “Differential-algebraic dynamic logic for differential-algebraic programs,” *J. Log. and Comput.*, vol. 20, no. 1, pp. 309–352, Feb. 2010.
13. J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou, “A calculus for hybrid CSP,” in *APLAS’10*, ser. LNCS, 2010, vol. 6461, pp. 1–15.

14. L. Zou, N. Zhan, S. Wang, M. Fränzle, and S. Qin, “Verifying Simulink diagrams via a hybrid hoare logic prover,” in *EMSOFT’13*, 2013, pp. 1–10.
15. L. Zou, N. Zhan, S. Wang, and M. Fränzle, “Formal verification of Simulink/Stateflow diagrams,” in *ATVA’15*, ser. LNCS, 2015, vol. 9346, pp. 464–481.
16. L. Zou, J. Lv, S. Wang, N. Zhan, T. Tang, L. Yuan, and Y. Liu, “Verifying Chinese train control system under a combined scenario by theorem proving,” in *VSTTE’13*, ser. LNCS, 2014, vol. 8164, pp. 262–280.
17. N. Zhan, S. Wang, and H. Zhao, “Formal modelling, analysis and verification of hybrid systems,” in *Unifying Theories of Programming and Formal Engineering Methods*, ser. LNCS, 2013, vol. 8050, pp. 207–281.
18. S. Wang, N. Zhan, and L. Zou, “An improved HHL prover: An interactive theorem prover for hybrid systems,” in *ICFEM’15*, ser. LNCS, vol. 9407, 2015, pp. 382–399.
19. *Simulink Design Verifier User’s Guide*, 2010, <http://www.manualslib.com/manual/392930/Matlab-Simulink-Design-Verifier-1.html#manual>.
20. Z. Han and P. J. Mosterman, “Towards sensitivity analysis of hybrid systems using simulink,” in *HSCC 2013*, 2013, pp. 95–100.
21. S. Tripakis, C. Sofronis, P. Caspi, and A. Curic, “Translating discrete-time Simulink to Lustre,” *ACM Trans. Embedded Comput. Syst.*, vol. 4, no. 4, pp. 779–818, 2005.
22. N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Marainchi, “Defining and translating a “safe” subset of Simulink/Stateflow into lustre,” in *EMSOFT’04*. ACM, 2004, pp. 259–268.
23. A. Cavalcanti, P. Clayton, and C. O’Halloran, “Control law diagrams in circus,” in *FM’05*, ser. LNCS, vol. 3582, 2005, pp. 253–268.
24. J. Woodcock and A. Cavalcanti, “The semantics of circus,” in *ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users*, ser. Lecture Notes in Computer Science, vol. 2272. Springer, 2002, pp. 184–203.
25. B. Meenakshi, A. Bhatnagar, and S. Roy, “Tool for translating Simulink models into input language of a model checker,” in *ICFEM’06*, ser. LNCS, vol. 4260, 2006, pp. 606–620.
26. V. Sfyrla, G. Tsiligiannis, I. Safaka, M. Bozga, and J. Sifakis, “Compositional translation of simulink models into synchronous BIP,” in *IEEE Fifth International Symposium on Industrial Embedded Systems, SIES 2010*. IEEE, 2010, pp. 217–220.
27. S. Bliudze and J. Sifakis, “The algebra of connectors - structuring interaction in BIP,” *IEEE Trans. Computers*, vol. 57, no. 10, pp. 1315–1330, 2008.
28. C. han Yang and V. Vyatkin, “Transformation of simulink models to IEC 61499 Function Blocks for verification of distributed control systems,” *Control Eng. Pract.*, vol. 20, no. 12, pp. 1259–1269, 2012.
29. C. Zhou and R. Kumar, “Semantic translation of simulink diagrams to input/output extended finite automata,” *Discrete Event Dynamic Systems*, vol. 22, no. 2, pp. 223–247, Jun. 2012.
30. S. Minpoli and G. Frehse, “SL2SX translator: from simulink to SpaceEx verification tool,” in *HSCC’16*, 2016.
31. C. Chen, J. S. Dong, and J. Sun, “A formal framework for modeling and validating Simulink diagrams,” *Formal Asp. Comput.*, vol. 21, no. 5, pp. 451–483, 2009.
32. P. Boström, “Contract-based verification of simulink models,” in *ICFEM 2011*, ser. Lecture Notes in Computer Science, vol. 6991. Springer, 2011, pp. 291–306.
33. P. Roy and N. Shankar, “Simcheck: a contract type system for simulink,” *ISSE*, vol. 7, no. 2, pp. 73–83, 2011.
34. V. Preoteasa and S. Tripakis, “Refinement calculus of reactive systems,” in *EMSOFT 2014*, 2014, pp. 2:1–2:10.
35. I. Dragomir, V. Preoteasa, and S. Tripakis, “Compositional semantics and analysis of hierarchical block diagrams,” in *SPIN 2016*, ser. Lecture Notes in Computer Science, vol. 9641, 2016, pp. 38–56.

36. S. Wang, N. Zhan, and D. Guelev, “An assume/guarantee based compositional calculus for hybrid CSP,” in *TAMC’12*, ser. LNCS, 2012, vol. 7287, pp. 72–83.
37. D. Guelev, S. Wang, and N. Zhan, “Hoare reasoning about HCSP in the duration calculus,” *Submitted*, 2013.
38. C. Hoare and J. He, *Unifying Theories of Programming*. Prentice Hall Englewood Cliffs, 1998, vol. 14.
39. H. Zhao, M. Yang, N. Zhan, B. Gu, L. Zou, and Y. Chen, “Formal verification of a descent guidance control program of a lunar lander,” in *FM’14*, ser. LNCS, 2014, vol. 8442, pp. 733–748.
40. E. A. M. Fränzle, and C. Herde, “SAT modulo ODE: A direct SAT approach to hybrid systems,” in *ATVA’08*, ser. LNCS, vol. 5311, 2008, pp. 171–185.
41. X. Chen, E. Ábrahám, and S. Sankaranarayanan, “Flow*: An analyzer for non-linear hybrid systems,” in *CAV’13*, ser. LNCS, vol. 8044, 2013, pp. 258–263.