# A denotational semantics of Simulink with higher-order UTP

Xiong Xu [a,c], Bohua Zhan [a,b], Shuling Wang [a], Jean-Pierre Talpin [c],
Naijun Zhan [a,b,*]

[a] *Institute of Software, Chinese Academy of Sciences, Beijing, China*
[b] *University of Chinese Academy of Sciences, Beijing, China*
[c] *Inria, Rennes, France*

A B S T R A C T

Matlab/Simulink is a de-facto industrial standard for modelling embedded systems. Reflecting the complexity of cyber-physical system (CPS) design, the semantics of Simulink is complex, mixing discrete and continuous time and events. In this paper, we define a compositional semantics of hierarchical Simulink diagrams using Higher-order Unifying Theories of Programming (HUTP) for CPS design. The HUTP theory satisfies the suitable algebraic properties to serve as a mathematical foundation for expressing the semantics of CPSs, in particular Simulink diagrams. We characterise a class of well-formed Simulink diagrams and prove the determinacy of their HUTP semantics. Moreover, we construct a framework for proving the consistency between Simulink diagrams and their translation to HCSP (Hybrid Communicating Sequential Processes). Finally, we provide a case study to illustrate and justify this translation.

## 1. Introduction

Cyber-Physical Systems (CPSs) are networked computing units controlling physical plants as diverse as grids, factories, supply chains, ground, sea, air and space transportation systems. CPSs are complex to design, verify and maintain, while often entrusted safety-critical roles. The efficient and verified development of safe and reliable CPSs is hence a priority mandated by many standards, yet a notoriously difficult and challenging field of engineering and research. Matlab/Simulink is a de-facto industrial standard for modelling cyber-physical systems. Reflecting the complexity of CPS design, Simulink is known to have a complex semantics, which need to describe interactions between discrete and continuous time behaviours, trigger events, hierarchical structure, and so on.

Model-based design (MBD) [11] has long become a predominant approach to break down the difficulties and challenges in CPS design into abstracted and comprehensible elements. Hoare and He's Unifying Theories of Programming (UTP) [12] is built upon the mathematical foundations of theorem proving and has both the core simplicity and the necessary extensibility to capture models of imperative and concurrent software, hardware, and physics found in CPS design under a common relational calculus suitable for design and verification.

Hybrid systems, which could be subsumed in the domain of CPSs, seamlessly integrate discrete behaviour with continuous dynamical systems, and have been extended to capture probabilistic, stochastic, time-delayed behaviours and even more complex features. In previous works [23], we defined one such conservative extension to Hoare and He's UTP theory with

higher-order quantification, i.e., the Higher-order UTP (HUTP), to provide a formal semantics for modelling and verifying hybrid systems, mixing discrete real-time processes and continuous dynamics. Within HUTP, we defined a calculus of *normal hybrid designs* to model and analyse hybrid systems. A normal hybrid design describes a contract between the component and its environment, and therefore supports the decomposition of engineering tasks to resolve system design complexity. Normal hybrid design as a first-class notion in the HUTP theory enjoys some desired algebraic properties, and therefore can serve as a semantic foundation for CPS design.

In [29,28], we introduced methods for translation of Simulink and Stateflow diagrams to Hybrid Communicating Sequential Processes (HCSP), in order to verify them using the Hybrid Hoare Logic prover implemented in Isabelle/HOL [27,22]. The correctness of the translation can be proved using HUTP. Concretely, we define the respective HUTP semantics for Simulink and HCSP, and then compare the HUTP representations of Simulink diagrams and their HCSP models to check the semantic consistency. In [23], we defined a formal semantics for Simulink based on normal hybrid designs. However, the normal-hybrid-design semantics is complex, which pose difficulties for ensuing analysis and verification. The complexity comes from (1) involvement of a large number of communications (including the communications between atomic discrete blocks), and (2) the use of normal hybrid design, although intuitive for system design, makes the definitions long and cumbersome. Moreover, compositional semantics for hierarchical Simulink subsystems is not considered.

Therefore, we introduce in this paper a new compositional formalisation of denotational semantics for hierarchical Simulink diagrams based on HUTP, featuring both discrete and continuous behaviours, as well as composition using normal, enabled and triggered subsystems. The expressivity of the present denotational semantics is well-suited for verifying correctness of translation from Simulink to other formalisms, such as HCSP [29], differential dynamic logic [14] and hybrid automata [1]. We exercise this capability by constructing a framework for proving the semantic consistency between Simulink diagrams and their corresponding HCSP models, and provide a case study to demonstrate and justify our translation of Simulink into HCSP.

In summary, the main contributions of this paper comprise:

- A notion of Simulink processes and their parallel composition based on conjunction of relations, which simplifies the HUTP theory for Simulink;
- A denotational semantics for hierarchical Simulink diagrams based on Simulink processes, reflecting the composability of subsystems and therefore following the principle of modular design;
- Notions of well-formedness of Simulink diagrams, and proof of semantic determinacy for these diagrams;
- A framework for proving correctness of translation from Simulink to HCSP, which is illustrated with a simple case study.

*Paper organisation.* The rest of the paper is organised as follows. Section 2 retrospects some preliminary concepts of Simulink, UTP and Higher-order UTP. Section 3 defines the notion of Simulink processes which serve as the semantic foundation for Simulink. Starting from Simulink blocks, Section 4 defines the HUTP semantics for Simulink diagrams by Simulink processes, and proves determinacy of the semantics for well-formed diagrams. Section 5 defines the compositional HUTP semantics for hierarchical Simulink diagrams containing normal, triggered and enabled subsystems. In Section 6, we illustrate by a case study how to prove the semantic consistency between Simulink diagrams and the corresponding HCSP models. Section 7 addresses the related work and Section 8 concludes this paper and discusses future work.

## 2. Preliminaries

In this section, we will present the preliminaries on Simulink, classical UTP, and our previous work on the higher-order UTP for hybrid systems.

### 2.1. Simulink

Simulink [16] is a widely-used design environment for building embedded control systems, with support for graphical modelling and efficient numerical simulation. Dynamic systems, possibly combining continuous and discrete behaviours, can be modelled by Simulink block diagrams. A rich set of fixed-step and variable-step solvers is provided for simulating dynamic systems. Fig. 1 shows how a simple plant-control system can be modelled in Simulink.

Blocks are the basic units for building Simulink models. Each block is defined with input and output ports, an output method that defines how the output values are calculated, optional internal states and corresponding update methods that define how the states are changed. It may also contain user-defined parameters that alter the functionality, such as the symbol parameter "+-" for Add block, resulting in Subtract; the parameter of threshold for Switch block, and so on. Sample time is one of the most important parameters of a block and specifies the rate of execution when the block executes the output method and the update method (if it exists). Among the different types of sample time, three basic ones are frequently used: discrete with sample time st for some st > 0, continuous with sample time 0, and inherited. For the inherited case, the sample time is not defined explicitly, but instead determined from the context of the corresponding block through a process called sample time propagation. For instance, if the sample times of all the input signals of an inherited block are known, then sample time of the block is computed as the greatest common divisor of the sample times
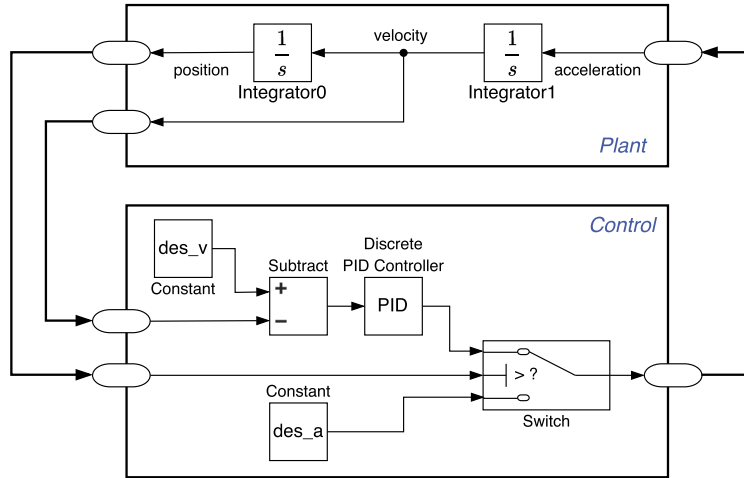
**Fig. 1.** A Simulink diagram of a plant-control model.

of these input signals. According to sample time, blocks can be categorised into two kinds: discrete and continuous blocks. Simulink provides discrete and continuous solvers to compute the states of blocks at each time step respectively.

Blocks are connected using lines to transfer signals from one block to another. The signals are time-varying and can be considered as functions mapping from real time to values. For discrete blocks the functions are piecewise constant. Blocks in a diagram may have different sample times, e.g. a multi-rate discrete system with discrete blocks that sample at different rates, or a hybrid continuous and discrete system. For such diagrams, the simulator must meet the precision specified on the continuous states, and hit all the sample times for the discrete states. The simulator needs to sort (or schedule) the blocks to be executed in a certain order. This may not be possible if there are *algebraic loops* in the diagram, in which case the diagram may be considered to be invalid. The blocks which maintain state variables such as the Integrator or Unit Delay blocks can break the loop.

Blocks can be grouped into subsystems to establish a hierarchical structure on Simulink diagrams. We consider three types of subsystems: normal subsystems, triggered subsystems and enabled subsystems. A normal subsystem executes as a single unit within the model. It can specify its system sample time and its execution is equivalent to executing the blocks inside the subsystem. Both triggered and enabled subsystems are conditionally executed subsystems. A triggered subsystem is defined with inherited sample time, that runs when the trigger signal is rising, falling, or either (rising or falling) through a zero value. An enabled subsystem runs when its control signal is positive.

A hierarchical Simulink model is thus composed of blocks, subsystems, and lines between them. After a Simulink model is built, it is ready for simulation. Each step of simulation corresponds to one sample time of the overall diagram. At each step, first compute the internal state and output of each block by invoking the corresponding output and update methods in the correct order; second, choose appropriate ODE solvers to compute evolution of continuous blocks through time. If there are triggered subsystems or integrator blocks with resets, zero crossings may need to be computed. The process ends when the given simulation time is reached.

### 2.2. Unifying theories of programming

Hoare and He's Unifying Theories of Programming (UTP) [12] is an alphabetised refinement calculus unifying heterogeneous programming paradigms. An alphabetised relation consists of an alphabet $\alpha(P)$, containing its variables $x$ and primes $x'$, and a relational predicate $P$ referring to this vocabulary. The terms $x$ and $x'$ are called observable variables: $x$ is observable at the start of execution and $x'$ is observable at the end of execution. The behaviour of a program is encoded as a relation between the observable variables $x$ and $x'$. In particular, assignment, sequential composition, conditional statement, non-deterministic choice, and recursion of imperative programs can be specified as alphabetised relations below, where $\boldsymbol{x}$ and $\boldsymbol{x}'$ are sequences or vectors of variables, $\boldsymbol{x}\backslash\{x\}$ ($\boldsymbol{x}'\backslash\{x'\}$) denotes excluding $x$ ($x'$) from $\boldsymbol{x}$ ($\boldsymbol{x}'$). To start with, the relational calculus comprises all operators of first-order logic.

$$x := e \mathrel{\widehat{=}} x' = e \land \boldsymbol{x}'\backslash\{x'\} = \boldsymbol{x}\backslash\{x\}$$

$$P \mathbin{;} Q \mathrel{\widehat{=}} \exists \boldsymbol{x}_* \cdot P[\boldsymbol{x}_*/\boldsymbol{x}'] \land Q[\boldsymbol{x}_*/\boldsymbol{x}]$$

$$P \triangleleft b \triangleright Q \mathrel{\widehat{=}} (b \land P) \lor (\neg b \land Q)$$

$$P \sqcap Q \mathrel{\widehat{=}} P \lor Q$$

$$P \sqcup Q \mathrel{\widehat{=}} P \land Q$$

Conventionally, $\sqcap$ is an algebraic sibling for $\wedge$ and $\sqcup$ for $\vee$. In the equal tradition of UTP [12,23], however, they denote $\vee$ and $\wedge$, respectively. We will follow UTP's convention in this paper.

Assignment $x := e$ is defined by observing the update $x'$ of variable $x$ once its value $e$ is evaluated, leaving other variables in the alphabet $\boldsymbol{x}$ unchanged. Sequence $P \, \fatsemi \, Q$ is modelled by locally binding, through $\boldsymbol{x}_*$, the final state $\boldsymbol{x}'$ of $P$ and the initial state $\boldsymbol{x}$ of $Q$, both of which are instantiated to $\boldsymbol{x}_*$. Note that $\fatsemi$ requires $\alpha_{out}(P) = \alpha'_{in}(Q)$, where $\alpha_{out}(P)$ and $\alpha_{in}(Q)$ denote the sets of output and input variables in $\alpha(P)$ and $\alpha(Q)$, respectively, and $\alpha'_{in}(Q)$ is the primed version by priming all the variables in $\alpha_{in}(Q)$. The conditional $P \lhd b \rhd Q$ evaluates as $P$ if $b$ is true and as $Q$ otherwise. $P \sqcap Q$ non-deterministically chooses $P$ or $Q$, and $P \sqcup Q$ is a conjunction of $P$ and $Q$.

Let $P$ and $Q$ be two predicates with the same alphabet, say $\{\boldsymbol{x}, \boldsymbol{x}'\}$. Then, $Q$ is a *refinement* of $P$, denoted $P \sqsubseteq Q$, if $\forall \boldsymbol{x}, \boldsymbol{x}' \cdot Q \Rightarrow P$. In addition, $P \sqsubseteq Q$ iff $P \sqcap Q = P$ iff $P \sqcup Q = Q$. With respect to the refinement order $\sqsubseteq$, the least $(\mu)$ and greatest $(\nu)$ fixed points of a function $F$ between programs can be defined as follows:

$$\mu F \ \widehat{=} \ \bigsqcap \{X \mid F(X) \sqsubseteq X\}$$
$$\nu F \ \widehat{=} \ \bigsqcup \{X \mid X \sqsubseteq F(X)\}$$

The notion of healthiness conditions plays an important role in the UTP theory. If a predicate satisfies $P = \mathcal{H}(P)$, then it is said to be $\mathcal{H}$-healthy. In other words, a healthiness condition $\mathcal{H}$ defines an invariant predicate set $\{X \mid \mathcal{H}(X) = X\}$, and is required to be idempotent ($\mathcal{H} \circ \mathcal{H} = \mathcal{H}$), which means that taking the medicine twice leaves you as healthy as taking it once (no overdoses). So, in UTP, the healthy predicates of a theory are the fixed points of idempotent functions. When $\mathcal{H}$ is monotonic on a complete lattice $(\mathbb{C}, \sqsubseteq)$, then according to the Knaster-Tarski theorem [20], the UTP theory satisfying $\mathcal{H}$ forms a complete lattice $\{X \in \mathbb{C} \mid \mathcal{H}(X) = X\}$. Additionally, recursion can be well defined. Distinct healthiness conditions can be composed to capture the characteristics of different programming paradigms. Concretely, a programming paradigm can be defined by a collection of healthiness conditions $\mathcal{H}_1, \mathcal{H}_2, \cdots, \mathcal{H}_n$. Their composition $\mathcal{H}_1 \circ \mathcal{H}_2 \circ \cdots \circ \mathcal{H}_n$ forms the semantic model of the domain-specific paradigm under consideration. For example, in Section 3.1, we introduce healthiness conditions characterising Simulink processes.

### 2.3. The higher-order UTP for hybrid systems

Higher-order UTP (HUTP) [23] is a conservative extension to Hoare and He's UTP theory which supports the specification of discrete, real-time and continuous dynamics, concurrency and communication, and higher-order quantification. In [23], we defined a formal semantics for Simulink based on a notion of normal hybrid designs. However, this semantics is complex and difficult to analyse for reasons given in Section 1. In this paper, we instead consider an abstracted HUTP semantics for Simulink, based on the notion of *abstract hybrid processes* proposed in [23] as future work. While having weaker algebraic structure than normal hybrid designs (e.g., chaos is not a left zero of sequential composition), abstract hybrid processes are simpler, of sufficient expressivity to define a semantics of Simulink, and are more comfortable for verification.

#### 2.3.1. Abstract hybrid processes

As mentioned in [23], HUTP separates the concerns in hybrid system design into time, state and trace. We introduce the notion of time by two observational variables $ti, ti' : \mathbb{R}_{\geq 0} \cup \{+\infty\}$ to specify the start- and end-time of the observed behaviour. The notion of state is represented by real-time variables and their derivatives, which are functions over time, and differential relations over them that are very powerful to express all kinds of continuous dynamics. Therefore, there are three versions for each state variable $v$:

- $v \in \mathbb{D}$ stands for its initial value in the domain $\mathbb{D}$, where $\mathbb{D}$ could be a Banach space;
- the primed version $v' \in \mathbb{D}$ stands for the final value, i.e., the output state variable; and
- the real time version $\underset{\sim}{v} : [ti, ti') \to \mathbb{D}$ stands for its dynamic trajectory from the start time $ti$ to the end time $ti'$, and $\dot{\underset{\sim}{v}} : (ti, ti') \rightharpoonup \mathbb{D}$ is a partial function denoting the derivative of $\underset{\sim}{v}$.

Timed traces $tr$ and $tr'$ record the execution history and capture communication behaviours, where $tr$ represents the timed trace before the process is started and $tr'$ stands for timed trace up to the moment of observation. However, in this paper, no communication is involved and the parallel composition is based on shared variables, so timed traces are abstracted away, which is the main feature of abstract hybrid processes.

We use the boldface symbols $\boldsymbol{v}$, $\boldsymbol{v}'$, $\underset{\sim}{\boldsymbol{v}}$ and $\dot{\underset{\sim}{\boldsymbol{v}}}$ to denote respective vectors of input, output, real-time state variables and their derivatives. The alphabet our theory depends on is $\{ti, ti', \boldsymbol{v}, \underset{\sim}{\boldsymbol{v}}, \dot{\underset{\sim}{\boldsymbol{v}}}, \boldsymbol{v}'\}$ by default. Therefore, first-order predicate $P(\boldsymbol{x}, \boldsymbol{x}')$ used in classical UTP [12] can be extended to higher-order differential relation $\underset{\sim}{P}(ti, ti', \boldsymbol{v}, \underset{\sim}{\boldsymbol{v}}, \dot{\underset{\sim}{\boldsymbol{v}}}, \boldsymbol{v}')$. However, not all higher-order differential relations are expected, such as $ti > ti'$ indicating time going backwards. Thus, we use *healthiness conditions* to exclude the ill behaviours. As introduced in [23], the features of abstract hybrid processes can be captured by the following four healthiness conditions ($\mathcal{H}_1$ is defined for traces, hence not applicable for abstract hybrid processes):

- Time must be irreversible:

$$\mathcal{H}_0^{\mathrm{A}}(X) \ = \ X \wedge ti \leq ti'$$

- If the preceding process does not terminate, i.e., $ti = +\infty$, the current process should do nothing but keep the time observation unchanged, i.e.,

$$\mathcal{H}_2^{\mathrm{A}}(X) = (ti = ti') \lhd ti = +\infty \rhd X$$

  where $P \lhd b \rhd Q \ \widehat{=} \ (b \wedge P) \vee (\neg b \wedge Q)$.

- If the current process does not terminate, i.e., $ti' = +\infty$, the values of the output state variables are unobservable, i.e.,

$$\mathcal{H}_3^{\mathrm{A}}(X) = (\exists \boldsymbol{v}' \cdot X) \lhd ti' = +\infty \rhd X$$

- If the process evolves for a period of time, i.e., $ti < ti'$, the real-time value $\underline{\boldsymbol{v}}$ should stay right-continuous (RC) and semi-differentiable (SD). Let $\boldsymbol{v}_k$, $\underline{\boldsymbol{v}}_k$, and $\boldsymbol{v}_k'$ denote the $k$-th variable in $\boldsymbol{v}$, $\underline{\boldsymbol{v}}_k$, and $\boldsymbol{v}_k'$, respectively. Then, we define

$$RC \,\widehat{=}\, \forall k \cdot \forall t \in [ti, ti') \cdot \exists d \cdot \underline{\boldsymbol{v}}_k(t) = \lim_{\delta \to 0^+} \underline{\boldsymbol{v}}_k(t + \delta) = d$$
$$SD \,\widehat{=}\, \forall k \cdot \forall t \in (ti, ti') \cdot \exists d_0 \cdot \lim_{\delta \to 0^+} \left( \underline{\boldsymbol{v}}_k(t + \delta) - \underline{\boldsymbol{v}}_k(t) \right) / \delta = d_0$$
$$\wedge \, \exists d_1 \cdot \lim_{\delta \to 0^-} \left( \underline{\boldsymbol{v}}_k(t + \delta) - \underline{\boldsymbol{v}}_k(t) \right) / \delta = d_1$$

  The healthiness condition

$$\mathcal{H}_4(X) = X \wedge RC \wedge SD$$

  rules out some ill behaviours, such as the Dirichlet function (returning 1 if $t$ is a rational number and 0 otherwise) and the Weierstrass function (continuous everywhere but differentiable nowhere).

**Remark 1.** Note that $\mathcal{H}_3^{\mathrm{A}}$ does not mean that the values of $\boldsymbol{v}$ exist at infinity. The existential quantifier just indicates that the output $\boldsymbol{v}'$ can take arbitrary values, i.e., chaos. In addition, the output of a process exhibiting Zeno-behaviour should also be unobservable (chaos). However, it cannot be captured by abstract hybrid processes as the trace information is abstracted away.

An abstract hybrid process is a fixed point of $X = \mathcal{H}_{\mathrm{HP}}^{\mathrm{A}}(X)$, where

$$\mathcal{H}_{\mathrm{HP}}^{\mathrm{A}} \,\widehat{=}\, \mathcal{H}_0^{\mathrm{A}} \circ \mathcal{H}_2^{\mathrm{A}} \circ \mathcal{H}_3^{\mathrm{A}} \circ \mathcal{H}_4$$

It is proved in [23] that $\mathcal{H}_{\mathrm{HP}}^{\mathrm{A}}$ is idempotent and monotonic, which indicates that abstract hybrid processes form a complete lattice under the refinement order $\sqsubseteq$.

## 3. Simulink processes in HUTP

Based on abstract hybrid processes, we propose a new notion of Simulink processes which can serve as the semantic foundation for Simulink. We further define parallel composition of Simulink processes as conjunction of relations. Finally, we define some syntactic sugar to simplify the ensuing presentations.

### 3.1. Simulink processes

The semantics of Simulink can be represented by a subset of abstract hybrid processes subject to additional healthiness conditions. First, we assume that the execution of Simulink diagrams will consume time ($ti < ti'$). This corresponds to the requirement that simulation will last for non-zero amount of time. Moreover, we require that simulations will always terminate ($ti' < +\infty$). These two properties can be captured by the following healthiness condition:

$$\mathcal{H}_{\mathrm{SIM}}(X) = X \wedge ti < ti' < +\infty$$

It can be proved that $\mathcal{H}_{\mathrm{SIM}}$ is idempotent and monotonic, which indicates that

$$\mathcal{H}_{\mathrm{SIM}}^{\mathrm{A}} \,\widehat{=}\, \mathcal{H}_{\mathrm{SIM}} \circ \mathcal{H}_{\mathrm{HP}}^{\mathrm{A}}$$

also forms a complete lattice under the refinement order. We call the $\mathcal{H}_{\mathrm{SIM}}^{\mathrm{A}}$-healthy relations *Simulink processes*, and we prove the following property, which reveals that $\mathcal{H}_0^{\mathrm{A}}$, $\mathcal{H}_2^{\mathrm{A}}$ and $\mathcal{H}_3^{\mathrm{A}}$ are redundant and therefore simplifies the representation of Simulink processes.

**Property 2.** $\mathcal{H}_{\mathrm{SIM}}^{\mathrm{A}} \equiv \mathcal{H}_{\mathrm{SIM}} \circ \mathcal{H}_4$.

**Proof.** It can be checked that $\mathcal{H}_{\mathrm{SIM}} \circ \mathcal{H}_0^{\mathrm{A}}(X) = \mathcal{H}_{\mathrm{SIM}} \circ \mathcal{H}_2^{\mathrm{A}}(X) = \mathcal{H}_{\mathrm{SIM}} \circ \mathcal{H}_3^{\mathrm{A}}(X) = \mathcal{H}_{\mathrm{SIM}}(X)$.  □

We next describe the meet ($\sqcap$, $\lor$), join ($\sqcup$, $\land$) and sequential composition ($\mathbin{;}$) operations on Simulink processes. They are specialisations of corresponding operations for general hybrid processes defined in [23]. The sequential composition of two Simulink processes P and Q is defined as follows:

$$P \mathbin{;} Q \mathrel{\widehat{=}} \exists ti_0, \boldsymbol{v}_0 \cdot P[ti_0, \boldsymbol{v}_0/ti', \boldsymbol{v}'] \land Q[ti_0, \boldsymbol{v}_0/ti, \boldsymbol{v}]$$

provided that $\alpha_{out}(P) = \alpha'_{in}(Q)$, where $\alpha_{out}(P)$ and $\alpha_{in}(Q)$ denote the sets of output and input variables in the respective alphabets of P and Q, and $\alpha'_{in}(Q)$ is the primed version by priming all the variables in $\alpha_{in}(Q)$. If $\alpha_{out}(P) \neq \alpha'_{in}(Q)$, then we can extend the alphabets by

$$\alpha_{out}^{+}(P) = \alpha_{in}^{+\prime}(Q) \mathrel{\widehat{=}} \alpha_{out}(P) \cup \alpha'_{in}(Q)$$

to ensure the well-definedness of $\mathbin{;}$. The meet and join operations simply correspond to union and intersection of relations. We then prove $\mathbin{;}$, $\lor$ and $\land$ are $\mathcal{H}_{\text{SIM}}^{\text{A}}$-preserving, and the proofs for other operations on Simulink processes are similar.

**Property 3.** *If* P *and* Q *are* $\mathcal{H}_{\text{SIM}}^{\text{A}}$*-healthy, so are* $P \mathbin{;} Q$, $P \lor Q$, *and* $P \land Q$.

**Proof.** By the definition of $\mathcal{H}_{\text{SIM}}^{\text{A}}$,

$$\begin{aligned}
P \mathbin{;} Q &= 0 < ti < ti' < +\infty \land P \land RC(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \land SD(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \mathbin{;} \\
&\quad 0 < ti < ti' < +\infty \land Q \land RC(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \land SD(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \\
&= \exists ti_0, \boldsymbol{v}_0 \cdot 0 < ti < ti_0 < +\infty \land 0 < ti_0 < ti' < +\infty \\
&\quad \land P[ti_0, \boldsymbol{v}_0/ti', \boldsymbol{v}'] \land Q[ti_0, \boldsymbol{v}_0/ti, \boldsymbol{v}] \\
&\quad \land RC(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti_0) \land RC(\underaccent{\tilde}{\boldsymbol{v}}, ti_0, ti') \land SD(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti_0) \land SD(\underaccent{\tilde}{\boldsymbol{v}}, ti_0, ti') \\
&= \exists ti_0, \boldsymbol{v}_0 \cdot 0 < ti < ti_0 < ti' < +\infty \\
&\quad \land P[ti_0, \boldsymbol{v}_0/ti', \boldsymbol{v}'] \land Q[ti_0, \boldsymbol{v}_0/ti, \boldsymbol{v}] \\
&\quad \land RC(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \land SD(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti')
\end{aligned}$$

where $RC$ and $SD$ denote $\underaccent{\tilde}{\boldsymbol{v}}$ is right continuous and semi-differentiable as specified in healthiness condition $\mathcal{H}_4$. We can also prove

$$\begin{aligned}
P \lor Q &= 0 < ti < ti' < +\infty \land P \land RC(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \land SD(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \lor \\
&\quad 0 < ti < ti' < +\infty \land Q \land RC(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \land SD(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \\
&= 0 < ti < ti' < +\infty \land (P \lor Q) \land RC(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \land SD(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \\
P \land Q &= 0 < ti < ti' < +\infty \land P \land RC(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \land SD(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \land \\
&\quad 0 < ti < ti' < +\infty \land Q \land RC(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \land SD(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \\
&= 0 < ti < ti' < +\infty \land (P \land Q) \land RC(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti') \land SD(\underaccent{\tilde}{\boldsymbol{v}}, ti, ti')
\end{aligned}$$

According to the above results, we can prove that $P \mathbin{;} Q$, $P \lor Q$ and $P \land Q$ are $\mathcal{H}_{\text{SIM}}^{\text{A}}$-healthy.  $\square$

### 3.2. Parallel composition

Of all the operations, parallel composition is the most important. In [23], we assume that the state variables of different processes are disjoint. Based on this assumption, a parallel-by-merge scheme is given. In this paper, we relax this assumption: state variables can be shared among processes. Intuitively, the combination is well-behaved because although variables are shared, the value of each variable is controlled by at most one process and only read by others. Hence, under some additional assumptions, we can prove that there exists unique assignment to all variables given the values of input variables to the overall process.

Therefore, the parallel-by-merge scheme (Fig. 1 of [23]) can be revisited to represent the parallel composition by shared state variables in this paper. The parallel-by-merge, originated from [12], is a typical scheme to define parallel composition in UTP [23,10]. Intuitively, parallel processes first execute independently and their respective outputs are fed into the merge predicate $M$. Then, $M$ produces the merged result as the output of the parallel composition. Each merge predicate reflects a parallel scheme, therefore the parallel composition is parametric over $M$, which is indicated by the notation $\parallel_M$. Concretely, let P and Q be the parallel processes with respective state variables $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$ (which are not necessarily disjoint), then

$$P \parallel_M Q \mathrel{\widehat{=}} \mathcal{H}_{\text{SIM}}^{\text{A}}((P_X \land Q_Y) \mathbin{;} M)$$

where $P_X$ ($Q_Y$) makes an $X$ ($Y$)-version of P (Q) by adding the time variable $ti'$ in P (Q) with the $X$ ($Y$)-subscript, i.e.,

$$P_X \mathrel{\widehat{=}} P \mathbin{\mathring{\,}} (ti = ti'_X \wedge \boldsymbol{v}_0 = \boldsymbol{v}'_0) = P[ti'_X/ti']$$
$$Q_Y \mathrel{\widehat{=}} Q \mathbin{\mathring{\,}} (ti = ti'_Y \wedge \boldsymbol{v}_1 = \boldsymbol{v}'_1) = Q[ti'_Y/ti']$$

**Remark 4.** Note that the $\mathcal{H}^{A}_{SIM}$-healthiness of parallel composition $P\|_M Q$ is enforced. Otherwise, $\mathcal{H}^{A}_{SIM}$-healthiness could be violated, because the merge predicate $M$ can be arbitrary. We could investigate well-defined merge predicates that guarantee $\mathcal{H}^{A}_{SIM}$-healthiness by definition (just as the merge predicate SIM, which does), but it is not the concern in this paper.

For Simulink, we define a new merge predicate:

$$SIM \mathrel{\widehat{=}} ti_X = ti_Y = ti' \wedge \boldsymbol{v}'_0 = \boldsymbol{v}_0 \wedge \boldsymbol{v}'_1 = \boldsymbol{v}_1$$

It states that the parallel processes are synchronous on time ($ti_X = ti_Y$), i.e., their termination time should be identical ($+\infty$ for non-termination); and the output values of the shared state variables $\boldsymbol{v}_0 \cap \boldsymbol{v}_1$ should keep consistent. We denote the parallel operator defined by SIM as $\|_{SIM}$. For brevity, in the remainder, we write $\|$ for $\|_{SIM}$ unless otherwise stated. The following property states that $\|$ is equivalent to conjunction.

**Property 5.** $P\|Q \equiv P \wedge Q$ *if* P *and* Q *are Simulink processes.*

**Proof.** According to the definition, $P\|Q = \mathcal{H}^{A}_{SIM}((P_X \wedge Q_Y) \mathbin{\mathring{\,}} SIM)$, where

$$
\begin{aligned}
(P_X \wedge Q_Y) \mathbin{\mathring{\,}} SIM &= (P[ti'_X/ti'] \wedge Q[ti'_Y/ti']) \mathbin{\mathring{\,}} \\
&\quad (ti_X = ti_Y = ti' \wedge \boldsymbol{v}'_0 = \boldsymbol{v}_0 \wedge \boldsymbol{v}'_1 = \boldsymbol{v}_1) \\
&= \exists ti^{\star}_X, ti^{\star}_Y, \boldsymbol{v}^{\star}_0, \boldsymbol{v}^{\star}_1 \cdot P[ti^{\star}_X, \boldsymbol{v}^{\star}_0/ti', \boldsymbol{v}'_0] \wedge Q[ti^{\star}_Y, \boldsymbol{v}^{\star}_1/ti', \boldsymbol{v}'_1] \\
&\quad \wedge (ti^{\star}_X = ti^{\star}_Y = ti' \wedge \boldsymbol{v}'_0 = \boldsymbol{v}^{\star}_0 \wedge \boldsymbol{v}'_1 = \boldsymbol{v}^{\star}_1) \\
&= P \wedge Q
\end{aligned}
$$

Since P and Q are $\mathcal{H}^{A}_{SIM}$-healthy, $P \wedge Q$ is also $\mathcal{H}^{A}_{SIM}$-healthy (Property 3). Then, we can get

$$P\|Q = \mathcal{H}^{A}_{SIM}((P_X \wedge Q_Y) \mathbin{\mathring{\,}} SIM) = \mathcal{H}^{A}_{SIM}(P \wedge Q) = P \wedge Q$$

The property is proved.  □

Although parallel composition is equivalent to conjunction in essence, we distinguish the two concepts in this paper. Concretely, parallel composition between blocks in a Simulink diagram or within a subsystem is called *conjunction*; while parallel composition between subsystems is called *parallel composition*. Consider the Simulink diagram in Fig. 1, where each block can be translated to a Simulink process. The semantics of subsystem Plant can be defined by the conjunction Integrator0 $\wedge$ Integrator1, while the semantics of the whole diagram can be defined by the parallel composition Plant$\|$Control, which is logically equivalent to Plant $\wedge$ Control.

### 3.3. Syntactic sugar

For brevity in the ensuing presentation, we introduce some syntactic sugar for the HUTP representation of Simulink semantics. Notice that the following notations are different from the definitions in [23]. Let $\underline{P}$ denote a predicate relating $\underline{\boldsymbol{v}}$ and $\dot{\underline{\boldsymbol{v}}}$, then

$$\lceil \underline{P} \rfloor \mathrel{\widehat{=}} \mathcal{H}^{A}_{SIM}\left(\forall t \in (ti, ti') \cdot \underline{P}(\underline{\boldsymbol{v}}(t), \dot{\underline{\boldsymbol{v}}}(t))\right)$$

is a continuous process reflecting the flow of $\underline{\boldsymbol{v}}$ over the time interval $(ti, ti')$ for $ti < ti'$, and it states that $\underline{P}$ holds at every instant $t$ from $ti$ to $ti'$. Note that although the input and output state variables $\boldsymbol{v}$ and $\boldsymbol{v}'$ do not appear in $\lceil \underline{P} \rfloor$, they are in the alphabet of $\lceil \underline{P} \rfloor$, or in other words, $\boldsymbol{v}$ and $\boldsymbol{v}'$ can take arbitrary values. We can also bind $\boldsymbol{v}$ and $\boldsymbol{v}'$ to the initial and final values of $\underline{\boldsymbol{v}}$, respectively, resulting in the following definitions:

$$\llbracket \underline{P} \rfloor \mathrel{\widehat{=}} \boldsymbol{v} = \underline{\boldsymbol{v}}(ti) \wedge \lceil \underline{P} \rfloor$$
$$\lceil \underline{P} \rrbracket \mathrel{\widehat{=}} \lceil \underline{P} \rfloor \wedge \boldsymbol{v}' = \underline{\boldsymbol{v}}(ti'^{-})$$
$$\llbracket \underline{P} \rrbracket \mathrel{\widehat{=}} \boldsymbol{v} = \underline{\boldsymbol{v}}(ti) \wedge \lceil \underline{P} \rfloor \wedge \boldsymbol{v}' = \underline{\boldsymbol{v}}(ti'^{-})$$

Especially, we define

$$\mathsf{Idle} \mathrel{\hat{=}} [\![\dot{\underset{\sim}{\boldsymbol{v}}} = \boldsymbol{0}]\!]$$

Besides, we add subscripts to the above definitions to constrain the duration. For example,

$$[\![\underset{\sim}{P}]\!]_d \mathrel{\hat{=}} [\![\underset{\sim}{P}]\!] \wedge ti' - ti = d$$
$$[\![\underset{\sim}{P}]\!]_{\leq d} \mathrel{\hat{=}} [\![\underset{\sim}{P}]\!] \wedge ti' - ti \leq d$$

Note that the above continuous processes are all Simulink processes as they are $\mathcal{H}_{\mathrm{SIM}}^{\mathrm{A}}$-healthy.

A causal sequence of operations or events which is assumed to take no time is called super-dense computation [15]. Under super-dense computation, the time to compute the discrete operations is negligible. However, the causal order of computations is still significant. Under the assumption of super-dense computation, a discrete process is defined by

$$[P] \mathrel{\hat{=}} ti = ti' < +\infty \wedge P$$

where $P$ denotes a predicate relating $\boldsymbol{v}$ and $\boldsymbol{v}'$. It executes instantly at time $ti = ti'$, rather than continuously over a time interval. Note that $[P]$ is not a Simulink process as its duration is 0. This would violate the healthiness condition $\mathcal{H}_{\mathrm{SIM}}$. However, the sequential composition of $[P]$ and a Simulink process is usually $\mathcal{H}_{\mathrm{SIM}}^{\mathrm{A}}$-healthy, as demonstrated in the later content. We define

$$\mathsf{Skip} \mathrel{\hat{=}} ti = ti' < +\infty \wedge \boldsymbol{v} = \boldsymbol{v}'$$

Similar to Property 16 in [23], it can be proved that $(\mathsf{Skip} \mathbin{\fatsemi} \mathsf{P}) = (\mathsf{P} \mathbin{\fatsemi} \mathsf{Skip}) = \mathsf{P}$ for any Simulink process P.

Since Simulink processes form a complete lattice according to the discussion at the end of Section 2.3.1, recursion can be defined. Theoretically, recursion is denoted by the fixed points of the equation $X = F(X)$, where $F$ constructs the body of the recursion. If $F$ is monotonic, the fixed points of $X = F(X)$ also form a complete lattice by the Knaster-Tarski theorem [20]. The least fixed point is denoted by $\mu X.F(X)$, based on which we can define

$$\mathsf{P}^* \mathrel{\hat{=}} \mu X.(\mathsf{Skip} \quad \vee \quad \mathsf{P} \mathbin{\fatsemi} X)$$

where P is a Simulink process.

## 4. Semantics for Simulink blocks

In this section, we give the HUTP semantics of Simulink blocks in terms of Simulink processes. A (non-hierarchical) Simulink diagram consists of blocks graphically connected by directed lines. Each such connection is the output signal of a unique block. We represent a signal by a variable $x$ defined as a real-valued function of time $x \in F \mathrel{\hat{=}} \mathbb{R}_{\geq 0} \to \mathbb{R}$. A Simulink block can be represented by the tuple $(\mathcal{I}, \mathcal{O}, \mathcal{S}, \mathcal{R})$, where $\mathcal{I}$ is the set of input variables, $\mathcal{O}$ is the set of output variables, $\mathcal{S}$ is the set of internal state variables, and $\mathcal{R}$ is a relation between the signals $F^{\mathcal{I}}$, $F^{\mathcal{S}}$ and $F^{\mathcal{O}}$. In the following, we use $\boldsymbol{x}(t)$ for the vector of input variables as a function of time, $\boldsymbol{y}(t)$ for the vector of output variables, and $\boldsymbol{s}(t)$ for the vector of state variables. Note that $\boldsymbol{s}$ is different from the state variables $\boldsymbol{v}$ in HUTP (Section 2.3), and the latter is actually the group of $\boldsymbol{x}$, $\boldsymbol{s}$ and $\boldsymbol{y}$.

**Example 6.** A continuous Add block specifies that the output signal $y$ is the sum of the two input signals $x_0$ and $x_1$. Here $\mathcal{I} = \{x_1, x_2\}$, $\mathcal{O} = \{y\}$, $\mathcal{S} = \emptyset$, and the relation for $\mathcal{R}$ is given by

$$\forall t \geq 0 \cdot y(t) = x_1(t) + x_2(t).$$

**Example 7.** A discrete Add block with sample time $\mathsf{st} > 0$ specifies that the output is updated to the sum of inputs whenever the time is a multiple of st, and keeps constant otherwise. Here, $\mathcal{I}$, $\mathcal{O}$ and $\mathcal{S}$ are the same as before. The relation for $\mathcal{R}$ is given by

$$\forall k \in \mathbb{N} \cdot \forall t \in [k \cdot \mathsf{st}, (k+1)\mathsf{st}) \cdot y(t) = x_1(k \cdot \mathsf{st}) + x_2(k \cdot \mathsf{st}).$$

**Example 8.** A continuous Switch block with condition ">0" specifies that the output $y$ is equal to the top input $x_1$ if the middle input $x_2$ satisfies the condition; and the bottom input $x_3$ otherwise ($x_2 \leq 0$). Here $\mathcal{I} = \{x_1, x_2, x_3\}$, $\mathcal{O} = \{y\}$ and $\mathcal{S} = \emptyset$. The relation for $\mathcal{R}$ is given by

$$\forall t \geq 0 \cdot y(t) = x_1(t) \triangleleft x_2(t) > 0 \triangleright y(t) = x_3(t).$$

**Example 9.** A Unit Delay block with sample time $\mathsf{st} > 0$ and initial value $v_0$ updates its state whenever the time is a multiple of st, and outputs the previous value of state. Here $\mathcal{I} = \{x\}$, $\mathcal{O} = \{y\}$ and $\mathcal{S} = \{s\}$. The relation for $\mathcal{R}$ is given by

$$\forall k \in \mathbb{N} \cdot \forall t \in [k \cdot \text{st}, (k+1)\text{st}) \cdot s(t) = x(k \cdot \text{st})$$

$$\wedge \quad \forall t \in [0, \text{st}) \cdot y(t) = v_0 \quad \wedge$$

$$\forall k \in \mathbb{N} \cdot \forall t \in [(k+1)\text{st}, (k+2)\text{st}) \cdot y(t) = s(k \cdot \text{st}).$$

**Example 10.** An Integrator block with initial state $s_0$ specifies that its state is the integral of the input signal and the output signal is consistent with the state. Here $\mathcal{I} = \{x\}$, $\mathcal{O} = \{y\}$ and $\mathcal{S} = \{s\}$. The relation for $\mathcal{R}$ is given by

$$y(0) = s(0) = s_0 \wedge \forall t > 0 \cdot \dot{s}(t^+) = x(t) \wedge s(t^-) = s(t) = y(t).$$

Given a Simulink diagram consisting of blocks $\{b_i\}_{1 \le i \le m}$. Let $\mathcal{I}(b_i)$, $\mathcal{O}(b_i)$, $\mathcal{S}(b_i)$ and $\mathcal{R}(b_i)$ be the sets of input variables, output variables, state variables, and relation for block $b_i$, respectively. We require the state variables $\mathcal{S}(b_i)$ are disjoint from each other and from the input/output variables. Let $\{v_j\}_{1 \le j \le \ell}$ be the set of variables denoting the lines (signals) connecting blocks of the Simulink diagram. Each $v_j$ is in at most one $\mathcal{O}(b_i)$. The semantics of the Simulink diagram is a relation on $v_i(t)$, defined to be the conjunction of the relation for each block:

$$\mathcal{R} = \bigwedge\nolimits_{1 \le i \le m} \mathcal{R}(b_i)$$

Following the above analysis, we can define the HUTP semantics for (non-hierarchical) Simulink diagrams. The definition is bottom-up as we start from the individual blocks, then combine them to form the semantics of the entire diagram.

*4.1. Discrete blocks*

A discrete block is specified by a sample time $\text{st} > 0$, initial state $s_0$, and two functions $f$ and $g$ for updating the state and computing the output, respectively. The values of state and output variables of a discrete block are constant on each time interval $[k \cdot \text{st}, (k+1)\text{st})$ for $k \in \mathbb{N}$. Hence, we only need to specify their values at times $k \cdot \text{st}$. They satisfy the following equations:

$$s(k \cdot \text{st}) = f(x(k \cdot \text{st}), s((k-1) \cdot \text{st}))$$

$$y(k \cdot \text{st}) = g(x(k \cdot \text{st}), s((k-1) \cdot \text{st}))$$

where we take $s((k-1)\text{st})$ to be $s_0$ for $k = 0$. The main idea here is that the output and state at the current round is computed from the input at *current* round and state at *previous* round.

For example, the discrete Add block in Example 7 is given by

$$y(k \cdot \text{st}) = g(x_1(k \cdot \text{st}), x_2(k \cdot \text{st})) = x_1(k \cdot \text{st}) + x_2(k \cdot \text{st}).$$

There is no need for $f$ as there are no state variables. The discrete Unit Delay block in Example 9 is given by

$$s(k \cdot \text{st}) = f(x(k \cdot \text{st}), s((k-1) \cdot \text{st})) = x(k \cdot \text{st})$$

$$y(k \cdot \text{st}) = g(x(k \cdot \text{st}), s((k-1) \cdot \text{st})) = s((k-1) \cdot \text{st}).$$

Now we describe how to encode the above formulas using the HUTP language. A discrete block can either be *stateful* or *stateless*. For a stateless discrete block, there is no need for the function $f$. The computation of $g$ is instant and can be expressed by the following discrete process:

$$\text{Comp} \mathrel{\widehat{=}} [y' = g(x')].$$

Intuitively, this means that the output $y'$ is computed from the input values only after they are computed by other processes at the same round, that is after the values of $x'$ are all available. This will enforce the ordering between computation of different blocks, as we will demonstrate afterwards.

After the computation, the block will keep quiescent for the period of st (sample time), i.e., the output $y$ remains unchanged, specified by the following continuous process:

$$\text{Period} \mathrel{\widehat{=}} \lceil \dot{\underset{\sim}{y}} = \mathbf{0} \rfloor_{\text{st}}$$

Thus, the hybrid process of the stateless discrete block is defined by

$$\text{DisBlock} \mathrel{\widehat{=}} (\text{Comp} \mathbin{\fatsemi} \text{Period})^* \mathbin{\fatsemi} \text{Comp} \mathbin{\fatsemi} \text{Tail}$$

where

$$\text{Tail} \mathrel{\widehat{=}} \lceil \dot{\underset{\sim}{y}} = \mathbf{0} \rfloor_{\le \text{st}}$$

means that the block can terminate at the times $k \cdot \mathsf{st}$ or within the time intervals $(k \cdot \mathsf{st}, (k+1)\mathsf{st})$.

For a stateful discrete block, its state variables $\boldsymbol{s}$ should be initialised, given by

$$\mathsf{Init} \mathrel{\widehat{=}} [\boldsymbol{s}' = \boldsymbol{s}_0]$$

The state variables $\boldsymbol{s}$ and output variables $\boldsymbol{y}$ are updated periodically according to functions $f$ and $g$, respectively. The update is instant and can be described by the following discrete process:

$$\mathsf{Comp}' \mathrel{\widehat{=}} [\boldsymbol{s}' = f(\boldsymbol{x}', \boldsymbol{s}) \wedge \boldsymbol{y}' = g(\boldsymbol{x}', \boldsymbol{s})]$$

The waiting period of the stateful discrete block is represented by the following continuous process:

$$\mathsf{Period}' \mathrel{\widehat{=}} \llbracket \underset{\sim}{\dot{\boldsymbol{s}}} = \dot{\boldsymbol{y}} = \boldsymbol{0} \rrbracket_{\mathsf{st}}$$

During the period, state variables $\boldsymbol{s}$ and output variables $\boldsymbol{y}$ keep unchanged. Thus, similar to DisBlock, the hybrid process of the stateful discrete block is given by

$$\mathsf{DisBlockSt} \mathrel{\widehat{=}} \mathsf{Init} \,\mathring{,}\, (\mathsf{Comp}' \,\mathring{,}\, \mathsf{Period}')^* \,\mathring{,}\, \mathsf{Comp}' \,\mathring{,}\, \mathsf{Tail}'$$

where

$$\mathsf{Tail}' \mathrel{\widehat{=}} \llbracket \underset{\sim}{\dot{\boldsymbol{s}}} = \dot{\boldsymbol{y}} = \boldsymbol{0} \rrbracket_{\leq \mathsf{st}}$$

**Theorem 11.** DisBlock *and* DisBlockSt *are Simulink processes.*

**Proof.** The sequential composition $\mathsf{Comp} \,\mathring{,}\, \mathsf{Period}$ can be expanded to

$$[\boldsymbol{y}' = g(\boldsymbol{x}')] \,\mathring{,}\, \llbracket \dot{\boldsymbol{y}} = \boldsymbol{0} \rrbracket_{\mathsf{st}} = \boldsymbol{y}(ti) = g(\underset{\sim}{\boldsymbol{x}}(ti)) \wedge \lceil \dot{\boldsymbol{y}} = \boldsymbol{0} \rceil_{\mathsf{st}}$$

which is $\mathcal{H}^{\mathsf{A}}_{\mathsf{SIM}}$-healthy according to the definition in Section 3.3. Similarly, we can prove $\mathsf{Comp} \,\mathring{,}\, \mathsf{Tail}$ is also $\mathcal{H}^{\mathsf{A}}_{\mathsf{SIM}}$-healthy. According to Property 3 and by induction on the number of iterations of $*$, DisBlock is $\mathcal{H}^{\mathsf{A}}_{\mathsf{SIM}}$-healthy. Similarly, we can also prove DisBlockSt is $\mathcal{H}^{\mathsf{A}}_{\mathsf{SIM}}$-healthy. $\square$

**Example 12.** Consider two discrete blocks in sequence. One block $\mathsf{B}_1$ has input line $x$ and output line $y$, and set $y := x + 1$ every sample time 1; the other block $\mathsf{B}_2$ has input line $y$ and output line $z$, and set $z := 2 \cdot y$ every sample time 1. The Simulink processes for $\mathsf{B}_1$ and $\mathsf{B}_2$ are given by:

$$\llbracket \mathsf{B}_1 \rrbracket_{\mathsf{HUTP}} \mathrel{\widehat{=}} ([y' = x' + 1] \,\mathring{,}\, \llbracket \dot{y} = 0 \rrbracket_1)^* \,\mathring{,}\, [y' = x' + 1] \,\mathring{,}\, \llbracket \dot{y} = 0 \rrbracket_{\leq 1}$$
$$\llbracket \mathsf{B}_2 \rrbracket_{\mathsf{HUTP}} \mathrel{\widehat{=}} ([z' = 2 \cdot y'] \,\mathring{,}\, \llbracket \dot{z} = 0 \rrbracket_1)^* \,\mathring{,}\, [z' = 2 \cdot y'] \,\mathring{,}\, \llbracket \dot{z} = 0 \rrbracket_{\leq 1}$$

We first rewrite the above two definitions to corresponding logical equations. By the definition of sequential composition $\mathring{,}$, the definition for $[y' = x' + 1] \,\mathring{,}\, \llbracket \dot{y} = 0 \rrbracket_1$ in $\llbracket \mathsf{B}_1 \rrbracket_{\mathsf{HUTP}}$ expands to

$$[y' = x' + 1] \,\mathring{,}\, \llbracket \underset{\sim}{\dot{y}} = 0 \rrbracket_1 \tag{1}$$

$$= (ti = ti' < +\infty \wedge y' = x' + 1) \,\mathring{,}\, \tag{2}$$

$$\begin{pmatrix} ti < ti' < +\infty \wedge x = \underset{\sim}{x}(ti) \wedge y = \underset{\sim}{y}(ti) \\ \wedge \forall t \in (ti, ti') \cdot \dot{y}(t) = 0 \wedge ti' - ti = 1 \\ \wedge RC(\underset{\sim}{x}, \underset{\sim}{y}, ti, ti') \wedge SD(\underset{\sim}{x}, \underset{\sim}{y}, ti, ti') \end{pmatrix} \tag{3}$$

$$= \exists t_0, x_0, y_0 \cdot ti = ti_0 < +\infty \wedge y_0 = x_0 + 1$$
$$\quad \wedge ti_0 < ti' < +\infty \wedge x_0 = \underset{\sim}{x}(ti_0) \wedge y_0 = \underset{\sim}{y}(ti_0)$$
$$\quad \wedge \forall t \in (ti_0, ti') \cdot \dot{y}(t) = 0 \wedge ti' - ti_0 = 1$$
$$\quad \wedge RC(\underset{\sim}{x}, \underset{\sim}{y}, ti_0, ti') \wedge SD(\underset{\sim}{x}, \underset{\sim}{y}, ti_0, ti')$$

$$= ti' - ti = 1 \wedge \underset{\sim}{y}(ti) = \underset{\sim}{x}(ti) + 1 \wedge \forall t \in (ti, ti + 1) \cdot \dot{y}(t) = 0$$
$$\quad \wedge RC(\underset{\sim}{x}, \underset{\sim}{y}, ti, ti + 1) \wedge SD(\underset{\sim}{x}, \underset{\sim}{y}, ti, ti + 1)$$

Note that although $\underset{\sim}{x}$ does not appear in $\llbracket \dot{y} = 0 \rrbracket_1$, it is in the alphabet of $\llbracket \mathsf{B}_1 \rrbracket_{\mathsf{HUTP}}$. Therefore, we cannot remove $x = \underset{\sim}{x}(ti)$ from $\llbracket \dot{y} = 0 \rrbracket_1$ (see (3)). Besides, by $\mathcal{H}^{\mathsf{A}}_4$, the continuous state variables in $\llbracket \dot{y} = 0 \rrbracket_1$ are right continuous and semi-differentiable during the period, specified by *RC* and *SD*. Then, by induction, we can get

$$([y' = x' + 1] \,\mathring{,}\, [\![\dot{\underset{\sim}{y}} = 0]\!]_1)^*$$

$$= \text{Skip} \lor \begin{pmatrix} \exists n \in \mathbb{N}^+ \cdot ti' - ti = n \\ \land \forall k \in \mathbb{N}_{<n} \cdot \underset{\sim}{y}(ti+k) = \underset{\sim}{x}(ti+k) + 1 \\ \land \forall t \in (ti+k, ti+k+1) \cdot \dot{\underset{\sim}{y}}(t) = 0 \\ \land RC(\underset{\sim}{x}, \underset{\sim}{y}, ti, ti+n) \land SD(\underset{\sim}{x}, \underset{\sim}{y}, ti, ti+n) \end{pmatrix} \qquad (4)$$

where $\mathbb{N}^+ \triangleq \mathbb{N} \setminus \{0\}$ and $\mathbb{N}_{<n} \triangleq \{k \in \mathbb{N} \mid k < n\}$. Similar to (1),

$$[y' = x' + 1] \,\mathring{,}\, [\![\dot{\underset{\sim}{y}} = 0]\!]_{\leq 1} = 0 < ti' - ti \leq 1 \land \underset{\sim}{y}(ti) = \underset{\sim}{x}(ti) + 1$$
$$\land \forall t \in (ti, ti') \cdot \dot{\underset{\sim}{y}}(t) = 0$$
$$\land RC(\underset{\sim}{x}, \underset{\sim}{y}, ti, ti') \land SD(\underset{\sim}{x}, \underset{\sim}{y}, ti, ti')$$

Based on the above results, $[\![B_1]\!]_{\text{HUTP}}$ expands to

$$\exists n \in \mathbb{N} \cdot n < ti' - ti \leq n + 1 \land$$
$$\forall k \in \mathbb{N}_{<n} \cdot \underset{\sim}{y}(ti+k) = \underset{\sim}{x}(ti+k) + 1 \land \underset{\sim}{y}(ti+n) = \underset{\sim}{x}(ti+n) + 1$$
$$\land \forall t \in (ti+k, ti+k+1) \cdot \dot{\underset{\sim}{y}}(t) = 0 \land \forall t \in (ti+n, ti') \cdot \dot{\underset{\sim}{y}}(t) = 0$$
$$\land RC(\underset{\sim}{x}, \underset{\sim}{y}, ti, ti') \land SD(\underset{\sim}{x}, \underset{\sim}{y}, ti, ti')$$

Similarly, $[\![B_2]\!]_{\text{HUTP}}$ expands to

$$\exists n \in \mathbb{N} \cdot n < ti' - ti \leq n + 1 \land$$
$$\forall k \in \mathbb{N}_{<n} \cdot \underset{\sim}{z}(ti+k) = 2 \cdot \underset{\sim}{y}(ti+k) \land \underset{\sim}{z}(ti+n) = 2 \cdot \underset{\sim}{y}(ti+n)$$
$$\land \forall t \in (ti+k, ti+k+1) \cdot \dot{\underset{\sim}{z}}(t) = 0 \land \forall t \in (ti+n, ti') \cdot \dot{\underset{\sim}{z}}(t) = 0$$
$$\land RC(\underset{\sim}{y}, \underset{\sim}{z}, ti, ti') \land SD(\underset{\sim}{y}, \underset{\sim}{z}, ti, ti')$$

The connection of $B_1$ and $B_2$ can be defined by $[\![B_1]\!]_{\text{HUTP}} \land [\![B_2]\!]_{\text{HUTP}}$, i.e.,

$$\exists n \in \mathbb{N} \cdot n < ti' - ti \leq n + 1 \land$$
$$\forall k \in \mathbb{N}_{<n} \cdot \underset{\sim}{y}(ti+k) = \underset{\sim}{x}(ti+k) + 1 \land \underset{\sim}{z}(ti+k) = 2 \cdot \underset{\sim}{y}(ti+k)$$
$$\land \underset{\sim}{y}(ti+n) = \underset{\sim}{x}(ti+n) + 1 \land \underset{\sim}{z}(ti+n) = 2 \cdot \underset{\sim}{y}(ti+n)$$
$$\land \forall t \in (ti+k, ti+k+1) \cdot \dot{\underset{\sim}{y}}(t) = \dot{\underset{\sim}{z}}(t) = 0$$
$$\land \forall t \in (ti+n, ti') \cdot \dot{\underset{\sim}{y}}(t) = \dot{\underset{\sim}{z}}(t) = 0$$
$$\land RC(\underset{\sim}{x}, \underset{\sim}{y}, \underset{\sim}{z}, ti, ti') \land SD(\underset{\sim}{x}, \underset{\sim}{y}, \underset{\sim}{z}, ti, ti')$$

This example demonstrates that the parallel composition of the HUTP semantics for $B_1$ and $B_2$ simplifies to the desired form, enforcing that the computation in $B_1$ is performed before that of $B_2$ at every sample time. We further note that the values of $\underset{\sim}{y}$ and $\underset{\sim}{z}$ are determined given values of input signal $\underset{\sim}{x}$.

## 4.2. Continuous blocks

We consider two kinds of continuous blocks: *computation* blocks and Integrator blocks. A computation block is a stateless block with sample time 0 (see Examples 6 and 8). When building Simulink diagrams, the sample time of computation blocks are usually inherited from integrator blocks by sample time propagation. It is specified by a function $g$ from its input $\boldsymbol{x}$ to its output $\boldsymbol{y}$, so its relation is specified by $\forall t \geq 0 \cdot \boldsymbol{y}(t) = g(\boldsymbol{x}(t))$. Hence, its HUTP representation is

$$\text{ConBlock} \triangleq \lceil \underset{\sim}{\boldsymbol{y}} = g(\underset{\sim}{\boldsymbol{x}}) \rceil$$

**Remark 13.** For a continuous block, our concern is the evolution of its output signals ($\boldsymbol{y}$) according to its input signals ($\boldsymbol{x}$) rather than its initial and/or final observations ($\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{x}'$, and $\boldsymbol{y}'$). Thus, we use $\lceil \cdot \rfloor$ rather than $[\![ \cdot ]\!]$ in ConBlock.

The Integrator block is already given in Example 10, and its HUTP representation is given by

$$\text{IntBlock} \triangleq \underset{\sim}{y}(ti) = \underset{\sim}{s}(ti) = s_0 \land \lceil \dot{\underset{\sim}{s}}^+ = \underset{\sim}{x} \land \underset{\sim}{s}^- = \underset{\sim}{s} = \underset{\sim}{y} \rfloor$$

where $s_0$ is the initial state of $\underset{\sim}{s}$, and $\dot{\underset{\sim}{s}}^+$ and $\underset{\sim}{s}^-$ denote the right-hand derivative and the left limit of $\underset{\sim}{s}$, respectively. Since $\underset{\sim}{x}$ could be discontinuous but at least right continuous as stated by $\mathcal{H}_4$, we use $\dot{\underset{\sim}{s}}^+$ rather than $\dot{\underset{\sim}{s}}$ in the representation. Besides, $\underset{\sim}{s}$ should be continuous and the output signal should keep consistent with the state, so we need the condition $\underset{\sim}{s}^- = \underset{\sim}{s} = \underset{\sim}{y}$.

**Theorem 14.** ConBlock *and* IntBlock *are Simulink processes.*

**Proof.** ConBlock and IntBlock are $\mathcal{H}^{A}_{SIM}$-healthy by the definition of $\lceil \cdot \rfloor$ specified in Section 3.3. $\quad\square$

**Property 15.** $\lceil \underset{\sim}{P} \rfloor \wedge \lceil \underset{\sim}{Q} \rfloor = \lceil \underset{\sim}{P} \wedge \underset{\sim}{Q} \rfloor$.

**Proof.** According to the definition of $\lceil \cdot \rfloor$,

$$
\begin{aligned}
\lceil \underset{\sim}{P} \rfloor \wedge \lceil \underset{\sim}{Q} \rfloor = {} & 0 < ti < ti' < +\infty \wedge \forall t \in (ti, ti') \cdot \underset{\sim}{P}(\boldsymbol{v}(t), \dot{\boldsymbol{v}}(t)) \\
& \wedge RC(\boldsymbol{v}, ti, ti') \wedge SD(\boldsymbol{v}, ti, ti') \wedge \\
& 0 < ti < ti' < +\infty \wedge \forall t \in (ti, ti') \cdot \underset{\sim}{Q}(\boldsymbol{v}(t), \dot{\boldsymbol{v}}(t)) \\
& \wedge RC(\boldsymbol{v}, ti, ti') \wedge SD(\boldsymbol{v}, ti, ti') \\
= {} & \lceil \underset{\sim}{P} \wedge \underset{\sim}{Q} \rfloor
\end{aligned}
$$

The property is proved. $\quad\square$

**Example 16.** Consider an Integrator block $B_3$ with state $s$, input line $z$ and output line $x$, and $s$ is set to 0 initially. The Simulink process for $B_3$ is given by:

$$
\begin{aligned}
[\![ B_3 ]\!]_{\mathsf{HUTP}} = {} & \underset{\sim}{x}(ti) = \underset{\sim}{s}(ti) = 0 \wedge \lceil \dot{\underset{\sim}{s}}^+ = \underset{\sim}{z} \wedge \underset{\sim}{s}^- = \underset{\sim}{s} = \underset{\sim}{x} \rfloor \\
= {} & \begin{pmatrix} ti < ti' < +\infty \wedge \underset{\sim}{x}(ti) = \underset{\sim}{s}(ti) = 0 \\ \wedge \forall t \in (ti, ti') \cdot \dot{\underset{\sim}{s}}(t^+) = \underset{\sim}{z}(t) \wedge \underset{\sim}{s}(t^-) = \underset{\sim}{s}(t) = \underset{\sim}{x}(t) \\ \wedge RC(\underset{\sim}{x}, \underset{\sim}{s}, \underset{\sim}{z}, ti, ti') \wedge SD(\underset{\sim}{x}, \underset{\sim}{s}, \underset{\sim}{z}, ti, ti') \end{pmatrix}
\end{aligned}
$$

### 4.3. Composition

A (non-hierarchical) Simulink diagram is composed of discrete and continuous blocks connected by lines. Given such a diagram, we can construct a directed graph $\mathcal{G}$, called its *causality graph*, as follows. The vertices of $\mathcal{G}$ are the input/output variables, and there is an edge from $v_i$ to $v_j$ if $v_i$ is the input and $v_j$ is the output of some non-delay discrete or computation block $B_k$. Note that the discrete delay block and the integrator block are excluded. If $\mathcal{G}$ is acyclic, then the diagram is said to be well-formed. Otherwise, there exist some loops among discrete and/or computation blocks called *algebraic loops* (also called logical loops in [29]), which may not always admit a solution. Actually, the cycle-freedom of causality graphs is a necessary condition for Simulink diagrams to behave well. In particular, it allows to avoid straightforward deadlocks. To our knowledge, should the causality graph of a diagram contain a cycle, the tool Matlab/Simulink would reject it, returning an error or a warning. Accordingly, we only consider Simulink diagrams with acyclic causality graphs in this paper.

If the diagram is well-formed, its HUTP semantics can be described by the parallel composition of the atomic blocks it contains. Specifically, given a well-formed diagram consisting of $n$ blocks whose semantics are represented by $P_i$ ($1 \le i \le n$), the semantics of the diagram is denoted by the following parallel composition, which is equivalent to the conjunction by Property 5:

$$
P_1 \| \cdots \| P_n \equiv \bigwedge_{i=1}^{n} P_i
$$

We say the semantics of a diagram is *determined* if, given any choice of input signals to the overall diagram, there are unique functions for all output and state variables that satisfies $\bigwedge_{i=1}^{n} P_i$. We wish to prove that under additional conditions related to the unique solvability of ODEs, the HUTP semantics of a well-formed diagram is determined. Before proving this result, we prove the following lemmas.

**Lemma 17.** *Consider a well-formed Simulink diagram consisting of n discrete blocks whose semantics are represented respectively by* $P_i$. *Given any choice of input signals to the overall diagram, there are unique functions for all input, output, and state variables that satisfy* $\bigwedge_{i=1}^{n} P_i$. *Moreover, let* st *be the sample time of the diagram (greatest common divisor of the sample times of the blocks), then the values of output and state variables depend only on the values of input variables at multiples of* st, *and they are constant over each time interval* $[k \cdot st, (k+1) \cdot st)$.

**Proof.** Since the causality graph $\mathcal{G}$ of the Simulink diagram is acyclic, we can choose a topological ordering $\underset{\sim}{v}_1, \cdots, \underset{\sim}{v}_m$ for the input and output variables of the blocks in the diagram. For brevity, we assume that the sample time of the diagram is 1. We prove by induction on $k$ that there exist unique values for input, output and state variables on each time interval $[k, k+1)$. First, consider the base case $k = 0$, we perform a second induction on the index $i$ in the ordering $\underset{\sim}{v}_i$. For the variable $\underset{\sim}{v}_i$, by the induction hypothesis, we can assume $\underset{\sim}{v}_j(0)$ is uniquely determined for each $j < i$. $\underset{\sim}{v}_i$ is either an input to the overall diagram, or the output of some block $\mathsf{B}_j$. If $\mathsf{B}_j$ is a delay block, then $\underset{\sim}{v}_i(0)$ is given by the initial value of the state. Otherwise, according to the definition of $\mathcal{G}$, all input variables of the block occur earlier in the topological order, whose values at 0 are uniquely determined by induction, so again $\underset{\sim}{v}_i(0)$ is uniquely determined. Since the values of state variables at time 0 denote a function of input variables at time 0 and the initial state, they are also determined.

Now consider the inductive case $k+1$. Again, we induct on the index $i$ in the ordering $\underset{\sim}{v}_i$. If $\underset{\sim}{v}_i$ is an input to the overall diagram, then it is already determined. So suppose $\underset{\sim}{v}_i$ is the output of some block $\mathsf{B}_j$. If the sample time of block $\mathsf{B}_j$ is a multiple of $k+1$, again we divide into cases for delay block and non-delay block. For the delay block, the value of $\underset{\sim}{v}_i(k+1)$ is given by the value of state at a previous time. For non-delay blocks, the value of $\underset{\sim}{v}_i(k+1)$ is a function of $\underset{\sim}{v}_j(k+1)$ for $j < i$ and state variables at time $k$. In both cases the value is determined. Finally, if the sample time of $\mathsf{B}_j$ is not a multiple of $k+1$, we have $\underset{\sim}{v}_i(k+1) = \underset{\sim}{v}_i(k)$. This shows $\underset{\sim}{v}_i(k+1)$ is determined for all $1 \le i \le m$. Then, since the state variables at time $k+1$ denote a function of variables $\underset{\sim}{v}_i(k+1)$ and state variables at time $k$, they are determined as well.

In this way, we construct the values of all $\underset{\sim}{v}_i$, as well as that of the state variables, at the integer time points. In the process, we have considered relations for all blocks. Hence, the solution we obtained satisfies the relation $\bigwedge_{i=1}^{n} \mathsf{P}_i$. Finally, by the construction in this proof, it is clear that the output and state variables depend only on the values of input variables at each integer $k$, and are constant over each time interval $[k, k+1)$. $\square$

**Lemma 18.** *For a well-formed Simulink diagram consisting of continuous blocks, let $\boldsymbol{v}$ be the line variables of the diagram, where $\boldsymbol{x}$ denote the input variables to the diagram and $\boldsymbol{s}$ and $\boldsymbol{y}$ represent the state and output variables of the Integrator blocks in the diagram. Then,*

*(1) its semantics can be expressed in the form of*

$$\boldsymbol{y}(ti) = \boldsymbol{s}(ti) = \boldsymbol{s}_0 \wedge \lceil \underset{\sim}{P}(\underset{\sim}{\boldsymbol{v}}) \wedge \dot{\underset{\sim}{\boldsymbol{s}}}^+ = E(\underset{\sim}{\boldsymbol{x}} \uplus \underset{\sim}{\boldsymbol{s}}) \wedge \underset{\sim}{\boldsymbol{s}}^- = \underset{\sim}{\boldsymbol{s}} = \underset{\sim}{\boldsymbol{y}} \rfloor \tag{5}$$

*where $\boldsymbol{s}_0$ are the initial states of $\boldsymbol{s}$, $\underset{\sim}{P}$ is a relation only relating $\underset{\sim}{\boldsymbol{v}}$, and $E$ is a (vector) function in terms of variables in $\underset{\sim}{\boldsymbol{x}} \uplus \underset{\sim}{\boldsymbol{s}}$;*

*(2) if the function $E$ satisfies the global Lipschitz condition, given any choice of input signals to the overall diagram, there are unique functions for all input, output, and state variables that satisfy the semantics.*

**Proof.** (1) Assume the diagram consists of $m$ Integrator blocks and $n$ computation blocks. Label the Integrator blocks by $\mathsf{B}_i$ for $1 \le i \le m$, and the computation blocks by $\mathsf{B}_j$ for $m+1 \le j \le m+n$. Let $a_i$, $s_i$ and $y_i$ be the input, state and output variables of an Integrator block $\mathsf{B}_i$, respectively, and $\boldsymbol{b}_j$ and $\boldsymbol{c}_j$ be the input and output variables of a computation block $\mathsf{B}_j$, respectively. According to the semantics of these continuous blocks (see Section 4.2) and Property 15, the semantics of the diagram can be defined by

$$\mathsf{Init} \wedge \mathsf{Evolve} \tag{6}$$

where

$$\mathsf{Init} \widehat{=} \bigwedge_{i=1}^{m} \underset{\sim}{y}_i(ti) = \underset{\sim}{s}_i(ti) = s_{i,0}$$

$$\mathsf{Evolve} \widehat{=} \Big\lceil \bigwedge_{i=1}^{m} \dot{\underset{\sim}{s}}_i^+ = \underset{\sim}{a}_i \wedge \underset{\sim}{s}_i^- = \underset{\sim}{s}_i = \underset{\sim}{y}_i \wedge \bigwedge_{j=m+1}^{m+n} \underset{\sim}{\boldsymbol{c}}_j = g_j(\underset{\sim}{\boldsymbol{b}}_j) \Big\rfloor$$

where $s_{i,0}$ is the initial value of $s_i$. For each computation block $\mathsf{B}_j$, it defines a variable substitution mapping $\Gamma_{jk}$. Concretely, it maps each output variable $c_{jk} \in \boldsymbol{c}_j$ of the block to an expression $g_{jk}$ on the input variables $\boldsymbol{b}_j$, i.e., $\Gamma_{jk}(c_{jk}) = g_{jk}(\boldsymbol{b}_j)$, where all the $g_{jk}$ form the function $g_j$. Since the diagram is well-formed, the input and output variables of all computation blocks in the diagram form a directed acyclic graph. Therefore, all the mapping functions $\Gamma_{jk}$ can be composed to form a function $\Gamma$ that maps the input variable $a_i$ (which could be some $c_{j,k}$) of each Integrator block $\mathsf{B}_i$ to $\Gamma(a_i)$ which is an expression on $\boldsymbol{x} \uplus \boldsymbol{s}$, denoted $e_i(\boldsymbol{x} \uplus \boldsymbol{s})$. Note that for $a_i \notin \mathrm{dom}(\Gamma)$, we let $\Gamma(a_i) = a_i$. All the expressions $e_i(\boldsymbol{x} \uplus \boldsymbol{s})$ form the expression function $E$. Besides, $\bigwedge_{j=m+1}^{m+n} \underset{\sim}{\boldsymbol{c}}_j = g_j(\underset{\sim}{\boldsymbol{b}}_j)$ denotes the relation $\underset{\sim}{P}$ in the representation. In summary, the formula of (6) can be expresses in the form of (5).

(2) According to Equation (5), the right-hand derivative of $\underset{\sim}{\boldsymbol{s}}$ always exists, which means $\underset{\sim}{\boldsymbol{s}}$ is piecewise differentiable. By induction on the time intervals where $\underset{\sim}{\boldsymbol{s}}$ is differentiable, and using the fact that $E$ satisfies the global Lipschitz condition, we obtain the existence and uniqueness of the solution.

The lemma is proved. $\square$

**Theorem 19.** *For a well-formed diagram consisting of discrete and continuous blocks, if the expression function $E$ of the continuous sub-diagram (consisting of the continuous blocks) satisfies the global Lipschitz condition, then the HUTP semantics of the entire diagram is determined and can be represented by a Simulink process.*

**Proof.** By Property 3 and Theorems 11 and 14, the HUTP semantics is a Simulink process. To show that the HUTP semantics of the combination of discrete and continuous sub-diagrams is determined, we perform an induction on multiples of the sample time st of the discrete sub-diagram (as defined in Lemma 17). At each step $k$, the computation of the discrete diagram provides the initial conditions at time $k \cdot$ st for evolution of the continuous diagram, and the continuous evolution provides the initial value for the discrete diagram at time $(k + 1) \cdot$ st. Hence determinacy follows from Lemma 17 and Lemma 18. □

**Example 20.** The connection of $B_1$, $B_2$ (Example 12) and $B_3$ (Example 16) forms a closed Simulink diagram with the causality graph $\mathcal{G} = \{(x, y), (y, z)\}$ acyclic, hence the diagram is well-formed. Then, the HUTP semantics of this diagram is $[\![B_1]\!]_{\mathsf{HUTP}} \wedge [\![B_2]\!]_{\mathsf{HUTP}} \wedge [\![B_3]\!]_{\mathsf{HUTP}}$, expanding as follows:

$$
\begin{pmatrix}
\underset{\sim}{x}(ti) = \underset{\sim}{s}(ti) = 0 \wedge \\
\exists n \in \mathbb{N} \cdot n < ti' - ti \leq n + 1 \wedge \forall k \in \mathbb{N}_{<n} \cdot \\
\underset{\sim}{y}(ti + k) = \underset{\sim}{x}(ti + k) + 1 \wedge \underset{\sim}{z}(ti + k) = 2 \cdot \underset{\sim}{y}(ti + k) \wedge \\
\underset{\sim}{y}(ti + n) = \underset{\sim}{x}(ti + n) + 1 \wedge \underset{\sim}{z}(ti + n) = 2 \cdot \underset{\sim}{y}(ti + n) \wedge \\
\forall t \in (ti + k, ti + k + 1) \cdot \dot{\underset{\sim}{y}}(t) = \dot{\underset{\sim}{z}}(t) = 0 \wedge \dot{\underset{\sim}{s}}(t) = \underset{\sim}{z}(t) \wedge \\
\forall t \in (ti + n, ti') \cdot \dot{\underset{\sim}{y}}(t) = \dot{\underset{\sim}{z}}(t) = 0 \wedge \dot{\underset{\sim}{s}}(t) = \underset{\sim}{z}(t) \wedge \\
\forall t \in (ti, ti') \cdot \underset{\sim}{s}(t^-) = \underset{\sim}{s}(t) = \underset{\sim}{x}(t) \wedge \\
RC(\underset{\sim}{x}, \underset{\sim}{y}, \underset{\sim}{z}, \underset{\sim}{s}, ti, ti') \wedge SD(\underset{\sim}{x}, \underset{\sim}{y}, \underset{\sim}{z}, \underset{\sim}{s}, ti, ti')
\end{pmatrix}
$$

Since $\underset{\sim}{z}$ is differentiable within the intervals $(ti + k, ti + k + 1)$ and $(ti + n, ti')$, $\dot{\underset{\sim}{s}}(t^+)$ is replaced with $\dot{\underset{\sim}{s}}(t)$ in the above formula. For ensuring the continuity of $\underset{\sim}{s}$, there should be $\underset{\sim}{s}^- = \underset{\sim}{s}$ during the period. For the above formula, we get the following unique solution:

$$
\forall t \in [ti, ti') \cdot \exists d \in \mathbb{R} \cdot d = t - ti \wedge
\begin{pmatrix}
\underset{\sim}{s}(t) &=& 2 \cdot 3^{\lfloor d \rfloor}(d - \lfloor d \rfloor) + 3^{\lfloor d \rfloor} - 1 \\
\underset{\sim}{x}(t) &=& 2 \cdot 3^{\lfloor d \rfloor}(d - \lfloor d \rfloor) + 3^{\lfloor d \rfloor} - 1 \\
\underset{\sim}{y}(t) &=& 3^{\lfloor d \rfloor} \\
\underset{\sim}{z}(t) &=& 2 \cdot 3^{\lfloor d \rfloor}
\end{pmatrix}
$$

which is exactly the semantics of the Simulink diagram (here $\lfloor \cdot \rfloor : \mathbb{R} \to \mathbb{Z}$ is the floor function).

## 5. Hierarchical Simulink diagrams

Modular design is a design principle that subdivides a system into smaller parts called modules (or subsystems), which can be independently created, modified, replaced, or exchanged with other modules or between different systems. The modelling of hierarchical Simulink diagrams reflects the principle of modular design: a Simulink diagram is composed of hierarchical subsystems, which may include enabled or triggered behaviours. In this section, we establish the HUTP semantics for *normal*, *triggered* and *enabled* subsystems, which forms hierarchical Simulink diagrams.

### 5.1. Normal subsystems

A normal subsystem groups a set of atomic Simulink blocks together, and will execute them as a single unit. Simulink distinguishes the input (output) variables $\boldsymbol{i}$ ($\boldsymbol{o}$) as seen from within the subsystem and the input (output) variables $\bar{\boldsymbol{i}}$ ($\bar{\boldsymbol{o}}$) as seen from outside (as lines in the overall diagram). For normal subsystems, we will identify the variables $\boldsymbol{i}$ with $\bar{\boldsymbol{i}}$, and variables $\boldsymbol{o}$ with $\bar{\boldsymbol{o}}$ (see $x$ and $y$ in Fig. 2). Later on, we may not identify $\boldsymbol{i}$ with $\bar{\boldsymbol{i}}$ for triggered and enabled subsystems.

A normal subsystem is *well-formed* if its causality graph is acyclic. The effect of executing a well-formed normal subsystem is equivalent to executing the corresponding Simulink diagram consisting of the same set of blocks. Therefore, the semantics of a normal subsystem can be defined by the conjunction of the semantics of all the blocks it contains, as specified in Section 4.3.

In the translation algorithm from Simulink to HCSP presented in [29], the subsystem is flattened by connecting the in-ports and out-ports as seen from inside with the corresponding in-ports and out-ports on the outside. The result of this process is shown on the right side of Fig. 2. This flattening makes the translation process easier to implement, and is necessary for collecting together all continuous blocks in the diagram for translation to a single ODE. However, it violates to some extent the principle of modular design, i.e., the hierarchical structure of the Simulink diagram is not reflected in the translated HCSP process. In this paper, subsystems are not flattened and therefore the structure of Simulink diagrams can be reflected in their HUTP semantics. For example, the Simulink diagram on the left of Fig. 2 is composed of two well-formed subsystems Subsystem0 and Subsystem1 whose semantics are given by

$$[\![\mathsf{Subsystem0}]\!]_{\mathsf{HUTP}} \ \widehat{=} \ [\![\mathsf{Int0}]\!]_{\mathsf{HUTP}} \wedge [\![\mathsf{Bias0}]\!]_{\mathsf{HUTP}}$$

$$[\![\mathsf{Subsystem1}]\!]_{\mathsf{HUTP}} \ \widehat{=} \ [\![\mathsf{Int1}]\!]_{\mathsf{HUTP}} \wedge [\![\mathsf{Bias1}]\!]_{\mathsf{HUTP}}$$
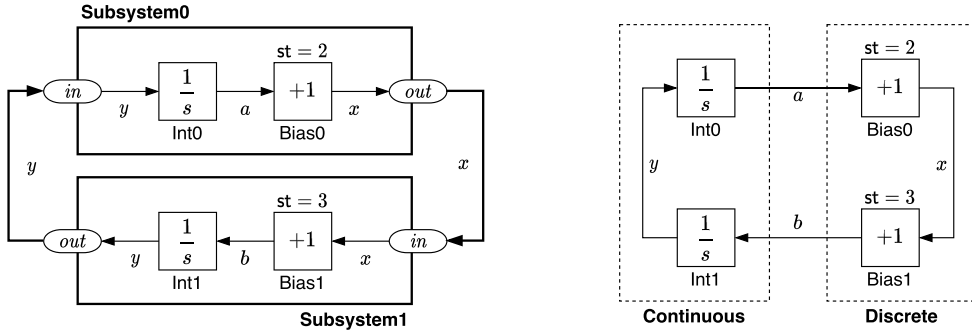
**Fig. 2.** A well-formed Simulink diagram composed of two subsystems. The left is the original hierarchical diagram and the right is the flattened form.
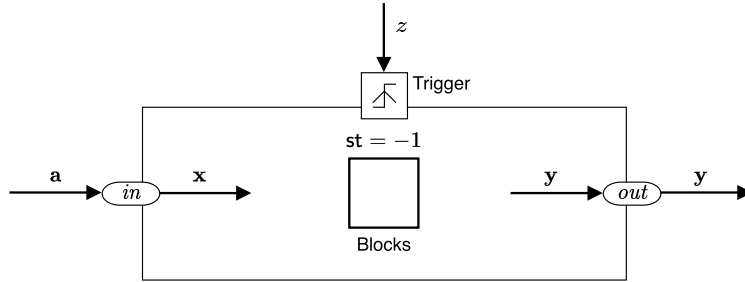


**Fig. 3.** A sketch of a triggered subsystem with rising edge trigger.

where $[\![\text{Bias0}]\!]_{\mathsf{HUTP}}$ and $[\![\text{Bias1}]\!]_{\mathsf{HUTP}}$ are discrete processes (Section 4.1), and $[\![\text{Int0}]\!]_{\mathsf{HUTP}}$ and $[\![\text{Int1}]\!]_{\mathsf{HUTP}}$ are continuous processes (Section 4.2).

### 5.2. Triggered subsystems

A triggered subsystem only contains blocks with inherited sample time $(-1)$. Such a block has no specified sample time, whose execution depends solely on the triggering signal. Concretely, the blocks execute at the instant when the trigger condition on the trigger line holds. A sketch of a triggered subsystem with rising edge trigger is shown in Fig. 3. Similar to well-formed normal subsystems, a triggered subsystem is well-formed if its causality graph is acyclic. The trigger port senses the input signal $z$ in real time. In this paper, we assume that each triggered subsystem has only one trigger line. There are three basic trigger types: rising, falling and either. For the rising edge trigger, the subsystem is triggered at time $t$ whenever (1) $z$ rises from negative to non-negative at $t$ or (2) $z$ rises from non-positive to positive at $t$. Formally,

$$\underset{\sim}{z}(t^-) < 0 \wedge \underset{\sim}{z}(t) \geq 0 \quad \vee \quad \underset{\sim}{z}(t^-) \leq 0 \wedge \underset{\sim}{z}(t) > 0$$

The triggering conditions for the other two trigger types can be defined similarly. Therefore, in this section, we only define the HUTP semantics for triggered subsystems with rising edge trigger. In this paper, we assume that the trigger line of each triggered subsystem is the output of a discrete block, hence piecewise constant with some sample time st. Treatment of continuous triggering will be more complicated, involving analysis of zero-crossing detection and potential cascade of zero-crossings [2].

Now we consider the HUTP semantics of well-formed triggered subsystems. The subsystem is triggered at current time iff (1) the previous value of the signal $z$ is less than 0 and the current value of $z$ reaches or crosses 0; or (2) the previous value of $z$ is not greater than 0 and the current value of $z$ crosses 0. After that, the signal $z$ will keep "not triggering" for some period until it satisfies the trigger condition again. Therefore, the behaviour of $z$ between two adjacent triggering time instants can be defined by

$$\text{Trigger} \mathrel{\widehat{=}} [z < 0 \wedge z' \geq 0 \quad \vee \quad z \leq 0 \wedge z' > 0] \mathbin{\overset{\circ}{\scriptstyle 9}} [\![\neg\text{trigger}]\!]$$

where

$$\text{trigger} \mathrel{\widehat{=}} z^- < 0 \wedge z \geq 0 \quad \vee \quad z^- \leq 0 \wedge z > 0$$

When triggered, the subsystem gets the latest values from the input ports, given by the relation $\underset{\sim}{\pmb{x}}(ti) = \underset{\sim}{\pmb{a}}(ti)$, where $ti$ is the current triggering time, $\pmb{x}$ denote the input variables from within the subsystem while $\pmb{a}$ denote the input variables to the
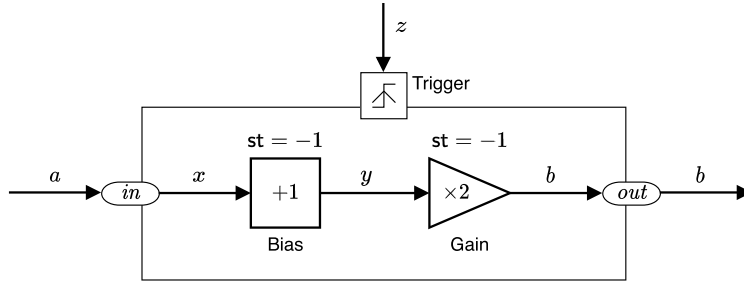
**Fig. 4.** An example of a triggered subsystem with rising edge trigger.

overall subsystem (see Fig. 3). The reason we distinguish $\boldsymbol{x}$ and $\boldsymbol{a}$ is that their values are not the same at all times: during the idle period of the subsystem, $\boldsymbol{x}$ will keep unchanged while $\boldsymbol{a}$ can change dynamically according to the behaviour of its source subsystem. The input variables $\boldsymbol{x}$ synchronise with the input variables $\boldsymbol{a}$ from the outside only when the triggering signal arrives. However, for output variables, it is not necessary to distinguish the output variables to within the subsystem and the output variables from the overall subsystem, because the output variables are controlled by the subsystem solely and they will not be modified by other subsystems, i.e., they always keep consistent.

After the input synchronisation ($\boldsymbol{x}$ obtain the values of $\boldsymbol{a}$), the subsystem will perform the computation and then keep idle for some period, which can be represented by the conjunction of the idle process $\lceil \dot{\underset{\sim}{\boldsymbol{v}}} = \boldsymbol{0} \rfloor$, where $\boldsymbol{v}$ are the variables inner the subsystem, and the continuous processes $\lceil \underset{\sim}{P_i} \rfloor$ ($1 \leq i \leq n$) of all blocks in the subsystem. Therefore, according to Property 15, which indicates

$$\lceil \dot{\underset{\sim}{\boldsymbol{v}}} = \boldsymbol{0} \rfloor \wedge \bigwedge_{i=1}^{n} \lceil \underset{\sim}{P_i} \rfloor = \lceil \dot{\underset{\sim}{\boldsymbol{v}}} = \boldsymbol{0} \wedge \bigwedge_{i=1}^{n} \underset{\sim}{P_i} \rfloor$$

the behaviour of the subsystem can be captured by

$$\mathsf{SubSys} \mathrel{\widehat{=}} \underset{\sim}{\boldsymbol{x}}(ti) = \underset{\sim}{\boldsymbol{a}}(ti) \wedge \lceil \dot{\underset{\sim}{\boldsymbol{v}}} = \boldsymbol{0} \wedge \bigwedge_{i=1}^{n} \underset{\sim}{P_i} \rfloor$$

Before triggering, the variables $\boldsymbol{v}$ of the lines in the subsystem should be initialised, because if the subsystem is not triggered at the beginning, it should be guaranteed that the values of $\boldsymbol{v}$ are valid. By default, $\boldsymbol{v}$ are initialised to $\boldsymbol{0}$ in Simulink. If the subsystem is not triggered initially, the values of $\boldsymbol{v}$ will keep constant. Besides, the trigger line $z$ should also be initialised, because at the beginning, it will compare the initial value of $z$ with the current value of $z$ to determine if the subsystem should be triggered at the time. By default, we also set $z$ to 0 initially. Then, the initialisation is specified by

$$\mathsf{InitTrig} \mathrel{\widehat{=}} [\boldsymbol{v}' = \boldsymbol{0} \wedge z' = 0] \mathbin{\mathring{,}} (\mathsf{Trigger} \wedge \mathsf{SubSys} \quad \vee \quad \mathsf{Idle})$$

In summary, the semantics of triggered subsystem can be defined by

$$\mathsf{TrigSubSys} \mathrel{\widehat{=}} \mathsf{InitTrig} \mathbin{\mathring{,}} (\mathsf{Trigger} \wedge \mathsf{SubSys})^*$$

When creating the causality graph of a triggered subsystem, we add edges from the trigger line to each input line inside the subsystem, since whether the input lines receive values from the outside is determined by the trigger line. For the example in Fig. 4, the causality graph $\mathcal{G}$ is given by $\{(z, x), (a, x), (x, y), (y, b)\}$.

**Example 21.** Consider the triggered subsystem in Fig. 4, we only analyse SubSys. Concretely, SubSys is equivalent to

$$\underset{\sim}{x}(ti) = \underset{\sim}{a}(ti) \wedge \lceil \dot{\underset{\sim}{x}} = \dot{\underset{\sim}{y}} = \dot{\underset{\sim}{b}} = 0 \ \wedge \ \underset{\sim}{b} = 2 \cdot \underset{\sim}{y} \wedge \underset{\sim}{y} = \underset{\sim}{x} + 1 \rfloor$$

which can expand as follows:

$$\underset{\sim}{x}(ti^+) = \underset{\sim}{x}(ti) = \underset{\sim}{a}(ti) = \underset{\sim}{a}(ti^+) \wedge \underset{\sim}{y}(ti) = \underset{\sim}{y}(ti^+) \wedge \underset{\sim}{b}(ti) = \underset{\sim}{b}(ti^+) \wedge \tag{7}$$

$$\forall t \in (ti, ti') \cdot \dot{\underset{\sim}{x}}(t) = \dot{\underset{\sim}{y}}(t) = \dot{\underset{\sim}{b}}(t) = 0 \wedge \underset{\sim}{b}(t) = 2 \cdot \underset{\sim}{y}(t) \wedge \underset{\sim}{y}(t) = \underset{\sim}{x}(t) + 1 \tag{8}$$

$$\wedge RC(\underset{\sim}{a}, ti, ti') \wedge SD(\underset{\sim}{a}, ti, ti')$$

This process specifies the behaviour that the subsystem is triggered at the beginning and then keeps idle for some period. First, the subsystem gets the latest input at $ti$, i.e., $\underset{\sim}{x}(ti) = \underset{\sim}{a}(ti)$; then, according to (8), we can get the relation $\underset{\sim}{b}(ti^+) = 2 \cdot \underset{\sim}{y}(ti^+) \wedge \underset{\sim}{y}(ti^+) = \underset{\sim}{x}(ti^+) + 1$; combining (7), we can infer that

$$\underset{\sim}{b}(ti) = 2 \cdot \underset{\sim}{y}(ti) \wedge \underset{\sim}{y}(ti) = \underset{\sim}{x}(ti) + 1 \tag{9}$$

which indicates the subsystem executes the computation at the beginning when triggered; afterwards, $x$, $y$ and $b$ keep constant from $ti$ to $ti'$, and therefore the relation between these variables always holds during the period (see (8)).
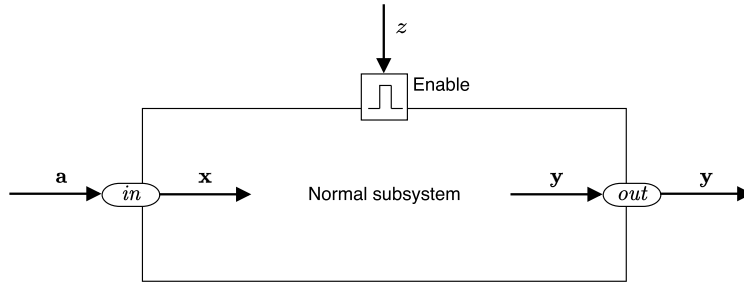
**Fig. 5.** A sketch of an enabled subsystem.

**Theorem 22.** *If the triggered subsystem is well-formed, then* TrigSubSys *is a Simulink process with determined semantics.*

**Proof.** According to Property 3 and by induction on the number of iterations of $*$, TrigSubSys is a Simulink process. When the triggering signal $z$ is determined, the triggering times are determined as well. When triggered, the subsystem gets the latest inputs and performs the execution expressed by a relation between the variables at the time (see (9)). Since the subsystem is well-formed, the solution of the relation is unique given any choice of input signals. The signals then keep constant during the period until the next trigger time arrives. Therefore, the semantics represented by TrigSubSys is determined. □

*5.3. Enabled subsystems*

An enabled subsystem is similar to a normal subsystem except that its execution depends on the enabled signal: it executes as usual when the value of the enabled signal is larger than 0 and keeps idle otherwise. A sketch of an enabled subsystem is illustrated in Fig. 5. Similar to triggered subsystems, we assume that each enabled subsystem has only one enabling line $z$.

The evolution of the enabled signal $z$ can be seen as an interleaving of positive and non-positive phases. Concretely, the evolution of $z$ can be defined by

$$(z \leq 0 \wedge \lceil \underset{\sim}{z} > 0 \rfloor\!\rfloor \quad \vee \quad z \geq 0 \wedge \lceil \underset{\sim}{z} \leq 0 \rfloor\!\rfloor)^*$$

In the beginning of a positive phase, it checks if the value of $z$ from the preceding phase (which should be non-positive) is non-positive ($z \leq 0$); if so, the value of $z$ will keep positive for some period ($\lceil \underset{\sim}{z} > 0 \rfloor\!\rfloor$). The behaviour of non-positive phases is similar.

In a positive phase, the inner input variables $\boldsymbol{x}$ should keep consistent with the input variables $\boldsymbol{a}$ to the overall subsystem, and then the subsystem executes as if it were a normal subsystem. Concretely, let $\mathsf{P}_i$ ($1 \leq i \leq n$) be the process denoting the semantics of the $i$-th block in the subsystem, then the behaviour of the enabled subsystem in a positive phase can be described by

$$\mathsf{Enabled} \mathrel{\widehat{=}} z \leq 0 \wedge \lceil \underset{\sim}{z} > 0 \rfloor\!\rfloor \quad \wedge \quad \lceil \underset{\sim}{\boldsymbol{a}} = \underset{\sim}{\boldsymbol{x}} \rfloor \wedge \bigwedge\nolimits_{i=1}^{n} \mathsf{P}_i \wedge \boldsymbol{v}' = \underset{\sim}{\boldsymbol{v}}(ti'^{-})$$

where $\boldsymbol{v}$ are the variables of all the lines in the subsystem. Note that $\boldsymbol{a} \cap \boldsymbol{v} = \emptyset$ and $\boldsymbol{x} \subseteq \boldsymbol{v}$. In each non-positive phase, the subsystem does nothing and keep idle:

$$\mathsf{Disabled} \mathrel{\widehat{=}} z \geq 0 \wedge \lceil \underset{\sim}{z} \leq 0 \rfloor\!\rfloor \quad \wedge \quad \lceil\!\lceil \underset{\sim}{\dot{\boldsymbol{v}}} = \boldsymbol{0} \rfloor$$

In addition, similar to triggered subsystems, the variables $\boldsymbol{v}$ of the lines in the subsystem should also be initialised. By default, $\boldsymbol{v}$ are initialised to $\boldsymbol{0}$. If the subsystem is disabled during the initial period, i.e., $z$ keeps non-positive, then $\boldsymbol{v}$ will keep constant:

$$\mathsf{Disabled}' \mathrel{\widehat{=}} \lceil \underset{\sim}{z} \leq 0 \rfloor\!\rfloor \quad \wedge \quad \lceil \underset{\sim}{\boldsymbol{v}} = \boldsymbol{0} \rfloor$$

Otherwise, i.e., $z$ keeps positive initially, the subsystem is enabled during the period:

$$\mathsf{Enabled}' \mathrel{\widehat{=}} \lceil \underset{\sim}{z} > 0 \rfloor\!\rfloor \quad \wedge \quad \bigwedge\nolimits_{i=1}^{n} \mathsf{P}_i \wedge \boldsymbol{v}' = \underset{\sim}{\boldsymbol{v}}(ti'^{-})$$

In summary, the semantics of an enabled subsystem can be denoted by

$$\mathsf{EnSubSys} \mathrel{\widehat{=}} (\mathsf{Enabled}' \vee \mathsf{Disabled}') \,\fatsemi\, (\mathsf{Enabled} \vee \mathsf{Disabled})^*$$

As with triggered subsystems, when creating the causality graph of an enabled subsystem, we add edges from the enabling line to each input line inside the subsystem. An enabled subsystem is well-formed if its causality graph is acyclic.

**Theorem 23.** *If the enabled subsystem is well-formed, then* EnSubSys *is a Simulink process with determined semantics.*

**Proof.** According to Property 3 and Theorem 19 and by induction on the number of iterations of $*$, EnSubSys is a Simulink process. Whether the subsystem is enabled or not depends on the enabling signal $z$. During the enabled period, the subsystem performs as a (well-formed) normal subsystem with determined semantics; during the disabled period, the subsystem keeps quiescent. In summary, the semantics represented by EnSubSys is determined. $\square$

*5.4. Composite systems*

A complex Simulink diagram is usually composed of hierarchical subsystems each encapsulating specific functions. Preferably, the hierarchical structure should be preserved when defining the Simulink semantics. In this section, we show how modular design can be taken into account when defining the HUTP semantics of hierarchical Simulink diagrams.

Overall, we adopt the bottom-up approach for defining the semantics. We start from the normal, enabled and triggered subsystems specified from Sections 5.1 to 5.3. Each well-formed subsystem can be treated as a unit, and we obtain the *causality relation* between its input and output variables by abstracting from the causality graph of the subsystem. For example, the causality graph of the triggered subsystem in Fig. 4 is $\{(z, x), (a, x), (x, y), (y, b)\}$, yielding the causality relation of the subsystem: $\{(z, b), (a, b)\}$. The causality graph of a high-level subsystem is then defined in terms of the causality relations of its component subsystems. We say a high-level subsystem is *well-formed* if (1) all its subsystems are well-formed and (2) the causality graph of the high-level subsystem is acyclic. For a well-formed high-level subsystem, its semantics can be represented by the parallel composition of the subsystems it contains.

**Theorem 24.** *For a well-formed high-level subsystem, if the semantics of its subsystems are determined, then its HUTP semantics is determined and can be represented by a Simulink processes.*

**Proof.** Since the semantics of the subsystems in the high-level subsystem are determined, the theorem can be proved from Theorems 19, 22 and 23 and by the well-formedness of the high-level subsystem (the causality graph of variables constructed from the causality relations of its subsystems is acyclic). $\square$

Subsystems on the same level can be connected together by parallel composition. In this way, a Simulink diagram can be organised as a composite system composed of hierarchical subsystems, from atomic (normal, triggered and enabled) subsystems to high-level subsystems.

**Example 25.** Consider the Simulink diagram on the left of Fig. 2, it is a structured diagram consisting of two well-formed subsystems. Note that although the causality graphs of these two subsystems are $\{(a, x)\}$ and $\{(b, y)\}$, respectively, the causality relations of these two subsystems are empty, as there is no causality relation between $x$ and $y$. Therefore, this diagram, which is a high-level subsystem, is well-formed. Then, we can define its semantics by the parallel composition of the two subsystems:

$$[\![\text{Diagram}]\!]_{\text{HUTP}} \mathrel{\widehat{=}} [\![\text{Subsystem0}]\!]_{\text{HUTP}} \parallel [\![\text{Subsystem1}]\!]_{\text{HUTP}}$$

where $[\![\text{Subsystem0}]\!]_{\text{HUTP}}$ and $[\![\text{Subsystem1}]\!]_{\text{HUTP}}$ are referred in Section 5.1.

## 6. Case study: proving the semantic consistency between Simulink and HCSP

In this section, we illustrate by an example how to prove the semantic consistency between Simulink diagrams and the translated HCSP models based on the simplified HUTP semantics of Simulink. The example is shown in Fig. 2 which is borrowed from Section 6.4 of [23] but with a slight modification. The syntax and semantics of HCSP can be found in [26]. Consider the Simulink diagram in Fig. 2, we set the initial values of variables $y$ and $a$ to 0, then the corresponding HCSP model is shown as follows, by the translation algorithm in [29].

$$[\![\text{Diagram}]\!]_{\text{HCSP}} \mathrel{\widehat{=}} \text{ConSubDiag} \| \text{DisSubDiag}$$

$$\text{ConSubDiag} \mathrel{\widehat{=}} y := 0; a := 0;$$

$$\left( \langle \dot{a} = y, \dot{y} = b \& \textbf{true} \rangle \rhd [\![\ \left( \begin{array}{l} ch_a!a \to \text{skip} \\ ch_b?b \to \text{skip} \end{array} \right) \right)^*$$

$$\text{DisSubDiag} \mathrel{\widehat{=}} t := 0; \left( \begin{array}{l} t\%2 == 0 \to (ch_a?a; x := a + 1); \\ t\%3 == 0 \to (b := x + 1; ch_b!b); \\ \textbf{wait}(1) \end{array} \right)^*$$

where we now use $\parallel$ to denote the parallel operator in HCSP (and use $\parallel_{\text{SIM}}$ to denote parallel operator on Simulink processes). As shown in Fig. 2 (right), the diagram consisting of two subsystems is flattened and then divided into two sub-diagrams:

(1) the continuous sub-diagram ConSubDiag consisting of continuous blocks Int0 and Int1;
(2) the discrete sub-diagram DisSubDiag consisting of discrete blocks Bias0 and Bias1.

The parallel composition of ConSubDiag and DisSubDiag is by communication: ConSubDiag evolves along the ODEs $\dot{a} = y, \dot{y} = b$ and then gets interrupted by the communication along $ch_a$ or $ch_b$. It sends the value of $a$ to DisSubDiag via channel $ch_a$ and receives the value of $b$ from DisSubDiag via channel $ch_b$. Notice that shared variables are not allowed in the context of HCSP.

In the latest version of our tool chain MARS,[1] we implemented a new translation algorithm from Simulink to HCSP, avoiding the use of communication. Hence the Simulink diagram is translated to one sequential HCSP process. For the diagram in Fig. 2 (right), it is translated to the following process:

$$\llbracket \text{Diagram} \rrbracket_{\text{HCSP}} \mathrel{\widehat{=}} tt := 0; a := 0; y := 0; \tag{10}$$

$$\begin{pmatrix} tt\%2 == 0 \rightarrow x := a + 1; \\ tt\%3 == 0 \rightarrow b := x + 1; \\ \langle \dot{a} = y, \dot{y} = b, \dot{tt} = 1 \& \mathbf{true} \rangle \rhd_1 \text{skip} \end{pmatrix}^{+}$$

In the above process, we use $tt$ to denote the time variable, and it is set to 0 initially. The initial values of $a$ and $y$, the output lines of two Integrator blocks, are also set to 0. Then, at each iteration, we execute the discrete blocks according to a topological order. In the discrete sub-diagram in Fig. 2 (right), Bias0 is prior to Bias1 as the input of the latter depends on the output of the former. For Bias0, it can execute (update the output $x$ according to the latest input $a$) if the current time is a multiple of the sample time 2, and it does nothing otherwise, depicted by the HCSP process $tt\%2 == 0 \rightarrow x := a + 1$. The process of Bias1 is similar. After the discrete blocks update their outputs, the whole diagram waits for 1 time unit which is the greatest common divisor of the sample times of Bias0 and Bias1. During the waiting period, the continuous blocks Int0 and Int1 can evolve. The evolution can be described by an ODE $\langle \dot{a} = y, \dot{y} = b, \dot{tt} = 1 \& \mathbf{true} \rangle \rhd_1 \text{skip}$, which indicates the ODE can only evolve for 1 time unit ($\rhd_1$) and then does nothing (skip). The superscript $+$ means the loop iterates at least once.

We will prove the consistency between the example and the new translated HCSP process (10), by comparing the HUTP semantics of Diagram and $\llbracket \text{Diagram} \rrbracket_{\text{HCSP}}$. The HUTP semantics of HCSP has already been given in [23], but it is based on normal hybrid designs. Since defining the new HUTP semantics of HCSP in terms of abstract hybrid processes is not the concern of this paper, we simply present the definitions which are sufficient to express the HUTP semantics of $\llbracket \text{Diagram} \rrbracket_{\text{HCSP}}$ of (10):

- The skip statement terminates immediately having no effect on variables, and it is modelled as the rational identity:

  $$\llbracket \text{skip} \rrbracket_{\text{HUTP}} \mathrel{\widehat{=}} \text{Skip}$$

- The assignment of the value $e$ to a variable $x$ is modelled as setting $x$ to $e$ and keeping all other variables constant if $e$ can be successfully evaluated. Let the alphabet be $\{ti, ti', \mathbf{v}, \mathbf{v}'\}$, where $x \in \mathbf{v}$, then

  $$\llbracket x := e \rrbracket_{\text{HUTP}} \mathrel{\widehat{=}} [\varphi(e) \wedge x' = e \wedge \mathbf{v}' \backslash \{x'\} = \mathbf{v} \backslash \{x\}]$$

  where $\varphi(e)$ specifies the condition by which $e$ can be evaluated.

- The sequential composition $P; Q$ behaves as $P$ first and as $Q$ afterwards:

  $$\llbracket P; Q \rrbracket_{\text{HUTP}} \mathrel{\widehat{=}} \llbracket P \rrbracket_{\text{HUTP}} \mathbin{\mathring{,}} \llbracket Q \rrbracket_{\text{HUTP}}$$

- The alternative $B \rightarrow P$, where $B$ is a Boolean expression, behaves as $P$ if $B$ is true; otherwise, it does nothing:

  $$\llbracket B \rightarrow P \rrbracket_{\text{HUTP}} \mathrel{\widehat{=}} \llbracket P \rrbracket_{\text{HUTP}} \lhd B \rhd \text{Skip}$$

- The recursion $P^+$ can be defined as the least fixed point:

  $$\llbracket P^+ \rrbracket_{\text{HUTP}} \mathrel{\widehat{=}} \llbracket P \rrbracket_{\text{HUTP}}^{+} = \mu X.(\llbracket P \rrbracket_{\text{HUTP}} \quad \vee \quad \llbracket P \rrbracket_{\text{HUTP}} \mathbin{\mathring{,}} X)$$

- A continuous evolution statement $\langle F(\dot{\mathbf{s}}, \mathbf{s}) = 0 \& B \rangle$ says that the process keeps waiting, and meanwhile keeps continuously evolving following the differential equations $F$, until the domain constraint $B$ is violated:

---

[1] https://gitee.com/bhzhan/mars.git.

$$\llbracket \langle F(\dot{\boldsymbol{s}}, \boldsymbol{s}) = 0 \& B \rangle \rrbracket_{\mathsf{HUTP}} \mathrel{\widehat{=}} \mathsf{Exit} \;\vee\; (\mathsf{ODE} \, \mathbin{\raise1pt\hbox{$\mathrm{\scriptstyle 9}$}} \mathsf{Exit})$$

where

$$\mathsf{Exit} \mathrel{\widehat{=}} [\neg B(\boldsymbol{s}) \wedge \boldsymbol{s}' = \boldsymbol{s}]$$

$$\mathsf{ODE} \mathrel{\widehat{=}} \llbracket F(\dot{\boldsymbol{s}}, \boldsymbol{s}) = 0 \wedge B[\underline{\boldsymbol{s}}/\boldsymbol{s}] \rrbracket$$

- $\langle F(\dot{\boldsymbol{s}}, \boldsymbol{s}) = 0 \& B \rangle \trianglerighteq_d \mathsf{P}$ behaves like $\langle F(\dot{\boldsymbol{s}}, \boldsymbol{s}) = 0 \& B \rangle$, if the evolution terminates before $d$ time units. Otherwise, after $d$ time units of evolution, it moves on to execute P:

$$\llbracket \langle F(\dot{\boldsymbol{s}}, \boldsymbol{s}) = 0 \& B \rangle \trianglerighteq_d \mathsf{P} \rrbracket_{\mathsf{HUTP}} \mathrel{\widehat{=}} \mathsf{Exit} \vee (\mathsf{ODE}_{<d} \, \mathbin{\raise1pt\hbox{$\mathrm{\scriptstyle 9}$}} \mathsf{Exit})$$
$$\vee \, (\mathsf{ODE}_d \, \mathbin{\raise1pt\hbox{$\mathrm{\scriptstyle 9}$}} \llbracket \mathsf{P} \rrbracket_{\mathsf{HUTP}})$$

where

$$\mathsf{ODE}_{<d} \mathrel{\widehat{=}} \llbracket F(\dot{\boldsymbol{s}}, \boldsymbol{s}) = 0 \wedge B[\underline{\boldsymbol{s}}/\boldsymbol{s}] \rrbracket_{<d}$$
$$\mathsf{ODE}_d \mathrel{\widehat{=}} \llbracket F(\dot{\boldsymbol{s}}, \boldsymbol{s}) = 0 \wedge B[\underline{\boldsymbol{s}}/\boldsymbol{s}] \rrbracket_d$$

The semantics of other statements, like communication ($ch?x$ and $ch!e$), ODE with communication interruption ($\langle F(\dot{\boldsymbol{s}}, \boldsymbol{s}) = 0 \& B \rangle \trianglerighteq \|_{i \in I} (io_i \longrightarrow \mathsf{P}_i)$) and parallel composition (P∥Q), can also be represented by abstract hybrid processes. Since they are not involved in $\llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}}$ of (10), we will not introduce the details. Note that the above HUTP representations of the HCSP processes are abstract hybrid processes rather than Simulink processes, because $\llbracket \mathsf{skip} \rrbracket_{\mathsf{HUTP}}$ and $\llbracket x := e \rrbracket_{\mathsf{HUTP}}$ are not Simulink processes as they violate healthiness condition $\mathcal{H}_{\mathsf{SIM}}$.

Hence, the HUTP representation of $\llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}}$ of (10) can expand to

$$\llbracket \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}} \rrbracket_{\mathsf{HUTP}}$$
$$= \llbracket tt := 0 \rrbracket_{\mathsf{HUTP}} \, \mathbin{\raise1pt\hbox{$\mathrm{\scriptstyle 9}$}} \llbracket a := 0 \rrbracket_{\mathsf{HUTP}} \, \mathbin{\raise1pt\hbox{$\mathrm{\scriptstyle 9}$}} \llbracket y := 0 \rrbracket_{\mathsf{HUTP}} \mathbin{\raise1pt\hbox{$\mathrm{\scriptstyle 9}$}}$$
$$\left( \begin{array}{l} \llbracket tt\%2 == 0 \to x := a+1 \rrbracket_{\mathsf{HUTP}} \mathbin{\raise1pt\hbox{$\mathrm{\scriptstyle 9}$}} \\ \llbracket tt\%3 == 0 \to b := x+1 \rrbracket_{\mathsf{HUTP}} \mathbin{\raise1pt\hbox{$\mathrm{\scriptstyle 9}$}} \\ \llbracket \langle \dot{a} = y, \dot{y} = b, \dot{tt} = 1 \& \mathbf{true} \rangle \trianglerighteq_1 \mathsf{skip} \rrbracket_{\mathsf{HUTP}} \end{array} \right)^+$$
$$= \underline{a}(ti) = 0 \wedge b = \underline{b}(ti) \wedge x = \underline{x}(ti) \wedge \underline{y}(ti) = 0 \wedge \underline{tt}(ti) = 0$$
$$\wedge \exists n \in \mathbb{N}^+ \cdot ti + n = ti' \wedge \forall k \in \mathbb{N}_{<n} \cdot \underline{tt}(ti + k) = k \wedge$$
$$\left( \underline{x}(ti+k) = \underline{a}(ti+k) + 1 \triangleleft k\%2 = 0 \triangleright \underline{x}(ti+k) = \underline{x}(ti+k-1) \right)$$
$$\wedge \left( \underline{b}(ti+k) = \underline{x}(ti+k) + 1 \triangleleft k\%3 = 0 \triangleright \underline{b}(ti+k) = \underline{b}(ti+k-1) \right)$$
$$\wedge \forall t \in (ti+k, ti+k+1) \cdot \dot{\underline{a}}(t) = \underline{y}(t) \wedge \dot{\underline{y}}(t) = \underline{b}(t) \wedge \dot{\underline{tt}}(t) = 1$$
$$\wedge \dot{\underline{b}}(t) = \dot{\underline{x}}(t) = 0 \wedge \forall t \in (ti, ti') \cdot \underline{a}(t^-) = \underline{a}(t) \wedge \underline{y}(t^-) = \underline{y}(t)$$
$$\wedge RC(\underline{a}, \underline{b}, \underline{x}, \underline{y}, \underline{tt}, ti, ti') \wedge SD(\underline{a}, \underline{b}, \underline{x}, \underline{y}, \underline{tt}, ti, ti') \wedge$$
$$a' = \underline{a}(ti'^-) \wedge b' = \underline{b}(ti'^-) \wedge x' = \underline{x}(ti'^-) \wedge y' = \underline{y}(ti'^-) \wedge tt' = \underline{tt}(ti'^-)$$

According to the above result, $\llbracket \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}} \rrbracket_{\mathsf{HUTP}}$ is $\mathcal{H}_{\mathsf{SIM}}$-healthy, i.e., it is a Simulink process. For the HUTP semantics of the left diagram in Fig. 2, we can get

$$\llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HUTP}} = \llbracket \mathsf{Subsystem0} \rrbracket_{\mathsf{HUTP}} \parallel \llbracket \mathsf{Subsystem1} \rrbracket_{\mathsf{HUTP}}$$
$$\equiv \llbracket \mathsf{Subsystem0} \rrbracket_{\mathsf{HUTP}} \wedge \llbracket \mathsf{Subsystem1} \rrbracket_{\mathsf{HUTP}} \quad (\text{Property 5})$$

where ∥ denotes $\parallel_{\mathsf{SIM}}$, and

$$\llbracket \mathsf{Subsystem0} \rrbracket_{\mathsf{HUTP}} = \underline{a}(ti) = 0 \wedge \exists n \in \mathbb{N} \cdot 2n < ti' - ti \leq 2(n+1)$$
$$\wedge \forall k \in \mathbb{N}_{<n} \cdot \underline{x}(ti+2k) = \underline{a}(ti+2k) + 1$$
$$\wedge \underline{x}(ti+2n) = \underline{a}(ti+2n) + 1$$
$$\wedge \forall t \in (ti+2k, ti+2k+2) \cdot \dot{\underline{a}}(t) = \underline{y}(t) \wedge \dot{\underline{x}}(t) = 0$$
$$\wedge \forall t \in (ti+2n, ti') \cdot \dot{\underline{a}}(t) = \underline{y}(t) \wedge \dot{\underline{x}}(t) = 0$$
$$\wedge \forall t \in (ti, ti') \cdot \underline{a}(t^-) = \underline{a}(t)$$
$$\wedge RC(\underline{a}, \underline{x}, \underline{y}, ti, ti') \wedge SD(\underline{a}, \underline{x}, \underline{y}, ti, ti')$$

$$\llbracket \mathsf{Subsystem1} \rrbracket_{\mathsf{HUTP}} = \underset{\sim}{y}(ti) = 0 \land \exists m \in \mathbb{N} \cdot 3m < ti' - ti \le 3(m+1)$$

$$\land \forall k \in \mathbb{N}_{<m} \cdot \underset{\sim}{b}(ti + 3k) = \underset{\sim}{x}(ti + 3k) + 1$$

$$\land \underset{\sim}{b}(ti + 3m) = \underset{\sim}{x}(ti + 3m) + 1$$

$$\land \forall t \in (ti + 3k, ti + 3k + 3) \cdot \underset{\sim}{\dot{y}}(t) = \underset{\sim}{b}(t) \land \underset{\sim}{\dot{b}}(t) = 0$$

$$\land \forall t \in (ti + 3m, ti') \cdot \underset{\sim}{\dot{y}}(t) = \underset{\sim}{b}(t) \land \underset{\sim}{\dot{b}}(t) = 0$$

$$\land \forall t \in (ti, ti') \cdot \underset{\sim}{y}(t^-) = \underset{\sim}{y}(t)$$

$$\land RC(\underset{\sim}{b}, \underset{\sim}{x}, \underset{\sim}{y}, ti, ti') \land SD(\underset{\sim}{b}, \underset{\sim}{x}, \underset{\sim}{y}, ti, ti')$$

Note that in the above formulas, for brevity, we omit the state variables and replace them by the corresponding output variables, because the state variable and the output variable of an Integrator block are consistent in this setting.

However, $\llbracket \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}} \rrbracket_{\mathsf{HUTP}} \not\equiv \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HUTP}}$ in two aspects: (1) there are more variables involved in $\llbracket \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}} \rrbracket_{\mathsf{HUTP}}$, such as $tt$; (2) the duration of $\llbracket \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}} \rrbracket_{\mathsf{HUTP}}$ is always a positive integer ($ti' - ti = n \in \mathbb{N}^+$) but the one of $\llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HUTP}}$ can be any positive real number. Therefore, we formulate a notion of equivalence under which $\llbracket \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}} \rrbracket_{\mathsf{HUTP}}$ and $\llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HUTP}}$ can be considered to be equivalent.

**Definition 26** *(Equivalence).* Let Diagram be a Simulink diagram, $S$ be the set of variables occurring in Diagram, and $N$ be a positive integer. Then Diagram and its translation to HCSP, i.e., $\llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}}$, are equivalent with respect to $S$ until time $N$, if $\llbracket \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}} \rrbracket_{\mathsf{HUTP}}$ is equivalent to $\llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HUTP}}$ on the variables in $S$ under the constraint $ti' - ti = N$, denoted

$$\llbracket \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}} \rrbracket_{\mathsf{HUTP}} \equiv_{N,S} \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HUTP}}$$

If $\forall N \in \mathbb{N}^+ \cdot \llbracket \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}} \rrbracket_{\mathsf{HUTP}} \equiv_{N,S} \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HUTP}}$, then

$$\llbracket \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HCSP}} \rrbracket_{\mathsf{HUTP}} \equiv_S \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HUTP}}$$

Intuitively, we compare two HUTP representations (with potentially different alphabets) by projecting them on their shared variables first, and then by checking whether the two resulting representations are equivalent within the same duration.

Since we are only concerned with the variables relating to the lines in the Simulink diagram, we let $S \mathrel{\widehat{=}} \{\underset{\sim}{a}, \underset{\sim}{b}, \underset{\sim}{x}, \underset{\sim}{y}\}$. Then we expand $(ti' - ti = N) \land \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HUTP}}$. Before expanding, we reformulate $(ti' - ti = N) \land \llbracket \mathsf{Subsystem0} \rrbracket_{\mathsf{HUTP}}$ to observe the finer-grained behaviour of Subsystem0, i.e., we shorten the step size of Subsystem0 from 2 to 1. Therefore,

$$(ti' - ti = N) \land \llbracket \mathsf{Subsystem0} \rrbracket_{\mathsf{HUTP}} = (ti' - ti = N) \land \llbracket \mathsf{Int0} \rrbracket_{\mathsf{HUTP}} \land \llbracket \mathsf{Bias0} \rrbracket_{\mathsf{HUTP}}$$

can expand to

$$ti' - ti = N \land \underset{\sim}{a}(ti) = 0 \land \forall K \in \mathbb{N}_{<N} \cdot$$

$$\underset{\sim}{x}(ti + K) = \underset{\sim}{a}(ti + K) + 1 \lhd K\%2 = 0 \rhd \underset{\sim}{x}(ti + K) = \underset{\sim}{x}(ti + K - 1)$$

$$\land \forall t \in (ti + K, ti + K + 1) \cdot \underset{\sim}{\dot{a}}(t) = \underset{\sim}{y}(t) \land \underset{\sim}{\dot{x}}(t) = 0$$

$$\land \forall t \in (ti, ti') \cdot \underset{\sim}{a}(t^-) = \underset{\sim}{a}(t) \land RC(\underset{\sim}{a}, \underset{\sim}{x}, \underset{\sim}{y}, ti, ti') \land SD(\underset{\sim}{a}, \underset{\sim}{x}, \underset{\sim}{y}, ti, ti')$$

We can also reformulate $(ti' - ti = N) \land \llbracket \mathsf{Subsystem1} \rrbracket_{\mathsf{HUTP}}$ by shortening the step size from 3 to 1:

$$ti' - ti = N \land \underset{\sim}{y}(ti) = 0 \land \forall K \in \mathbb{N}_{<N} \cdot$$

$$\underset{\sim}{b}(ti + K) = \underset{\sim}{x}(ti + K) + 1 \lhd K\%3 = 0 \rhd \underset{\sim}{b}(ti + K) = \underset{\sim}{b}(ti + K - 1)$$

$$\land \forall t \in (ti + K, ti + K + 1) \cdot \underset{\sim}{\dot{y}}(t) = \underset{\sim}{b}(t) \land \underset{\sim}{\dot{b}}(t) = 0$$

$$\land \forall t \in (ti, ti') \cdot \underset{\sim}{y}(t^-) = \underset{\sim}{y}(t) \land RC(\underset{\sim}{b}, \underset{\sim}{x}, \underset{\sim}{y}, ti, ti') \land SD(\underset{\sim}{b}, \underset{\sim}{x}, \underset{\sim}{y}, ti, ti')$$

Therefore, we can get

$$(ti' - ti = N) \land \llbracket \mathsf{Diagram} \rrbracket_{\mathsf{HUTP}} = (ti' - ti = N) \land \llbracket \mathsf{Subsystem0} \rrbracket_{\mathsf{HUTP}} \land \llbracket \mathsf{Subsystem1} \rrbracket_{\mathsf{HUTP}}$$

which can expand to

$$ti' - ti = N \wedge \underline{a}(ti) = 0 \wedge \underline{y}(ti) = 0 \wedge \forall K \in \mathbb{N}_{<N} \cdot$$

$$\left( \underline{x}(ti + K) = \underline{a}(ti + K) + 1 \lhd K\%2 = 0 \rhd \underline{x}(ti + K) = \underline{x}(ti + K - 1) \right)$$

$$\wedge \left( \underline{b}(ti + K) = \underline{x}(ti + K) + 1 \lhd K\%3 = 0 \rhd \underline{b}(ti + K) = \underline{b}(ti + K - 1) \right)$$

$$\wedge \forall t \in (ti + K, ti + K + 1) \cdot \dot{\underline{a}}(t) = \underline{y}(t) \wedge \dot{\underline{y}}(t) = \underline{b}(t) \wedge \dot{\underline{b}}(t) = \dot{\underline{x}}(t) = 0$$

$$\wedge \forall t \in (ti, ti') \cdot \underline{a}(t^-) = \underline{a}(t) \wedge \underline{y}(t^-) = \underline{y}(t)$$

$$\wedge RC(\underline{a}, \underline{b}, \underline{x}, \underline{y}, ti, ti') \wedge SD(\underline{a}, \underline{b}, \underline{x}, \underline{y}, ti, ti')$$

and it can be proved that $[\![ [\![ \text{Diagram} ]\!]_{\mathsf{HCSP}} ]\!]_{\mathsf{HUTP}} \equiv_{N,S} [\![ \text{Diagram} ]\!]_{\mathsf{HUTP}}$ for any $N \in \mathbb{N}^+$, i.e.,

$$[\![ [\![ \text{Diagram} ]\!]_{\mathsf{HCSP}} ]\!]_{\mathsf{HUTP}} \equiv_S [\![ \text{Diagram} ]\!]_{\mathsf{HUTP}}$$

Comparing with the proof of semantic consistency based on normal hybrid designs in Section 6.4 of [23], the proof in this section is more formal and concise, because we adopt Simulink processes as the semantic foundation for Simulink. After all, Simulink processes, a subset of abstract hybrid processes, are much simpler than normal hybrid designs, and arguably more suitable for defining the Simulink semantics.

## 7. Related works

There is a large amount of existing work on formal semantics of Simulink and translation of Simulink models to other languages as part of various system design workflows. Initial work focused on the discrete part of Simulink. Tripakis et al. described a translation of discrete Simulink to the data-flow language Lustre [21]. Dragomir et al. translate Simulink's hierarchical block diagrams into an algebra of transformers connected together via series, parallel and feedback operators [8]. In follow-up work, Preoteasa et al. proved the determinacy of these translations, showing that the semantics of the resulting model does not depend on the various choices made during translation [19]. The proofs are formalised in the interactive theorem prover Isabelle/HOL. These works resulted in the Refinement Calculus of Reactive Systems (RCRS) toolset for modelling and reasoning about reactive systems [9]. Compared to RCRS, we consider in addition the continuous blocks in Simulink (without discretisation), triggered and enabled subsystems, and multi-rate systems, establishing the determinacy of the semantics in this more general setting. Ye et al. present a compositional assume-guarantee reasoning framework to provide a purely relational mathematical semantics for discrete time Simulink diagrams, and then to verify the diagrams against the contracts in the same semantics in UTP [25]. However, the work only captures single sampling rate Simulink models, while multi-rate models are not supported by the reasoning framework.

Existing work take different approaches to formalise the semantics of continuous blocks in Simulink. Some focused on describing in detail how ODE solving and zero-crossing detection are performed. For example, Bouissou et al. gave an operational semantics for both continuous-time and discrete-time blocks in Simulink that emphasises the details of numerical simulation [4]. Other works focused on first giving a mathematically precise definition of the semantics, and then possibly consider connections to numerical simulation results. Lee and Zheng [13] detailed the issues that arise when defining semantics of hybrid systems. They described a semantic model where each signal is given by a function from tags to states, where each tag consists of a time and an integer, thus able to describe multiple computation steps at a single time point. While they give semantics for HyVisual (part of the Ptolemy framework), many of the ideas apply to Simulink as well. Benveniste et al. [3] gave an alternative semantic model based on the theory of non-standard analysis, which is able to handle cascades of zero-crossings resulting continuous triggers in the system. Based on this model, Bourke et al. proposed Zélus [6], extending a Lustre-like synchronous language with ODEs. The Zélus language is then used to give semantics to a large collection of Simulink blocks [5]. Compared to semantics based on tags and non-standard analysis, we use a simpler semantic model based on functions from real numbers. On the other hand we currently do not consider continuous triggers and hence cascaded zero-crossings.

Several existing work connected translation from Simulink to models of hybrid systems with verification using either model checking or theorem proving. Librenz et al. [14] proposed a translation from Simulink diagrams to differential dynamic logic [18], for verification in the KeYmaera X tool. The work of Agrawal et al. [1] provides a characterisation of Simulink as a translation or interpretation as hybrid automata. Minopoli et al. [17] translate Simulink into SpaceEx models. Our work originated from the initial translation method of Zou et al. [29] from Simulink models into HCSP, within the MARS platform for analysis, verification and simulation of hybrid systems [7], and the theory of higher-order UTP [23]. We then proposed a unified graphical co-modelling, analysis and verification of CPSs by combining AADL and Simulink/Stateflow [24] based on these works. Compared to these previous works, our method yields simpler translated results, and permits easier proofs of semantic determinacy as well as correctness of translation.

## 8. Conclusions and future works

Reflecting the complexity of cyber-physical systems design, the semantics of Simulink models is highly complex. In the aim of breaking down this complexity, we abstract the meaning of hierarchical Simulink diagrams into logically and

mathematically comprehensible terms, by employing a notion of Simulink processes, a subset of abstract hybrid processes, defined in HUTP. Based on our HUTP semantics of Simulink, we construct a framework for proving Simulink diagrams consistent with their translation into HCSP. We provide a case study that illustrates and justifies this translation.

*Future works.* As mentioned in Section 5.1, existing translation procedures from Simulink to HCSP begin by flattening some subsystems, which undermines the modular design of Simulink diagrams. Therefore, we will consider improving the translation algorithm to take modular design into account. In this paper, we only introduce parts of the new HUTP semantics of HCSP when proving the semantic consistency in Section 6. In the future, we will provide the complete definition of the new HUTP representation of HCSP, covering communication and ODE with communication interrupts, and prove its consistency with the operational semantics of HCSP. Finally, based on the HUTP representation, we will provide a systematic proof (not just by examples) for the correctness of the translation algorithm from Simulink to HCSP.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Acknowledgement

## References

[1] A. Agrawal, G. Simon, G. Karsai, Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations, Electron. Notes Theor. Comput. Sci. 109 (2004) 43–56.
[2] A. Benveniste, T. Bourke, B. Caillaud, M. Pouzet, Non-standard semantics of hybrid systems modelers, J. Comput. Syst. Sci. 78 (2012) 877–910.
[3] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.B. Raclet, P. Reinkemeier, A.L. Sangiovanni-Vincentelli, W. Damm, T.A. Henzinger, K.G. Larsen, Contracts for system design, Found. Trends Electron. Des. Autom. 12 (2018) 124–400.
[4] O. Bouissou, A. Chapoutot, An operational semantics for Simulink's simulation engine, SIGPLAN Not. 47 (2012) 129–138.
[5] T. Bourke, F. Carcenac, J. Colaço, B. Pagano, C. Pasteur, M. Pouzet, A synchronous look at the Simulink standard library, ACM Trans. Embed. Comput. Syst. 16 (2017) 176:1–176:24.
[6] T. Bourke, M. Pouzet, Zélus: a synchronous language with ODEs, in: 16th International Conference on Hybrid Systems: Computation and Control, HSCC, 2013, pp. 113–118.
[7] M. Chen, X. Han, T. Tang, S. Wang, M. Yang, N. Zhan, H. Zhao, L. Zou, MARS: a toolchain for modelling, analysis and verification of hybrid systems, in: Provably Correct Systems, in: NASA Monographs in Systems and Software Engineering, Springer, 2017, pp. 39–58.
[8] I. Dragomir, V. Preoteasa, S. Tripakis, Compositional semantics and analysis of hierarchical block diagrams, in: 23rd International Symposium on Model Checking Software, SPIN, Springer, 2016, pp. 38–56.
[9] I. Dragomir, V. Preoteasa, S. Tripakis, The refinement calculus of reactive systems toolset, Int. J. Softw. Tools Technol. Transf. 22 (2020) 689–708.
[10] S. Foster, A. Cavalcanti, S. Canham, J. Woodcock, F. Zeyda, Unifying theories of reactive design contracts, Theor. Comput. Sci. 802 (2020) 105–140.
[11] D.D. Gajski, S. Abdi, A. Gerstlauer, G. Schirner, Embedded System Design: Modeling, Synthesis, Verification, Springer-Verlag, 2009.
[12] C.A.R. Hoare, J. He, Unifying Theories of Programming, Prentice Hall, Englewood Cliffs, 1998.
[13] E.A. Lee, H. Zheng, Operational semantics of hybrid systems, in: Hybrid Systems: Computation and Control, 8th International Workshop, HSCC, 2005, pp. 25–53.
[14] T. Liebrenz, P. Herber, S. Glesner, Deductive verification of hybrid control systems modeled in Simulink with KeYmaera X, in: International Conference on Formal Engineering Methods, ICFEM, Springer, 2018, pp. 89–105.
[15] Z. Manna, A. Pnueli, Verifying hybrid systems, in: R.L. Grossman, A. Nerode, A.P. Ravn, H. Rischel (Eds.), Hybrid Systems, Springer, Berlin, Heidelberg, 1993, pp. 4–35.
[16] MathWorks, Simulink® User's Guide, http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf, 2013.
[17] S. Minopoli, G. Frehse, SL2SX translator: from Simulink to SpaceEx models, in: 19th International Conference on Hybrid Systems: Computation and Control, HSCC, 2016, pp. 93–98.
[18] A. Platzer, Differential dynamic logic for hybrid systems, J. Autom. Reason. 41 (2008) 143–189.
[19] V. Preoteasa, I. Dragomir, S. Tripakis, Mechanically proving determinacy of hierarchical block diagram translations, in: 20th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI, 2019, pp. 577–600.
[20] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pac. J. Math. 5 (1955) 285–309.
[21] S. Tripakis, C. Sofronis, P. Caspi, A. Curic, Translating discrete-time Simulink to Lustre, ACM Trans. Embed. Comput. Syst. 4 (2005) 779–818.
[22] S. Wang, N. Zhan, L. Zou, An improved HHL prover: an interactive theorem prover for hybrid systems, in: International Conference on Formal Engineering Methods, ICFEM, Springer, 2015, pp. 382–399.
[23] X. Xu, J.P. Talpin, S. Wang, B. Zhan, N. Zhan, Semantics foundation for cyber-physical systems using higher-order UTP, ACM Trans. Softw. Eng. Methodol. (2022).
[24] X. Xu, S. Wang, B. Zhan, X. Jin, J.P. Talpin, N. Zhan, Unified graphical co-modeling, analysis and verification of cyber-physical systems by combining AADL and Simulink/Stateflow, Theor. Comput. Sci. 903 (2022) 1–25.
[25] K. Ye, S. Foster, J. Woodcock, Compositional assume-guarantee reasoning of control law diagrams using UTP, in: From Astrophysics to Unconventional Computation, Springer, 2020, pp. 215–254.
[26] N. Zhan, S. Wang, H. Zhao, Formal Verification of Simulink/Stateflow Diagrams (a Deductive Approach), Springer, 2017.
[27] L. Zou, J. Lv, S. Wang, N. Zhan, T. Tang, L. Yuan, Y. Liu, Verifying Chinese train control system under a combined scenario by theorem proving, in: Verified Software: Theories, Tools, Experiments, VSTTE, 2013, pp. 262–280.

[28] L. Zou, N. Zhan, S. Wang, M. Fränzle, Formal verification of Simulink/Stateflow diagrams, in: 13th International Symposium on Automated Technology for Verification and Analysis, ATVA, Springer, 2015, pp. 464–481.

[29] L. Zou, N. Zhan, S. Wang, M. Fränzle, S. Qin, Verifying Simulink diagrams via a hybrid Hoare logic prover, in: International Conference on Embedded Software, EMSOFT, IEEE, 2013, pp. 1–10.