Verified Numerical Semantics and Code Extraction for Simulink

Yuzhen Qi, Shuling Wang, Xiangyu Jin, Xiong Xu, and Naijun Zhan

***Abstract*—Simulink is widely used in safety-critical domains for modeling, simulating, and generating code for embedded control systems. However, the trustworthiness of Simulink models is either by simulation or by manual proof possibly with some interactive proof assistant using existing approaches. The former lacks rigorous correctness guarantees, while the latter lacks efficiency, therefore cannot scale up to large systems. In this paper, we consider to explore the possibility to combine these two techniques together by providing a verified numerical semantics for Simulink formalized in Isabelle/HOL. Our semantics integrates a formally verified ODE solver with affine arithmetic for local error control, that formally defines error propagation through the execution of discrete–continuous diagrams and hierarchical conditional subsystems, establishing global error bounds with respect to an exact semantics. Based on the verified semantics, we leverage Isabelle's code extraction mechanism to automatically generate executable code. We evaluate our framework on three case studies with increasing complexity and diverse Simulink features, demonstrating that the generated trajectories remain within the computed error bounds while significantly reducing manual proof effort. This work provides an end-to-end formal framework combining mechanized rigor, numerical error control, and executable code generation for Simulink, enabling both reliable simulation and formal verification.**

***Index Terms*—Simulink continuous-discrete diagrams, Numerical semantics, Interactive theorem proving, Code generation**

I. Introduction

Simulink [1] is an industry-standard environment for modeling, simulating, and generating code for embedded control systems. It is widely adopted in safety-critical domains such as automotive, aerospace, and industrial automation, where system failures can have catastrophic consequences. Simulink provides two essential capabilities: simulation, which solves the continuous dynamics described by ordinary differential equations (ODEs) via numerical solvers while coordinating the execution of discrete blocks at their sample times; and automatic code generation, which translates Simulink models into C/C++ code. Despite its widespread use, Simulink lacks an official formal semantics. The simulation engine and code generator are industrial implementations whose correctness relies on testing rather than formal guarantees. This raises fundamental concerns: simulation in Simulink provides no ODE error bounds and explores only specific trajectories; code generation provides no formal assurance that the implementation faithfully reflects the model. Bridging this gap between industrial practice and formal guarantees is a grand challenge in embedded systems design.

Several lines of work have formalized Simulink semantics. Exact semantics [2]–[9] provide mathematical rigor, some with mechanized proofs in theorem provers [8]–[10]. However, they focus on computing or verifying exact ODE solutions, which lacks efficiency and does not scale up to large systems. Numerical semantics or approximate code generation [11]–[14] enable more efficient execution by approximating continuous dynamics, yet lack mechanized correctness formalization. Meanwhile, verified ODE solvers [15] provide rigorous error control for continuous dynamics alone, but do not address the discrete–continuous interplay central to Simulink, whose complexity arises from different sample rates, conditional subsystems, and feedback loops. To address the above limitations, we present a verified numerical semantics for Simulink formalized in Isabelle/HOL. It integrates error analysis and control for ODE solving, captures the compositional behavior of hierarchical systems combining discrete and continuous dynamics, and provides a path to more reliable executable code via Isabelle's code extraction mechanism.

This semantics builds upon an existing exact semantics [10], introducing several key innovations. Exact ODE solutions are replaced by numerical integration to obtain computable trajectory approximations, using an adaptive Runge–Kutta method with rigorous local error control. This method builds upon a formally verified ODE solver framework [15], which provides certified Runge–Kutta implementations with adaptive step-size control and affine arithmetic for tight error enclosures. Crucially, we integrate this solver into the complete Simulink semantic framework, handling the interaction between continuous dynamics and discrete control in hierarchical diagrams. To this end, we formally define how local errors propagate through mixed discrete–continuous systems, specifically, how the execution of different components influences error accumulation. Moreover, we address the merging of execution paths that arises from the nondeterminism inherent in approximating conditional subsystems. By solving these challenges, we finally establish a global error bound between the numerical and exact semantics, which depends on both the solver's local error tolerance and the propagation of errors throughout the entire system execution.

Yuzhen Qi was with the Key Laboratory of System Software, Institute of Software, Chinese Academy of Sciences, Beijing, China, when this work was performed. He is now with the Institute for Advanced Algorithms Research, Shanghai, China (e-mail: qiyz@ios.ac.cn).

Shuling Wang, Xiangyu Jin, and Xiong Xu are with the National Key Laboratory of Space Integrated Information Systems, Institute of Software, Chinese Academy of Sciences, Beijing, China (e-mail: {wangsl, jinxy, xux}@ios.ac.cn).

Naijun Zhan is with the School of Computer Science and the MoE Key Laboratory of High Confidence Software Technologies, Peking University, Beijing, China, and also with Zhongguancun Laboratory, Beijing, China (e-mail: njzhan@pku.edu.cn).

From this verified numerical semantics, we then obtain executable code via Isabelle's code extraction. Leveraging Isabelle's built-in mechanism for translating higher-order logic definitions into executable programs, we automatically generate SML code that directly implements the formal semantics. While the extraction mechanism resides within Isabelle, it is widely regarded as substantially more reliable than Simulink's unverified code generator. This yields, for the first time, a fully formalized and executable toolchain for Simulink models, spanning from high-level semantics to runnable code with formally traceable guarantees.

Building on these results, we obtain a unified framework for both simulation and verification of Simulink models. First, the extracted code enables trusted simulation: it executes models with formally guaranteed error bounds, providing an independent check against Simulink's own simulation results. Second, the established error control and bound support formal verification via over-approximation: if the numerical trajectory remains within a safe region, the exact trajectory is guaranteed to stay within that region expanded by the error bound. We evaluate our framework on three case studies, comparing the extracted code's output against Simulink simulations and validating the formal error bounds, thereby enabling verification. The evaluation also highlights the reduction in manual effort compared to existing verification using exact semantics [10], where proofs required more than one thousand of lines of interactive theorem proving for a PID control example, now replaced by automated execution with formal guarantees.

Contributions. In summary, this paper makes the following contributions:

- A formally verified numerical semantics for Simulink in Isabelle/HOL, which covers continuous dynamics, discrete control, their interaction and hierarchical composition, and is the first executable numerical semantics for Simulink with machine-checked correctness guarantees.
- Formal error control and propagation in Isabelle/HOL. By integrating a verified ODE solver, we formally establish global error bounds between the numerical and exact semantics, accounting for both ODE approximation errors and their propagation through discrete/continuous interactions.
- A trusted path from semantics to executable code. Using Isabelle's code extraction, we generate SML programs that directly implement the numerical semantics, yielding a fully formalized toolchain from high-level models to runnable code.
- An end-to-end framework unifying simulation and verification. We demonstrate the framework on three case studies, validating the error bounds against Simulink simulations and illustrating its potential for verifying safety-critical embedded systems.

Paper outline. After presenting the related work, the rest of the paper is organized as follows. Section II introduces the preliminary knowledge about Simulink and ODE solving methods. Section III presents the formal syntax of Simulink diagrams in our framework. Section IV describes the verified ODE solver we employ and its adaptation to our semantics. The main contributions are presented in two parts: the numerical semantics of Simulink in Section V, and its Isabelle implementation together with code generation in Section VI. Section VII illustrates our approach using three case studies, and Section VIII concludes the paper.

Related work. Prior work on Simulink semantics divides into exact and numerical approaches. Exact semantics provide rigorous mathematical foundations. Xu et al. [7] formalize a denotational semantics for Simulink diagrams in the hybrid Unifying Theories of Programming (UTP) and prove its determinacy. [6] propose a contract-based semantics based on a set of composition operators and a verification framework via refinement calculus. However, neither approach provides machine-checked formalization, and they do not scale to large systems due to the computational cost of solving ODEs exactly. More work translates Simulink diagrams to various formal models for verification [2], [4], [16]–[18]. However, these approaches enable verification but do not aim at executable implementations. In [3], [13], [14], Simulink diagrams are translated to HCSP formal models for verification, and subsequently discretized to generate code with approximate error control. However, neither the translation nor the discretization process is machine-checked. Numerical semantics approximate continuous dynamics to enable efficient execution. HySon [12] provides numerical semantics using zonotope arithmetic to bound truncation error during simulation and further supports zero-crossing detection. It shows that error-aware simulation is feasible, but its error bounds are computed by unverified floating-point implementations, without machine-checked formalization. The work [8], [9] defines a mechanized execution semantics of Simulink diagrams in the Refinement Calculus of Reactive Systems (RCRS). However, RCRS handles the ODEs of continuous blocks using fixed-step Euler assignments, without formal error control.

Building on the need for numerically solving ODEs with guaranteed error bounds, a gap left by existing Simulink formalizations, several works have mechanized the verification of numerical ODE solvers in theorem provers. Faissole et al. [19] formally verified in Coq the rounding error analysis of explicit Runge–Kutta methods under floating-point arithmetic, deriving strict error bounds for linear systems. Park and Thies [20] proposed a Coq framework based on Taylor models and power series, enabling certified program extraction for nonlinear polynomial ODEs with arbitrarily prescribed precision. Hickman et al. [21] integrated a computer algebra system (CAS) into Isabelle/HOL, leveraging symbolic computation to verify symbolic solutions of ODEs and improve automation. Immler [15], [22] pioneered in Isabelle/HOL, combining affine arithmetic with Runge–Kutta methods to define a verified ODE solver, refined to floating-point code via the Autoref framework and automatically generating trusted SML code. This work yields rigorous numerical enclosures and serves as the basis for the numerical semantics of Simulink in the present paper. Extending formalization to implementable code,

Bryant et al. [23] presented a verification framework based on interaction trees in Isabelle/HOL, allowing users to declare numerical programs with invariant annotations and export machine-verified code via Isabelle's code generator. Notably, all the above work focuses on solving isolated ODE systems, rather than the mixed discrete/continuous setting of Simulink.

II. Preliminaries

A. An Overview of Simulink

Simulink [1] is a widely used graphical environment for modeling and simulating dynamic systems. A model is built by connecting blocks from a rich library; blocks can be grouped into subsystems, which may themselves contain subsystems, forming a hierarchical structure. This paper presents a verified numerical semantics for Simulink and demonstrates its extraction into executable code via Isabelle, strengthening correctness through the combination of the formally verified semantics and the extraction process. Below, we introduce the basic elements of Simulink.

a) Blocks, Subsystems, and Diagrams: Each block has input and output ports, and may have internal state. Its behavior is defined by two functions: an output function that computes the outputs, and an update function that determines the next state, both based on the current inputs and state. Blocks execute according to a *sample time*: a discrete block (sample time > 0) updates only at integer multiples of its sample time, while a continuous block (sample time $= 0$) evolves continuously. Among continuous blocks, those without internal state (e.g., arithmetic operators) are called *calculational blocks*; integrators are the prototypical continuous blocks with state, representing ordinary differential equations.

Subsystems encapsulate a set of blocks. Three types are relevant to this paper: atomic subsystems (executed as a single unit), enabled subsystems (execute only when an enable signal is true), and triggered subsystems (execute on a rising or falling edge of a trigger signal). The overall diagram may mix discrete and continuous components, then its execution is paced by the *fundamental sample time*, defined as the greatest common divisor of all discrete block sample times. Without loss of generality, we assume the fundamental sample time is normalized to 1 throughout the paper.

b) A Running Example: Figure 1 shows a running example of a closed-loop control system. The plant is a continuous first-order unstable system modeled by an integrator with positive feedback. The controller is a discrete PID implemented using Unit Delay blocks: the error signal e (difference between a constant setpoint and the sampled plant output) is processed through proportional, integral, and derivative paths, and the resulting control signal c is fed back to the plant. This example illustrates the interaction between continuous dynamics and discrete control, and will be used to demonstrate our numerical semantics and code generation, as well as to highlight the reduction in verification effort compared to prior exact-semantics approaches.

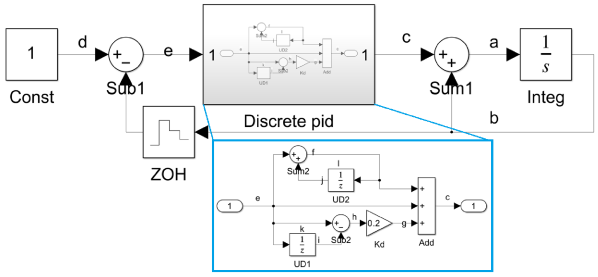

Fig. 1. Simulink diagram of a discrete PID control system

c) Data Dependencies: Connections between blocks impose an implicit execution order: if block B depends on the output of block A, then A must execute before B. When such dependencies cycle exclusively through stateless blocks, an algebraic loop arises, producing an ill-defined instantaneous system of equations. Stateful blocks, such as integrators and Unit Delays, resolve such cycles because their outputs depend on internal states rather than immediate inputs. In our example (Figure 1), the feedback loop is therefore well-defined: it is broken by the integrator in the plant and by the discrete sampling within the PID controller.

This paper formalizes a numerical semantics for Simulink in Isabelle/HOL with formally verified error bounds, and demonstrates how to extract executable code directly from this verified semantics. The following sections define the syntax and semantics step by step, with its error bounds relative to an ideal (exact) semantics established, and then present the code generation mechanism.

B. Numerical Methods for Ordinary Differential Equations

The Initial Value Problem (IVP) for an ODE system is formulated in the following form: $\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}), \mathbf{y}(t_0) = \mathbf{y}_0$, where $\mathbf{y}(t) = [y_1(t), y_2(t), \ldots, y_n(t)] \in \mathbb{R}^n$ denotes an n-dimensional vector of functions, $t \in \mathbb{R}$ is the scalar independent variable, $\frac{d\mathbf{y}}{dt} = [\frac{dy_1}{dt}, \frac{dy_2}{dt}, \ldots, \frac{dy_n}{dt}]^T$ is the vector of the first-order derivatives of the functions with respect to t, $\mathbf{f}(t, \mathbf{y})$ is an n-dimensional continuous vector field mapping from an open set $D \subseteq \mathbb{R} \times \mathbb{R}^n$ to $\mathbb{R}^n$. The vector $\mathbf{y}$ starts at t_0 with initial value $\mathbf{y}_0$. The existence and uniqueness of a solution to this IVP are guaranteed by the well-known Picard–Lindelöf theorem (also known as the Cauchy–Lipschitz theorem).

Theorem 1 (Picard-Lindelöf)**.** *Let D be an open set in $\mathbb{R}\times\mathbb{R}^n$, and let $(t_0, \mathbf{y}_0) \in D$. Suppose the function $\mathbf{f}(t, \mathbf{y})$ satisfies the following conditions:*

1) *$\mathbf{f}$ is continuous on D;*
2) *$\mathbf{f}$ is locally Lipschitz in $\mathbf{y}$ on D, i.e. for any compact subset $K \subset D$, there exists a constant L_k such that for all $(t, \mathbf{y}_1), (t, \mathbf{y}_2) \in K$, $\|\mathbf{f}(t, \mathbf{y}_1) - \mathbf{f}(t, \mathbf{y}_2)\| \leq L_K \|\mathbf{y}_1 - \mathbf{y}_2\|$.*

Then there exists $\delta > 0$ such that the IVP admits a unique solution $\mathbf{y}(t)$ on the interval $[t_0 - \delta, t_0 + \delta]$.

If the vector field satisfies a stronger global Lipschitz condition on the entire space, the theorem extends to a global

existence and uniqueness result, guaranteeing that a unique solution exists on the whole time domain. However, for the vast majority of real-world systems, analytical solutions are intractable, which necessitates the application of numerical methods to compute approximate solutions.

Numerical methods operate by discretizing the continuous ODE system and constructing a sequence of discrete vector approximations $\mathbf{y}_n \approx \mathbf{y}(t_n)$ at a set of time points $t_0 < t_1 < \cdots < t_N$. For example, one-step methods admit the general formulation $\mathbf{y}_{n+1} = \mathbf{y}_n + h\Phi(t_n, \mathbf{y}_n, h; \mathbf{f})$, where $h = t_{n+1} - t_n$ is the step size, and Φ is the increment function. Starting from the initial condition $(t_0, \mathbf{y}_0)$, the formula is applied iteratively to produce a sequence of discrete points approximating the exact solution trajectory. The simplest one-step scheme is the Euler method, which relies on a first-order Taylor expansion at the left endpoint of the interval, leading to the update $y_{n+1} = y_n + hf(t_n, y_n)$. Runge–Kutta methods achieve higher precision by evaluating multiple intermediate slopes within each step and forming a refined increment function via weighted averaging. The classical fourth-order Runge–Kutta (RK4) method proceeds by computing four intermediate slopes k_1, k_2, k_3, k_4, where each slope is evaluated at a point determined by a combination of the previous slopes and the step size h. The next approximation is then updated using a weighted combination of these slopes:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

A critical performance metric for any numerical method is its order of accuracy, which quantifies the rate at which the approximation error decreases as the step size is reduced. The local truncation error is the error incurred in a single step of the method, assuming that the previous approximation $\mathbf{y}_n$ is exact. The global error is the cumulative error over the entire interval of integration, resulting from the propagation of local truncation errors across all steps. Higher-order methods thus exhibit faster convergence to the exact solution as the step size h approaches zero. The Euler method is a first-order scheme with local truncation error $O(h^2)$ and global error $O(h)$. By contrast, RK4 is a fourth-order method with local truncation error $O(h^5)$ and global error $O(h^4)$.

III. Formal Syntax of Simulink Diagrams

Before giving the formal semantics, a formal syntax for describing graphical Simulink diagrams is necessary. In this work, we adopt the formal syntax of Simulink blocks and diagrams defined in [10], which is presented as follows.

Diagrams	*Diag*	:=	**B**
Blocks	*B*	:=	*dB* | *cB* | *Sub*
Discrete	*dB*	:=	$(\mathbf{i}, \mathbf{o}, \mathbf{s}, \mathbf{s}_0, st, \mathbf{di}, \mathbf{f}, \mathbf{g})$
Continuous	*cB*	:=	$(\mathbf{i}, \mathbf{o}, \mathbf{s}_0, \mathbf{f}, isC)$
Subsystem	*Sub*	:=	$(\mathbf{i}, \mathbf{o}, \mathbf{B}, st, Sty)$
	Sty	:=	Atomic | Enab *sre sig* | Trig *tty sig*
Functions	$\mathbf{f} \ni \mathbf{f}_o$	:	$\mathbb{R}^{\lvert\mathbf{s}\rvert} \times \mathbb{R}^{\lvert\mathbf{i}\rvert} \to \mathbb{R}$
	$\mathbf{g} \ni \mathbf{g}_s$	:	$\mathbb{R}^{\lvert\mathbf{s}\rvert} \times \mathbb{R}^{\lvert\mathbf{i}\rvert} \to \mathbb{R}$

The above definition denotes sequences in boldface, where a sequence is either empty (ϵ) or recursively formed as $a \cdot \mathbf{a}$. A Simulink diagram is formally defined as a sequence of blocks, denoted by **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: **i**, **o** and **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, **di** is a sequence of Boolean values indicating whether the corresponding input is instantaneously required by the outputs, **f** and **g** two sequences of functions that map states and inputs to the outputs and updated states, respectively.

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 **i** and **o** denote the input and output variables, respectively, $\mathbf{s}_0$ specifies the initial values of the states, **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 **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 **f** defines functions that map the inputs to the derivatives of the outputs.

A subsystem block is represented as a tuple $(\mathbf{i}, \mathbf{o}, \mathbf{B}, st, Sty)$, where **i** and **o** are defined as above, **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.

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.

a) Example: Take Figure 1 as an example. The Unit Delay block UD1 has input e, output i, state variable k, initial state 1 and sample time 1; it is defined as $(e, i, k, 1, 1, 0, \lambda s.\lambda i.s, \lambda s.\lambda i.i)$. According to the definition, the output depends not on the current input but on the previous state k, while the state is updated to the current input value e. The calculational block Sum1 has inputs c and b, output a, and implements $a = c + b$; it is defined by $(c \cdot b, a, -, \lambda i_1\ i_2.\ i_1 + i_2, 1)$. The Integrator block Integ has input a, output b, initial value 0.5, and models the ordinary differential equation $\dot{b} = a$; it is correspondingly defined as $(a, b, 0.5, \lambda i.\ i, 0)$. The subsystem Discrete pid is atomic with input e and output c, consisting of the blocks within

the blue box, with sample time 1; it is thus represented as $(e, c, \mathsf{Sum2} \cdot \mathsf{UD1} \cdot \mathsf{Sub2} \cdot \mathsf{UD2} \cdot \mathsf{Kd} \cdot \mathsf{Add}, 1, \mathsf{Atomic})$.

Well-formedness Constraints A structure *Diag* defined according to the above syntax is a *valid Simulink diagram* iff it satisfies a set of well-formedness conditions, collectively denoted by the predicate *wf*(*Diag*)(adapted from [10]). These conditions include, for instance, that block outputs are distinct, and that the lengths of outputs and states match those of the output and state functions, respectively. In addition to these well-formedness conditions, a syntactic constraint requires that subsystems contain only discrete blocks. This design choice ensures modular composability in the semantics: since continuous integrators must be solved collectively via a global ODE system, consistent with Simulink's solving strategy, subsystems containing integrators are required to be flattened before semantic execution. As discussed in the existing work [10], this syntactic restriction on subsystems can be relaxed, and other Simulink features not covered, such as derivative blocks, variable transport/time delay, and certain complex subsystem types *etc*, can be incorporated through syntactic and semantic extensions. With this syntax, a representative set of Simulink blocks covering the essential functionality for modeling embedded control systems can be formalized [10]. Table I lists the specific blocks formalized in this framework.

TABLE I
SUPPORTED SIMULINK SUBSET

Category	Blocks / Features
Sources	Constant, Clock*, Sine*, Pulse Generator*, Chirp Signal*
Continuous	Integrator* (reset, saturation, external IC, state port), State-Space*, First Order Hold*
Discontinuities	Saturation, Coulomb & Viscous Friction, Dead Zone, Relay(discrete)
Discrete	Discrete PID Controller, Unit Delay, Zero-Order Hold
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)
User-Defined Functions	Fcn
Simulink Extras	Fahrenheit to Celsius, Celsius to Fahrenheit
Subsystems	Atomic, Enabled (held/reset), Triggered (rising/falling edge), Hierarchical
Data Types	Real-valued scalars and vectors

* Composed of continuous blocks; cannot be placed inside subsystems.

IV. AN ADAPTED VERIFIED ODE SOLVER

This section introduces a verified ODE solver developed in Isabelle/HOL, which serves as the foundation for defining the numerical semantics of Simulink. We begin by presenting the general concepts of affine forms and affine arithmetic, followed by the verified ODE solver adapted from the work of Immler [15].

A. Affine Forms and Arithmetic

Affine forms represent sets of high-dimensional vectors and compute over-approximations of results. Unlike interval arithmetic, which loses variable dependencies and suffers from the wrapping effect (e.g., $x-x$ fails to yield zero), affine forms capture linear dependencies between variables by introducing noise symbols ε_i, yielding tighter enclosures. Specifically, an affine form $\hat{x}$ represents a set of values as

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i$$

where x_0 is the central value, $x_i\varepsilon_i$ are noise terms, and ε_i are noise symbols satisfying $|\varepsilon_i| \leq 1$. The degree of $\hat{x}$, denoted as $deg(\hat{x})$, is the maximum index of non-zero coefficients of the noise terms. The corresponding set represented by the above affine form is

$$set(\hat{x}) = \{x_0 + \sum_{i=1}^{n} x_i \varepsilon_i \mid \varepsilon_i \in [-1, 1]\}$$

We define aform $:= \mathbb{R} \times (\mathbb{N} \to \mathbb{R})$ to denote an affine form, where the first term is the central value and the second term maps the index of a noise term i to its coefficient x_i, with only a finite number of non-zero coefficients.

To derive an over-approximation of the value set of operation results, it is necessary to define corresponding operations applicable to affine forms according to common basic function operations on real values, such as $+$, $-$, $\times$, /, power, *etc*. For an operation $op(x_1, ..., x_n)$ defined on real values, its corresponding operation on affine forms, denoted by $\hat{op}$, must satisfy two conditions: the result is an affine form, and it satisfies the inclusion relation:

$$set(\hat{op}(\hat{x}_1, ..., \hat{x}_n)) \supseteq \{op(x_1, ..., x_n) \mid \forall i.\ x_i \in set(\hat{x}_i)\}$$

Affine forms can accurately propagate dependencies under linear operations; when dealing with nonlinear operations such as multiplication, higher-order noise is encapsulated by introducing new noise symbols. For example, if $\hat{a} = a_0 + \sum_{i=1}^{n} a_i\varepsilon_i$ and $\hat{b} = b_0 + \sum_{i=1}^{n} b_i\varepsilon_i$, we can define

$$\hat{a} \hat{+} \hat{b} = a_0 + b_0 + \sum_{i=1}^{n} (a_i + b_i)\varepsilon_i$$

$$\hat{a} \hat{\times} \hat{b} = a_0 b_0 + \sum_{i=1}^{n} (a_0 b_i + a_i b_0)\varepsilon_i + (\sum_{i=1}^{n} |a_i|)(\sum_{i=1}^{n} |b_i|)k$$

where k is a new noise symbol different from all ε_i (it can also be taken as ε_{n+1} here). Repeated affine operations tends to generate excess noise symbols, which reduces computational efficiency. New noise symbols introduced by these operations are independent of all existing ones and have no linear dependencies. Consequently, these noise terms can be merged into a single term whose coefficient is the sum of the absolute values of the original coefficients, while the represented set remains unchanged. For example, if

$$\hat{a} = a_0 + \sum_{i=1}^{n} a_i\varepsilon_i + b_1 k_1 + b_2 k_2$$
$$\hat{a}' = a_0 + \sum_{i=1}^{n} a_i\varepsilon_i + (|b_1| + |b_2|)k$$

we can conclude that set $set(\hat{a}) = set(\hat{a}')$.

To improve computational efficiency, the error-bounded affine form aform_err := aform$\times\mathbb{R}$ is defined as a product type where the last term stores the independent term's coefficient. It

has the same degree as the first term. Basic operations on affine forms (aform) can be extended to error-bounded affine forms (aform_err). Taking addition and multiplication as examples: if $\tilde{a} = (a_0 + \sum_{i=1}^{n} a_i\varepsilon_i, e_a)$ and $\tilde{b} = (b_0 + \sum_{i=1}^{n} b_i\varepsilon_i, e_b)$, then

$$\tilde{a}\tilde{+}\tilde{b} = (a_0 + b_0 + \sum_{i=1}^{n}(a_i + b_i)\varepsilon_i, e_a + e_b)$$

$$\tilde{a}\tilde{\times}\tilde{b} = (a_0b_0 + \sum_{i=1}^{n}(a_0b_i + a_ib_0)\varepsilon_i, \\ (e_a + \sum_{i=1}^{n}|a_i|)(e_b + \sum_{i=1}^{n}|b_i|) + e_ab_0 + e_ba_0)$$

In the ODE numerical semantics, another error source arises from using finite-precision binary fractions to approximate real numbers, called rounding error. Corresponding affine forms can be defined with respect to rounding error, ensuring the result set is an over-approximation of the original number.

1) Our Extensions: In our framework, we define the conversion functions between aform and aform_err, preserving the set of value and dependencies of noise terms.

$$\text{af\_to\_afe}(\hat{x}) = (\hat{x}, 0)$$
$$\text{afe\_to\_af}(n, (\hat{x}, a)) = \hat{x} + a \cdot \varepsilon_n$$

where n is the index of the newly added noise symbol. Specially, if a is a real number, we use $\hat{a}$ to denote the affine form with center a and all noise term coefficients being 0, and $\tilde{a} = \text{af\_to\_afe}(\hat{a})$ is the corresponding error-bounded form. In our semantics, we use both versions of affine forms to represent the over-approximation of exact trajectories as needed. Since they are convertible between each other, we will not distinguish them in the context.

In the presence of conditional execution in Simulink, we often need to merge two affine forms. The merge operation must produce a single affine form that simultaneously encloses both input forms. Let $\tilde{x}$ and $\tilde{y}$ be two affine forms with errors, we define their *linear merge* as follows:

$$\tilde{x} \oplus_n \tilde{y} = 0.\tilde{5}\tilde{\times}(\tilde{x}\tilde{+}\tilde{y})\tilde{+}0.5\varepsilon_n\tilde{\times}(\tilde{x}\tilde{-}\tilde{y})$$

where $n > max(deg(\tilde{x}), deg(\tilde{y}))$ and ε_n is independent of all noise terms in $\tilde{x}$ and $\tilde{y}$. This operation satisfies that $set(\tilde{x} \oplus_n \tilde{y}) \supseteq set(\tilde{x}) \cup set(\tilde{y})$ and if $\tilde{x} = \tilde{y}$, then $\tilde{x} \oplus_n \tilde{y} = \tilde{x}$. Since the set represented by an affine form is always convex, any affine form containing both $set(\tilde{x})$ and $set(\tilde{y})$ must enclose all points on the line segment connecting any two points in the union set. Consequently, this merging operation yields a relatively tight enclosure.

B. An Adapted Verified ODE Solver

The numerical semantics for continuous diagrams lies in employing numerical algorithms to solve ODEs. To ensure the performance of the code generated from numerical semantics and to facilitate global error analysis and local error control, we utilize the ODE solver implemented by Immler [15] in Isabelle. This solver contributes in the following aspects: First, it formalizes the initial value problems (IVPs) and provides a machine-checked proof of the Picard–Lindelöf theorem, guaranteeing local and global existence and uniqueness of solutions. Second, it implements a Runge–Kutta method with adaptive step size and local error control, using affine forms and arithmetic to represent and compute the sets that over-approximate the exact solutions rather than single points, thereby enabling rigorous error estimation. In particular, at each step, the discretization error (from the Taylor remainder) and floating-point rounding error are modeled as affine terms and incorporated. Third, it supports automatic generation of executable SML code from the formal ODE algorithms.

The Runge–Kutta Algorithm implemented by Immler consists of three stages: It first certifies the existence of a solution via fixed-point iteration, then calculates an estimation set containing errors, and finally dynamically adjusts next step size based on errors.

- Before performing numerical integration, the algorithm first ensures that a unique solution exists within the current step size h and remains in a bounded region. It defines a set operator $Q_h(X) = X_0 + [0, h] \cdot f(X)$, an over-approximation of the Picard operator, and searches for a set C satisfying the post-fixed point condition $Q_h(C) \subseteq C$. Starting from an initial guess (e.g., X_0), the function `cert-stepsize` iteratively computes $Q_h(C)$; If containment holds, the unique solution is guaranteed to exist and lie within C. If the iteration fails to converge, the step size h is halved and the process restarts.
- Next, after certifying the step size h and the coarse estimation set C, the algorithm performs a single Runge–Kutta step and computes the local error. The function rk2-remainder$_h(X_0, C)$ computes the Lagrange remainder as an affine form representing all possible errors (denoted by R), from which width(R) yields the maximum error value. Meanwhile, rk2$_h(X_0)$ computes a second-order Runge–Kutta step from the initial set X_0 with step size h, yielding a numerical solution set. Adding the remainder to this result produces a set (say X_1), which over-approximates the set of exact solutions.
- Finally, it handles error control and step size adjustment. The function Threshold$(X_1, rtol, atol)$ provides the error limit given the relative tolerance *rtol*, absolute tolerance *atol*, and the over-approximation of the exact solution X_1. If the error ε does not meet the requirements, the step size is shortened, and the process returns to the first step for recalculation. Otherwise, the step is accepted, and adapt-stepsize(h, ε) is used to adjust the step size based on error ε, yielding h_{next} as the initial trial step size for the next iteration.

Through $X_1 = X_{approx}\hat{+}R$, the algorithm guarantees that the true solution $\varphi(x_0, h)$ lies within the computed set X_1. By monitoring the remainder, it dynamically adjusts the step size, increasing it in smooth regions for efficiency and decreasing it where changes are rapid to maintain accuracy. Repeating this process from the initial time until the target termination time yields the numerical algorithm with local error control, denoted *step_until_afe*$(optns, ode, X_0, t)$. Here,

optns represents parameters including precision, relative error tolerance and absolute error tolerance, *ode* is the ODE, and X_0 is the initial affine form; the function returns a list of affine forms over-approximating the reachable set at time t.

In practical cases, results for the entire process may be required to facilitate plotting or determining ranges over the whole interval. To this end, we introduce a new function *step_inter_afe* to the above solver, which, building upon *step_until_afe*, additionally records the result of each step and its corresponding time, as well as the over-approximation enclosure over that interval obtained by substituting the entire time interval into the step size (i.e., $\text{rk2}_{[0,h]}(X_0)$).

V. Numerical Semantics for Simulink

The operational semantics of a Simulink diagram specifies how the values of its variables, both outputs and states, evolve over time through the execution of its blocks. In this section, we introduce a numerical semantics of Simulink that approximates the behavior of the ODE system within it while accounting for two critical concerns: first, error control ensures that the approximated values remain within a prescribed tolerance of the exact solution; second, error propagation captures how numerical errors accumulate and propagate through the execution of Simulink discrete-continuous blocks.

To address these concerns, we define the numerical semantics as transitions over timed tubes. Formally, a timed tube $\tilde{h} : \mathbb{R}^+ \to (\text{Var} \to \text{aform\_err})$ assigns to each time point in $[0, +\infty)$ an affine form with error for every variable. Thus, at each time point, the semantics returns an over-approximation, represented as an affine form, that is guaranteed to contain the exact system solution, and simultaneously provides a formal error bound. Based on this definition, the numerical semantics is defined as transitions over timed tubes. Next we define different forms of transitions for the numerical semantics of distinct Simulink components, corresponding to their execution over time.

A. Numerical Semantics for Discrete Diagrams

Figure 2 presents the numerical semantics of discrete blocks, subsystems and diagrams, which are defined both at discrete time instants n and over the open intervals $(n, n+1)$ for $n \in \mathbb{N}$.

1) Discrete Blocks: Intuitively, the update of the outputs and states of a discrete block *dB* with sample time *st* satisfies:

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

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. Formally, the semantics of a discrete block *dB* at a given time point n is governed by two separate rules (dB-O) and (dB-S), corresponding to its output computation and state update respectively. Both rules take an initial timed tube $\tilde{h}$ under the enabled condition *ena* and reset condition *res*, and produce a new tube $\tilde{h}'$. In rule (dB-O), $\tilde{preS}$ computes the previous states, i.e.

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

$\tilde{\mathbf{o}}'$ returns the output value at the previous time point. The output $\mathbf{o}$ at time n is then updated to $\tilde{o}_{new}$, which is computed by applying the output function $\tilde{\mathbf{f}}_o$ to inputs $\mathbf{i}$ and previous states $\tilde{\mathbf{s}}'$, if the block is enabled to execute; otherwise, it retains its value from the previous time point. (dB-O) can be explained similarly.

The execution of a discrete subdiagram (i.e., a block list) is equal to the execution of all blocks within it in an order that conforms to their data dependency. It is defined by rule (dDig), which proceeds in two phases separating output and state updates: first, the outputs of the blocks are computed in an order that respects the data dependency relation, ensuring that a block whose input depends on another block's output executes after that block (implemented by the *sort* function in (dDig-O)); subsequently, the states of all blocks are updated in an arbitrary order, as no data dependencies remain at this stage (dDig-S).

The above numerical semantics for discrete blocks and block lists closely mirrors the structure of the precise operational semantics in [10], with two key distinctions: States are replaced by tubes, and functions over states are lifted to their tube counterparts to handle error propagation as defined in the previous section. However, as we will see, subsystems introduce differences due to the nondeterminism that arises when evaluating their execution conditions over a tube.

2) Discrete Subsystems: The semantics for subsystems are defined by three rules: (Atomic), (Enabled), and (Triggered), depending on the subsystem type. For atomic subsystems, the internal blocks are executed by recursively invoking the semantics of the inside discrete blocks. For enabled and triggered subsystems, however, the semantics must additionally account for the enabled or triggered conditions. We first introduce predicates that encode these conditions.

In the exact semantics, evaluating a condition on a concrete state yields a definite true or false. As an instance, in the exact semantics, we define the reset condition as follows:

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

which determines whether the subsystem should be reset at time t, depending on the reset signal *sig* and the reset condition *sre*. In the numerical semantics, however, the state becomes a tube, a set of possible concrete states. Consequently, a condition may evaluate to true for some concrete states and false for others. To capture this, let $\tilde{sup}(\tilde{x})$ and $\tilde{inf}(\tilde{x})$ denote the supremum and infimum of the set represented by $\tilde{x}$, respectively. We then define the following predicates to capture the possible truth of conditions under over-approximation.

$$\tilde{En}_u(sig, \tilde{h}, t) = (\tilde{sup}(\tilde{h}(t)(sig)) > 0)$$
$$\tilde{Res}_u(sig, sre, \tilde{h}, t) = (sre = 0 \wedge \tilde{sup}(\tilde{h}(t)(sig)) > 0 \wedge \tilde{inf}(\tilde{h}(t-1)(sig)) \leq 0)$$
$$\tilde{Trig}_u(sig, tty, \tilde{h}, t) = ((tty = 0 \ or \ 2) \wedge \tilde{sup}(\tilde{h}(t)(sig)) > 0 \wedge \tilde{inf}(\tilde{h}(t-1)(sig)) \leq 0 \vee (tty = 1 \ or \ 2) \wedge \tilde{inf}(\tilde{h}(t)(sig)) < 0 \wedge \tilde{sup}(\tilde{h}(t-1)(sig)) \geq 0)$$

For each of the above predicates, we define a dual predicate $\tilde{En}_d$, $\tilde{Res}_d$ and $\tilde{Trig}_d$, obtained by changing $\tilde{sup}$ with $\tilde{inf}$, $\tilde{inf}$ with $\tilde{sup}$, and taking the negation of the original condition. All the predicates now characterize whether a behavior is possible. For example, if $\tilde{Res}_u$ holds, meaning there may exist some state in the tube where the reset condition is true, then the subsystem is possibly reset. Conversely, if $\tilde{Res}_d$ holds, meaning there may exist some state where the reset condition is false, then the subsystem is possibly not reset.

Building on the above definitions, the semantics of an enabled subsystem Enab *sre sig* is defined by first considering several distinct cases and then obtaining the resulting tube by taking an affine over-approximation of the tubes computed from these cases, ensuring that the final tube remains representable as a single affine form and furthermore provides a sound over-approximation of the exact solutions. Rule (Enabled) defines the execution of an enabled subsystem *Sub* under external enabled condition *ena* and reset condition *res*, starting from the initial tube $\tilde{h}$. The execution of *Sub* corresponds to running its internal subdiagram **B** with a new enabled condition, i.e. the *conjunction* of *ena* and the enabled condition imposed by *Sub*, and a new reset condition, i.e. the *disjunction* of *res* and the reset condition imposed by *Sub*. Based on this, we consider three specific cases:

- If the enabled condition imposed by *Sub* is true, the overall enabled condition for **B** follows the external *ena*. The overall reset condition for **B** is then determined as follows: if the reset condition imposed by *Sub* is true, the overall reset condition is *True*; otherwise, it follows the external *res*. As shown in the rule, these two subcases yield tubes $\tilde{h}_1$ and $\tilde{h}_2$, respectively.
- If the enabled condition imposed by *Sub* is false, its reset condition must also be false. Consequently, the overall enabled condition for **B** becomes *False*, while the overall reset condition follows the external *res*. This case yields tube $\tilde{h}_3$.

In order to define the final affine over-approximation, we introduce a function $\mathcal{M}(n, b_1, \tilde{h}_1, b_2, \tilde{h}_2)$, which conditionally merges the two tubes at time point n: If b_1 is false, the result is $\tilde{h}_2$; if b_1 is true and b_2 is false, the result is $\tilde{h}_1$; if both b_1 and b_2 are true, the resulting tube is obtained from $\tilde{h}_1$ by replacing its value at time n with the linear merge of the two affine forms $\tilde{h}_1(n)(v)$ and $\tilde{h}_2(n)(v)$ at degree d. Formally,

$$\begin{aligned}\mathcal{M}(n, b_1, \tilde{h}_1, b_2, \tilde{h}_2) = b_1?(b_2?(\lambda t\ v.(t = n)?\\ \tilde{h}_1(t)(v) \oplus_d \tilde{h}_2(t)(v) : \tilde{h}_1(t)(v)) : \tilde{h}_1) : \tilde{h}_2\end{aligned}$$

where the degree d is defined as follows:

$$\begin{aligned}d = 1 + \max(&\mathit{Deg}(\tilde{h}_1(n)), \mathit{Deg}(\tilde{h}_1(n-1)),\\ &\mathit{Deg}(\tilde{h}_2(n)), \mathit{Deg}(\tilde{h}_2(n-1)))\end{aligned}$$
$$\mathit{Deg}(\tilde{h}_1(n)) = \max\{\mathit{deg}(\tilde{h}_1(n)(v)) \,|\, v \in \mathit{dom}(\tilde{h}_1) \wedge v \textit{ is updated}\}$$

Here, $\mathit{Deg}(\tilde{h}_1(n))$ denotes the maximum degree among the variables of $\tilde{h}_1$ that have been updated at time n. The index d is used to refer a fresh noise symbol when merging $\tilde{h}_1(n)(v)$ and $\tilde{h}_2(n)(v)$. To ensure independence, d must be larger than the values of $\mathit{Deg}(\tilde{h}_1)$ and $\mathit{Deg}(\tilde{h}_2)$ at both time points n and n–1. This avoids the dependencies with the variables that have not yet been updated.

Using the merge operator $\mathcal{M}$, the semantics defined by rule (Enabled) first merges $\tilde{h}_1$ and $\tilde{h}_2$ based on the values of $\tilde{Res}_u(sig, sre, \tilde{h}, n)$ and $\tilde{Res}_d(sig, sre, \tilde{h}, n)$, and then merges the resulting tube with $\tilde{h}_3$ based on the values of $\tilde{En}_d(sig, sre, \tilde{h}, n)$ and $\tilde{En}_u(sig, sre, \tilde{h}, n)$. Intuitively, if $\tilde{En}_u$ is false, meaning the subsystem is not possibly enabled, the result is $\tilde{h}_3$. Otherwise, if $\tilde{En}_u$ is true and $\tilde{En}_d$ is false, meaning the subsystem is possibly enabled but not possibly not enabled, i.e., it is surely enabled, the result is $\tilde{h}_4$, which is then obtained based on the reset conditions $\tilde{Res}_u$ and $\tilde{Res}_d$ (understood similarly). If both $\tilde{En}_u$ and $\tilde{En}_d$ are true, meaning the subsystem is possibly enabled and also possibly not enabled, the result is the merge of $\tilde{h}_3$ and $\tilde{h}_4$. This definition guarantees that the resulting tube is an affine over-approximation of the exact semantics of the enabled subsystem.

The semantics of a triggered subsystem is defined by rule (Triggered); we omit the detailed explanation here, as it follows similarly to that of the enabled subsystem.

3) Execution over an Interval: The previous definitions establish the numerical semantics of discrete Simulink components at individual time points. The final three rules extend this to execution over the interval $(n, n+1)$ for a discrete block *dB*, a subsystem *Sub*, and a discrete diagram **B**, respectively. In each case, the values of all outputs and states are retained from time n, while all other variables remain unchanged.

Discussion. In the current numerical semantics, discrete component conditions and state transitions are evaluated only at fundamental sample times n. Events between sample points, such as zero-crossings triggered by continuous signals, are therefore not captured. We leave this for future work. A possible extension is to incorporate set-based event location (e.g., sign-change detection with Hermite-Birkhoff root-finding, as in HySon [12]) to handle such events within an interval.

B. Numerical Semantics for Continuous Diagrams

Given a continuous diagram **B**, we denote its calculational blocks and integrators by $\mathbf{B}_C$ and $\mathbf{B}_I$, respectively. Calculational blocks are stateless and introduce data dependencies between blocks; therefore, they must be sorted together with discrete blocks based on these dependencies before execution. To align with the semantics of discrete blocks, we define two separate rules: (Cal) for execution at time points and (Cal-ivl) for execution over open intervals. In both rules, the outputs of a calculational block are computed by applying its output function to the inputs, while other variables remain unchanged.

Integrator blocks are handled collectively as an initial value problem of an ODE system, defined over an initial time point followed by a sequence of left-open, right-closed intervals. Previous work [10] introduced a method to extract the ODE system from the whole diagram by combining all integrators, eliminating intermediate calculational block variables, and obtaining an ODE system expressed solely over integrator variables. We adopt this method and use *getODE*(**B**) to return the ODE system of diagram **B**. Rule (Integ-0) defines the

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

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

$$\frac{sort(\mathbf{B}) \equiv B \cdot \mathbf{B}' \quad \langle B, ena, res, \tilde{h}\rangle \xrightarrow{n}_{\mathcal{D}} \tilde{h}_1 \quad \langle \mathbf{B}', ena, res, \tilde{h}_1\rangle \overset{n}{\Longrightarrow}_{\mathcal{DO}} \tilde{h}_2}{\langle \mathbf{B}, ena, res, \tilde{h}\rangle \overset{n}{\Longrightarrow}_{\mathcal{DO}} \tilde{h}_2}\ \text{dDig-O} \qquad \frac{\mathbf{B} \equiv B \cdot \mathbf{B}' \quad \langle B, ena, res, \tilde{h}\rangle \overset{n}{\rightsquigarrow}_{\mathcal{D}} \tilde{h}_1 \quad \langle \mathbf{B}', ena, res, \tilde{h}_1\rangle \overset{n}{\Longrightarrow}_{\mathcal{DS}} \tilde{h}_2}{\langle \mathbf{B}, ena, res, \tilde{h}\rangle \overset{n}{\Longrightarrow}_{\mathcal{DS}} \tilde{h}_2}\ \text{dDig-S}$$

$$\frac{\langle \mathbf{B}, ena, res, \tilde{h}\rangle \overset{n}{\Longrightarrow}_{\mathcal{DO}} \tilde{h}_1 \quad \langle \mathbf{B}, ena, res, \tilde{h}_1\rangle \overset{n}{\Longrightarrow}_{\mathcal{DS}} \tilde{h}_2}{\langle \mathbf{B}, ena, res, \tilde{h}\rangle \overset{n}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_2}\ \text{dDig}$$

$$\frac{Sub \equiv (\mathbf{i}, \mathbf{o}, \mathbf{B}, st, \mathsf{Atomic}) \quad \langle \mathbf{B}, ena, res, \tilde{h}\rangle \overset{n}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_1}{\langle Sub, ena, res, \tilde{h}\rangle \xrightarrow{n}_{\mathcal{D}} \tilde{h}_1}\ \text{Atomic}$$

$$\frac{\begin{array}{c} Sub \equiv (\mathbf{i}, \mathbf{o}, \mathbf{B}, st, (\mathsf{Enab}\ sre\ sig)) \quad \langle \mathbf{B}, ena, \mathrm{True}, \tilde{h}\rangle \overset{n}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_1 \\ \langle \mathbf{B}, ena, res, \tilde{h}\rangle \overset{n}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_2 \quad \langle \mathbf{B}, \mathrm{False}, res, \tilde{h}\rangle \overset{n}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_3 \\ \tilde{h}_4 = \mathcal{M}(n, \tilde{Res}_u(sig, sre, \tilde{h}, n), \tilde{h}_1, \tilde{Res}_d(sig, sre, \tilde{h}, n), \tilde{h}_2) \\ \tilde{h}_5 = \mathcal{M}(n, \tilde{En}_u(sig, \tilde{h}, n), \tilde{h}_4, \tilde{En}_d(sig, \tilde{h}, n), \tilde{h}_3) \end{array}}{\langle Sub, ena, res, \tilde{h}\rangle \xrightarrow{n}_{\mathcal{D}} \tilde{h}_5}\ \text{Enabled}$$

$$\frac{\begin{array}{c} Sub \equiv (\mathbf{i}, \mathbf{o}, \mathbf{B}, st, (\mathsf{Trig}\ tty\ sig)) \quad \langle \mathbf{B}, ena, res, \tilde{h}\rangle \overset{n}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_1 \quad \langle \mathbf{B}, \mathrm{False}, res, \tilde{h}\rangle \overset{n}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_2 \\ \tilde{h}_3 = \mathcal{M}(n, \tilde{Trig}_u(sig, tty, \tilde{h}, n), \tilde{h}_1, \tilde{Trig}_d(sig, tty, \tilde{h}, n), \tilde{h}_2) \end{array}}{\langle Sub, ena, res, \tilde{h}\rangle \xrightarrow{n}_{\mathcal{D}} \tilde{h}_3}\ \text{Triggered}$$

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

$$\frac{Sub.\mathbf{B} = bs \quad \langle bs, ena, res, \tilde{h}\rangle \overset{(n,n+1)}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_1}{\langle Sub, ena, res, \tilde{h}\rangle \xrightarrow{(n,n+1)}_{\mathcal{D}} \tilde{h}_1}\ \text{dS-ivl}$$

$$\frac{sort(\mathbf{B}) \equiv B \cdot \mathbf{B}' \quad \langle B, ena, res, \tilde{h}\rangle \xrightarrow{(n,n+1)}_{\mathcal{D}} \tilde{h}_1 \quad \langle \mathbf{B}', ena, res, \tilde{h}_1\rangle \overset{(n,n+1)}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_2}{\langle \mathbf{B}, ena, res, \tilde{h}\rangle \overset{(n,n+1)}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_2}\ \text{dDig-ivl}$$

Fig. 2. The Numerical Semantics of Discrete Blocks, Subsystems and Diagrams

initialization of $\mathbf{B}_I$, while rule (Integ-ivl) defines its execution over the interval $(n, n+1]$ by solving $\dot{\mathbf{s}} = \mathbf{f}(\mathbf{s}, \mathbf{i})$. This invokes the function *nSol* to compute the numerical values as timed tubes, starting from the initial tube $\tilde{h}$ and using the precision settings *optns*.

C. Numerical Semantics for Hybrid Diagrams

The interaction between discrete and continuous components is captured by the overall semantics of the diagram, defined by rules (Diag-0) and (Diag-ivl). Given a hybrid diagram $\mathbf{B}$, the first rule initializes its integrators $\mathbf{B}_I$ and then executes its discrete and calculational blocks $\mathbf{B}_D, \mathbf{B}_C$ at once. The second rule specifies the execution order of all components over the interval $(n, n+1]$ based on data dependencies. The first two phases $\mathbf{B}_D$ over $(n, n+1)$ and $\mathbf{B}_I$ over $(n, n+1]$, can be interchanged without affecting the semantics, as they operate on disjoint variables (outputs and states, respectively) and both depend only on values at time n. In contrast, $\mathbf{B}_C$ must execute after both $\mathbf{B}_D$ and $\mathbf{B}_I$, because its inputs over $(n, n+1)$ may depend on their outputs. Finally, at the right endpoint $n+1$, $\mathbf{B}_D$ and $\mathbf{B}_C$ must execute together, as they both rely on inputs at $n+1$ and may have mutual dependencies.

Finally, this section defines a continuous-time numerical semantics for Simulink, defined over entire intervals rather than only at discrete points, so the value at any arbitrary time point can be obtained directly. At each time point, the semantics retains the entire tube (represented as an affine form) rather than selecting a single representative value, preserving all uncertainty information. For code generation or simulation, a concrete trajectory can be extracted, for instance, by taking the center point of the tube, as demonstrated in next section.

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

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

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

$$\frac{getODE(\mathbf{B}) = (\dot{\mathbf{s}} = \mathbf{f}(\mathbf{s}, \mathbf{i})) \quad \tilde{\mathbf{q}} = nSol(optns, \mathbf{f}, \mathbf{s}, n, \tilde{h})}{\langle \mathbf{B}_I, \mathbf{B}, ena, res, \tilde{h}\rangle \overset{(n,n+1]}{\Longrightarrow}_{\mathcal{I}} \tilde{h}[\mathbf{s} \mapsto (t, \tilde{\mathbf{q}}(t))) \mid \forall t \in (n, n+1]]}\ \text{Integ-ivl}$$

$$\frac{\langle \mathbf{B}_I, \mathbf{B}, ena, res, \tilde{h}\rangle \Longrightarrow^{0}_{\mathcal{I}} \tilde{h}_1 \quad \langle \mathbf{B}_D \cdot \mathbf{B}_C, ena, res, \tilde{h}_1\rangle \Longrightarrow^{0}_{\mathcal{D}} \tilde{h}_2}{\langle \mathbf{B}, ena, res, \tilde{h}\rangle \Longrightarrow^{0} \tilde{h}_2}\ \text{Diag-0}$$

$$\frac{\begin{array}{c}\langle \mathbf{B}_D, ena, res, \tilde{h}\rangle \overset{(n,n+1)}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_1 \quad \langle \mathbf{B}_I, \mathbf{B}, ena, res, \tilde{h}_1\rangle \xrightarrow{(n,n+1]}_{\mathcal{I}} \tilde{h}_2 \\ \langle \mathbf{B}_C, ena, res, \tilde{h}_2\rangle \overset{(n,n+1)}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_3 \quad \langle \mathbf{B}_D \cdot \mathbf{B}_C, ena, res, \tilde{h}_3\rangle \overset{n+1}{\Longrightarrow}_{\mathcal{D}} \tilde{h}_4\end{array}}{\langle \mathbf{B}, ena, res, \tilde{h}\rangle \overset{(n,n+1]}{\Longrightarrow} \tilde{h}_4}\ \text{Diag-ivl}$$

Fig. 3. The Numerical Semantics of Continuous and Hybrid Diagrams

From the tube, we can also extract the upper and lower bounds for each variable at any time point using $\tilde{sup}(\tilde{x})$ and $\tilde{inf}(\tilde{x})$, thereby determining the *error upper bound* between the numerical result (i.e., the center point) and the exact solution.

VI. Implementation and Code Generation in Isabelle

All the numerical semantics of Simulink presented above have been formalized in Isabelle, from which executable code is extracted [1]. This section describes these two aspects.

A. Formalization of the Numerical Semantics

This section presents and discusses several key aspects of the implementation. In the Isabelle formalization, a Simulink diagram is represented as a list of blocks. Each block is either atomic or a composite subsystem, which itself contains a list of blocks. This recursive definition captures arbitrary nesting of diagrams, reflecting their hierarchical structure and forming the basis for the semantic definitions.

a) Subsystems: According to the definition in Section V-A, the semantics of a subsystem may involve multiple possible execution branches over a tube, leading to multiple candidate tubes that must be merged into a single over-approximation. We define the operator $\oplus_d$ to merge two affine forms, which is implemented as union_approx in Isabelle:

```
fun union_approx :: nat ⇒ nat ⇒ aform_err ⇒ aform_err ⇒ aform_err where
  union_approx p n a b = the (approx_floatarith p
      (Add (Mult (Num 0.5) (Add (Var 1) (Var 2)))
           (Mult (Var 0) (Add (Var 1) (Minus (Var 2)))))
      [((0,pdev_upd zero_pdevs n 0.5),0),a,b])
```

Given precision p and index n for a new noise symbol, it merges the error-bounded affine forms a and b. Here, *approx_floatarith* p F *afes* returns the result of evaluating the function expression F on the list of error-bounded affine forms *afes* with precision p. Based on this, we implement the merge semantic function $\mathcal{M}(n, b_1, \tilde{h}_1, b_2, \tilde{h}_2)$ as conflu:

```
fun conflu :: nat ⇒ nat ⇒ veced_t option ⇒ veced_t option ⇒ veced_t option where
  conflu p n None H = H |
  conflu p n H None = H |
  conflu p n (Some Hed1) (Some Hed2) =
    Some (λt. if t=n then
      let d = max (max (snd (Hed1 n)) (snd (Hed2 n)))
          (max (snd (Hed1 (n-1))) (snd (Hed2 (n-1)))) in
      (λv. union_approx p d (fst (Hed1 t) v) (fst (Hed2 t) v),d+1)
    else Hed1 t)
```

Here, Hed_1 and Hed_2 are tubes annotated with degrees. Specifically, $Hed_1\ t$ yields not only a set of error-bounded affine forms but also their maximum degree, accessed via *snd* $(Hed_1\ t)$. By appending degree information to the tubes, we avoid redundant computations, thereby improving efficiency. When one of the tubes is missing (i.e., one input is *None*), the function simply returns the other tube.

The numerical semantics of (sub)diagrams and subsystems are defined via mutual recursion. For instance, the output computation rule for subdiagrams at sampling instants (Rule dDig-O) and the rules for subsystems (Atomic, Enabled, and Triggered) are mutually recursive. To simplify proofs, the Isabelle implementation avoids explicit mutual recursion by folding the semantic definition into a single recursive function. Rule dDig-O is implemented by the function exeDL_oe_att,

[1] All the formalization in Isabelle/HOL can be found at https://gitee.com/infiniteob/sim-sem-numeric/tree/master/

which incorporates the subsystem semantics and is defined in a self-recursive manner:

```
fun exeDL_oe_att :: nat ⇒ block list ⇒ var list ⇒ floatarith list⇒
bool ⇒ bool ⇒ nat ⇒ veced_t ⇒ veced_t where
  exeDL_oe_att p [] rol fas exe res n Hed = Hed |
  exeDL_oe_att p (b#bl) rol fas exe res n Hed =
   (λHed. case b of
        (cB il ol _ _ dir) ⇒ (if dir then exeCale_att ··· else Hed)
        |(dB il ol sl vl st _ ofl sfl) ⇒
          (if ((get_st b)|n ∨ (get_st b)=0) ∧ exe
            then exeD_oe_att ··· else exeD_oe_not_att ···)
        |(Sub il ol bs st typ) ⇒ (case typ of
          Atomic ⇒ (exeDL_oe_att p bs rol fas exe res n Hed)
          |(Enab re var) ⇒the (conflu p n
          (if (Sup_afe p (fst (Hed n) var)) > 0 then (conflu p n
           (if (Inf_afe p (fst (Hed (n-1)) var))≤0∧re=0 then Some
            (exeDL_oe_att p bs rol fas exe True n Hed) else None)
           (if (Sup_afe p (fst (Hed (n-1)) var))>0∨re≠ 0 then Some
            (exeDL_oe_att p bs rol fas exe res n Hed) else None))
          else None)
          (if (Inf_afe p (fst (Hed n) var)) ≤ 0 then Some
            (exeDL_oe_att p bs rol fas False res n Hed) else None))
          |(Trig trgtyp var) ··· ···
```

As defined above, this function processes each block in the list inductively: for continuous or discrete blocks, it directly invokes the block's semantics; for subsystems, it determines possible execution paths based on their type and signal bounds, then recursively calls *exeDL_oe_att* on their block lists and merges all possible execution results using conflu.

b) Integrators: The key part of the numerical semantics for continuous (sub-)diagrams *Diag* is solving the integrators. As previously mentioned, this first requires deriving the system of ODEs corresponding to all integrators, and then solving the resulting initial value problem. By using *getODE* [10], we can obtain the ODE system describing the continuous behavior of *Diag*. To solve such ODEs, we use the function *one_step_until_time_afm* defined in Immler's ODE numerical solving theory, which is the concrete implementation of *step_until_afe* mentioned in Section IV, to define the function solus. The inputs of solus include: a record parameter *optns* containing precision p, relative tolerance *rtol*, and absolute tolerance *atol*; the continuous variables *vl* contained in the ODE, serving as indices to query specified variables in the computed list of affine forms; the ODE expressions *ode_fas*, for which we will explain in the next section how to obtain; the initial value set $X0$; and the time t to be solved. This function returns a set of error-bounded affine forms representing an over-approximation of the ODE solution at time t, along with the degree of this set of affine forms:

```
definition solus optns vl ode_fas X0 t =
(let ode_ops=(init_ode_ops False False (the_odo ode_fas true_form))
  in (case (one_step_until_time_afm optns ode_ops X0 t)
   of dRETURN (((l,r),(res,der)),_) ⇒
   (λ c. (res ! (index vl c),0),degree_aforms res)))
```

B. Code Generation in Isabelle

Although the numerical semantics defined in Isabelle can be evaluated via the **value** command, its execution efficiency is limited and unsuitable for large-scale execution. To obtain a practically usable implementation, we leverage Isabelle's code extraction mechanism to automatically generate efficient executable code, in our case SML, directly from the formalized numerical semantics. This yields a trustworthy executor that produces numerical results with formal error bounds, enabling efficient execution, rapid simulation and formal verification. Especially, as a validation, we can compare the generated code's execution results with those of Simulink, confirming the consistency of the numerical behavior.

In the previous section, we noted that some semantic definitions for discrete blocks resemble those of the exact semantics [10], with the key difference that values are replaced by the tubes. However, there are many important changes. To ensure the Isabelle definitions are executable, we made substantial adjustments to the semantics when adapting it to the numerical setting. We represent the values of all variables at any time, previously represented as a map, as a list of affine forms. Meanwhile, we convert the block functions into versions that operate on affine forms, in the form of corresponding arithmetic expressions: For integrators, we provide the arithmetic expressions of the ODEs, denoted as *ode_fas*, for performing numerical integration on affine forms; and for the state and output functions of all discrete and calculational blocks, we provide arithmetic expressions *s_fas* and *o_fas* to represent the corresponding functions, respectively. At last, we change some definitions not executable, e.g. the ones using existential quantifiers, into equivalent executable versions. Once these preparations are complete, we use Isabelle's code generator to produce executable SML programs from the numerical semantics.

As shown in Figure 4, *ex_pid_fas* represents the arithmetic expressions of the ODEs in a given Simulink model *ex_pid*, which can be automatically constructed via **schematic_goal**. *ex_pid_sfas* and *ex_pid_ofas* are the expression lists for the state and output functions, respectively, obtained using the same method. H is the initial tube where the center, noise term coefficients, and error are all zeros. H' is the final tube obtained after executing the numerical semantics of *ex_pid* over the interval $[0, 4]$. Given the numerical semantics of *ex_pid*, the **export_code** command can directly generate SML code for this numerical computation process.

VII. Case Studies

In this section, we consider three case studies that vary in Simulink features, with increasing scale and complexity.

A. The PID Control Example

The PID control example in Figure 1 (hereafter referred to as *Diag*) is taken from [10]. The requirement is, under the control of the PID controller, the process variable b of the plant converges to the set point $d = 1$. Previous work [10] used the exact semantics to rigorously verify the example and proves the following theorem.

Theorem 2 (Convergence to set point [10])**.** *Let $\varepsilon > 0$ be an arbitrary precision, then there exists $N \in \mathbb{N}$ such that for all*

```
declare interpret_floatariths.simps [code del]
schematic_goal ex_pid_ofas:
  "get_exps_out ex_pid X = interpret_floatariths ?fas X"
  apply normalization
  by (reify_floatariths)
concrete_definition ex_pid_ofas uses ex_pid_ofas
schematic_goal ex_pid_sfas:
  "get_exps_st ex_pid X = interpret_floatariths ?fas X"
  apply normalization
  by (reify_floatariths)
concrete_definition ex_pid_sfas uses ex_pid_sfas
schematic_goal ex_pid_fas:
  "get_ODE1 ex_pid X = interpret_floatariths ?fas X"
  apply normalization
  by (reify_floatariths)
concrete_definition ex_pid_fas uses ex_pid_fas
schematic_goal ex_pid_fas_compact:
  "get_ODE1_compact ex_pid X = interpret_floatariths ?fas X"
  apply normalization
  by (reify_floatariths)
concrete_definition ex_pid_fas_compact uses ex_pid_fas_compact
abbreviation H where "H ≡ (λt v.(pdevs_of_real 0,0),0)"
definition "H' = exeDisConDiag_Hed_tilltime optns0 ex_pid
                 ex_pid_fas ex_pid_sfas ex_pid_ofas 4 H"
export_code H' in SML module_name ex_pid
```

Fig. 4. Code generation for a Given Simulink Diagram *ex_pid*

```
fun exeDisConDiag_Hed_tilltime p bl ode_fas s_fas o_fas n hed =
  (if equal_nata n zero_nata
    then exeDisDiag_Hed_attime p bl s_fas o_fas zero_nata
           (exeINTBlks_Hed_init bl hed)
    else exeDisDiag_Hed_attime p bl s_fas o_fas (suc (minus_nata n one_nata))
           (exeCalBlks_Hed_intertime p bl o_fas
             (of_nat semiring_1_real (minus_nata n one_nata)) one_reala
             (exeINTBlks_Hed_inter bl ode_fas
               (of_nat semiring_1_real (minus_nata n one_nata)) one_reala
               (exeDisDiag_Hed_intertime bl (minus_nata n one_nata)
                 (exeDisConDiag_Hed_tilltime p bl ode_fas s_fas o_fas
                   (minus_nata n one_nata) hed)))));
```

Fig. 5. Partial SML code generated for the numerical semantics function

$n \in \mathbb{N}, t \in \mathbb{R}$, *if* $n \geq N$ *and* $\langle Diag, True, False, h_0 \rangle \overset{[0,n]}{\Longrightarrow} h$, *then for any* $t \in [N, n]$, $|h(b)(t) - 1| \leq \varepsilon$ *holds.*

This result rigorously establishes the property of the example over the whole infinite interval $[0, +\infty)$. However, the proof process involves numerous complex intermediate definitions and requires nearly one thousand lines of manually written Isabelle proof code. In practice, proving properties over the entire time domain is not always necessary; verifying system behavior over a finite time domain often suffices. For this example, we will prove instead that within a finite timed interval, the value of b remains very close to 1. Specifically, the maximum deviation of b from 1 is bounded by 0.113 on $[10, 12]$ and by 0.063 on $[12, 15]$. This is achieved by generating and executing code using the numerical semantics presented here.

Figure 5 displays a portion of the generated SML code. We used the exported code to compute the ranges of relevant variables for *Diag* on the interval $[0, 15]$, as shown in Figure 6. Due to the high precision of the results, the representation of the ranges is not visually distinct in the plot. The figure presents not only the numerical semantics results for five variables of this example but also partial exact semantics results, including the values of these variables on $[0, 2]$ and the values of b and d on $[0, 15]$. These relatively precise values

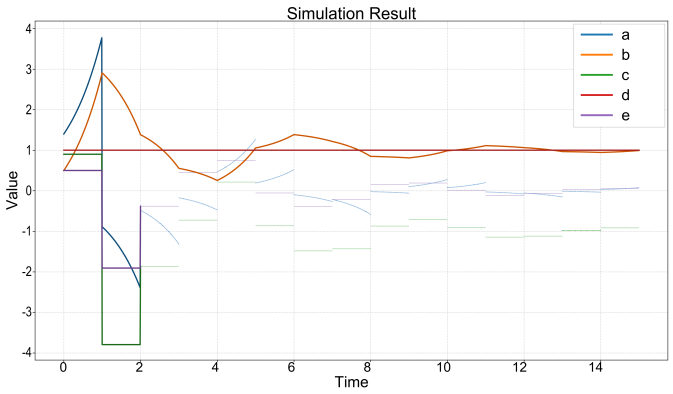

Fig. 6. Numerical results(thin) and exact results(bold) of *Diag* over $[0, 15]$

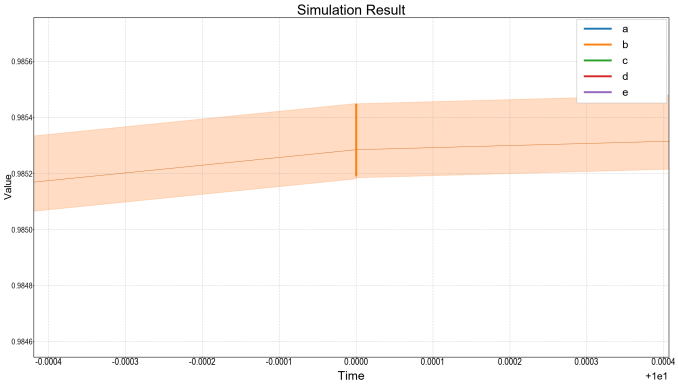

Fig. 7. Detailed view of variable b at $t = 10$

were obtained by substituting time into rigorously derived expressions from the previous work, and as seen in the figure, they align highly consistently with the numerical semantics results. The exact traces for a, c, and e are shown only up to $t = 2$; their closed-form derivations are labor-intensive and these intermediate variables are not of direct interest.

Figure 7 zooms in on b at $t = 10$. The light orange area is the over-approximation enclosure between sampling points, the dark orange vertical line the tighter bound at $t = 10$, and the dark orange thin line the exact value, which lies entirely within both. The center, upper, and lower bounds of $H'(10)(b)$ are $(2066365)_{21}$, $(2066637)_{21}$, and $(2066093)_{21}$; rounded to 5 decimal places, $b(10) \in [0.98519, 0.98545]$, with the error from the estimate 0.98532 not exceeding 0.00013. By examining all sampling points and enclosures, the maximum deviation of b from 1 is bounded by 0.113 on $[10, 12]$ and 0.063 on $[12, 15]$. The generated code computes the numerical semantics over $[0, 15]$ with a wall-clock time of 12.7 s(0.571 s for execution, with the remainder consumed by compilation and startup).

B. Water Tank Control System

The second case study is a Water Tank Control System (Figure 8, *Diag_WT*) adapted from the Simulink library. The

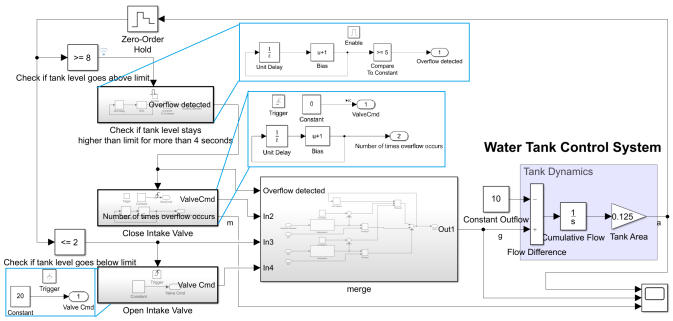

Fig. 8. Simulink diagram of the Water Tank Control System (*Diag_WT*)

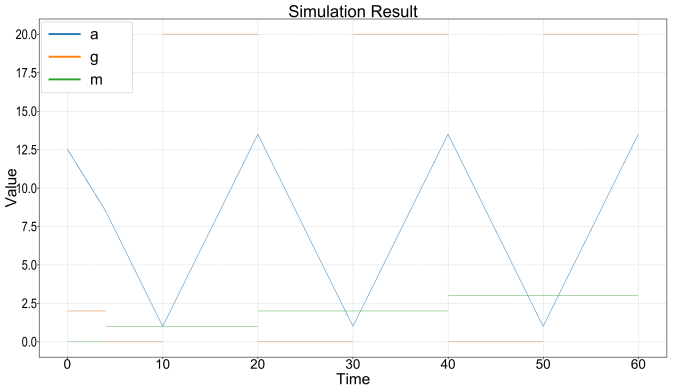

Fig. 9. Numerical results for *Diag_WT* over $[0, 60]$: m (green, exact), a (blue, with error enclosure), and g (orange, exact)

blue area Tank Dynamics models the water level a via an integrator. When $a < 2$, the triggered subsystem Open Intake Valve sets the intake flow g to 2; when $a > 8$ for 4 consecutive seconds, Close Intake Valve sets g to 0 and increments the overflow counter m by 1. A Merge block (modeled as an atomic subsystem) selects the most recently triggered branch as g, with initial value 2.

We are concerned with two properties over $[0, 60]$: the overflow counter m should not exceed 5, and the water level a should stay within the range $[0, 13]$. *Diag_WT* is substantially more complex than the PID example: it contains 31 blocks, including one integrator, two triggered subsystems, one enabled subsystem with state reset, and one atomic subsystem with nested subsystems. The generated SML code computes the numerical semantics over $[0, 60]$ with a wall-clock time of 12.3 s and a running time of 0.791 s, demonstrating that our extracted code remains efficient for models with tens of blocks, hierarchical and conditional subsystems.

Figure 9 shows the results. The overflow counter m is computed exactly; its maximum on $[0, 60]$ is 4 at $t = 60$, so $m(t) < 5$. For the water level, $a(0) = 12.5$. On $[0, 4)$, $\inf(\tilde{h}(t)(a)) \geq 8.49996$, so neither triggered subsystem fires. At $t = 4$, the timer activates Close Intake Valve: g drops to 0, m increments to 1. Key numerical bounds:

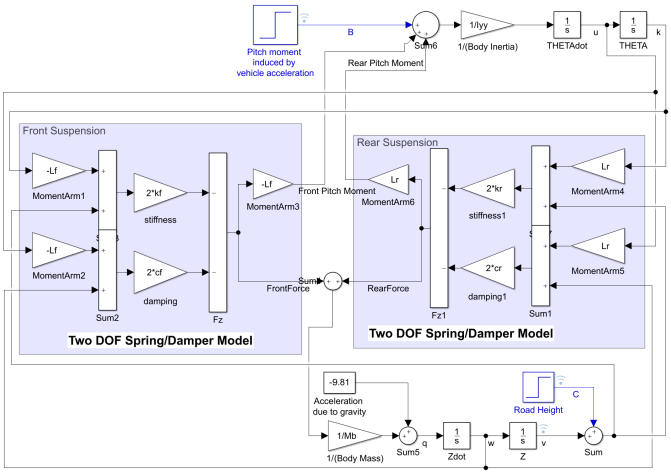

Fig. 10. Simulink diagram of the Vehicle Suspension Model

$a(20) \in [13.49989, 13.50004]$, $a(50) \in [0.9997, 1.0001]$, and $a(t) \in [0.9997, 13.50004]$ globally. The upper bound exceeds 13 by 0.50004, attributable to the 4 s detection delay.

C. Vehicle Suspension Model

We consider a two-degree-of-freedom (bounce and pitch) Vehicle Suspension Model adapted from the Simulink library, shown in Figure 10. The blue areas Front Suspension and Rear Suspension model the front and rear suspensions as coupled continuous subsystems; their outputs are combined with step-disturbance inputs for pitch moment and road height to form a closed-loop system. The diagram contains 29 blocks including 4 integrators.

We verify two properties under step disturbances. **(1)** A pitch moment step at $t = 1$ (from 0 to 100) drives the pitch angle k toward a new equilibrium; we require its peak on $[1, 3]$ to stay below 0.01. As shown in Figure 11, the peak of k occurs near $t \approx 1.4716$ and does not exceed 0.00129, satisfying the requirement with comfortable margin. **(2)** A road-height step at $t = 5$ (from 0 to 0.01) induces a vertical acceleration q; we require $|q| < 1$ on $[0, 7]$. Figure 12 shows that $|q|$ peaks near the step, where $q \in [-0.82777, -0.80715]$, thus $|q| < 0.83 < 1$.

The generated SML code computes the numerical semantics over $[0, 7]$ with a wall-clock time of 81 s and a running time of 68.59 s, dominated by ODE integration. Table II breaks down the solver performance by interval ($[i - 1, i]$): for each interval, *Steps* counts the ODE integration steps, *AvgPic* the average Picard iterations per step, and *Total* the computation time. Interval 6 is notably expensive (181 steps, 32.1 s) due to the road-height step disturbance.

Across the three studies, the cost shifts from simple PID control system (PID, wall-clock 12.7 s) through hierarchical and conditional subsystems (Water Tank, 12.3 s) to a more complex ODE system (Suspension, 81 s). The suspension model's per-step cost grows with noise-symbol count in the affine forms, which increases linearly with step count, leading

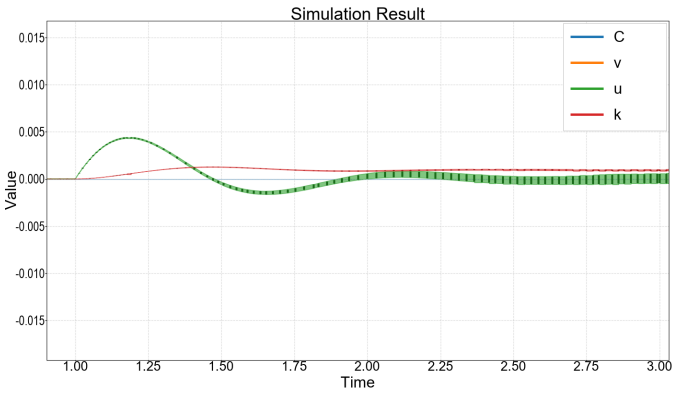

Fig. 11. Pitch angle k (red) and angular velocity u (green) on $[1, 3]$

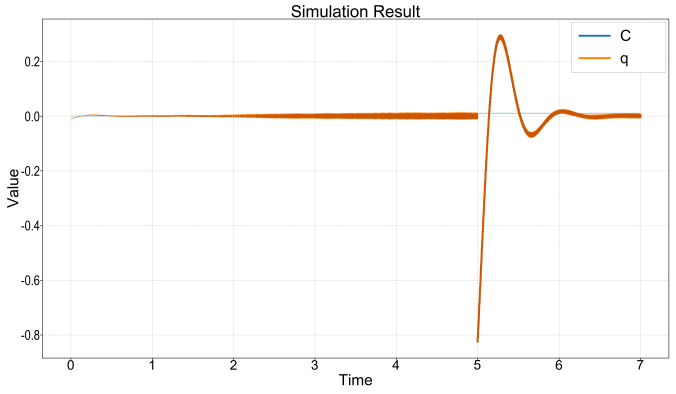

Fig. 12. Vertical acceleration q (orange) and road height C (blue) on $[0, 7]$

to roughly quadratic growth with simulation time. Picard iterations (tens per step) multiply this further. Key optimization directions include error-bounded affine forms to curb noise growth and better initial guesses to reduce iteration counts.

VIII. Conclusion

This paper presents a verified numerical semantics for Simulink formalized in Isabelle/HOL, bridging the gap between rigorous exact semantics and executable simulation. By integrating a certified ODE solver with affine arithmetic and adaptive step-size control, we provide formal local error bounds and establish global error propagation through mixed discrete–continuous systems. Leveraging Isabelle's code extraction, we generate executable code that directly reflects the verified semantics, yielding a trustworthy toolchain from models to runnable code. The approach is evaluated on several case studies covering different Simulink features and levels of complexity, demonstrating its feasibility and the resulting code's compliance with formal error bounds.

Future work includes improving the ODE solver performance to enable scaling to larger industrial models, such as automotive or aerospace control systems. We also plan to extend the semantics to support additional Simulink features and to support C code generation for embedded deployment.

TABLE II
ODE Solver Performance by Interval

Int	Steps	AvgPic	Total (s)
1	39	44.9	0.893
2	52	37.6	2.552
3	41	31.7	2.112
4	50	38.7	4.183
5	55	35.7	5.929
6	181	24.5	32.105
7	52	37.2	20.513

References

[1] MathWorks, *Simulink® User's Guide*, 2018a, http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf.

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

[3] L. Zou, N. Zhan, S. Wang, M. Fränzle, and S. Qin, "Verifying Simulink diagrams via a hybrid Hoare logic prover," in *EMSOFT 2013*. IEEE, 2013, pp. 9:1–9:10.

[4] T. Liebrenz, P. Herber, and S. Glesner, "Deductive verification of hybrid control systems modeled in Simulink with Keymaera X," in *ICFEM 2018*, ser. LNCS, vol. 11232. Springer, 2018, pp. 89–105.

[5] T. Bourke, F. Carcenac, J. Colaço, B. Pagano, C. Pasteur, and M. Pouzet, "A synchronous look at the Simulink standard library," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 176:1–176:24, 2017.

[6] Q. Sun, W. Zhang, C. Wang, and Z. Liu, "A contract-based semantics and refinement for hybrid simulink block diagrams," *J. Syst. Archit.*, vol. 143, p. 102963, 2023.

[7] X. Xu, B. Zhan, S. Wang, J. Talpin, and N. Zhan, "A denotational semantics of Simulink with higher-order UTP," *J. Log. Algebraic Methods Program.*, vol. 130, p. 100809, 2023.

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

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

[10] Y. Qi, S. Wang, X. Li, B. Zhan, and N. Zhan, "Formal semantics for hierarchical simulink diagrams in Isabelle/HOL," *Journal of Systems Architecture*, vol. 174, p. 103724, 2026.

[11] O. Bouissou and A. Chapoutot, "An operational semantics for Simulink's simulation engine," *SIGPLAN Not.*, vol. 47, no. 5, p. 129–138, 2012.

[12] O. Bouissou, S. Mimram, B. Strazzulla, and A. Chapoutot, "Set-based simulation for design and verification of Simulink models," in *ERTS*, 2014, toulouse, France, February 2014.

[13] G. Yan, L. Jiao, S. Wang, L. Wang, and N. Zhan, "Automatically generating systemC code from HCSP formal models," *TOSEM*, vol. 29, no. 1, pp. 4:1–4:39, 2020.

[14] S. Wang, Z. Ji, X. Xu, B. Zhan, Q. Gao, and N. Zhan, "Formally verified c code generation from hybrid communicating sequential processes," in *ICCPS 2024*, 2024, pp. 123–134.

[15] F. Immler, "A verified ODE solver and Smale's 14th problem," Ph.D. dissertation, Technische Universität München, 2018.

[16] A. Agrawal, G. Simon, and 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.

[17] C. Chen, J. Dong, and J. Sun, "A formal framework for modeling and validating Simulink diagrams," *Formal Aspects Comput.*, vol. 21, no. 5, pp. 451–483, 2009.

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

[19] F. Faissole, "Formally-verified round-off error analysis of runge–kutta methods," *Journal of Automated Reasoning*, vol. 68, no. 1, p. 1, 2023.

[20] S. Park and H. Thies, "A coq formalization of taylor models and power series for solving ordinary differential equations," in *ITP 2024*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2024.

[21] T. J. Hickman, C. Laursen, and S. Foster, "Certifying differential equation solutions from computer algebra systems in Isabelle/HOL," in *NASA Formal Methods Symposium*. Springer, 2021, pp. 190–207.

[22] F. Immler, "Verified reachability analysis of continuous systems," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 37–51.
[23] D. Bryant and J. J. Huerta y Munive, "Verifying numerical methods with Isabelle/HOL," in *CPP 2024*, 2024, pp. 14–29.

Yuzhen Qi received the bachelor's degree from Shandong University and the master's degree in software engineering from the Institute of Software, Chinese Academy of Sciences (ISCAS). He is currently with the Institute for Advanced Algorithms Research, Shanghai, China. His research interests include formal methods, interactive theorem proving, and formal modeling and verification.

Shuling Wang received the Ph.D. degree from Peking University. She subsequently conducted postdoctoral research at the United Nations University International Institute for Software Technology and ISCAS. She is currently an associate researcher at ISCAS. Her research interests include the formal modeling and verification of cyber-physical and embedded systems, program verification, and interactive theorem proving.

Xiangyu Jin received the Ph.D. degree in software engineering from the Institute of Software, Chinese Academy of Sciences (ISCAS). He is currently a postdoctoral researcher at the National Key Laboratory of Space Integrated Information Systems, ISCAS. His research interests include formal methods, hybrid systems, and formal modeling and verification.

Xiong Xu received the Ph.D. degree from the Institute of Software, Chinese Academy of Sciences (ISCAS), where he is currently a research assistant. His research interests center on design theories for safety-critical cyber-physical systems, including model-based design, hybrid systems, process algebra, and session types.

Naijun Zhan received the bachelor's and master's degrees from Nanjing University and the Ph.D. degree from the Institute of Software, Chinese Academy of Sciences (ISCAS). He is currently a Boya Distinguished Professor with the School of Computer Science, Peking University. Before joining Peking University, he was a research fellow at the University of Mannheim, Germany, and subsequently held the positions of associate professor, professor, and distinguished professor at ISCAS. His research interests include the formal design of real-time, embedded, and hybrid systems, as well as program verification.