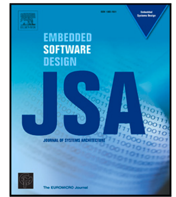
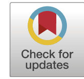
Journal of Systems Architecture 174 (2026) 103724

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc





Formal semantics for hierarchical Simulink diagrams in Isabelle/HOL ☆

Yuzhen Qi [a], Shuling Wang [b,*], Xing Li [a], Bohua Zhan [c], Naijun Zhan [d]

[a] Key Laboratory of System Software (Chinese Academy of Sciences), Institute of Software, CAS, Beijing, 100190, China
[b] National Key Laboratory of Space Integrated Information Systems, Institute of Software, CAS, Beijing, 100190, China
[c] Huawei Technologies Co., Ltd., Shenzhen, 518129, China
[d] School of Computer Science, Peking University, Beijing, 100871, China

ARTICLE INFO

Keywords:
Simulink
Discrete
Continuous
Denotational and operational semantics
Consistency
Isabelle/HOL

ABSTRACT

Simulink is widely used in the design of safety-critical embedded systems, including avionics and automotive applications. While it offers simulation for model validation, formal verification remains essential to rigorously ensure system correctness. Existing approaches often translate Simulink diagrams into third-party formal models, however, the lack of a rigorously defined semantics for Simulink can lead to inconsistencies between the original diagrams and their translated formal counterparts. In this paper, we present a formal semantic foundation for a core subset of Simulink by defining both denotational and operational semantics. The denotational semantics offers a mathematical interpretation of the diagram's input–output behavior, faithfully capturing its hierarchical structure. In contrast, the operational semantics specifies the concrete execution of Simulink diagrams, resolving block execution order, solving continuous dynamics, and coordinating hybrid discrete-continuous interactions. Both semantics have been fully formalized in Isabelle/HOL, and we have established their consistency by proving the existence and uniqueness of the timed state trajectories defined by the denotational semantics. Furthermore, to facilitate application, we developed a translator that automatically converts Simulink graphical diagrams into their Isabelle representation. Our formal semantics supports the rigorous analysis of Simulink diagram properties, as demonstrated through a PID control example. The semantics also establishes a foundation for validating simulation results and ensuring consistency between Simulink models and other formal models, thus enabling sound verification.

1. Introduction

Simulink has been widely used in model-driven design of embedded systems, particularly in safety-critical domains such as avionics and automotive systems. It provides efficient simulation for model validation, however, simulation-based analysis remains inherently incomplete as it can only examine a finite set of scenarios. In contrast, formal verification checks properties against all possible inputs, making it indispensable for safety-critical applications. The commercial tool Simulink Design Verifier (SLDV) enables property verification but is limited to discrete-time systems. Crucially, its lack of formal semantics for both models and properties may yield unsound results, as demonstrated by the bugs and conflicting verification results in [1]. In academia, an alternative approach translates Simulink diagrams into third-party formal models for verification [2–4]. However, most existing work fail to preserve semantic consistency between the original diagram and its formal representation, compromising the validity of verification results for original Simulink diagrams. Establishing such consistency requires precise formal semantics for Simulink. Current efforts either restrict Simulink semantics to a limited subset of Simulink (e.g., excluding continuous-time semantics or hierarchical subsystems [5,6]), or lack machine-checked formalization [7,8].

In this paper, we establish a formal semantic foundation for a core subset of Simulink, covering both discrete/continuous blocks, hierarchical subsystems, and their composition. We begin by defining a denotational semantics for Simulink diagrams, which provides a mathematical interpretation of the diagrams as sets of timed states that record the evolution of all variables with respect to time. The denotational semantics preserves the graphical hierarchy of Simulink diagrams, and captures the input–output relationships of blocks. We then define an operational semantics for Simulink diagrams, which specifies the actual execution of Simulink diagrams, in the form of timed state transitions. It determines the execution order of blocks by

☆ This work has been partially funded by the National Key R&D Program of China under grant No. 2022YFA1005103, the National Natural Science Foundation of China under grant No. 62432005 and No. 62572459.
* Corresponding author.
E-mail address: wangsl@ios.ac.cn (S. Wang).

https://doi.org/10.1016/j.sysarc.2026.103724
Received 24 September 2025; Received in revised form 25 January 2026; Accepted 25 January 2026
Available online 4 February 2026
1383-7621/© 2026 Elsevier B.V. All rights reserved, including those for text and data mining, AI training, and similar technologies.

sorting them based on data dependencies, solves continuous dynamics by formulating all integrator blocks as systems of ordinary differential equations, coordinates hybrid discrete-continuous interactions, and moreover, enables modular execution by processing subsystems recursively while preserving their hierarchical structures.

We have formalized both semantics in Isabelle/HOL and furthermore proven their consistency, establishing the existence and uniqueness of the timed state trajectories defined by the denotational semantics. Specially, the *existence* theorem states that the concrete timed states produced by the operational semantics are contained within the denotational semantics set, while the *uniqueness* theorem states that the denotational semantics set contains exactly one unique timed state for any given inputs. An immediate corollary of these theorems is the determinism of the operational semantics across all block execution orders that adhere to data dependencies. All above definitions and proofs related to the two semantics are formalized in Isabelle/HOL. To facilitate practical application, we developed a translator that automatically converts Simulink graphical diagrams into their Isabelle representation. The proposed semantics then enables the formal analysis of Simulink diagrams in Isabelle. As a case study, we apply it to formalize a discrete PID controller and verify a key safety property that the continuous plant converges to the setpoint under the control. Furthermore, the semantics can be used to prove consistency between Simulink models and other formal representations, thereby ensuring sound verification; it also offers a basis for validating numerical simulation results.

This work also constitutes a key component of MARS [9], an integrated framework for modeling, analyzing and verifying hybrid systems. The MARS toolchain begins with a graphical model combining AADL and Simulink/Stateflow, which is then transformed into a formal HCSP model [10] for verification, and finally, generates C code from the verified HCSP model. A central challenge is to ensure the correctness of the translation from Simulink/Stateflow to HCSP. In previous work [11], we formalized the syntax and semantics of Stateflow models in Isabelle/HOL. As future work, we plan to establish the semantic consistency between Simulink diagrams and their corresponding HCSP models by combining the semantics of Simulink developed in this paper with the existing semantics of Stateflow from [11].

After presenting the related work, the remainder of this paper is organized as follows: Section 2 gives a brief introduction of Simulink. Section 3 presents the syntax of Simulink blocks and diagrams. Section 4 and Section 5 present the denotational semantics and the operational semantics of Simulink respectively. Section 6 proves the consistency between the two semantics. Section 7 presents the main implementation issues in Isabelle/HOL and Section 8 illustrates our approach via three case studies. Section 9 discusses the limitation of our approach and presents the possible extensions. Section 10 concludes with the future work.

1.1. Related work

Earlier work focus on defining the semantics of the discrete-time subset of Simulink [5,6,12–14]. Later work translate Simulink to formal models for verification. Agrawal et al. [15] translated Simulink diagrams to hybrid automata using graph transformation rules. Filipovikj et al. [16] provided an execution order preserving transformation of Simulink blocks into UPPAAL for statistical model checking. Chen et al. [17] applied a Timed Interval Calculus specification language to complement Simulink with formal verification. Zou et al. [2] proposed a translation from Simulink to Hybrid CSP and used Hybrid Hoare logic for verification. Herber et al. [3] proposed a transformation from Simulink to differential dynamic logic and applied KeYmaera X theorem prover for verification. Bourke et al. [4] formalized Simulink diagrams through Zélus [18] for formal analysis, a synchronous language extending Lustre with ordinary differential equations (ODEs). While these approaches account for both discrete and continuous blocks, they give an indirect characterization of Simulink through the semantics of translated formal models, for which the correctness of the translation process is difficult to verify and furthermore the applicability is constrained to the target formalism. Our approach avoids this because it provides a direct formal semantics for Simulink diagrams, without translation to an intermediate formalism. The denotational semantics precisely defines constraints on system states, while the operational semantics defines state transitions. Both are defined directly over the diagrams rather than rely on other translations.

Bouissou et al. [19] present an operational semantics for the simulation engine of Simulink by formalizing continuous and discrete blocks with numerical semantics. In contrast to their approach, our semantics provides an exact solution of continuous dynamics independent of any specific simulation algorithms, which avoids numerical errors and preserves analytical properties of continuous systems that may be lost during numerical discretization. We also plan to formalize a numerical semantics which can provide a bound for difference of operational semantics with itself in Isabelle/HOL. Xu et al. [20] formalize a denotational semantics for hierarchical Simulink diagrams using the hybrid Unifying Theories of Programming (UTP) and prove its determinacy. Their determinacy result aligns with our theorems concerning the existence and uniqueness of our denotational semantics, both requiring the loop-free condition and the Lipschitz condition for the underlying ODEs. The difference is that, their determinacy is proved via induction on an ordering of signal updates derived from input–output relations, a notion weaker than our refined data dependency, whereas our approach constructs an executable operational semantics and furthermore all proofs are mechanized. Liu et al. [7,8] proposed a contract-based semantics for Simulink diagrams based on a set of contract composition operators and a verification framework via refinement calculus. However, neither of them provides machine-checked formalization of their semantics, which given the inherent complexity of Simulink's behavior, is essential for the well-definedness and consistency of the semantic definitions.

Dragomir et al. [21–25] defined the execution semantics of Simulink hierarchical block diagrams via the series, parallel and feedback operators in Refinement Calculus of Reactive Systems (RCRS). This line of work is most closely related to ours. They define a fine-grained input–output dependency relation which, for Simulink blocks, coincides with our refined notion of data dependency between blocks. Their approach also disallows instantaneous dependency loops. Based on the dependency relation, blocks are first sorted, then the parallel and series composition operators are applied accordingly, with a final decision on whether to apply feedback composition based on the existence of (non-instantaneous) cycles. It is noteworthy that their parallel operator implicitly permits all valid execution permutations, which aligns with the role of deep permutation in our semantics. Consequently, as demonstrated in their work [24], the determinacy of the semantics is formally proven in Isabelle/HOL. However, RCRS only addresses the discrete case in essence as the ODEs of continuous blocks are approximated by fixed-step Euler assignments.

2. An overview of simulink

In Simulink [26], blocks serve as the basic units for constructing Simulink models and can be connected to form block diagrams. Blocks can also be grouped into subsystems, which may recursively contain subsystems inside to form a hierarchical structure. A Simulink diagram is thus composed of connected blocks and subsystems. Fig. 1 models a continuous physical system and its corresponding PID controller, which form a closed-loop system. In the diagram, the continuous plant is modeled as an inherently unstable first-order system, where the `Integ` block is driven by positive feedback from `Sum1`. To stabilize this, the `Discrete pid` subsystem functions as a feedback controller that computes a correction signal c based on the error e—the difference between the constant setpoint and the plant's sampled output. Within this discrete

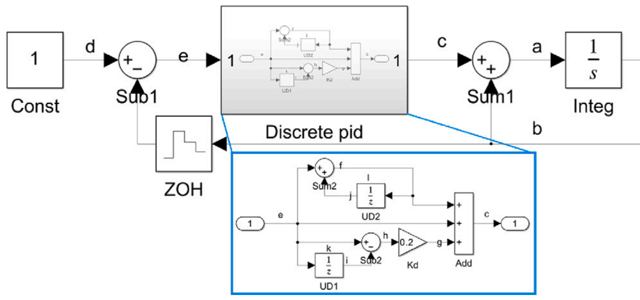


Fig. 1. A Simulink diagram of a discrete PID control example.

controller, the Unit Delay block UD1 serves as a differentiator by holding the previous value to estimate the rate of change, while the feedback loop between Sum2 and UD2 acts as a discrete integrator that sums past errors. These derivative and integral components are combined with the proportional error and scaled by Kd to generate the control signal, which adjusts the plant's input dynamics to counteract the instability and force the system state b to converge to the target setpoint. We will use this example as a running case study throughout this paper.

Blocks, subsystems and diagrams A block has inputs and outputs, as well as internal states. Its dynamic behavior is governed by two functions: an output function that calculates the outputs and a state update function that computes the next state values, both using current inputs and the previous states. The sample time of a block defines how frequently the computation of outputs and states is performed. Based on sample times, blocks are categorized into two types: discrete blocks for $st>0$, that execute only at the integer multiples $k\cdot st$ for $k \in \mathbb{N}$; and continuous blocks for $st=0$, that execute continuously over time. In Fig. 1, the discrete Unit Delay block UD1 delays the input with one time step, composing a differentiator in PID module, while the continuous integrator block Integ representing ordinary differential equations (ODE) models the evolution of a positive feedback system. Non-integrator blocks may also be configured as continuous (termed *calculational blocks*), e.g. the Sum1 block in Fig. 1. A calculational block refers to a stateless continuous block whose output at any time instant is the result of applying a mathematical function to its inputs. Blocks in the categories listed in Table 1, such as Math Operations, Logical Operators, Relational Operators, Signal Routing, and Discontinuities, are all calculational blocks.

Blocks are connected through signals, which are represented as variables and interpreted as mappings from time to values, called *timed state* in our semantics. If a connection exists from block A to block B, and B depends on the outputs of A, then A must execute before B. When the data dependency relations form a loop, the behavior of the diagram becomes ill-defined. Blocks such as Unit Delay and Integrator can break the mutual dependency in loops, as their outputs are depending on internal states rather than immediate inputs. In Fig. 1, the entire diagram forms a feedback control loop: at the beginning of each time step, the Discrete pid takes the difference between Const and Integ, computes the new control signal, and sends it back to Integ for use in the next time step.

A hybrid discrete-continuous diagram contains both discrete and continuous blocks, and its time step is determined by the *fundamental sample time*, i.e. the greatest common divisor of all the discrete blocks within the diagram. Throughout the paper, we assume that the fundamental sample time is 1, without loss of generality. For diagrams with non-integer sample times, we show in Section 8.2 how they can be transformed into equivalent diagrams with an integer fundamental sample time through time scaling. A subsystem consists of blocks and may be organized hierarchically, containing other subsystems inside. We consider three types of subsystems in this paper: atomic subsystems, that execute a set of encapsulated blocks as a single unit; enabled and triggered subsystems, both of which are conditional subsystems and will execute when the corresponding control signals hold. An atomic subsystem is illustrated in Fig. 1, whereas examples of enabled and triggered subsystems are provided in Section 8.2.

Simulink offers a comprehensive library of predefined blocks, covering a wide range of functionalities for modeling dynamic systems. These blocks enable users to construct complex models efficiently without starting from scratch. The syntax and semantics defined in this paper support a wide subset of Simulink blocks, as presented in Table 1.

3. Syntax of simulink blocks and diagrams

In this section, we will define the formal syntax of Simulink blocks and diagrams. All definitions and proofs given in this paper have been formalized in Isabelle/HOL, but for ease of understanding, we write them in usual mathematical notations. The syntax for Simulink diagrams, blocks and subsystems is defined in Box I.

The syntax definition uses boldface to represent sequences. A sequence may be an empty sequence ϵ or recursively a concatenation of an element a with a sequence of elements $\mathbf{a}$, denoted $a\cdot\mathbf{a}$. A Simulink diagram is formally defined as a sequence of blocks, denoted by $\mathbf{B}$. These blocks fall into three categories: discrete blocks dB, continuous blocks cB, and subsystems Sub.

A discrete block is represented as a tuple $(\mathbf{i}, \mathbf{o}, \mathbf{s}, \mathbf{s}_0, st, \mathbf{di}, \mathbf{f}, \mathbf{g})$, where: $\mathbf{i}$, $\mathbf{o}$ and $\mathbf{s}$ denote the input, output, and state variables of the block, respectively, $\mathbf{s}_0$ specifies the initial values for states, st is the sample time, $\mathbf{di}$ is a sequence of Boolean values indicating whether the corresponding input is instantaneously required by the outputs, $\mathbf{f}$ and $\mathbf{g}$ two sequences of functions that map states and inputs to the outputs and updated states, respectively. Take the Unit Delay block UD1 in Fig. 1 as an example, which has input e, output i, state variable k, initial state 1 and sample time 1, and is formally defined as $(e, i, k, 1, 1, 0, \lambda s.\lambda i.s, \lambda s.\lambda i.i)$. In this case, the input is not instantaneously required by the output. As specified by the functions, the output takes the value of the previous state k, while the state is updated to the current input value e.

Continuous blocks comprise calculational blocks and integrators. Both are formally defined by a tuple of the form $(\mathbf{i}, \mathbf{o}, \mathbf{s}_0, \mathbf{f}, isC)$, where $\mathbf{i}$ and $\mathbf{o}$ denote the input and output variables, respectively, consistent with their definitions for discrete blocks, $\mathbf{s}_0$ specifies the initial values of the states, $\mathbf{f}$ is a sequence of functions defining the output behavior, and isC is a Boolean value indicating whether the block is calculational. If isC is true, the block is calculational and stateless. In this case, $\mathbf{s}_0$ is disregarded, and $\mathbf{f}$ is a sequence of functions that map inputs directly to outputs. If isC is false, the block is an integrator: its outputs coincide with its states, $\mathbf{s}_0$ gives their initial values, and $\mathbf{f}$ defines functions that map the inputs to the derivatives of the outputs. For example, the calculational block Sum1 in Fig. 1, with inputs c and b and output a, implements the equation $a = c + b$, and is therefore defined by the tuple: $(c\cdot b, a, -, \lambda i_1\ i_2.\ i_1 + i_2, 1)$; the Integrator block Integ has input a, output b, and initial value 0.5. It implements the ordinary differential equation $\dot{b} = a$ via the function $\lambda i.\ i$. It is correspondingly defined as: $(a, b, 0.5, \lambda i.\ i, 0)$. Note that the output a of Sum1 is connected to the input of Integ, which is also denoted by a in this context.

A subsystem block is represented as a tuple $(\mathbf{i}, \mathbf{o}, \mathbf{B}, st, Sty)$, where $\mathbf{i}$ and $\mathbf{o}$ are defined as above, $\mathbf{B}$ the sequence of blocks contained within the subsystem, st the subsystem's sample time, which is equal to the greatest common divisor of all internal blocks' sample times, and Sty the type of the subsystem. Atomic subsystems require no additional parameters, whereas enabled and triggered subsystems are parameterized as follows: Enab $sre\ sig$, with sig indicating whether it is enabled to execute and sre indicating whether to reset the states of the subsystem upon each enabling; Trig $tty\ sig$, with $tty \in \mathbb{N}$ a natural number representing the trigger type (0 for rising edge, 1 for falling edge and 2 for either) and sig the variable providing the trigger signal. Take Discrete pid in Fig. 1 as an example. It is an atomic subsystem

Table 1
The subset of Simulink blocks supported by our approach.

Category	Blocks
Sources	Constant, Clock, Sine, Discrete/Continuous Pulse Generator
Continuous	Integrator (Limited), State-Space
Discontinuities	Saturation, Coulomb and Viscous Friction, Dead Zone
Discrete	Discrete PID Controller, Unit Delay
Logical & Bit	Logical Operators, Relational Operators
Math Operations	Add, Bias, Gain, MinMax, Abs, Product, Sqrt, Square, Sign, Sine Wave function
Signal Routing	Switch, Multiport Switch
Signal Attributes	IC (Initial Condition for signals)
Subsystems	Atomic/Enabled/Triggered Subsystems

$$
\begin{array}{llll llll}
\text{Diagrams} & Diag & := & \mathbf{B} & \text{Blocks} & B & := & dB \mid cB \mid Sub \\
\text{Discrete} & dB & := & (\mathbf{i},\mathbf{o},\mathbf{s},\mathbf{s}_0,st,\mathbf{di},\mathbf{f},\mathbf{g}) & \text{Continuous} & cB & := & (\mathbf{i},\mathbf{o},\mathbf{s}_0,\mathbf{f},isC) \\
\text{Subsystem} & Sub & := & (\mathbf{i},\mathbf{o},\mathbf{B},st,Sty) & Sty & := & \multicolumn{2}{l}{\mathsf{Atomic} \mid \mathsf{Enab}\ sre\ sig \mid \mathsf{Trig}\ tty\ sig} \\
\text{Functions} & \mathbf{f}\ni\mathbf{f}_o & : & \mathbb{R}^{|\mathbf{s}|}\times\mathbb{R}^{|\mathbf{i}|}\to\mathbb{R} & \mathbf{g}\ni\mathbf{g}_s & : & \mathbb{R}^{|\mathbf{s}|}\times\mathbb{R}^{|\mathbf{i}|}\to\mathbb{R} &
\end{array}
$$

Box I.

with input e and output c, consisting of the blocks within the blue box, with sample time 1, thus can be represented as $(e, c, \mathsf{Sum2}\cdot\mathsf{UD1}\cdot\mathsf{Sub2}\cdot\mathsf{UD2}\cdot\mathsf{Kd}\cdot\mathsf{Add}, 1, \mathsf{Atomic})$.

A Simulink diagram is composed of interconnected blocks and subsystems, where the connections between them are reflected by their inputs and outputs. Specifically, if the output of one block serves as the input of another, these blocks are considered connected. Thus, it is natural to represent a Simulink diagram as a sequence of blocks, capturing both the components and their interconnections.

The syntax of blocks and diagrams defined above must satisfy a set of well-formedness conditions ensuring that a structure $Diag$ defined within this syntax corresponds to a valid Simulink diagram. These conditions are collectively denoted by the predicate $wf(Diag)$. The conditions include the outputs of blocks should be distinct, the outputs and states must match the output functions and state functions in length respectively, *etc*. In addition to the well-formedness conditions, we impose a syntactic restriction that subsystems contain only discrete blocks. This design choice ensures modular composability in the semantics: As continuous integrators must be solved collectively via a global ODE system, consistent with Simulink's solving strategy, subsystems containing integrators are therefore required to be flattened before semantic evaluation. We avoid this unfolding in semantics, instead, the syntax restriction can be relaxed by pre-flattening any subsystem containing continuous integrators during the diagram construction phase. With the proposed syntax, we have represented a wide range of Simulink blocks, as listed in Table 1.

Discussion of limitations Some Simulink features are not currently handled in our formalization. These mainly include blocks involving differentiation (Derivative blocks), zero-crossing, dynamics-dependent blocks such as variable transport/time delay, dashboard components, model verification blocks, certain complex subsystem types, messages and data store memory. Among these, zero-crossing detection has been separately discussed in Section 9; Model verification blocks serve a purpose distinct from the modeling of systems, as verification can be conducted within Isabelle after translation; Derivative blocks are excluded due to the well-known challenges in guaranteeing well-defined solutions under general assumptions; Blocks such as dashboard are primarily related to user interface and visualization, which fall outside the focus of our behavioral modeling. Importantly, most remaining features could be incorporated through syntactic and semantic extensions. However, to maintain focus and clarity in presenting our core contribution, we have limited the scope accordingly.

4. Denotational semantics

The denotational semantics of a Simulink diagram is interpreted over timed states $h : \mathrm{Var} \to \mathbb{R} \to \mathrm{Val}$, which assign to each variable $x \in \mathrm{Var}$ (including inputs, outputs and state variables of the diagram) a value over the whole time interval $[0, \infty)$, i.e. $h(x)(t)$ denotes the value of variable x at time t. Given a time domain $\mathcal{T}$, the denotational semantics of a diagram over $\mathcal{T}$ is defined as the set of all timed states that satisfy the behavioral constraints of the diagram throughout $\mathcal{T}$. The structure of $\mathcal{T}$ reflects the hybrid discrete-continuous nature of Simulink diagrams. Specifically, since discrete blocks execute at isolated time instants, their behavior is captured at single points n (where $n \in \mathbb{N}$). In contrast, continuous blocks evolve over intervals and may also be influenced by discrete events at endpoints. Therefore, the semantic definition must separately account for both single time instants n and continuous intervals, such as $(n, n+1)$ or $[n, n+1]$, to accurately model the system's behavior over each type of domain.

Formally, given a diagram $Diag$, its denotational semantics over a time domain $\mathcal{T}$ is denoted by $[\![Diag]\!]_{\mathcal{T}}^{(ena,res)}$, where $\mathcal{T}$ is either a discrete time instant $n \in \mathbb{N}$, or a continuous interval such as $(n, n+1)$ or $[n, n+1]$. The denotational semantics of diagrams or blocks over the set $\mathcal{T}_1 \cup \mathcal{T}_2$ is equal to the intersection of its denotational semantics over $\mathcal{T}_1$ and over $\mathcal{T}_2$. Thus, we can compose the denotational semantics over $[0, n]$ from the denotational semantics over discrete time instants and continuous intervals. The Boolean flags ena and res indicate whether the diagram is enabled and whether a state reset is required, respectively. By default, the top-level diagram executes with $ena = True$ and $res = False$. These parameters are updated when execution encounters enabled or triggered subsystems within $Diag$, as presented in Fig. 2, which provides the full denotational semantics for Simulink blocks and diagrams.

Discrete blocks The denotational semantics of a discrete block dB is defined by $[\![dB]\!]_{\mathcal{T}}^{(ena,res)}$. The two parameters ena and res denoting whether dB is enabled and whether a state reset is needed, are essential as the behavior of a discrete block may be affected when it resides within an enabled or triggered subsystem. In the semantics, let $\mathbf{f}_o$ and $\mathbf{g}_s$ denote the functions of dB corresponding to the output o and state s, respectively; and define $preS(dB, h, t, res)$ to return the state values prior to update at time t, depending on the reset signal res,

$$preS(dB, h, t, res) = (t = 0 \vee res)\ ?\ \mathbf{s}_0 : h(\mathbf{s})(t-1)$$

which equals to the initial values of the states at time 0 or when a reset is signaled, otherwise the state values from the previous time step. Notice that an external reset signal may occur at any time, potentially altering the state values before the next sampling instant.

$$
\begin{array}{ll}
h\in[\![dB]\!]_n^{(ena,res)} & \text{iff Let } \mathbf{s}' = preS(dB,h,n,res) \text{ in} \\
& \left(\begin{array}{l}
\text{if } (st|n \wedge ena) \\
\text{then}\left(\begin{array}{l}\forall o\in \mathbf{i}.h(o)(n) = \mathbf{f}_o(\mathbf{s}')(h(\mathbf{i})(n))\wedge \\ \forall s\in\mathbf{s}.h(s)(n) = \mathbf{g}_s(\mathbf{s}')(h(\mathbf{i})(n))\end{array}\right) \\
\text{else}\left(\begin{array}{l}\forall o\in\mathbf{o}.h(o)(n) = (n=0)\ ?\ 0 : h(o)(n-1) \\ \wedge\forall s\in\mathbf{s}.h(s)(n) = \mathbf{s}'_s\end{array}\right)
\end{array}\right) \\
h\in[\![dB]\!]_{(n,n+1)}^{(ena,res)} & \text{iff } \forall t\in(n,n+1).\forall v\in\mathbf{o}\cup\mathbf{s}.h(v)(t) = h(v)(n) \\
h\in[\![cB_C]\!]_n^{(ena,res)} & \text{iff } \forall o\in\mathbf{o}.h(o)(n) = \mathbf{f}_o(h(\mathbf{i})(n)) \\
h\in[\![cB_C]\!]_{(n,n+1)}^{(ena,res)} & \text{iff } \forall t\in(n,n+1).\forall o\in\mathbf{o}.h(o)(t) = \mathbf{f}_o(h(\mathbf{i})(t)) \\
h\in[\![cB_I]\!]_0^{(ena,res)} & \text{iff } \forall o\in\mathbf{o}.h(o)(0) = \mathbf{s}_o \\
h\in[\![cB_I]\!]_{[n,n+1]}^{(ena,res)} & \text{iff } \left(\begin{array}{l}(h(o) \text{ is continuous on } [n,n+1]\wedge \\ \forall t\in(n,n+1).(h(o))'(t) = \mathbf{f}_o(h(\mathbf{i})(t)))\end{array}\right) \\
h\in[\![Sub]\!]_n^{(ena,res)} & \text{iff } \left\{\begin{array}{ll}
\forall B\in\mathbf{B}.h\in[\![B]\!]_n^{(ena,res)} & \text{if } Sty = \mathsf{Atomic} \\
\forall B\in\mathbf{B}.h\in[\![B]\!]_n^{(ena\wedge En(sig,h,n),res\vee Res(sig,sre,h,n))} & \\
& \text{if } Sty = \mathsf{Enab}\ sig\ sre \\
\forall B\in\mathbf{B}.h\in[\![B]\!]_n^{(ena\wedge Trig(sig,tty,h,n),res)} & \text{if } Sty = \mathsf{Trig}\ tty\ sig
\end{array}\right. \\
[\![Sub]\!]_{(n,n+1)}^{(ena,res)} & = \cap_{B\in\mathbf{B}}[\![B]\!]_{(n,n+1)}^{(ena,res)} \\
[\![Diag]\!]_{\mathcal{T}}^{(ena,res)} & = \cap_{B\in top(Diag)}[\![B]\!]_{\mathcal{T}}^{(ena,res)}
\end{array}
$$

Fig. 2. The denotational semantics of Simulink diagrams.

Thus, previous state values are referenced at $t-1$ rather than $t-st$, based on the fact that the fundamental sample time is 1.

According to the semantics of dB, the behavior is defined over discrete time steps $n \in \mathbb{N}$ and the intervals between them, as follows: At time instant n, if the execution condition $st|n \wedge ena$ holds, meaning it is both a sampling instant and the block is enabled, then the block executes. Thus, the outputs and updated states are computed by applying their respective functions to the previous state values and current inputs. Otherwise, if either condition is false, both outputs and states remain unchanged from the previous time step, except at $n = 0$ or upon a state reset event, where they are initialized to their initial values. Between discrete time points, i.e., over the interval $(n, n+1)$, the outputs and states of dB remain constant, holding their values at n.

Continuous blocks and subdiagrams The denotational semantics for continuous blocks cB are defined for calculational blocks and integrators separately. For a calculational block cB_C, its output at any time point is determined by applying the corresponding output functions to the inputs. For an integrator cB_I, its semantics consists of an initialization at time 0, where outputs take their initial values, and the continuous integration over each interval $[n, n+1]$ for all $n \in \mathbb{N}$, where the outputs are continuous and their derivatives are governed by the corresponding derivative functions.

Discrete subsystems For defining the semantics of subsystems, we first introduce several predicates that encode the enabled and triggered conditions of subsystems: En determines whether the subsystem is enabled at time t depending on the value of enabling signal sig; Res determines whether the subsystem should be reset at time t, depending on the reset signal sig and the reset condition sre; $Trig$ determines whether the subsystem is triggered at time t depending on the triggering signal sig and the trigger type tty. Below the formal definitions are given.

$$En(sig, h, t) = (h(sig)(t) > 0)$$

$$Res(sig, sre, h, t) = (sre = 0 \wedge h(sig)(t) > 0 \wedge h(sig)(t-1) \leq 0)$$

$$
\begin{aligned}
Trig(sig, tty, h, t) = \ & ((tty = 0 \ or\ 2) \wedge h(sig)(t) > 0 \wedge h(sig)(t-1) \leq 0 \\
& \vee (tty = 1\ or\ 2) \wedge h(sig)(t) < 0 \wedge h(sig)(t-1) \geq 0)
\end{aligned}
$$

As shown in Fig. 2, for each subsystem type, the denotational semantics of Sub at time instant n is defined by intersecting the semantics of all its constituent blocks in $\mathbf{B}$, while enforcing their respective enabled, reset, and trigger conditions. On the interval $(n, n+1)$, the subsystem simply maintains the outputs and states of its internal blocks at their values from time n, so the denotational semantics is the intersection of the denotational semantics of all internal blocks.

Simulink diagrams Finally, the denotational semantics of a Simulink (sub-)diagram $Diag$, denoted by $[\![Diag]\!]_{\mathcal{T}}^{(ena,res)}$, is defined as the intersection of the semantics of its top-level blocks and subsystems, denoted by $top(Diag)$. The classification of each constituent block as discrete or continuous, a prerequisite for determining its semantic definition, is derived from its sampling time property. If $Diag$ is a top-level diagram, the enabled flag ena is set to $True$ and the reset flag res to $False$ by default, as none of its top-level components are nested within an enabled or triggered subsystem.

5. Operational semantics

The denotational semantics defined above provides a mathematical interpretation of Simulink behaviors through sets of timed states. We now present an operational semantics that defines the concrete execution of Simulink diagrams via transitions between timed states. Compared to denotational semantics, this transition-based operational semantics provides a step-wise construction of the timed state for a Simulink diagram executed over continuous time. Due to the hybrid nature of Simulink diagrams, the operational semantics defines transitions that update the state over both time instants and time intervals.

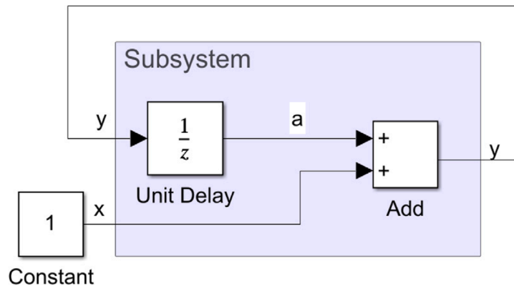


Fig. 3. A subsystem forming a non-direct feedthrough loop.

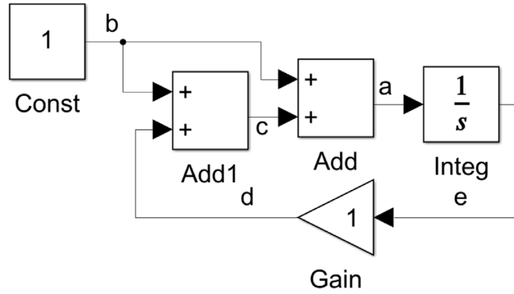


Fig. 4. An example of calculating $getODE$.

Each transition corresponds to the execution of the diagram at a specific time (particularly for discrete blocks at sample points) or throughout an interval (for continuous blocks), reflecting the computation of outputs and state updates over that evolution period.

Next we begin by introducing some key concepts and definitions, and then present the operational semantics.

5.1. Refined data dependency and block execution order

A Simulink diagram executes by evaluating its blocks and subsystems in an order determined by their data dependencies, where a block cannot rely on outputs from the blocks executed after it. Therefore, determining a proper execution order is crucial for correct execution. However, the data dependency implied by block connections is overly coarse. A connection from block A to block B does not necessarily mean B instantaneously depends on A's output. Given a block B, we define its *direct feedthrough inputs* $dfIn(B)$ as those inputs that instantaneously affect its output:

$$dfIn(B) = \begin{cases} \{i | i \in \mathbf{i} \wedge \mathbf{di}_i = True\} & B = dB(\mathbf{i}, _, _, _, _, \mathbf{di}, _, _) \\ \mathbf{i} \cap \bigcup_{b \in \mathbf{B}} dfIn(b) & B = Sub(\mathbf{i}, _, \mathbf{B}, _, \text{Atomic}) \\ (\mathbf{i} \cap \bigcup_{b \in \mathbf{B}} dfIn(b)) \cup \{sig\} & B = Sub(\mathbf{i}, _, \mathbf{B}, _, _(_, sig)) \end{cases}$$

As defined above, if B is a block, $dfIn(B)$ is determined by $\mathbf{di}$; If B is a subsystem, $dfIn(B)$ comprises the subset of its inputs that are the direct feedthrough inputs of its inside blocks, plus the external control signals (when present). Based on this notion, a *refined data dependency* can be defined: Block B is said to depend on block A, denoted by $A \rightharpoonup B$, if and only if some output of A is one of B's direct feedthrough inputs. Consider Fig. 3 where a loop exists. The subsystem has one direct feedthrough input x and one non-direct feedthrough input y (as Unit Delay does not instantaneously rely on its input). According to the $\rightharpoonup$ definition, no direct feedthrough loop exists for this example.

Recall that we define a diagram as a sequence of blocks in syntax. With the help of the data dependency relation $\rightharpoonup$, we define a recursive predicate *besorted* to represent that a diagram has been sorted according to $\rightharpoonup$.

$$besorted\ \epsilon = True \quad besorted\ B \cdot \mathbf{B} = besorted\ \mathbf{B} \wedge \forall A \in set(B \cdot \mathbf{B}). \neg(A \rightharpoonup B)$$

In the above definition, subsystems are considered as a whole and the sorting of blocks inside them is not considered. The predicate $total_besorted$ builds upon $besorted$ to represent that all blocks including those nested inside subsystems are recursively sorted. Having defined a diagram being sorted, we then present a recursive function $sort(\mathbf{B})$ to sort a given diagram $\mathbf{B}$, to be employed in the operational semantics:

$$sort(\mathbf{B}) = \begin{cases} \epsilon & \text{if find_0indegree}(\mathbf{B}) = (\text{None}, \mathbf{B}) \\ b \cdot sort(\mathbf{B}') & \text{if find_0indegree}(\mathbf{B}) = (\text{Some b}, \mathbf{B}') \end{cases}$$

which iteratively finds a zero-indegree block of an unsorted diagram using function $find_0indegree$ (which returns a pair consisting of one zero-indegree block and the remaining blocks after its removal), adds it to the sorted sequence, and terminates when no such block exists. Obviously, $sort(\mathbf{B})$ is always a subset of $\mathbf{B}$ and sorted. A Simulink diagram $\mathbf{B}$ is acyclic, denoted by $loop_free\ \mathbf{B}$, if and only if $length(sort(\mathbf{B}))=length(\mathbf{B})$. Based on *sort*, we define function *deep_sort* further to recursively perform sorting on nested blocks inside subsystems. We prove the correctness of the sorting function in Isabelle/HOL, i.e. $total_besorted\ (deep_sort\ \mathbf{B})$, denoted by **L1** for further reference. Meanwhile, the acyclicity for nested subsystems can be defined by checking nested blocks upon $loop_free$, denoted by $total_loop_free$. We prove that $total_loop_free\ \mathbf{B} \rightarrow \mathbf{B} \simeq (deep_sort(\mathbf{B}))$, denoted by **L2**, i.e., when $\mathbf{B}$ is total loop free, the resulting diagram after sorting and the original diagram are permutation of each other (denoted by $\simeq$).

5.2. Operational semantics of discrete subdiagrams

Before giving the semantics, we recall the update of the outputs and states of a discrete block dB with sample time st, formalized as follows:

$$o(k \cdot st) = \mathbf{f}_o(\mathbf{s}(k \cdot st - 1), \mathbf{i}(k \cdot st)) \quad s(k \cdot st) = \mathbf{g}_s(\mathbf{s}(k \cdot st - 1), \mathbf{i}(k \cdot st))$$

where $o \in \mathbf{o}$, and $s \in \mathbf{s}$ represent outputs and states of dB, $\mathbf{f}_o$ and $\mathbf{g}_s$ are corresponding output and state functions for o and s, respectively. Both outputs and states are computed based on the previous states and current inputs (that might be outputs of other blocks). Our operational semantics for discrete subdiagrams employs a two-phase execution process that separates output and state updates: First, at the output update phase, all blocks compute their outputs in the order according to their data dependencies; then, at the state update phase, as all outputs have already been computed, all blocks can update their states in any execution order.

Fig. 5 illustrates the operational semantics of discrete blocks, subsystems, and diagrams, which are formally defined both at discrete time instants n and over the open intervals $(n, n+1)$ for $n \in \mathbb{N}$. They are defined by distinct forms of transition relations, with the following explanations:

- $\langle dB, ena, res, h\rangle \xrightarrow{n}_D h'$ and $\langle dB, ena, res, h\rangle \overset{n}{\rightsquigarrow}_D h'$ specify the transition relations for output computation and state update, respectively, of a discrete block dB at time n. Each relation executes from an initial state h under the enabled condition ena and reset condition res, resulting in a new state h'. Rules (dB-O) and (dB-S) define the execution of a discrete block at time n, reflecting its behavior of computing new outputs and updating states at discrete sampling points, while incorporating the effects of ena and res. $h[\mathbf{o} \mapsto (n, \mathbf{o}_{new})]$ represents a new timed state obtained from h by updating the value of each $o \in \mathbf{o}$ at time n by the corresponding value in $\mathbf{o}_{new}$.
- $\langle \mathbf{B}, ena, res, h\rangle \overset{n}{\Longrightarrow}_D h'$ defines the execution of a discrete subdiagram, as defined by Rule (dDig). This execution consists of two phases: first, the output computations of all constituent discrete blocks and subdiagrams are performed in a sorted order determined by their data dependencies, as defined by Rule (dDig-O); subsequently, state updates are carried out in an arbitrary order, in accordance with Rule (dDig-S).

$$\frac{\begin{array}{c}\mathbf{s}' = preS(dB,h,n,res) \quad \mathbf{o}' = ((n=0)?0 : h(o)(n-1)) \\ \forall o \in \mathbf{o}.o_{new} = (st|n \wedge ena)?(\mathbf{f}_o(\mathbf{s}')(h(\mathbf{i})(n))) : \mathbf{o}'\end{array}}{\langle dB, ena, res, h\rangle \xrightarrow{n}_{\mathcal{D}} h[\mathbf{o} \mapsto (n, \mathbf{o}_{new})]} \text{dB-O}$$

$$\frac{\begin{array}{c}\mathbf{s}' = preS(dB,h,n,res) \\ \forall s \in \mathbf{s}.s_{new} = (st|n \wedge ena)?(\mathbf{g}_s(\mathbf{s}')(h(\mathbf{i})(n))) : \mathbf{s}'\end{array}}{\langle dB, ena, res, h\rangle \overset{n}{\rightsquigarrow}_{\mathcal{D}} h[\mathbf{s} \mapsto (n, \mathbf{s}_{new})]} \text{dB-S}$$

$$\frac{sort(\mathbf{B}) \equiv B \cdot \mathbf{B}' \quad \langle B, ena, res, h\rangle \xrightarrow{n}_{\mathcal{D}} h_1 \quad \langle \mathbf{B}', ena, res, h_1\rangle \overset{n}{\Longrightarrow}_{\mathcal{DO}} h_2}{\langle \mathbf{B}, ena, res, h\rangle \overset{n}{\Longrightarrow}_{\mathcal{DO}} h_2} \text{dDig-O}$$

$$\frac{\mathbf{B} \equiv B \cdot \mathbf{B}' \quad \langle B, ena, res, h\rangle \overset{n}{\rightsquigarrow}_{\mathcal{D}} h_1 \quad \langle \mathbf{B}', ena, res, h_1\rangle \overset{n}{\Longrightarrow}_{\mathcal{DS}} h_2}{\langle \mathbf{B}, ena, res, h\rangle \overset{n}{\Longrightarrow}_{\mathcal{DS}} h_2} \text{dDig-S}$$

$$\frac{\langle \mathbf{B}, ena, res, h\rangle \overset{n}{\Longrightarrow}_{\mathcal{DO}} h_1 \quad \langle \mathbf{B}, ena, res, h_1\rangle \overset{n}{\Longrightarrow}_{\mathcal{DS}} h_2}{\langle \mathbf{B}, ena, res, h\rangle \overset{n}{\Longrightarrow}_{\mathcal{D}} h_2} \text{dDig}$$

$$\frac{Sub \equiv (\mathbf{i}, \mathbf{o}, \mathbf{B}, st, \mathsf{Atomic}) \quad \langle \mathbf{B}, ena, res, h\rangle \overset{n}{\Longrightarrow}_{\mathcal{D}} h_1}{\langle Sub, ena, res, h\rangle \xrightarrow{n}_{\mathcal{D}} h_1} \text{Atomic}$$

$$\frac{\begin{array}{c}Sub \equiv (\mathbf{i}, \mathbf{o}, \mathbf{B}, st, (\mathsf{Enab}\ sre\ sig)) \quad ena' = ena \wedge En(sig,h,n) \\ res' = res \vee Res(sig, sre, h, n) \quad \langle \mathbf{B}, ena', res', h\rangle \overset{n}{\Longrightarrow}_{\mathcal{D}} h_1\end{array}}{\langle Sub, ena, res, h\rangle \xrightarrow{n}_{\mathcal{D}} h_1} \text{Enabled}$$

$$\frac{\begin{array}{c}Sub \equiv (\mathbf{i}, \mathbf{o}, \mathbf{B}, st, (\mathsf{Trig}\ tty\ sig)) \quad ena' = ena \wedge Trig(sig, tty, h, n) \\ \langle \mathbf{B}, ena', res, h\rangle \overset{n}{\Longrightarrow}_{\mathcal{D}} h_1\end{array}}{\langle Sub, ena, res, h\rangle \xrightarrow{n}_{\mathcal{D}} h_1} \text{Triggered}$$

$$\frac{\forall v \in \mathbf{o} \cup \mathbf{s}.v_{new} = h(v)(n)}{\langle dB, ena, res, h\rangle \xrightarrow{(n,n+1)}_{\mathcal{D}} h[v \mapsto (t, v_{new}) \mid \forall v \in \mathbf{o} \cup \mathbf{s}, \forall t \in (n, n+1)]} \text{dB-ivl}$$

$$\frac{Sub.\mathbf{B} = bs \quad \langle bs, ena, res, h\rangle \overset{(n,n+1)}{\Longrightarrow}_{\mathcal{D}} h_1}{\langle Sub, ena, res, h\rangle \xrightarrow{(n,n+1)}_{\mathcal{D}} h_1} \text{dS-ivl}$$

$$\frac{sort(\mathbf{B}) \equiv B \cdot \mathbf{B}' \quad \langle B, ena, res, h\rangle \xrightarrow{(n,n+1)}_{\mathcal{D}} h_1 \quad \langle \mathbf{B}', ena, res, h_1\rangle \overset{(n,n+1)}{\Longrightarrow}_{\mathcal{D}} h_2}{\langle \mathbf{B}, ena, res, h\rangle \overset{(n,n+1)}{\Longrightarrow}_{\mathcal{D}} h_2} \text{dDig-ivl}$$

Fig. 5. The operational semantics of discrete blocks, subsystems and subdiagrams.

- A discrete subsystem is handled as discrete blocks, so its semantics is defined using the same transition relation $\xrightarrow{n}_{\mathcal{D}}$. Depending on the type of subsystems, the semantics are specified by three rules: (Atomic), (Enabled), and (Triggered). The internal blocks are executed by recursively invoking the semantics of discrete subdiagrams, with additional enabled or triggered conditions incorporated based on the types. Note that the semantics of subdiagrams and subsystems are mutually recursive.
- $\langle dB/Sub, ena, res, h\rangle \xrightarrow{(n,n+1))}_{\mathcal{D}} h'$ denotes the execution of a discrete block dB or subsystem Sub from an initial state h throughout the interval $(n, n+1)$ respectively; finally, $\langle \mathbf{B}, ena, res, h\rangle \overset{(n,n+1)}{\Longrightarrow}_{\mathcal{D}} h'$ denotes the execution of a discrete diagram throughout the interval $(n, n+1)$. Simply, the values of all outputs and states preserve their values at time n, while other variables are unchanged. $h[v \mapsto (t, v_{new}) \mid \forall v \in \mathbf{o} \cup \mathbf{s}, \forall t \in (n, n+1)]$ denotes a new timed state obtained from h by updating the value of each v at time t to the corresponding v_{new}.

5.3. Operational semantics of continuous subdiagrams

A continuous subdiagram in diagram $\mathbf{B}$ comprises calculational blocks and integrators, denoted by $\mathbf{B}_C$ and $\mathbf{B}_I$, respectively. Since calculational blocks must be scheduled together with discrete blocks at

each fundamental sample time, separate semantic rules are required for points and open intervals to properly interoperate with discrete blocks. At every fundamental sample time, the integrator's computation must be completed before proceeding with the discrete part. The behavior of an integrator is described by an ODE and an initial value, and the solution to the ODE is defined over a continuous interval. Therefore, to define the integrator's semantics, the overall interval is partitioned into an initial point followed by a sequence of left-open, right-closed intervals. Their semantics are defined separately in Fig. 6.

- $\langle cB, ena, res, h\rangle \xrightarrow{T}_D h'$ defines the transition relations for the execution of a calculational block at a time instant $T = n$ and over an interval $T = (n, n+1)$, respectively. In both cases, as specified by Rules (Cal) and (Cal-ivl), the block's outputs are computed by applying their output functions to the inputs, while all other variables are not changed. This definition captures the stateless nature of calculational blocks, where outputs depend solely on instantaneous inputs.
  The operational semantics of a calculational subdiagram is similar to that of discrete subdiagrams but excludes state updates, specifically, following the rule (dDig-O). It begins by ordering all blocks according to their data dependencies and then executes them sequentially in the resulting order. Since discrete and calculational blocks are typically combined in the overall semantics of a diagram, we represent both using the same form of transition relations.
- $\langle \mathbf{B}_I, \mathbf{B}, ena, res, h\rangle \overset{T}{\Longrightarrow}_I h'$ defines the transition relation for the execution of all integrator blocks $\mathbf{B}_I$ within the whole diagram $\mathbf{B}$, for the initial time $T = 0$ and over the interval $T = (n, n+1]$, respectively. In contrast to other block types, the semantics of integrators cannot be defined in isolation. All integrators in a continuous diagram must be solved together as a system of ordinary differential equations (called an ODE system). Rule (Integ-0) initializes the states of all integrator blocks $\mathbf{B}_I$, returned by $outputs$. Rule (Integ-ivl) governs the continuous evolution of $\mathbf{B}_I$ over the interval $(n, n+1]$. It constructs the ODE system associated with $\mathbf{B}_I$ from $\mathbf{B}$ (defined by *getODE*) and then solves the resulting initial value problem with initial values given at time n over the interval $(n, n+1]$.

We illustrate the function *getODE* using the example continuous diagram bl shown in Fig. 4. The diagram bl includes an integrator block Integ and several calculational blocks, while the discrete block Const is outside it . The integrator Integ corresponds to the ODE $\dot{e} = a$. Here, a is the output of the calculational block Add, which implements the equation $a{=}b{+}c$. The variable c is itself the output of another calculational block Add1, defined by $c{=}b{+}d$. Furthermore, d is the output of the Gain block, defined as $d{=}e$. By successive substitution, we derive the ODE: $\dot{e}{=}b{+}(b{+}e)$, where all variables on the right-hand side are either integrator state variables (such as e) or external inputs to the continuous subdiagram (such as b). All intermediate variables, that are the outputs of certain calculational blocks, are eliminated during this process.

Although the behavior of these calculational blocks is incorporated within the integrators, their values are still retained in the operational semantics. This is because they are necessary for verifying the consistency between the operational and denotational semantics, and may also be required as inputs to other blocks.

5.4. Operational semantics of simulink diagrams

Above we have defined the operational semantics for core components of a Simulink diagram $\mathbf{B}$, including its discrete ($\mathbf{B}_D$), calculational ($\mathbf{B}_C$), and integrator ($\mathbf{B}_I$) subdiagrams. Fig. 6 bottom presents the operational semantics for a hybrid discrete-continuous diagram $\mathbf{B}$, where $\langle \mathbf{B}, ena, res, h\rangle \overset{T}{\Longrightarrow} h'$ defines the transition relation for the execution of a diagram $\mathbf{B}$ for the initial time $T = 0$ and over the interval $(n, n+1]$ respectively.

The execution order of different parts of $\mathbf{B}$ is critical for its operational semantics. Rule (Diag-0) performs the initialization: It first initializes the states of integrators, then updates the outputs and states of discrete and calculational blocks together. Especially, the latter sorts the combined discrete and calculational blocks according to their data dependencies and then executes them together in the order. Rule (Diag-ivl) defines the execution over $(n, n+1]$: First, the outputs and states of discrete blocks are updated over $(n, n+1)$, which depends solely on variable values at time n; next, the integrators are solved over $(n, n+1]$, also relying exclusively on values at n; then, the outputs of calculational blocks are updated over $(n, n+1)$, which depends on values from both integrators and discrete blocks over $(n, n+1)$; finally, the outputs and states of discrete and calculational blocks at time $n+1$ are updated, potentially incorporating values from integrators at time $n+1$.

The execution order of these different parts of Simulink diagrams is determined by data dependencies. If the order is violated, the system behavior would fail to satisfy the constraints defined in the denotational semantics, thereby leading to an inconsistency between the two semantics.

6. Consistency of denotational and operational semantics

From the definitions of the denotational and operational semantics, they exhibit fundamental differences. Notably, the operational semantics prescribes the execution order of blocks, and more importantly, alters the overall diagram structure in order to solve integrators. These substantial distinctions make it both non-trivial and essential to formally study the relationship between these two semantics. Establishing their consistency is critical for ensuring the correctness of the semantics, as it confirms that the formal mathematical meaning aligns with the executable behavior. In this section, we prove the consistency between the denotational and operational semantics for Simulink, expressed by the existence and uniqueness of the timed state trajectories defined by the denotational semantics.

6.1. The existence theorem

Theorem 1 (*Existence*). ***B** is a Simulink diagram. If the conditions wf **B**, $total_loop_free$ **B**, ODE_cond_global **B** and $None_Con_sub$ **B** hold, then for any initial state h_0, $\langle \boldsymbol{B}, True, False, h_0\rangle \overset{[0,n]}{\Longrightarrow} h$ implies $h \in [\![\boldsymbol{B}]\!]^{(True,False)}_{[0,n]}$.*

The conditions for this theorem include: $\mathbf{B}$ is a valid Simulink diagram (wf), $\mathbf{B}$ is loop free ($total_loop_free$), the ODE system of $\mathbf{B}$ satisfy the continuity and global Lipschitz conditions (ODE_cond_global, to guarantee the existence and uniqueness of solutions of corresponding ODEs), and the subsystems within $\mathbf{B}$ contain no continuous blocks ($None_Con_sub$). The last condition restricts the type of subsystems in order to have a modular operational semantics, while the first three conditions are necessary for a Simulink diagram to have well-defined behavior. For brevity, we refer to these first three conditions collectively as **WDEF B**.

The theorem is proved by decomposing the diagram into separate components, as defined in the semantics (Rules Diag-0, Diag-ivl), proving the required result for each component, and then re-composing them back into the diagram while verifying result preservation. This compositional approach relies on two fundamental properties we have formally proven: Locality, each component's operational semantics exclusively modifies its own output/state variables without affecting those of other components; existence preservation, the property of each component remains preserved under output and state updates of other components. We now present two representative lemmas for discrete and continuous integrator components respectively, highlighting their distinct behavioral properties. The first one states the existence of

$$\frac{\forall o \in \mathbf{o}.o_{new} = \mathbf{f}_o(h(\mathbf{i})(n))}{\langle cB, ena, res, h\rangle \xrightarrow{n}_{\mathcal{D}} h[\mathbf{o} \mapsto (n, \mathbf{o}_{new})]}\text{ Cal}$$

$$\frac{\forall t \in (n, n+1).o \in \mathbf{o}.o_{new} = \mathbf{f}_o(h(\mathbf{i})(t))}{\langle cB, ena, res, h\rangle \xrightarrow{(n,n+1)}_{\mathcal{D}} h[\mathbf{o} \mapsto (t, \mathbf{o}_{new}) \mid \forall t \in (n, n+1)]}\text{ Cal-ivl}$$

$$\frac{outputs(\mathbf{B}_I) = \mathbf{s}}{\langle \mathbf{B}_I, \mathbf{B}, ena, res, h\rangle \overset{0}{\Longrightarrow}_{\mathcal{I}} h[\mathbf{s} \mapsto \mathbf{s}_0]}\text{ Integ-0}$$

$$\frac{\begin{array}{c} getODE(\mathbf{B}) = (\dot{\mathbf{s}} = \mathbf{f}(\mathbf{s}, \mathbf{i})) \quad \mathbf{q}(t) \text{ is continuous over } [n, n+1] \\ \mathbf{q}(n) = h(\mathbf{s})(n) \quad \forall t \in (n, n+1).\dot{\mathbf{q}}(t) = \mathbf{f}(\mathbf{q}(t), h(\mathbf{i})(t)) \end{array}}{\langle \mathbf{B}_I, \mathbf{B}, ena, res, h\rangle \overset{(n,n+1]}{\Longrightarrow}_{\mathcal{I}} h[\mathbf{s} \mapsto (t, \mathbf{q}(t))) \mid \forall t \in (n, n+1]]}\text{ Integ-ivl}$$

$$\frac{\langle \mathbf{B}_I, \mathbf{B}, ena, res, h\rangle \overset{0}{\Longrightarrow}_{\mathcal{I}} h_1 \quad \langle \mathbf{B}_D \cdot \mathbf{B}_C, ena, res, h_1\rangle \overset{0}{\Longrightarrow}_{\mathcal{D}} h_2}{\langle \mathbf{B}, ena, res, h\rangle \overset{0}{\Longrightarrow} h_2}\text{ Diag-0}$$

$$\frac{\begin{array}{c} \langle \mathbf{B}_D, ena, res, h\rangle \overset{(n,n+1)}{\Longrightarrow}_{\mathcal{D}} h_1 \quad \langle \mathbf{B}_I, \mathbf{B}, ena, res, h_1\rangle \xrightarrow{(n,n+1]}_{\mathcal{I}} h_2 \\ \langle \mathbf{B}_C, ena, res, h_2\rangle \overset{(n,n+1)}{\Longrightarrow}_{\mathcal{D}} h_3 \quad \langle \mathbf{B}_D \cdot \mathbf{B}_C, ena, res, h_3\rangle \overset{n+1}{\Longrightarrow}_{\mathcal{D}} h_4 \end{array}}{\langle \mathbf{B}, ena, res, h\rangle \overset{(n,n+1]}{\Longrightarrow} h_4}\text{ Diag-ivl}$$

Fig. 6. The operational semantics of continuous subdiagrams and hybrid diagrams.

denotational semantics of discrete subdiagrams at all sampling instants $n \in \mathbb{N}$.

Lemma 1. *$\boldsymbol{B}$ is a discrete diagram. If wf $\boldsymbol{B}$ and $total_loop_free$ $\boldsymbol{B}$ hold, then $\langle \boldsymbol{B}, ena, res, h_0\rangle \overset{n}{\Longrightarrow}_{\boldsymbol{B}} h$ implies $h \in [\![\boldsymbol{B}]\!]_n^{(ena,res)}$.*

The proof for the discrete case must account for the sorting algorithm used when defining operational semantics, which determines block execution order through dependency analysis. It makes use of the following properties: (1) Function $deep_sort$ produces a totally ordered diagram respecting all data dependencies (**L1** in Section 5.1); (2) For loop-free diagrams, the sorted result is permutation to the original (**L2** in Section 5.1); (3) Permutation preserves well-formedness, i.e. $\forall \mathbf{B}\ \mathbf{B}'.\mathbf{B} \simeq \mathbf{B}' \longrightarrow wf\ \mathbf{B} \longrightarrow wf\ \mathbf{B}'$; and (4) Permutation-equivalent diagrams yield identical denotational semantics, i.e. $\forall \mathbf{B}\ \mathbf{B}'.\mathbf{B} \simeq \mathbf{B}' \longrightarrow [\![\mathbf{B}]\!]_n^{(ena,res)} = [\![\mathbf{B}']\!]_n^{(ena,res)}$.

Another important lemma is the existence of the operational semantics for all integrators over an interval $[n, n+1]$.

Lemma 2. *$\boldsymbol{B}$ is a Simulink diagram, and its discrete, integrator and calculational parts are denoted as $\boldsymbol{B}_D$, $\boldsymbol{B}_I$ and $\boldsymbol{B}_C$ respectively. If **WDEF** $\boldsymbol{B}$ holds, and if we have $\langle \boldsymbol{B}_I, ena, res, h_0\rangle \overset{(n,n+1]}{\Longrightarrow} h_1$ and $\langle \boldsymbol{B}_C, ena, res, h_1\rangle \overset{(n,n+1)}{\Longrightarrow} h_2$, and $h_2(v)(t) = h_1(v)(n)$ for all $t \in (n, n+1)$ and $v \in outputs(\boldsymbol{B}_D)$, then $h_2 \in [\![\boldsymbol{B}_I]\!]_{[n,n+1]}^{(ena,res)}$.*

where h_1, h_2 are respectively the timed states reached after the execution of $\mathbf{B}_I$ and $\mathbf{B}_C$ in sequence, as defined in the diagram execution (Rule Diag-ivl). Besides the well-defined conditions, it requires additionally that the outputs of discrete components $\mathbf{B}_D$ that the integrators depend on must remain invariant under the execution of calculational $\mathbf{B}_C$, which is actually ensured by the operational semantics of $\mathbf{B}_C$. The proof of this lemma relies primarily on two key properties. First, for any input v of an integrator in $\mathbf{B}_I$, the arithmetic expression obtained by substituting the outputs of calculational blocks with their computed values within $getODE$ is consistent with the denotational interpretation of v. This ensures the *correctness* of the derived ODE system for $\mathbf{B}_I$. Second, the inputs of each arithmetic expression are disjoint from the outputs of $\mathbf{B}_C$, demonstrating the *completeness* of the reduction by eliminating all intermediate outputs of $\mathbf{B}_C$.

6.2. The uniqueness theorem

Theorem 2 (*Uniqueness*)**.** *$\boldsymbol{B}$ is a Simulink diagram. If the conditions $\boldsymbol{WDEF}$ $\boldsymbol{B}$ and $None_Con_sub$ $\boldsymbol{B}$ hold, and if both $h_1, h_2 \in [\![\boldsymbol{B}]\!]_{[0,n]}^{(True,False)}$, and h_1, h_2 have the same values for external inputs, i.e. $\forall v \in inputs(\boldsymbol{B}) \setminus outputs(\boldsymbol{B}).\forall t \in [0,n].h_1(v)(t) = h_2(v)(t)$, then h_1, h_2 have the same values for all outputs and states, i.e. $\forall v \in (states(\boldsymbol{B}) \cup outputs(\boldsymbol{B})).\forall t \in [0,n].h_1(v)(t) = h_2(v)(t)$.*

Under the same conditions as the Existence theorem, plus the condition that h_1 and h_2 have identical values for all external inputs to $\mathbf{B}$, then they coincide on all states and outputs of $\mathbf{B}$. The proof of the uniqueness theorem follows the similar compositional approach as the Existence theorem.

One direct consequence of the existence and uniqueness of the denotational semantics is the determinism of the operational semantics under different execution orders (indicated by $\mathbf{B} \simeq \mathbf{B}'$), stated as follows.

Theorem 3 (*Determinism of Operational Semantics*)**.** *$\boldsymbol{B}$ and $\boldsymbol{B}'$ are two Simulink diagrams and $\boldsymbol{B} \simeq \boldsymbol{B}'$ holds. If they both satisfy the same conditions for the Existence theorem, then for any initial state h_0, $\langle \boldsymbol{B}, True, False, h_0\rangle \overset{[0,n]}{\Longrightarrow} h$ if and only if $\langle \boldsymbol{B}', True, False, h_0\rangle \overset{[0,n]}{\Longrightarrow} h$ for some h.*

7. Implementation

We have formalized the syntax and semantics of Simulink diagrams in Isabelle and proved all relevant theorems.[1] To apply the semantics, a Simulink diagram must first be represented in Isabelle according to the syntax given in Section 3. Manually writing such representation is often tedious and error-prone. To overcome this, we developed an automatic translator that converts graphical Simulink diagrams into their Isabelle representation. This allows us to rigorously analyze system behavior by applying the semantic rules within Isabelle.

7.1. Translation from Simulink diagrams to Isabelle representation

We have implemented a translator[2] that automatically translates Simulink diagrams into their Isabelle representation. Using Matlab function `save_system`, a Simulink diagram can first be exported into an XML file. In the Mars toolchain [9], we implemented a translator from Simulink to representations in Python. The translator reads a Simulink diagram in the XML format and outputs a Python object, which contains all the necessary information of the Simulink diagram required for syntax translation, including the blocks, their attributes, the nested subsystems, *etc*. Hence, after calling the translator from Simulink to Python, we traverse the resulting Python objects of the Simulink diagram and construct the Isabelle representation of the entire diagram according to our syntax.

For the use of the translator, it requires that the input Simulink diagrams first pass Simulink's own compilation, and moreover, it provides an explicit post-translation validation step to ensure the absence of algebraic loops. However, a formal re-verification of well-formedness for the transformed blocks is still required within Isabelle itself. Currently, all types of blocks listed in Table 1 are supported by the translator. They all successfully pass both the translator's checks and the subsequent well-formedness verification in Isabelle, providing an end-to-end sanity check for the supported Simulink components.

7.2. Formalization in Isabelle/HOL

In the remainder of this section, we present and discuss selected key aspects of this formalization. In Isabelle implementation, a Simulink diagram is represented as a list of blocks. Each block is either an atomic block or a composite subsystem, which itself contains a list of blocks. This recursive definition allows diagrams to be nested arbitrarily, reflecting their hierarchical structure. This is central to both our semantic definitions and the corresponding proofs.

7.2.1. Formalization of the operational semantics

As defined in Section 5, the operational semantics for (sub-)diagrams and subsystems are defined mutually recursive. For instance, the output computation rule for subdiagrams at time instants (Rule dDig-O) is mutually recursive with the rules for subsystems (Atomic, Enabled, and Triggered). To simplify proofs, the implementation in Isabelle avoids explicit mutual recursion by folding the semantic definitions into a single recursive function. Rule dDig-O is implemented by the following function $exeDL_o_att$, which incorporates the semantics of subsystems and is defined in a self-recursive manner.

```
fun exeDL_o_att :: block list ⇒ bool ⇒ bool ⇒ nat ⇒ timed_vars ⇒ timed_vars
where
  exeDL_o_att [] exe res n h = h|
  exeDL_o_att (b#bl) exe res n h = exeDL_o_att bl exe res n (case b of
    (cB il ol _ ofl dir) ⇒ (if dir then exeCal_att ... else h)|
    (dB il ol sl vl st _ ofl sfl) ⇒(if ((get_st b)|n ∧ (get_st b) = 0) ∧ exe
                         then exeD_o_att ... else exeD_o_not_att ... )|
    (Sub il ol bs st typ) ⇒ (case typ of
                         Atomic ⇒ ( exeDL_o_att bs exe res n h) |
                         Enab re var ⇒(if (h var n)>0
                            then (if (h var n-1)≤0∧re=0
                               then exeDL_o_att bs exe True n h
                               else exeDL_o_att  bs exe res n h)
                            else exeDL_o_att bs False res n h)|
                         Trig tty var ⇒
                   (if ((tty=0 ∨ tty=2) ∧ (h var n) > 0 ∧ (h var n-1) ≤0)
                     ∨ ((tty=1 ∨ tty=2) ∧ (h var n) < 0 ∧ (h var n-1) ≥0)
                         then exeDL_o_att bs exe res n h
                         else exeDL_o_att bs False res n h)))
```

As defined above, it processes each block in the block list by induction: for a calculational or discrete block, it directly invokes the semantics of that block; for a subsystem, it computes the environmental parameters based on its type and signals, then recursively invokes $exeDL_o_att$ on the block list of the subsystem.

Integrators The most challenging part of the operational semantics of continuous subdiagrams is solving the integrators. As mentioned previously, it needs to first derive the ODE system corresponding to all integrators and then solve the resulting initial value problem. Function $getODE$ extracts the ODE system from the whole diagram, and its core is the function get_ode_fn. Consider an integrator I representing the ODE $\dot{y} = v$ in diagram $Diag$, $get_ode_fn\ Diag\ v$ constructs a modified ODE $\dot{y} = f(vl)$ for I, where v is replaced by a composite expression function $f(vl)$ derived from the calculational blocks in $Diag$.

```
get_ode_fn Diag v = (let b = find (λb. v ∈ set (outputs b) ∧ isC b) Diag in
  (if b = None then (λl.(hd l), [v])
  else (let bl' = remove1 b Diag in
       let fvl = map (get_ode_fn bl') (inputs b) in
       let vl = concat (map snd fvl) in
       let fns = λl a b. (fst a + length (snd b) , (snd a)@[fst b
                 (take (length (snd b)) (drop (fst a) l))]) in
       let get_vals = λfvl l. snd (foldl (fns l) (0,[]) fvl) in
       let f = λl.(outfns b)[index (outputs b) v] (get_vals fvl l) in (f,vl))))
```

The above function $get_ode_fn\ Diag\ v$ returns a pair (f, vl), where f is a function for computing v in $Diag$, and vl are integrator variables or those external to the continuous part of $Diag$. If v is not the output of some calculational block b, then $f(vl)$ is v itself. Otherwise, let fvl be the results of applying get_ode_fn recursively to all inputs of b, then vl is obtained by concatenating all parameter variables obtained from fvl, and f is constructed by combining the functions in fvl, i.e. firstly computing the inputs of b by applying functions in fvl to values of their corresponding vl, via function get_vals, and then computing v by applying b's output function to the inputs.

Using $getODE$, we obtain the ODE system describing the continuous behavior of $Diag$. To solve such ODEs, we define function $solveODE$ using the function $solution$ defined in the locale $unique_on_closed$ in Isabelle/HOL's built-in Initial Value Problem theory, taking the following as inputs: an initial timed state h, a Simulink diagram $Diag$, an initial time $t0$ and an integration interval length t. The function returns the solution of the ODE, mapping time to the values of all variables throughout the integration interval:

```
solveODE h Diag t0 t = unique_on_closed.solution t0 {t0−−t0+t}
  (Var2Vec h t0) (getODE Diag) UNIV (SOME L. unique_on_univ Diag h t0 t L)
```

where $UNIV$ denotes the universal set of all variables and $unique_on_univ$ specifies the conditions that guarantee the existence and uniqueness of the solution to the ODE system. These conditions include: the integral interval $[t_0, t_0+t]$ (denoted by T) is compact; $UNIV$ (denoted by X) is a closed subset of the state space; the ODEs $getODE(Diag)$ are continuous

[1] The Isabelle implementation can be found at https://gitee.com/infiniteob/sim-sem/tree/master/.

[2] The translator can be found at https://gitee.com/lee_sen/mars/tree/lx/Simulink2Isabelle.

on $T \times X$ and satisfy the global Lipschitz condition with respect to state variables; and the ODEs exhibit a self-mapping property over T and X, defined as follows ($t0$ is the initial time, x_0 is the initial value at t_0 and f is the ODE function):

$$self_mapping\ T\ t0\ x0\ f\ X = T\ is\ interval \wedge t0 \in T \wedge x0 \in X \wedge \forall \tau \in T. \forall x : [t0, \tau] \rightarrow X. x(t0) = x0 \wedge x\ is\ continuous\ on\ [t0, \tau] \longrightarrow x(t0) + \int_{t0}^{\tau} f(t, x(t))dt \in X$$

which ensures that the Picard operator maps continuous functions from T to X to continuous functions from T to X.

7.2.2. Key proof techniques in Isabelle/HOL

Given the hierarchical structure of diagrams, properties of diagrams are generally proved by structural induction. To facilitate induction over nested subsystems, we define a function that computes the depth of a diagram, aka. a block list, as follows:

```
fun levelbl :: block list ⇒ nat where
  levelbl [] = 0|
  levelbl (b#bl) = max (levelbl bl) (case b of (d B il ol sl vl st dl ofl sfl) ⇒ 1
                    |(c B _ _ _ _ _) ⇒ 1 |(Sub _ _ bs _ _) ⇒ 1 + levelbl bs)
```

The depth of an empty list is 0, while the depth of a non-empty list without subsystems is 1, otherwise it is the maximum depth of the nested subsystems within the subsystems plus 1. When we need to apply the inductive hypothesis to the block lists within subsystems, we can induct on the depth.

Moreover, in the operational semantics of Simulink diagrams, blocks are ordered according to their data dependencies; however, multiple valid execution orders may exist. To address this, we define a permutation relation ($\simeq$ used previously) over diagrams, based on the standard list permutation relation ($\sim$):

```
inductive deep_perm :: block list ⇒ block list ⇒ bool (infixr ≃ 50) where
  deepin: length al = length bl ∧ (∀i <length al. (al_i=bl_i∨(∃il ol bs bs′ st typ.
        al_i=(Sub il ol bs st typ) ∧bl_i=(Sub il ol bs′ st typ) ∧bs≃bs′))) → al ≃ bl
| step: al ∼ bl → bl ≃ cl → al ≃ cl
```

As defined above, $al \simeq bl$ indicates that not only may the order of top-level blocks differ, but also the order of blocks within subsystems, recursively at all nested levels. We have proven that $\simeq$ preserves properties such as commutativity and transitivity, the structural invariants including diagram length and depth, and so on. These results ensure that some semantic definitions remain consistent across all valid permutations of block orderings.

Finally, proving properties of the semantics of diagrams relies on the condition $loop_free$, which also ensures the termination of certain recursive functions over diagrams. However, the original definition of $loop_free$, given in terms of the $sort$ function, is not sufficiently constructive for reasoning about loops. To overcome this limitation, we introduce a more foundational characterization of loops. We define the positive transitive closure $\rightharpoonup^+$ of the direct dependency relation $\rightharpoonup$, to indicate that there is a direct dependency path of positive length between two blocks. We further define a predicate $dp_loop(b) = \exists a.a \rightharpoonup^+ b \wedge a \rightharpoonup^+ a$ to denote that block b depends on some direct feedthrough loop. From this, we can derive the equivalence $(\forall b \in bl.dp_loop(b)) \longleftrightarrow (\forall b \in bl.\exists a \in bl.a \rightharpoonup b)$, which states that every block in bl depends on some node involved in a cycle, implying the absence of blocks with zero in-degree. This formulation allows us to reinterpret sorting in terms of loop freedom: $sort\ bl \sim filter\ (\lambda b.\neg dp_loop(b))\ bl$, meaning that $sort\ bl$ consists exactly of those blocks not involved in any direct feedthrough loop, up to permutation. Most importantly, we establish the key equivalence: $loop_free\ D \longleftrightarrow \forall b \in D.\neg dp_loop(b)$, connecting the notion of loop freedom with a more constructive characterization of loops. This facilitates inductive reasoning about loop structures and simplifies the proof of loop-related semantic properties.

Table 2
Lines of Code (LOC) in Isabelle by category.

	Category	LOC
Definitions	Syntax	125
	Denotational semantics	155
	Operational semantics	199
	Auxiliary definitions	161
Proofs	Discrete	2696
	Continuous	2872
	Consistency Theorems	5923
Applications	The PID control example	875
	Temperature Control example	180
	LiDAR scanning angle sampling example	116
	Block library	100
	Total	**13 402**

7.2.3. Statistics on the isabelle formalization

Table 2 lists the lines of code for each part in Isabelle implementation: 640 lines for definitions, 11,491 lines for proofs, 1271 lines for applications, totaling 13,402 lines.

8. Application to case studies

This section presents the application of our semantic framework to several case studies. First, the PID control example demonstrates the end-to-end workflow: the Simulink diagram is translated into Isabelle automatically, its execution trace over a time interval is computed and the key properties are formally verified using our semantics and the consistency theorem. Furthermore, we demonstrated the broad applicability of our semantic framework through two additional examples: one involving a multi-rate system with non-integer sample times, and another involving systems with triggered/enabled subsystems.

8.1. The PID control example

We now demonstrate the application of our semantics using the PID control example shown in Fig. 1, hereafter referred to as *Diag*. A crucial property of *Diag* is that the process variable b of the controlled plant converges to the setpoint $d = 1$ under the PID controller. In this section, we employ our formally defined semantics to model the behavior of this example and rigorously verify this property.

The blocks UD1, UD2, Sum1, Integ, and Discrete pid have been previously defined within our formal syntax. For the remaining blocks, the zero-order hold block ZOH is used to assign a discrete sampling time to Sub1, thus it is syntactically merged with Sub1 while retaining the designated sampling time. By applying the translator and block library, we obtain the Isabelle representation of *Diag* expressed in our syntax, as depicted in Fig. 7. With the syntax defined, to apply our semantics to $Diag$, the conditions required by the existence and uniqueness theorems on the semantics established in Section 6 must be verified. These include the well-formedness condition $wf\ Diag$, the absence of algebraic loops $total_loop_free\ Diag$, the global ODE condition $ODE_cond_global\ Diag$, and the type restriction to subsystems $None_Con_sub\ Diag$; all of these have been formally proven to hold for this example.

Upon completion of the above steps, the operational semantics of $Diag$ can be derived according to the definitions in Section 5. We have proven the following result about the execution of $Diag$ over the interval $[0, 2]$.

Lemma 3. $\langle Diag, True, False, h_0 \rangle \overset{[0,2]}{\Longrightarrow} h$

where h records the values of variables over $[0, 2]$ as defined in Fig. 8. At time 0, the execution initializes the output of Integ to 0.5. For $t \in [0, 1)$, the explicit solution for the continuous variable b is $1.4e^t - 0.9$, c

```
abbreviation ex_pid_Integ :: block where " ex_pid_Integ ≡ Integ (CHR ''a'') (CHR ''b'') 0.5 "
abbreviation ex_pid_Sum1 :: block where " ex_pid_Sum1 ≡ Add ''cb'' [True,True] (CHR ''a'') "
abbreviation ex_pid_Const :: block where " ex_pid_Const ≡ Const (CHR ''d'') 1 "
abbreviation ex_pid_Sub1 :: block where " ex_pid_Sub1 ≡ Add_d ''db'' [True,False] (CHR ''e'') 1 "
abbreviation ex_pid_Discrete_pid_Add :: block where " ex_pid_Discrete_pid_Add ≡ Add_d ''feg'' [True,True,True] (CHR ''c'') 1 "
abbreviation ex_pid_Discrete_pid_Kd :: block where " ex_pid_Discrete_pid_Kd ≡ Gain_d (CHR ''h'') (CHR ''g'') 1 0.2 "
abbreviation ex_pid_Discrete_pid_Sub2 :: block where " ex_pid_Discrete_pid_Sub2 ≡ Add_d ''ei'' [True,False] (CHR ''h'') 1 "
abbreviation ex_pid_Discrete_pid_Sum2 :: block where " ex_pid_Discrete_pid_Sum2 ≡ Add_d ''ej'' [True,True] (CHR ''f'') 1 "
abbreviation ex_pid_Discrete_pid_UD1 :: block where " ex_pid_Discrete_pid_UD1 ≡ UD (CHR ''e'') (CHR ''i'') (CHR ''k'') 1 1 "
abbreviation ex_pid_Discrete_pid_UD2 :: block where " ex_pid_Discrete_pid_UD2 ≡ UD (CHR ''f'') (CHR ''j'') (CHR ''l'') 0 1 "
definition ex_pid_Discrete_pid :: block where
"ex_pid_Discrete_pid ≡ Subsystem ''e'' ''c'' [ex_pid_Discrete_pid_Add,ex_pid_Discrete_pid_Kd,ex_pid_Discrete_pid_Sub2,
                                            ex_pid_Discrete_pid_Sum2,ex_pid_Discrete_pid_UD1,ex_pid_Discrete_pid_UD2] 1 Atomic"
definition " ex_pid ≡ [ex_pid_Integ,ex_pid_Sum1,ex_pid_Const,ex_pid_Sub1,ex_pid_Discrete_pid] "
```

Fig. 7. The Isabelle representation of the PID control example.

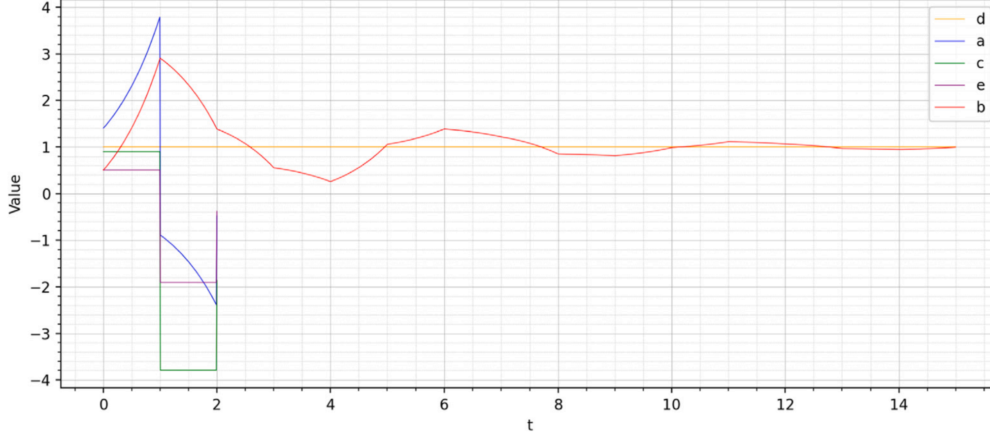


Fig. 8. The timed states of variables for the PID control example.

keeps constant 0.9, and a is $1.4e^t$, satisfying a=b+c and $\dot{b}$=a. For t∈[1, 2], the value of b becomes $(3.68e^{-1}-1.68)e^t+3.08e-4.58$. At $t = 1$, b exceeds the setpoint d and changes from increasing to decreasing, reflecting the process of the plant's output approaching the setpoint under PID control.

To analyze the convergence of the continuous process variable b, it is necessary to determine its values over the interval $[0, n]$ for arbitrary n. Using the denotational semantics, we establish that the value of b at each fundamental sampling instant k (denoted as b_k) satisfies a recurrence relation. Together with the initial values at $n = 0, 1, 2$, this recurrence leads to the following result.

Lemma 4. *Suppose* $h \in [\![Diag]\!]^{(True,False)}_{[0,n]}$ *and let* b_k *be the value of* b *at time instants* $k \in \{0, 1, \dots, n\}$ *in h, i.e.* $h(b)(k)$*, then h satisfies the following recurrence relation at time k:*

- $b_0 = 0.5$, $b_1 = 1.4e - 0.9$, $b_2 = 6.76e - 1.68e^2 - 4.58$.
- $\forall k{+}3 \leq n. b_{k+3} = (3.2{-}1.2e)b_{k+2}{+}(0.4e{-}1.4)b_{k+1}{+}(0.2{-}0.2e)b_k{+}e{-}1$

Due to the complexity of the coefficients, we construct a bounding sequence $c_k = \frac{16}{7}\left(\frac{7}{8}\right)^k$, and have proven the following two facts:

$\forall k. c_k \geq |b_k - 1|$, and $\lim_{k\to\infty} c_k = 0$

thereby obtaining $\lim_{k\to\infty} b_k = 1$.

Furthermore, we also derive from the denotational semantics of *Diag* that the values of b over each interval $[k, k{+}1]$ satisfy the following fact.

Lemma 5. *Suppose* $h \in [\![Diag]\!]^{(True,False)}_{[0,n]}$ *and let* b_k *be the value of* b *at time instants* $k \in \{0, 1, \dots, n\}$ *in h, then for any* $k < n$*, any* $t \in [k, k + 1]$*,*

$$h(b)(t) = \frac{b_{k+1} - b_k}{e - 1} e^{t-k} + \frac{e \cdot b_k - b_{k+1}}{e - 1}$$

From the consistency between the denotational and operational semantics, the above timed state $h \in [\![Diag]\!]^{(True,False)}_{[0,n]}$ is exactly the timed state constructed by the operational semantics. Therefore, we have obtained the continuous evolution of b over arbitrary intervals $[0, n]$ for any $n \in \mathbb{N}$ by following our semantics. Fig. 8 shows the values of b over the finite interval $[0, 15]$ according to our computed timed states, showing its convergence toward the setpoint value $d = 1$. But to rigorously establish this convergence property, we formally prove the following theorem in Isabelle/HOL, stating that b converges to the setpoint value 1.

Theorem 4 (*Convergence to Set Point*)**.** *Let* $\varepsilon > 0$ *be an arbitrary precision, then there exists* $N \in \mathbb{N}$ *such that for all* $n \in \mathbb{N}, t \in \mathbb{R}$*, if* $n \geq N$ *and* $\langle Diag, True, False, h_0\rangle \xRightarrow{[0,n]} h$*, then for any* $t \in [N, n]$*,* $|h(b)(t) - 1| \leq \varepsilon$ *holds.*

We have also compared the above results with the simulation results produced by Simulink for this example, which turn out to be consistent.

8.2. Other case studies

We have also applied our approach to two additional case studies. In the first example, we demonstrate how to handle multi-rate systems with non-integer sample times through time scaling, while the second example shows that our approach is applicable to Simulink diagrams containing triggered/enabled subsystems and state resets.

A multi-rate system Fig. 9 shows a temperature control system with two distinct discrete sample times. The original system (top), with a fundamental sample time less than 1, cannot be handled directly by our approach; and it is transformed via time scaling into the system shown below, which has a fundamental sample time of 1 and thus can be handled.

Given an arbitrary Simulink diagram D, suppose each discrete block i has a sample time of $\frac{p_i}{q_i}$, where p_i and q_i are coprime integers. Let n be the *least common multiple* of all denominators q_i, then the fundamental sample time of D can be set to $\frac{1}{n}$. We aim to define a transformation

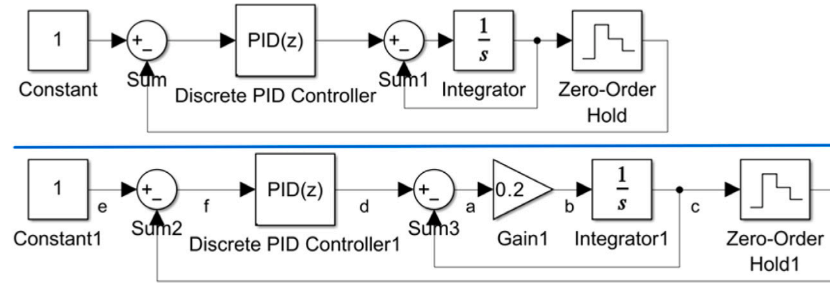


Fig. 9. A Temperature Control System.

process that turns the fundamental sample time to 1, while ensuring the semantics of the original and transformed diagrams are consistent modulo time scaling. Specifically, let h be the trace for D and h' for the transformed diagram, they satisfy that for every variable v in D and every time t, $h'(v)(nt) = h(v)(t)$. To construct a diagram D' whose execution trace is exactly h', we apply a general method as follows: multiply the sample time of every discrete block in D by n, and insert a Gain block with gain $\frac{1}{n}$ before the input of every integrator, while leaving all other parts unchanged.

In the upper half of Fig. 9, the Discrete PID Controller is a temperature controller that outputs a control signal, with a sample time of 0.4. Sum1 and Integrator together form a model of the ambient temperature, i.e. the plant to be controlled, and are both continuous blocks. Constant provides the reference setpoint, outputting a constant value of 1. Sum represents a low-frequency temperature sensor that outputs the difference between the ambient temperature and the setpoint, and its sample time is set to 0.6 via a Zero-Order Hold. The fundamental sample time of the entire system is $\frac{1}{5}$.

The lower half of Fig. 9 shows the transformed system. The key differences are: Gain1 block with gain $\frac{1}{5}$ is inserted before the integrator; the sample time of Zero-Order Hold1 is set to 3; and the sample time of Discrete PID Controller1 is set to 2.

A system with triggered/enabled subsystems Fig. 10 illustrates a LiDAR scanning angle sampling system, which includes both a triggered subsystem and an enabled subsystem with state reset capabilities. Scanner System models the process of how the scanning angle of the radar changes over time. It is an enabled subsystem with state reset functionality, receiving signal b as the enable signal. Inside this subsystem, there is a discrete accumulator that represents the periodically increasing scanning angle, with a sample time of 1. The input a denotes the scanning speed, while the output i provides the current scanning angle. Signal b indicates the operational status of the radar: when $b = 0$, the radar is off and all internal blocks maintain their outputs unchanged; when b transitions from 0 to 1, a scan starts, triggering the internal Unit Delay's state to reset, ensuring that each scan starts from the initial position. Angle Detector acts as an angle latch, implemented as a triggered subsystem. It reads the current angle i and updates its output h to match i every time a rising edge of the sampling signal e is detected. When not triggered, it holds the value of h constant.

Formalization. For both the case studies, we first translated them into our syntax in Isabelle using the translator we implemented. During translation, the well-formedness of the models was automatically checked. We then formally proved in Isabelle that both the models satisfy the conditions required by the existence and uniqueness theorems, i.e., the **WDEF** and $None_Con_Sub$ conditions, thereby enabling the use of denotational semantics to analyze and verify properties of their operational semantics, similar to the PID control example.

9. Extensions and discussion

9.1. The continuity and Lipschitz conditions

The theorems for the semantics established in Section 6 rely on the condition ODE_cond_global, i.e. the continuity and global Lipschitz conditions, to ensure the existence and uniqueness of the solution for the ODE system corresponding to the given Simulink diagram. Recall that the ODE system is derived using the $getODE$ function, which reduces the calculational blocks in the continuous subdiagram by incorporating their functionalities into the Integrator blocks.

We define a predicate $form_ode(\mathbf{B}, b)$ to determine whether a block b is a calculational block contributing to the ODE system of diagram $\mathbf{B}$, i.e. it recursively provides input to an Integrator block:

$$form_ode(\mathbf{B}, b) = b \in \mathbf{B} \wedge isC(b) \wedge$$
$$\exists a \in \mathbf{B}.\,(isI(a) \vee form_ode(\mathbf{B}, a)) \wedge outputs(b) \cap inputs(a) \neq \emptyset$$

For all blocks b in $\mathbf{B}$ satisfying $form_ode(\mathbf{B}, b)$, if they satisfy the continuity and the Lipschitz conditions, then the ODE system of $\mathbf{B}$ itself also satisfies these conditions. Now consider blocks in Table 1, we find that the blocks of the following types: Coulomb and Viscous Friction, Continuous Pulse Generator, IC (Initial Condition for signals), (Multiport) Switch, Sqrt, Sign, and Square, may violate the continuity and the Lipschitz conditions of the ODE system. However, these types of blocks can still be employed for constructing diagrams in our approach, provided that $form_ode(\mathbf{B}, b)$ does not hold, i.e. when their outputs are directed to discrete blocks. It is also noteworthy that Saturation and Dead Zone, although categorized under Discontinuities in Simulink block library, they satisfy both continuity and the Lipschitz condition.

9.2. Relaxation of the constraint on discrete subsystems

The constraint $None_Con_sub$ requires subsystems to contain only discrete blocks, in order to have a modular and compositional operational semantics for Simulink diagrams. This constraint can be relaxed in the denotational semantics by specifying that continuous blocks satisfy their equations only when enabled or triggered, while maintaining their states or outputs otherwise. We omit these details here and focus on the operational semantics of subsystems containing continuous blocks.

To relax the constraint in the operational semantics, we pursue two methods: by syntactic transformation, we unfold the internal structure of a subsystem containing continuous blocks and transform it into an equivalent form that satisfies the constraint; or alternatively, we extend the semantic framework itself to directly handle these subsystems. Next consider a Simulink diagram $\mathbf{B}$ that contains a subsystem S with

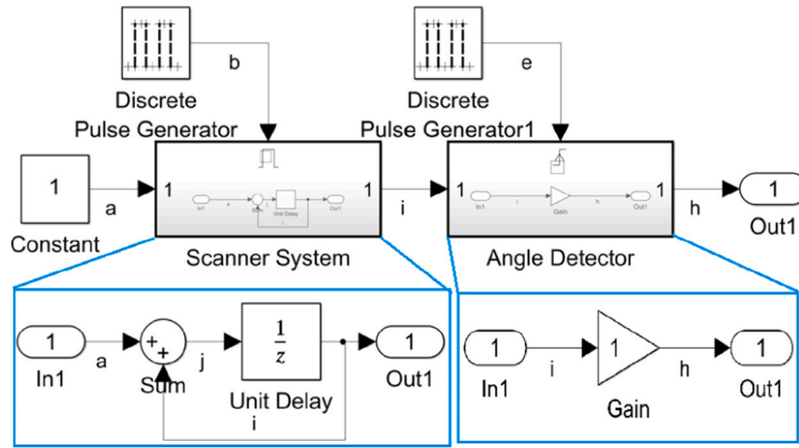


Fig. 10. A LiDAR Scanning Angle Sampling System.

internal continuous blocks. Based on the subsystem type, we discuss each method together with their specific limitations.

Syntactic transformation If S is an atomic subsystem, it is recursively flattened and its constituent atomic blocks are incorporated into the parent diagram $\mathbf{B}$, preserving their original interconnections. If S is a triggered subsystem, since Simulink prohibits triggered subsystems from containing integrators, S can only contain internal calculational blocks. In this case, as the triggering signal occurs only at integer multiples of the fundamental sample time of $\mathbf{B}$, each internal calculational block can be replaced by a stateless discrete block, defined with the fundamental sample time and an identical output update function. This transformation eliminates continuous blocks within S.

Enabled subsystems, however, present a fundamental limitation to the current semantic framework. Their dynamic behavior, which is determined by an enabling signal, prevents them from being transformed into a single, static ODE system. For example, an enabled subsystem containing an integrator corresponds to two distinct ODE systems: when enabled, the integrator evolves normally; when disabled, its state remains constant. Our current semantics cannot represent this conditional switching within a unified ODE. This may be addressed by extending the semantics to support time-varying or switched ODE systems that change based on the enabled signal.

Semantic extension We consider extending the current semantics to compositionally handle subsystems S containing continuous blocks.

If S is an atomic subsystem, its semantics is sketched as follows:

- First, if S contains integrators with states $\mathbf{s}$, we derive the ODE system corresponding to S. We apply $getODE(S)$ to obtain a partial ODE system $\dot{\mathbf{s}} = f(\mathbf{s}, \mathbf{x})$, where variables occurring in $\mathbf{x}$ are either external inputs to S, or outputs of discrete blocks within S. Note that this resulting ODE system is partial, as in the first case, the external inputs originating from the rest of diagram $\mathbf{B}$, indicated by $\mathbf{B} \setminus S$, might be outputs of calculational blocks; and moreover, $\mathbf{B} \setminus S$ may also contain integrators.
- Second, for external outputs o of S, we derive their expression functions by applying function $get_ode_fn(S, o)$. This yields $o = g(\mathbf{s}, \mathbf{x})$, with $\mathbf{s}, \mathbf{x}$ following the same notation as above.
- Third, the semantic definition of subsystem S produces three elements for integration: the discrete components S_D, the partial ODE system $\dot{\mathbf{s}} = f(\mathbf{s}, \mathbf{x})$, and its output functions $o = g(\mathbf{s}, \mathbf{x})$.
- At last, these are subsequently incorporated into the semantics of the parent diagram $\mathbf{B} \setminus S$. This integration proceeds as follows: first, merge the discrete part S_D with the discrete part of $\mathbf{B} \setminus S$; then treat the partial ODE as an integrator block and combine it with continuous part of $\mathbf{B} \setminus S$; next, with the help of the output functions of S, derive the complete ODE system for $\mathbf{B}$; finally follow the standard semantic rules, the semantics of the overall diagram $\mathbf{B}$ is defined.

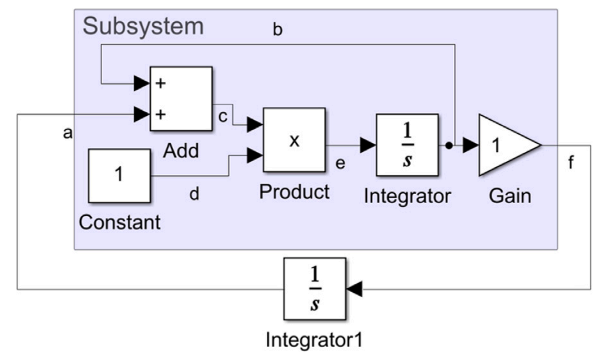


Fig. 11. An Atomic Subsystem with Continuous Blocks.

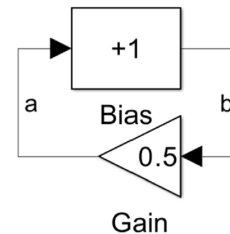


Fig. 12. A Diagram with Algebraic Loop.

We use the example subsystem in Fig. 11 to illustrate the above procedure. Its discrete part consists of a Constant block with output d, its output function is $f = b \cdot 1$, and its partial ODE is $\dot{b}=(b+a) \cdot d$. For the entire diagram, using the output function, the whole ODE system is obtained, plus the ODE equation for Integrator1: $\dot{a}=b \cdot 1$.

For triggered subsystems, since they are prohibited from containing integrators, we simply treat all internal calculational blocks as discrete blocks, holding their outputs constant over each time interval between triggering instants. Enabled subsystems are more complex, as we mentioned above. Their ODE system must be constructed conditionally based on the enabling signal. Specifically, the behavior of calculational blocks over a time interval depends on the enabling signal: if enabled, their behavior follows the same rules as in an atomic subsystem; if disabled, all internal block outputs and state variables remain constant.

9.3. Semantic extension of algebraic loops

The current semantic framework excludes diagrams **B** containing algebraic loops, i.e., those satisfying $\exists b.\, b \in \mathbf{B} \wedge dp_loop(b)$, i.e. a direct feedthrough loop exists. For such diagrams, the semantics can be extended to handle certain solvable algebraic loops. To simplify the discussion, here we consider diagrams **B** without nested subsystems.

Analogous to solving an ODE system, all blocks constituting an algebraic loop must be considered together to solve the underlying algebraic equations. At time t, suppose the blocks forming the loop in **B** produce a system of equations: $\mathbf{o} = F(\mathbf{o}, \mathbf{i}, t)$. Here $\mathbf{o}$ denotes the set of output variables of all discrete and calculational blocks, $\mathbf{i}$ denotes the states of integrators or other external inputs (treated as known at time t), t is explicitly included because discrete blocks may behave differently at different times. Function $F(\mathbf{o}, \mathbf{i}, t)$ can be regarded as an operator on $\mathbb{R}^n$, where $n = dim(\mathbf{o})$. As long as this operator admits a unique fixed point, the values of all output variables in $\mathbf{o}$ at time t are uniquely determined, thereby defining the operational semantics of diagram **B**.

For example, in Fig. 12, both Bias and Gain are calculational blocks. At time t, Bias imposes the constraint $b = a + 1$, and Gain imposes $a = 0.5 \cdot b$. With no integrators or external inputs, we obtain the equation system:

$$\begin{cases} b = a + 1 \\ a = 0.5 \cdot b \end{cases}$$

Solving it yields $a = 1$, $b = 2$. Consequently, the operational semantics updates the values of variables a and b at time t to 1 and 2, respectively.

The current semantic framework excludes algebraic loops for two main reasons. First, for a general diagram containing algebraic loops, it is difficult to determine whether the corresponding equation system admits a unique solution, making it hard to define a well-founded operational semantics. Second, Simulink discourages the use of algebraic loops in models, as they can significantly degrade simulation performance, even prevent successful simulation altogether and hinder code generation.

Although the existence and uniqueness of solutions cannot be guaranteed for arbitrary equation systems, one can analyze specific classes of equations based on their structure. In particular, if the operator F satisfies the condition that there exists a constant $k \in [0, 1)$ such that for all $X, Y \in \mathbb{R}^n$,

$$|F(X, I, t) - F(Y, I, t)| \leq k\,|X - Y|$$

then $F(\cdot, I, t)$ is a contraction mapping on $\mathbb{R}^n$. By the Banach fixed-point (contraction mapping) theorem [27], such an operator possesses a unique fixed point, which serves as the unique solution to the equation $X = F(X, I, t)$. This provides a sufficient condition under which the operational semantics for diagrams with algebraic loops can be rigorously defined.

9.4. Alignment of precise semantics and numerical simulation

Our semantics models trajectories as exact, piecewise-analytic functions, given by the solutions of ODEs interleaved with discrete events. In contrast, numerical simulators like Simulink approximate these trajectories through discrete-time integration using variable- or fixed-step solvers. A critical challenge in this approximation is the detection of discrete events, particularly zero-crossings. For detection of zero-crossing events, Simulink employs variable-step solvers to monitor sign changes of signals over integration intervals, and iteratively, reduces the time step using root-finding methods such as bisection or interpolation to precisely locate the instant at which a discrete event occurs. The accuracy of this detection depends on the solver's tolerance settings and step-size control. When selecting the step-size type, one must balance accuracy and efficiency by considering factors such as model stiffness, whether the model is intended for code generation, etc.

In principle, exact solutions allow our semantics to determine zero-crossing events precisely. However, our current implementation detects events only at integer multiples of the fundamental sample time. Detecting events within a sampling interval would require identifying the earliest zero-crossing time in that interval, a feature not implemented in the current framework.

We are currently deriving a numerical semantics for Simulink from the operational semantics presented in this paper. The derivation involves one key modification: replacing the exact solution of ODEs with a verified numerical ODE solver featuring adaptive step-size and error control [28]. This numerical semantics is being formalized in Isabelle, from which reliable executable code can be generated directly. Based on formal proofs in Isabelle, we aim to establish the conformance bounds: for any Simulink diagram satisfying the **WDEF** and $None_Con_Sub$ conditions, for any finite time interval, an error ϵ can be computed such that the trajectory produced by the numerical semantics remains within an ϵ-neighborhood of the exact trajectories determined by the operational semantics of this paper. This approach bridges our formal semantics with Simulink's simulation results while maintaining provable guarantees.

10. Conclusion

We present complementary denotational and operational semantics for Simulink diagrams, which cover Simulink's core features (discrete/continuous blocks, hierarchical subsystems) and a practical subset of Simulink block library. We establish and formally verify the consistency between the two semantics, demonstrating the existence and uniqueness of denotational semantics, and the determinism of the operational semantics upon different execution orders. All definitions and proofs are proved in Isabelle/HOL to achieve high correctness guarantee. We have also implemented a translator that converts Simulink graphical diagrams to Isabelle representations. As an illustration, we apply our approach to a PID control example, to formally verify its convergence property. Furthermore, this framework enables both consistency checking of Simulink's formal translations and simulation validation.

In the future, we plan to extend the semantic framework along multiple dimensions. First, we aim to integrate this work with our previously formalized Stateflow semantics to enable the formalization of Simulink/Stateflow hybrid models. Second, to bridge the gap between numerical simulation and precise semantics, we will develop in Isabelle/HOL a numerical semantics incorporating adaptive step-size control and error estimation, better reflecting real solver behavior and ultimately supporting a reliable code generator. Finally, we will extend support to more module and subsystem types and apply the framework to verify concrete model translations (e.g. the translation from Simulink/Stateflow to HCSP in the Mars toolchain [9]).

CRediT authorship contribution statement

Yuzhen Qi: Writing – original draft, Software, Formal analysis. **Shuling Wang:** Writing – review & editing, Supervision, Conceptualization. **Xing Li:** Software. **Bohua Zhan:** Formal analysis, Conceptualization. **Naijun Zhan:** Supervision, Methodology, Conceptualization.

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.

Data availability

No data was used for the research described in the article.

References

[1] J. Nellen, T. Rambow, M.T.B. Waez, E. Ábrahám, J.-P. Katoen, Formal verification of automotive Simulink controller models: Empirical technical challenges, evaluation and recommendations, in: FM'18, in: LNCS, vol. 10951, Springer, 2018, pp. 382–398.

[2] L. Zou, N. Zhan, S. Wang, M. Fränzle, S. Qin, Verifying simulink diagrams via a hybrid hoare logic prover, in: EMSOFT 2013, IEEE, 2013, pp. 9:1–9:10.

[3] T. Liebrenz, P. Herber, S. Glesner, Deductive verification of hybrid control systems modeled in simulink with KeYmaera X, in: ICFEM 2018, in: LNCS, vol. 11232, Springer, 2018, pp. 89–105.

[4] 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 (5s) (2017) 176:1–176:24.

[5] N. Izerrouken, M. Pantel, X. Thirioux, Machine-checked sequencer for critical embedded code generator, in: ICFEM 2009, in: LNCS, vol. 5885, Springer, 2009, pp. 521–540.

[6] R. Reicherdt, S. Glesner, Formal verification of discrete-time MATLAB/Simulink models using boogie, in: SEFM 2014, in: LNCS, vol. 8702, Springer, 2014, pp. 190–204.

[7] W. Zhang, Q. Sun, C. Wang, Z. Liu, Towards correctness proof for hybrid simulink block diagrams, J. Syst. Archit. 141 (2023) 102922.

[8] Q. Sun, W. Zhang, C. Wang, Z. Liu, A contract-based semantics and refinement for hybrid simulink block diagrams, J. Syst. Archit. 143 (2023) 102963.

[9] B. Zhan, X. Xu, Q. Gao, Z. Ji, X. Jin, S. Wang, N. Zhan, Mars 2.0: A toolchain for modeling, analysis, verification and code generation of cyber-physical systems, 2024, arXiv arXiv:2403.03035.

[10] C. Zhou, J. Wang, A.P. Ravn, A formal description of hybrid systems, in: Hybrid Systems, in: LNCS, vol. 1066, 1996, pp. 511–530.

[11] S. Yi, S. Wang, B. Zhan, N. Zhan, Machine-checked executable semantics of stateflow, in: ICFEM 2022, Springer, 2022, pp. 421–438.

[12] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, Translating discrete-time simulink to lustre, in: EMSOFT 2003, in: LNCS, vol. 2855, Springer, 2003, pp. 84–99.

[13] A. Cavalcanti, P. Clayton, C. O'Halloran, Control law diagrams in circus, in: FM 2005, in: LNCS, vol. 3582, Springer, 2005, pp. 253–268.

[14] B. Meenakshi, A. Bhatnagar, S. Roy, Tool for translating simulink models into input language of a model checker, in: ICFEM 2006, in: LNCS, vol. 4260, Springer, 2006, pp. 606–620.

[15] A. Agrawal, G. Simon, G. Karsai, Semantic translation of simulink/stateflow models to hybrid automata using graph transformations, in: GT-VMT@ETAPS 2004, vol. 109, Elsevier, 2004, pp. 43–56.

[16] P. Filipovikj, N. Mahmud, R. Marinescu, C. Seceleanu, O. Ljungkrantz, H. Lönn, Simulink to UPPAAL statistical model checker: Analyzing automotive industrial systems, in: FM 2016, in: LNCS, vol. 9995, Springer, 2016, pp. 748–756.

[17] C. Chen, J. Dong, J. Sun, A formal framework for modeling and validating Simulink diagrams, Form. Asp. Comput. 21 (5) (2009) 451–483.

[18] T. Bourke, M. Pouzet, Zélus: a synchronous language with ODEs, in: HSCC '13, ACM, 2013, pp. 113–118.

[19] O. Bouissou, A. Chapoutot, An operational semantics for Simulink's simulation engine, SIGPLAN Not. 47 (5) (2012) 129–138.

[20] X. Xu, B. Zhan, S. Wang, J. Talpin, N. Zhan, A denotational semantics of Simulink with higher-order UTP, J. Log. Algebraic Methods Program. 130 (2023) 100809.

[21] V. Preoteasa, S. Tripakis, Refinement calculus of reactive systems, in: EMSOFT 2014, IEEE, 2014, pp. 2:1–2:10.

[22] I. Dragomir, V. Preoteasa, S. Tripakis, Compositional semantics and analysis of hierarchical block diagrams, in: SPIN 2016, in: LNCS, vol. 9641, Springer, 2016, pp. 38–56.

[23] V. Preoteasa, S. Tripakis, Towards compositional feedback in non-deterministic and non-input-receptive systems, in: LICS 2016, ACM, 2016, pp. 768–777.

[24] V. Preoteasa, I. Dragomir, S. Tripakis, Mechanically proving determinacy of hierarchical block diagram translations, in: VMCAI 2019, in: LNCS, vol. 11388, Springer, 2019, pp. 577–600.

[25] I. Dragomir, V. Preoteasa, S. Tripakis, The refinement calculus of reactive systems toolset, Int. J. Softw. Tools Technol. Transf. 22 (6) (2020) 689–708.

[26] MathWorks, Simulink® user's guide, 2018, http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf.

[27] J.M. Ortega, W.C. Rheinboldt, Iterative solution of nonlinear equations in several variables, in: Classics in Applied Mathematics, vol. 30, SIAM, Philadelphia, 2000.

[28] F. Immler, A Verified ODE Solver and Smale's 14th Problem (Ph.D. thesis), Institut für Informatik, Technische Universität München, 2018.