

Mars 2.0: A Toolchain for Modeling, Analysis, Verification and Code Generation of Cyber-Physical Systems

BOHUA ZHAN, XIONG XU, QIANG GAO, ZEKUN JI, XIANGYU JIN, SHULING WANG*, and NAIJUN ZHAN, State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences, China

We introduce Mars 2.0 for modeling, analysis, verification and code generation of Cyber-Physical Systems. Mars 2.0 integrates Mars 1.0 with several important extensions and improvements, allowing the design of cyber-physical systems using the combination of AADL and Simulink/Stateflow (AADL \oplus S/S), which provide a unified graphical framework for modeling the functionality, physicality and architecture of the system to be developed. For a safety-critical system, formal analysis and verification of its combined AADL \oplus S/S model can be conducted via the following steps. First, the toolchain automatically translates AADL \oplus S/S models into Hybrid CSP (HCSP), an extension of CSP for formally modeling hybrid systems. Second, the HCSP processes can be simulated using the HCSP simulator, and to complement incomplete simulation, they can be verified using the Hybrid Hoare Logic prover in Isabelle/HOL, as well as the more automated HHLPy prover. Finally, implementations in SystemC or C can be automatically generated from the verified HCSP processes. The transformation from AADL \oplus S/S to HCSP, and the one from HCSP to SystemC or C, are both guaranteed to be correct with formal proofs. This approach allows model-driven design of safety-critical cyber-physical systems based on graphical and formal models and proven-correct translation procedures. We demonstrate the use of the toolchain on several benchmarks of varying complexity, including several industrial-sized examples.

CCS Concepts: • **Software and its engineering** \rightarrow *Formal software verification*; Semantics; • **Computer systems organization** \rightarrow **Embedded software**.

Additional Key Words and Phrases: Cyber-physical systems, graphical modeling, formal modeling, analysis, verification, code generation

ACM Reference Format:

Bohua Zhan, Xiong Xu, Qiang Gao, Zekun Ji, Xiangyu Jin, Shuling Wang, and Naijun Zhan. 2018. Mars 2.0: A Toolchain for Modeling, Analysis, Verification and Code Generation of Cyber-Physical Systems. 1, 1 (March 2018), 23 pages.

1 INTRODUCTION

Cyber-physical systems (CPSs) seamlessly integrate computational and physical capabilities, that expand the capabilities of physical world through communication, computation and control. CPSs are omnipresent in our daily life from aerospace, transportation, automotive, to health care, *etc.* All these systems are expected to achieve desired behaviors, especially for safety-critical ones, for which any failure may cause severe catastrophes. Model-based design (MBD), which aims to design efficient and reliable systems, has become a popular development approach in CPS industry. It controls complexity of systems by appropriate abstraction/refinement and composition/decomposition, and meanwhile,

*Corresponding author

Authors' address: Bohua Zhan, bzhan@ios.ac.cn; Xiong Xu, xux@ios.ac.cn; Qiang Gao, gaoqiang@ios.ac.cn; Zekun Ji, jizk@ios.ac.cn; Xiangyu Jin, jinxy@ios.ac.cn; Shuling Wang, wangsl@ios.ac.cn; Naijun Zhan, znj@ios.ac.cn, State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences, Zhongguancun South 4th Street, No 4, Haidian District, Beijing, China, 100190.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

it provides techniques based on (formal) modeling, analysis and verification in order to detect and correct errors in the early stage of design. In the MBD approach, a system is usually modeled at different levels of abstraction for different purposes, e.g. graphical models for engineering design and requirement analysis, formal models for formal verification, and executable code for implementation. However, these models at different levels must be consistent with each other to obtain a final correct implementation of the system.

The MBD of CPSs faces the following challenge: a CPS tightly combines heterogeneous components related to physicality (including hardware platform and physical processes), software and architecture, and their combinations. To design a CPS at system level, it is desirable to take modeling, analysis, verification and implementation of these different aspects into account uniformly. Unfortunately, most of the existing MBD approaches do not support this. For example, AADL (Architecture Analysis & Design Language) [29] supports the description of embedded system hardware and software architectures, relevant discrete behaviors, and their compositions based on connections. But it could not handle the continuous behavior of physical plants to be monitored and controlled by computers. On the other hand, Matlab's Simulink/Stateflow [21, 22] is an industrial model-based design environment that supports both discrete and continuous data flows, and flexible event-driven control based on hierarchical state machines. But it is not well-suited to describe system architectures and hardware platforms. To address this challenge, recent work [35] investigates the combination of AADL and Simulink/Stateflow called AADL \oplus S/S. AADL \oplus S/S provides a unified graphical framework for designing CPSs, which inherits the advantages of AADL and S/S, and avoids their disadvantages.

In addition to combination of AADL and Simulink/Stateflow, other aspects of the MBD approach, such as formal semantics of Simulink/Stateflow graphical models, formal verification [20, 34], and code generation from formal models [37] have also been investigated. Part of these works are implemented and combined into a toolchain called MARS [8], for **M**odelling, **A**nalysis and **V**eRification of Hybrid Systems. As shown in Fig. 1 left, MARS supports graphical modeling by Simulink/Stateflow and automatic transformation from graphical models to HCSP formal models for verification. In particular, HCSP, which is an abbreviation for hybrid communicating sequential processes [17, 43], is an extension of CSP with ordinary differential equations (ODEs) for modeling hybrid systems.

Since the publication of [8], several extensions and improvements are added into the toolchain. Most significantly, it incorporates the combination of AADL and Simulink/Stateflow (AADL \oplus S/S) in [35], and the co-simulation of combined AADL \oplus S/S models by translation to C code and defining their integration [39]. Other additions include a simulation tool for HCSP described in [39], and translation procedures from Simulink and Stateflow models into HCSP programs whose results are easier to understand and verify, as well as supporting more features (the procedure for Stateflow is described in [15], and the one for Simulink will be described in this paper). Finally, formal verification tools are significantly improved [31], and a new code generation process to C is added.

Presentations of the above works are scattered in various papers (some of which still unpublished). Moreover, there is not yet a comprehensive overview of how the entire toolchain can be applied to facilitate the MBD approach. Hence, we present in this paper an updated version of the toolchain, called Mars 2.0. We give an overview of the new components since [8], as well as how the toolchain as a whole is used in practice. The architecture of the toolchain is shown on the right of Fig. 1, where the green boxes (AADL and Simulink/Stateflow) indicate graphical models that serve as input to the toolchain; the yellow boxes indicate formal models (HCSP), and their simulation and verification tools; and the red boxes indicate generated code. The architecture of the previous version of this tool is shown on the left of Fig. 1, which is mainly composed of three parts: translators Sim2HCSP and HCSP2Sim, and an HHL prover. HCSP2Sim is used to justify the correctness of Sim2HCSP, and HHL prover verifies HCSP formal models. In Mars 2.0, the correctness of the translation procedures is proved formally, thus HCSP2Sim is no longer necessary.

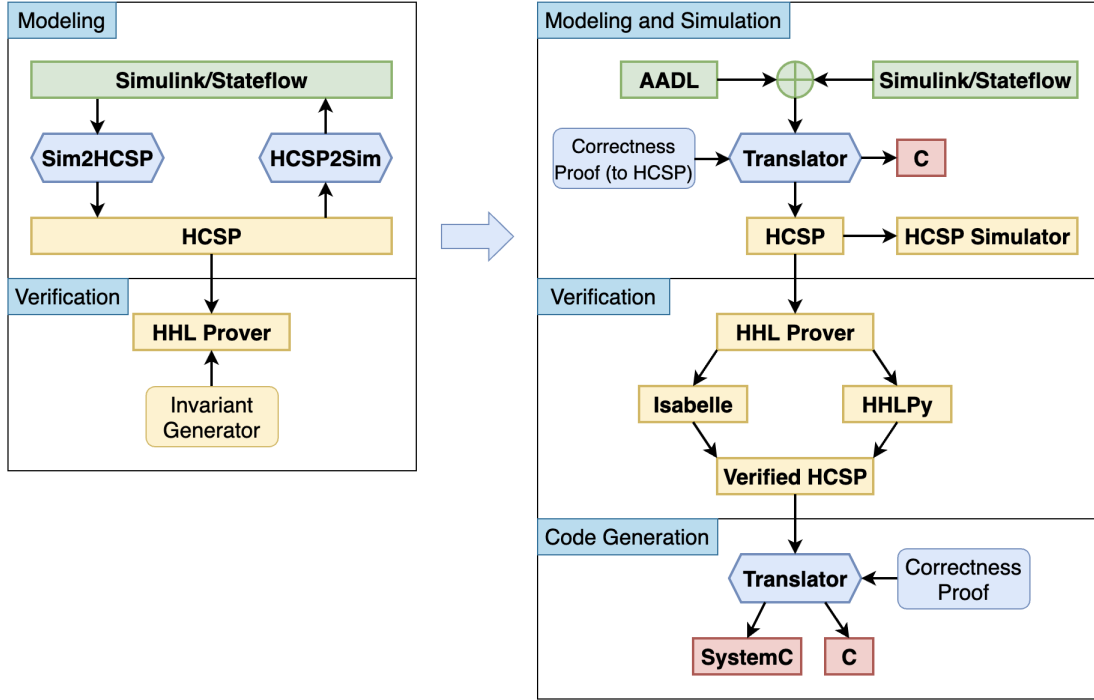


Fig. 1. Mars 1.0 (left, [8]) and 2.0 (right)

In summary, we present in this paper a unified account of the Mars 2.0 toolchain, as well as show some examples of its application on industrial-sized case studies. Moreover, descriptions of the following aspects of the toolchain have not appeared in other papers.

- We present a new algorithm for transforming Simulink models to sequential HCSP (the original algorithm in [8, 35] results in parallel processes). This gives HCSP programs that are easier to verify.
- We present some implementation details of code generation to C, in particular its support for extensions to HCSP. This includes preprocessing to handle channel parameters, and type inference for typing a well-formed HCSP program before it is transformed to C.
- We describe how the toolchain can be used to support the MBD approach, including a realistically-scaled example on the whole design of the Automatic Cruise Control system, the simulation and verification of several complex case studies.

After reviewing related works in this section, the paper is organized as follows: Sect. 2 presents modeling frameworks based on AADL \oplus S/S and HCSP respectively, and their simulation methods. Sect. 3 presents the translation from AADL \oplus S/S to HCSP. Sect. 4 presents the HHL prover for verifying HCSP models, including implementations in Isabelle/HOL and in Python. The code generation to C is presented in Sect. 5. We present experiments of using Mars 2.0 on several benchmarks in Sect. 6, and finally conclude in Sect. 7.

1.1 Related Work

Except for the aforementioned closely-related work, there exist a huge amount of work on the modeling, analysis, verification and code generation of CPSs. We briefly summarize them from two directions, which consider CPSs from a complete or a separate view respectively.

Several unified frameworks have been proposed for designing CPSs. The Metropolis design framework [2, 10] is a platform-based design environment for heterogeneous systems, that provides simulation, verification, and code synthesis by transforming all models to a unified meta-model language. However, it lacks support for physical plant modeling. Ptolemy [27] aims to design heterogeneous systems that combine different models of computation in terms of actors and provides modeling and simulation techniques for the combined models. Functional Mock-up Interface (FMI 3.0) [19] is an industrial standard maintained by the Modelica Association that enables the exchange and co-simulation of dynamic component models. It couples different simulation tools at system level by coordinating and synchronizing their respective executions. However, Ptolemy supports very limited facilities to model continuous behaviors [9], and furthermore, both Ptolemy and FMI are not designed for hardware architecture modeling and analysis.

Other lines of work consider limited perspectives of CPSs. UML, SysML [1] and MARTE [30] are traditional model based design environments for designing discrete systems, without support for physical plants. There are some works aiming for modeling and verifying continuous and hybrid behaviors, but without considering architectures. Zélus [7] extends the synchronous language Lustre [16] with ODEs and zero-crossing events for designing and implementing hybrid systems. It supports analysis of hybrid models by type systems and semantics [4, 7], and compilation that generates code for simulation and for embedded targets by source-to-source transformation [3, 6]. Differential dynamic logic is developed for reasoning about behaviors of hybrid dynamic models [25], and based on which the KeYmaera X prover [13] is implemented for safety analysis of dynamic systems. Their later work transformed verified high-level models of CPSs in differential dynamic logic to executable controllers, with the safety properties preserved [5, 14]. Though the correctness issue is addressed, only the discrete part of the model is considered.

There are a number of commercial tools supporting code generation from dynamic and hybrid systems, such as Simulink [21], AADL [24, 29], SCADE [12], Rational Rose [28], and TargetLink [33] etc. OSATE [24] is the AADL tool environment, which provides modeling and analysis of real-time systems in AADL and furthermore supports the automated code generation from AADL models including runtime behavior and scheduling to C code. However, they all validate the process of the code generation by simulation, without formal guarantee for correctness. SCADE [12] supports automatic code generation from their models and provides formal guarantee for correctness. It has been used for designing safety-critical embedded systems, but it is founded on the synchronous data-flow language Lustre [16], which does not address continuous plants. Moreover, all of the above work provide no support for architecture modeling and analysis.

2 MODELING AND SIMULATION

This section reviews the existing work on modeling and simulation: graphical modeling based on $AADL \oplus S/S$ and the co-simulation by transforming $AADL \oplus S/S$ to C, formal modeling based on HCSP and the simulation of HCSP.

2.1 Graphical Modeling and Simulation with $AADL \oplus S/S$

Using $AADL \oplus S/S$, a cyber-physical system is modeled with the following three layers:

Architecture layer The system architecture and its hardware platform are described by AADL components that define the structure, type and characteristics of composed hardware and software components.

Software layer The software behavior can be modeled either through AADL behavioral annexes or S/S diagrams.

Physical layer The physics of the cyber-physical system and its interaction with the hardware/software platform are modeled by S/S diagrams.

The simulation of $AADL \oplus S/S$ models is performed by executing the C code generated from the graphical models. The C code generation is divided into the following three parts.

Translating AADL to C. For each thread in the AADL part, a corresponding Thread object containing its component properties (such as dispatch protocol, priority, deadline, period, and minimum/maximum execution times) will be created. Besides the properties described in AADL, Thread also includes other properties like thread state.

Translating S/S to C. Matlab provides an automatic code generation tool to translate S/S diagrams into C code that can simulate the model step-by-step. To apply the code generation tool, we need to set some configuration parameters, such as the step size, the ODE solver, format of the generated code, etc. The C code generated from an S/S diagram by the tool can be roughly divided into three functions: initialization (input), computation (execute for one step), and finalization (output). Thus, the behavior of a Thread object thread can be defined by the following function pointers:

```
thread->initialize = initialization;
thread->compute = computation;
thread->finalize = finalization;
```

where *initialization*, *computation*, and *finalization* are the functions in the C file generated by Matlab's code generator.

Simulation. The C code of an $AADL \oplus S/S$ model is combined together through a function implementing the thread scheduling protocol such as the Highest Priority First (HPF) protocol. The communication between components is implemented by shared variables in the context of C code. The step size of the Matlab simulation is set to agree with that of AADL simulation. The simulation result can then be visualized using Python's plotting library, serving as a visual check that properties of the model are satisfied for the given initial state.

Fig. 2 shows the $AADL \oplus S/S$ model of an Isolette system which controls the temperature in an infant incubator. The system consists of one physical component describing the temperature evolution of the incubator, and this component is modeled by a Simulink diagram. The architecture part of the system is modeled using AADL. It consists of a central Processor with three component devices: Sensor, Controller, and Actuator. At the software layer, it consists of three threads with different priorities, which are scheduled according to the scheduling policy specified by the Processor. The simulation of this combined model can be found in [39].

2.2 Formal Modeling and Simulation with HCSP

Hybrid CSP (HCSP) is a formal language for describing hybrid systems which extends CSP by introducing differential equations for modeling continuous evolution, and interrupts for modeling the interaction between continuous evolution and communications. For full definition of HCSP, readers are referred to [17, 40]. Compared to the standard HCSP syntax, we make use of an extended language including datatypes such as strings and lists, operations on these datatypes,

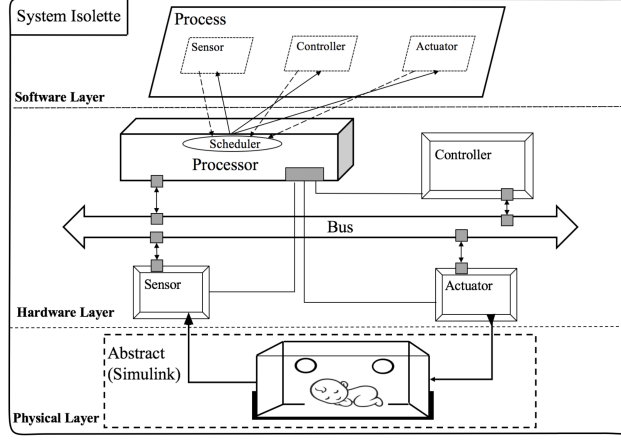


Fig. 2. Graphical model of Isolette system (from [39])

arrays of channels, and module definitions. We also allow definition of functions and procedures in a module. This makes translation of AADL \oplus S/S easier, but poses further challenges to simulation and code generation.

A simulator for the extended HCSP with a graphical user interface has been implemented. The backend of the simulator is implemented in Python. In particular, solving of ODEs is done using Python’s `scipy` package (function `solve_ivp`), which is also able to accurately calculate the time at which the boundary of the domain is reached using a root-finding algorithm. Finally, the simulator is linked to a web interface which is able to show the HCSP process in pretty-printed form, the steps of execution, and a plot of the variables in the process against time. This allows us to not only view the result of running an HCSP process, but also find out what went wrong if the process does not execute as expected.

As shown in Sect. 3, Mars 2.0 also supports the simulation of AADL \oplus S/S models by translating AADL \oplus S/S to HCSP first and then doing the simulation on the resulted HCSP models. Compared to the simulation by translating AADL \oplus S/S to C, the second approach guarantees the correctness of translation formally and thus is more reliable.

3 FROM AADL \oplus S/S TO HCSP

In this section, we first review translation procedures from AADL \oplus S/S that are introduced in existing work [35], where the translation procedures for Simulink and Stateflow are described in [44, 45]. Then, we introduce the new algorithms that translate Simulink and Stateflow diagrams to sequential HCSP processes.

3.1 Translation of AADL \oplus S/S

The translation of combined AADL \oplus S/S models to HCSP is introduced in [35]. We give a brief overview here. In addition to the AADL model and associated Simulink/Stateflow models, a configuration file is required, supplying additional information such as the correspondence between channel names in the AADL model and input/output variable names in the HCSP or Simulink/Stateflow models.

Translating threads. Each thread is translated into two HCSP processes. The *dispatch* process is used to dispatch the thread, according to its dispatch protocol. Currently the *periodic* and *aperiodic* protocols are supported. The *execution*

process models execution of the thread. It represents a state machine with 4 states: wait (waiting to be dispatched), ready (dispatched but not running), running (currently running) and await (waiting for some resource, such as bus access). The execution behavior of a thread is specified using either HCSP code directly, or using Simulink/Stateflow diagrams. In the latter case, the procedures given in [44, 45] are used to translate them into HCSP code, which is then spliced into the state machine model. In the latest version, proposed in this paper, we adopt the new translation algorithms for Simulink and Stateflow, which will be introduced in Sect. 3.2 and Sect. 3.3.

Translating the scheduler. The scheduler manages a group of threads and determines which thread to run at any time according to the scheduling protocol, which can be one of FIFO (first-in-first-out), RMS (rate-monotonic), DMS (deadline-monotonic) and HPF (highest priority first). For example, in the HPF case, each thread has a priority, and the scheduler maintains a pool of ready threads, at each time choosing the thread with the highest priority to run.

Translating devices and abstract components. Devices and abstract components (for modeling physical processes) are expressed using HCSP code directly, or using Simulink/Stateflow diagrams. As with threads, procedures in Sect. 3.2 and Sect. 3.3 can also be used to translate them into HCSP code, which is itself the translation of the component.

Translating buses and connections. Finally, we discuss translation of connections between threads. When two threads communicate directly without buses (e.g. when they are bound to the same processor), the connection is translated to a buffer modeled by a (stationary) ODE waiting for input and output communications. This models direct (asynchronous) communication between these threads.

When the communication is bound to one or more buses, additional code is required on the thread side to request and release bus access. Moreover, bus delay is modeled by inserting the corresponding delay in the implementation of the bus thread.

3.2 Translation of Simulink

The work [45] first introduced a translation algorithm from Simulink to HCSP, together with proof of correctness. In that work, a Simulink diagram is separated into discrete and continuous sub-diagrams, which are translated into separate HCSP processes. The entire Simulink diagram is then formed by parallel composition of these HCSP processes, with variables shared by the sub-diagrams transmitted by communication in HCSP. Communication is also used to model triggered subsystems. This method of translation has the advantage of putting plant and control in the Simulink model into separate processes. However, the extra communication complicates analysis and verification of the translated models, in particular those involving theorem proving.

The work [36] mentioned a different translation algorithm from Simulink to HCSP, that results in a sequential HCSP program. The focus of [36] is to define a denotational semantics for Simulink diagrams, and how this semantics can be used to prove the correctness of the translation from Simulink to HCSP using UTP theory, hence the translation itself is only described briefly and at a high level. In this paper, we give a systematic description of this translation algorithm. This algorithm differs from that in [45] mainly by removing the use of communication between discrete and continuous part of the diagram, as well as for triggered subsystems. This results in sequential programs that are easier to verify by theorem proving methods. Other improvements include giving a full treatment of discrete blocks with states, e.g. delay blocks, as well as computation blocks within the continuous part of the diagram. The Simulink blocks supported by our tool are summarized in Table 1.

Table 1. The Simulink blocks supported by Mars 2.0

Category	Blocks	Description
Sources	Constant	Outputs a constant signal.
	Clock	Outputs the current simulation time.
	Sine Wave	Outputs a sine wave.
	Signal Builder	Creates and generates interchangeable groups of signals whose waveforms are piecewise linear.
	Discrete Pulse Generator	Generates square wave pulses at regular intervals.
Continuous	Integrator	Continuous-time integration of the input signal.
	Transfer Function	Models a linear system by a transfer function of the Laplace-domain.
Discontinuities	Hit Crossing	Detects when the input signal reaches the crossing offset value in the direction specified by the crossing direction.
	Saturation	Limits input signal to the upper and lower saturation values.
Discrete	Discrete PID Controller	Implements the discrete-time PID control algorithm.
	Unit Delay	Samples and holds with one sample period delay.
Logical and Bit Operations	Logical Operators	Logical operators including AND, OR, NOT, and so on.
	Relational Operators	Applies the selected relational operator (such as \leq and $>$) to the inputs and outputs the result.
Math Operations	Add	Adds or subtracts inputs.
	Bias	Adds bias to input.
	Gain	Multiplies the input by a constant value.
	MinMax	Outputs min or max of input.
	Abs	Outputs the absolute value of input.
	Product	Multiplies or divides inputs.
	Sqrt	Outputs the square root of input.
	Square	Outputs the square of input.
Signal Routing	Data Store Memory	Defines and initializes a named shared data store, which is a memory region usable by Data Store Read and Data Store Write blocks that specify the same data store name.
	Mux	Multiplexes scalar or vector signals.
	Selector	Selects or reorders specified elements of a multidimensional input.
	Switch	Passes through the first input or the third input signal based on the value of the second input.
Subsystems	Normal Subsystems	Contains a subset of blocks within a model or system.
	Enabled Subsystems	If the value of the control signal is greater than zero, the subsystem executes.
	Triggered Subsystems	The subsystem executes once the value of the control signal crosses zero (falling, raising, or either).
Sinks	Scope	Displays input signals with respect to simulation time.

3.2.1 Causality graph. We first review some important concepts about Simulink models, in particular the causality graph. A Simulink model consists of blocks connected together by wires. Each block may be classified as *discrete* or *continuous*. Discrete blocks update periodically according to its *sample time*, while continuous blocks are updated continuously. Examples of discrete block include the gain block, which periodically compute its output as a certain multiple of its input, and the delay block, which delays its (discrete) input by one sample time.

Any discrete block can be described mathematically using an internal state, together with *update* (f) and *output* (g) functions. Consider a discrete block with sample time $st > 0$ and internal states s . Let x and y be the input and output signals of the block, respectively, then for each $k \in \mathbb{N}$, we have

$$\begin{aligned} s(k \cdot st) &= f(x(k \cdot st), s((k-1) \cdot st)) \quad (\text{update}) \\ y(k \cdot st) &= g(x(k \cdot st), s((k-1) \cdot st)) \quad (\text{output}) \end{aligned}$$

For example, the gain block has no internal state or update function, and its output function is given by:

$$y(k \cdot st) = a \cdot x(k \cdot st)$$

where y is the output signal, x is the input signal, a is the gain ratio, and st is the sample time. The delay block has a single state variable s , with update and output functions as follows:

$$\begin{aligned} s(k \cdot st) &= x(k \cdot st) \quad (\text{update}) \\ y(k \cdot st) &= s((k-1) \cdot st) \quad (\text{output}) \end{aligned}$$

The *causality graph* of the discrete part of the Simulink diagram is defined as follows. Each node of the graph corresponds to a wire. Each block connects each output wire to the set of input wires that the output function depends on. For example, the gain block connects its output to its input, but the delay block introduces no edges to the graph (as its output depends only on the internal state). We assume the Simulink diagrams under consideration are well-formed, in the sense that its causality graph is acyclic [36].

Examples of continuous blocks include the continuous version of the gain block, which always maintains its output as a certain multiple of its input, and the integrator block, which specifies its output as the integral of its input. Mathematically, the gain block can be defined as:

$$y(t) = a \cdot x(t) \quad \text{for all } t \geq 0$$

while the integrator block can be defined using an ODE:

$$\dot{y}(t) = x(t)$$

There are other continuous blocks that result in more complex ODEs. For example, the transfer function block used in the powertrain control benchmark (Sect. 6.2) with transfer function $\frac{1}{0.1s+1}$ is given mathematically by the following ODE: $\dot{y}(t) = -10y(t) + 10x(t)$. In general, a continuous block may also contain internal state variables, with some output defined as function of internal and input variables.

The causality graph of the continuous part of the Simulink diagram is defined as follows. Again, each node in the graph corresponds to a wire. Each block connects each output wire to the set of input wires it depends on through an equation, rather than through an ODE. For example, the continuous gain block connects its output to its input, while the integrator block and the above transfer function block introduces no edges. The entire Simulink diagram is well-formed if the combined causality graph of its continuous and discrete parts is acyclic.

3.2.2 Subsystems. Simulink allows hierarchical design of models using *subsystems*. They come in three types: normal subsystems, enabled subsystems, and triggered subsystems. All subsystems consists of blocks (and perhaps other subsystems) that can be considered as a single component of the model. Normal subsystems are executed at all times. Enabled subsystems are executed when a certain condition holds. Triggered subsystems are executed on the rising,

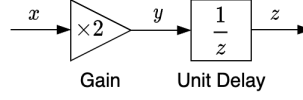


Fig. 3. Example of discrete diagram

falling, or either events of a triggering signal. We consider enabled and triggered subsystems consisting of discrete blocks only. The paper [36] gave the semantics and definition of causality graphs for all three types of subsystems.

3.2.3 Preprocessing. Before the main translation step, several preprocessing steps are performed on the Simulink diagram. First, normal and enabled subsystems are unfolded, with enabling condition recorded onto individual blocks of enabled subsystems. Next, we determine the sample times of blocks that are not explicitly specified. This is done by forward and backward propagation of sample times, according to the method given in the Simulink manual [21]. This classifies each block as either discrete or continuous. The sample time of the entire Simulink diagram is also determined at this step, as the greatest common divisor (GCD) of the sample times of the discrete blocks. The last preprocessing step separates the Simulink diagram into discrete and continuous parts, which are first translated individually (to be described below), then put together to form the translation result of the entire Simulink diagram.

3.2.4 Translating Discrete Diagrams. Each wire and internal state of discrete block corresponds to a variable in the translated HCSP program. To each discrete block B , we associate two HCSP code fragments $B.O$ and $B.U$ for output and update, respectively. For example, the gain block has:

$$\begin{aligned} B.O &\hat{=} y := a \cdot x \quad (\text{output}) \\ B.U &\hat{=} \text{skip} \quad (\text{update}) \end{aligned}$$

while the delay block has:

$$\begin{aligned} B.O &\hat{=} y := s \quad (\text{output}) \\ B.U &\hat{=} s := x \quad (\text{update}) \end{aligned}$$

Translation of the discrete part of the diagram begins by a topological sort of the causality graph, giving a linear order on the wires. By identifying each block with its output wire, this gives a linear order on the blocks as well. Suppose the discrete blocks are B_1, \dots, B_n according to this order, then the translation to HCSP first performs all output operations of blocks in order, then performs all update operations. That is:

$$\text{Discrete} \hat{=} B_1.O; \dots; B_n.O; B_1.U; \dots; B_n.U$$

For blocks with sample time different from the sample time of the entire diagram, a condition need to be added to take this into account. Likewise, the enabling condition need to be added for blocks within enabled subsystems.

Some initialization steps are also needed for the discrete part of the diagram. In particular, each block with internal state require initialization of its state. Triggered subsystems and Stateflow charts (to be discussed later) may also require initialization. We collect initialization of discrete (and later continuous) part of the diagram into the code fragment *Init*.

We give a concrete example to illustrate the translation procedure. Consider the diagram in Fig. 3. Suppose the topological order chosen for the causality graph is z, x, y , corresponding to the order Unit Delay, Gain on the blocks. Let s be the internal variable of the unit delay block. Then the result of translation is:

$$\text{Discrete} \hat{=} z := s; y := 2x; s := y$$

and the initialization is: $\text{Init}_d \hat{=} s := 0$.

3.2.5 Translating Continuous Diagrams. Each continuous block not involving ODEs can be specified by a function g from its input \mathbf{x} to its output \mathbf{y} . Therefore, such blocks can be represented by a variable substitution $\mathbf{y} \mapsto g(\mathbf{x})$. Let

$$G \hat{=} \{\mathbf{y}_1 \mapsto g_1(\mathbf{x}_1), \dots, \mathbf{y}_n \mapsto g_n(\mathbf{x}_n)\}$$

be the set of variable substitutions. Since the causality graph of the Simulink diagram is acyclic, there are no cycles in these variable substitutions, hence they form a well-defined substitution Γ by composition.

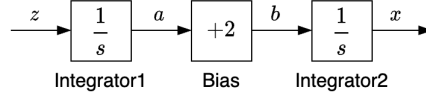


Fig. 4. Example of continuous diagram

Next, the translation of the continuous diagram consists of collecting together ODEs of the blocks, then performing the variable substitution Γ . This is represented as the ODE $\dot{\mathbf{y}} = \Gamma(\mathbf{x})$. The initialization Init_c gives the initial value of ODEs. We illustrate this with an example. Consider the diagram in Fig. 4. The two integrator blocks yield the ODE $\dot{a} = z, \dot{x} = b$. The bias block yields the substitution $b \mapsto a + 2$. Hence, the result of translation is the ODE $\dot{a} = z, \dot{x} = a + 2$. The initialization is: $\text{Init}_c \hat{=} a := 0; x := 0$.

3.2.6 Translating Triggered Subsystems. Triggered subsystems can be considered as a stateful block with two internal variables *pre* and *triggered*, to record respectively the last value of the trigger line and whether the subsystem was triggered at the last round. Initially, *pre* is set to 0 and *triggered* is set to **false**.

Let \mathbf{x} and \mathbf{y} be the input and the output of the triggered subsystem, and *cur* be the value of the trigger line. The trigger conditions $\text{trig}(\text{pre}, \text{cur})$ is defined as follows, depending on whether the trigger type is rising, falling, or either:

$$\begin{aligned} (\text{pre} < 0 \wedge \text{cur} \geq 0) \vee (\text{pre} \leq 0 \wedge \text{cur} > 0) & \quad (\text{rising}) \\ (\text{pre} > 0 \wedge \text{cur} \leq 0) \vee (\text{pre} \geq 0 \wedge \text{cur} < 0) & \quad (\text{falling}) \\ (\text{pre} < 0 \wedge \text{cur} \geq 0) \vee (\text{pre} > 0 \wedge \text{cur} \leq 0) & \quad (\text{either}) \\ \vee (\text{pre} = 0 \wedge \text{cur} > 0) \vee (\text{pre} = 0 \wedge \text{cur} < 0) & \end{aligned}$$

Let $\mathbf{y} := g(\mathbf{x})$ represent the computation performed by the (discrete) blocks in the triggered subsystem. At each round, this computation is performed if the subsystem was not triggered in the previous round ($\neg \text{triggered}$) and the trigger condition holds ($\text{trig}(\text{pre}, \text{cur})$). Then, we update the values of *pre* and *triggered* accordingly. In summary, the output and update processes of a discrete triggered subsystem DTrig can be defined as follows:

$$\begin{aligned} \text{DTrig.O} & \hat{=} \neg \text{triggered} \wedge \text{trig}(\text{pre}, \text{cur}) \rightarrow \mathbf{y} := g(\mathbf{x}) \\ \text{DTrig.U} & \hat{=} (\text{triggered}, \text{pre}) := (\text{TrigCond}, \text{cur}) \end{aligned}$$

3.2.7 Translating Simulink Diagrams. Translation of the entire Simulink diagram is formed by combining translation of its discrete and continuous parts. First, initialization of the discrete and continuous parts are performed. Then, a loop is entered where every round consists of the computation of the discrete diagram, followed by evolution of the continuous diagram for d time units, where d is the sample time of the entire diagram. That is:

$$\text{Diagram} \hat{=} \text{Init}_d; \text{Init}_c; \left(\text{Discrete}; t := 0; \langle \dot{\mathbf{y}} = \Gamma(\mathbf{x}), t = 1 \wedge t < \text{period} \rangle \right)^*$$

For example, for the combination of the diagrams in Fig. 3 and Fig. 4, the result of translation is (with sample time 1):

$$s := 0; a := 0; x := 0; (z := s; y := 2x; s := y; t := 0; \langle \dot{a} = z, \dot{x} = a + 2, \dot{t} = 1 \ \& \ t < 1 \rangle)^*$$

3.3 Translation of Stateflow

At a basic level, Stateflow diagrams consists of *states* and *transitions* between states. However, there are many additional features in Stateflow to provide various conveniences in modeling. They include hierarchical states, inter-level transitions, junctions, events and temporal events, messages, and so on. Moreover, Stateflow permits states to contain evolution by ODEs, making it a convenient tool for modeling hybrid systems, in particular switched systems. The existing work [15] describes in detail a translation procedure from Stateflow diagrams to HCSP, covering all of the above features. In this section, we give a brief overview of the main ideas, and an example of translation of ODEs in Stateflow, a feature that is used in the case study in Section 6.3.

The translation is organized into two stages: a syntax-directed translation stage, and a code-optimization stage. In the syntax-directed translation stage, each element in the Stateflow chart is translated into a single procedure according to the Stateflow semantics. For example, for each state, three procedures are created for the entry, during, and exit process of the state. The entry procedure calls the en-action of the state, then calls the entry procedures of its child states (in addition to updating some book-keeping variables). The during procedure tries the outgoing transitions, du-action of the state, inner transitions, and during procedures of active child states, in that order. The exit procedure of a state first performs the ex-action of the state, then calls the exit procedure of its child states. An event stack EL is used to keep track of current events. Raising an event is translated to first pushing the event onto the stack, call the execution of either the full chart (in the case of broadcast events) or a particular state (for directed events), and finally popping the event from the stack.

The above syntax-directed translation procedure is easy to understand and prove correct. However, it results in HCSP processes that consist of many procedure definitions, as well as other extraneous code. The second part of translation is a code-optimization stage. In this part, various optimization techniques common to compilers are performed. They include inlining of procedures, peephole optimization, constant propagation, and dead code elimination. The Stateflow translation is evaluated on 112 hand-designed examples, validating consistency of the semantics by comparing simulation results in Simulink and of the translated HCSP program.

3.3.1 Translation of continuous evolution. Stateflow permits specifying ODEs within a state. This can be combined with other features of Stateflow, in particular hierarchical states, resulting in potentially complex semantics. The translation from Stateflow to HCSP collects together ODEs within each state across the hierarchy, and determines which ODE to execute according to the current state. The boundary of each ODE correspond to conditions on outgoing transitions from the corresponding state.

Example. Consider the Stateflow diagram shown in Fig. 5 (reproduced from [31]). This diagram combines hierarchical states with specification of ODE within a state. The outer state specifies that variable x evolves according to $\dot{x} = y$. The inner states specify that variable t evolves according to $\dot{t} = 1$ and modifies the value of y on entry. Hence, the behavior of this diagram has y alternately taking values 1 and -1 , while x evolves in a sawtooth pattern. The translated HCSP contains the ODE evolution $\langle \dot{x} = y, \dot{t} = 1 \ \& \ t < 1 \rangle$ (where $t < 1$ is the negation of the condition on the outgoing transitions), together with code specifying updates to the variable y on transition between states A and B .

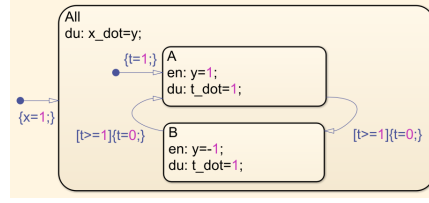


Fig. 5. Sawtooth example

4 VERIFICATION

In this section, we describe verification of HCSP processes by theorem proving. Verification using HHL Prover is based on hybrid Hoare logic, with two modes having been implemented, one in the proof assistant Isabelle, and one for sequential HCSP processes only but provides more automation.

4.1 Verification in Isabelle

HHL Prover in Isabelle is implemented based on the hybrid Hoare logic (HHL) for deductive verification of HCSP processes. HHL is first introduced and implemented in [20, 34], with assertions defined using Duration Calculus (DC) [42] and first-order logic. However, DC-based proof systems for HCSP are too complicated to be used in practice.

The HHL in Mars 2.0 is based on a recent improved proof system, mainly with the following differences: First, the semantics for HCSP processes is defined using traces, which record the ordered sequence of events consisting of communications and continuous evolution. Especially, each continuous evolution event records the readiness for potential communications. Traces can be combined in parallel with each other while guaranteeing that synchronized communications occur at the earliest possible time. Second, assertions in the HHL are defined as purely first-order predicates on traces. Based on the new assertions, for each construct in HCSP, a corresponding Hoare rule is defined in the weakest-precondition form. In particular, rules for combining assertions for parallel programs are defined based on the trace synchronizations, allowing us to conclude the specification of parallel processes from those of individual, sequential processes.

The prover in Isabelle has been applied to the case studies of lunar lander [41], Mars lander [38], and Chinese Train Control System (CTCS-3) [45]. More recently, we used the prover based on the new HHL to verify the correctness of the composition of one scheduler with two tasks for AADL thread scheduling.

4.2 Verification using HHLPy

HHLPy is a prover with a friendly graphical user interface for verifying properties of sequential HCSP processes [31]. The sequential part of HCSP processes contains discrete computation and continuous evolution statements, but not communication, interrupts or parallel processes. Properties specified in first-order logic are annotated as pre- and postconditions. Together with other annotations such as invariants and proof rules for ODEs, HHLPy can generate verification conditions automatically. These verification conditions are sent to the SMT solver Z3 [11] or Wolfram Engine. If all verification conditions are proved, the HCSP process is shown to satisfy the desired properties.

The Hoare logic rules in HHLPy include regular rules for reasoning about discrete programs, as well as rules for reasoning about ODEs such as differential weakening, differential invariant, differential cut, and differential ghost, which are adapted from differential dynamic logic [25, 26] to the case of HCSP processes. The main issue that need to

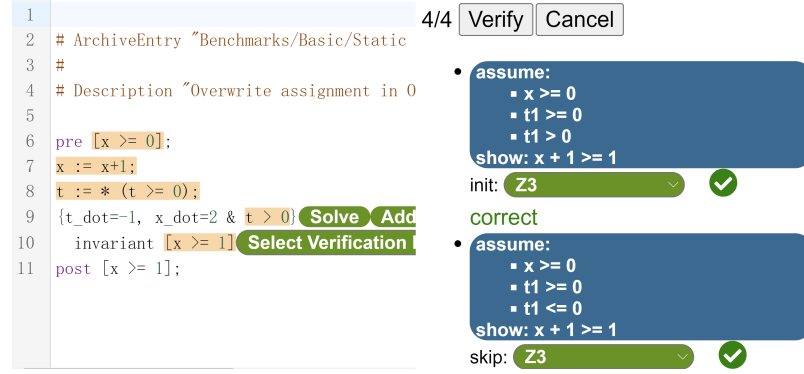


Fig. 6. Screenshot of user interface. The left panel is the editor area, where user can edit HCSP processes and annotations. The right panel shows verification conditions with their labels, solvers selected and results.

be addressed is the difference in semantics between HCSP and differential dynamic logic concerning the boundary of ODE evolution. For this, a new form of judgment, called *invariant triple* $\llbracket P \rrbracket \langle \dot{x} = e \rangle \llbracket Q \rrbracket$ is defined, stating that Q is an invariant of the ODE $\dot{x} = e$ under domain P . The rules of reasoning about ODEs are then defined in terms of both invariant triples and regular Hoare triples.

HHLPy is implemented in Python, with user interface implemented in JavaScript. Proof rules and the verification condition generation algorithm are implemented in the Python-based core engine. Fig. 6 shows a screenshot of the user interface. Specifically, the tool implements highlighting mechanisms to visualize different verification conditions. As shown in Fig. 6, relevant fragments for generating each verification condition are highlighted when hovering over it. In addition, the tool implements a labeling mechanism to distinguish between different verification conditions for proof reuse.

5 C CODE GENERATION

Automatic code generation forms the last part of the toolchain, which allows obtaining implementations of the system that are formally guaranteed to agree with the formal model. Code generation from HCSP to SystemC is introduced in [37], which gives a proof of correctness stated in terms of approximate bisimulation between the HCSP model and the generated SystemC code. In this section, we introduce code generation to C that is newly added in Mars 2.0. Compared to [37], we handle extensions to the HCSP language including different data types, module and channel parameters, and procedure definitions. Moreover, the synchronized communications of HCSP are implemented in C using the concurrency primitives in the POSIX pthreads library. In contrast to SystemC, which provides the same synchronized communication mechanisms as in HCSP, making their implementation more direct, the implementation of synchronized communication in C requires more care.

In other work currently under review, we presented the translation algorithm from HCSP to C, in particular how the synchronized communication in HCSP can be implemented using the pthreads library. We then prove a theoretical result stating the correctness of the implementation, in the sense of a bisimulation between the HCSP primitives and the translated concurrent C code under the interleaving model. The other part of the translation, discretization of ODEs, stays the same, and hence inherits the correctness result in terms of approximate bisimulation. The overall conclusion

is that the original HCSP code, including evolution following ODEs and interrupts, is approximately bisimilar to the generated C code.

We now give an overview of the code generation procedure to C, focusing on the implementation aspects.

5.1 Preprocessing

The first step performs several preprocessing steps to remove aspects of the extended HCSP language that cannot be translated directly. This includes in particular module and channel parameters. The translation from $AADL\oplus S/S$ to HCSP in Sect. 3.1 results in many processes that are instantiated from module templates. For example, the scheduler is parameterized by the processor ID, and the system may contain multiple schedulers instantiated from the same module with different processor IDs. In the first stage, we instantiate the modules into concrete HCSP processes.

Next, the translated HCSP code contains channels with parameters, which can both be matched in the external choice, and also be instantiated with variables. This is illustrated with the following (simplified) code fragment from the scheduler for the HPF protocol:

```
reqProcessor[0][_tid]?prior -->
  Pool_now := push(Pool_now, _tid);
  run_now := ...;
  run[0][run_now]!;
```

Intuitively, it says that whenever the scheduler on processor 0 receives a request from thread *tid*, indicating it is ready to run, the scheduler adds the thread to the pool of ready threads and recomputes the next thread to run based on priority, then sends the run signal to that thread via a communication. Both the matching on *tid* and the variable parameter *run_now* pose problems for code generation, as the latter assumes static channels. Hence, when the set of all threads on processor 0 is known, the above code fragment is expanded to include one communication for each thread. For example, if there are two threads with ID “t1” and “t2”, then the above code is expanded to:

```
reqProcessor[0]["t1"?prior -->
  Pool_now := push(Pool_now, "t1"); run_now := ...;
  if (run_now == "t1") { run[0]["t1"]!; }
  else if (run_now == "t2") { run[0]["t2"]!; }
reqProcessor[0]["t2"?prior -->
  Pool_now := push(Pool_now, "t2"); run_now := ...;
  if (run_now == "t1") { run[0]["t1"]!; }
  else if (run_now == "t2") { run[0]["t2"]!; }
```

After this expansion, all communications are on channels with concrete parameters.

5.2 Type inference

The extended HCSP language contains variables with several different types, including numbers, strings, lists, and so on. The simulator described in Sect. 2.2 is dynamically typed, figuring out types of variables at runtime. However, the same cannot be done for code generation to the C language. Hence, before code generation, type inference is performed on the entire HCSP program. This includes identifying types of each variable in each thread, as well as types of values communicated on each channel. We make the restriction that each variable in each thread can hold values of at most one type, and each channel can communicate values of at most one type. Type inference may need to be performed in multiple rounds, as knowledge of types may be passed between threads through a channel. The type inference

procedure flags an error if the above typing constraints are found to be violated, and also when it cannot conclude the type of some variable. In the latter case, extra initialization statements can be added for that variable to help the type inference procedure.

5.3 From HCSP to C

The main part of code generation translates each HCSP statement to the corresponding C code. For discretization of ODEs, we follow existing work [37]: When generating the C code, the step size of discretization is determined with respect to the given precision, then the state of the continuous evolution at the next time point is computed by the Runge-Kutta method. Thus, the C code generated by the continuous evolution statement $\langle F(\dot{s}, s) = 0 \& B \rangle$ is a loop structure that applies the Runge-Kutta method at each iteration, then waits for a step length of time before the next iteration, until the boundary condition B becomes false. The C code generated by the continuous interrupt statement $\langle F(\dot{s}, s) = 0 \& B \rangle \triangleright \parallel_{i \in I} (ch_i \triangleright \longrightarrow P_i)$ is also a loop structure, where each iteration performs a discrete version of the interrupt, which waits for the availability of communications among $\{ch_i\}_{i \in I}$ as well as for a time limit. If some communication can occur, the loop breaks and continues to implement the corresponding P_i ; otherwise, the state of continuous variables is updated and the loop is repeated, until B becomes false.

This stage results in C code that calls primitives corresponding to the implementation of time delay, input/output communications, and (discrete) interrupt. We provide a set of functions that implement the above time delay and communication primitives using the pthread library. There are two variants of implementation which differ in how to deal with passage of time. In the real time variant, actual system delays are inserted into the C code, so the execution of C code proceeds at the same speed as the original HCSP model. In the virtual time variant, execution of C code proceeds as fast as possible, which can be used as a simulator and obtains simulation results quickly.

We now briefly explain the implementation of communication primitives, leaving the details (as well as correctness proof) to another paper. The implementation maintains several global data structures that are protected by a single lock. All functions below acquire the lock at the beginning and release it at the end or when waiting for some event (time limit or communication) to occur.

Time delay. To realize the progress of time in each thread and synchronize time between different threads, a local clock is introduced to represent the time in each thread, and a global clock is introduced to represent the current global progress of simulation. The implementation of $\text{delay}(i, d)$ increases the local clock of thread i by d , and then waits for the global clock to catch up.

Input and output. Each channel is represented by a data structure with three fields: sender or receiver's token, channel content and channel direction. Each token defines whether the corresponding sender or receiver is available to participate in a communication. This takes the value of thread ID if some thread is ready to communicate, and -1 if otherwise.

Output $ch!e$ performs the following actions in sequence: set the token of $ch!$ to the thread number, put message e into channel ch , read the token of $ch?$ to see if it is available. If yes, perform the communication immediately, wait for the input side to perform corresponding actions, and then set the token of $ch!$ back to -1 ; otherwise, the thread will wait for $ch?$ to become available and then perform the communication. The input call $ch?x$ performs symmetric actions, except when the communication occurs, it will read the value from channel ch and assign it to variable x . In both cases, the thread releases the lock on the global data structures once to wait for the other side to perform corresponding actions, and the token in the channel reserves the communication so it cannot be interfered by other threads.

Interrupt. Interrupt combines the actions of time delay and waiting for a list of communications $\{io_i\}$. It performs the following actions: first check and modify the tokens corresponding to each communication $\{io_i\}$. If any communication is ready, it is performed and the call returns with the index of communication, to allow the corresponding ensuing action P_i to be performed. Otherwise it increments the local clock by the time delay of the interrupt, waits for all communications as well as for the global clock to catch up. If a communication occurs first, the local clock is set back to the value of global clock at the time, and the call returns the index of communication. If the global clock catches up first, the call returns to indicate no communication is performed within this step. In the case of communication, wait for the other side to perform corresponding actions as in the description of input and output above.

Main function. The main function of the generated code first initializes the global variables, and then starts one thread for the code generated from each HCSP process. After all threads finished running, it releases resources for the global variables.

6 EXPERIMENTS

In this section, we describe several case studies of using the toolchain in practice. The toolchain, as well as all of the case studies, are available for download at <https://github.com/intgq/Mars-2.0>.

6.1 Examples in Existing Work

We first give a brief summary of existing case studies using the Mars toolchain.

In [44, 45], modeling and verification of the Chinese High-speed Train Control System (CTCS) are studied using pure Simulink/Stateflow environment for modeling and then transforming to HCSP for verification. The verification shows that the train can successfully complete the level transition from CTCS-2 to CTCS-3 and the mode transition from full supervision to calling on simultaneously in a combined scenario.

In [41], the modeling and verification of the slow descent phase of lunar lander are studied. First, a closed loop model consisting of the lander's physical dynamics and the program controlling the lander is built; Second, the safety property is formally defined that if the initial conditions stay in a reasonable range, then the velocity during the landing must be maintained around a given value; Finally, the property is formally verified for the given model. In [38], the authors apply the Mars toolchain to model the descent guidance control phase of the recently launched Tianwen I mars lander, and verify that it correctly controls the velocity of the lander.

6.2 Simulation of Simulink/Stateflow Models

We tested the procedure for translating Simulink/Stateflow models in Section 3.2 and 3.3 on some more complex examples. The Powertrain control verification benchmark [18] consists of a series of models of powertrain control in automobile systems. The simpler model already uses many Simulink features, such as transfer functions (Transfer Fcn), Mux and Selectors (whose explanation can be found in Table 1) for dealing with vectors of signals, as well as Matlab expressions to specify polynomials. The Yoyo control example is representative of models where a Stateflow diagram is placed in the middle of a Simulink model, used to provide control input. It also contains Simulink blocks such as hit-cross that generates an event whenever some signal crosses a given threshold. We translated both models into HCSP, and performed simulation and code generation. The results of simulating the HCSP code in the powertrain control model is shown in Fig. 7. This agrees with the result of running the generated C program, as well as the simulation result within Simulink.

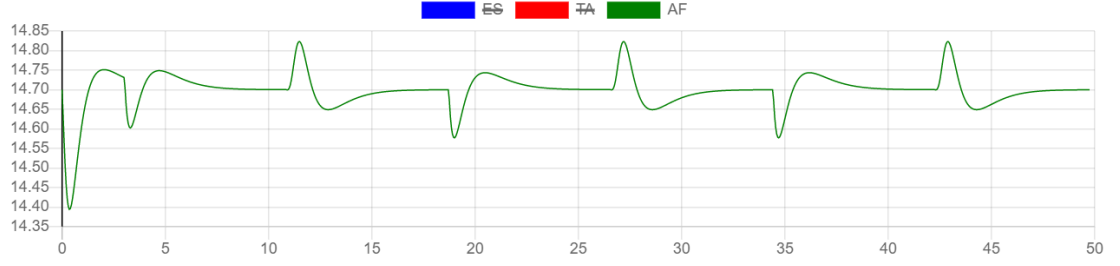


Fig. 7. Simulation result of powertrain control in the HCSP simulator. The x-axis is time and the y-axis is AF (air-fuel ratio)

6.3 Translation and Verification of Switched Systems

In this part, we present a case study on modeling switched systems using Stateflow, followed by verification using HHLPy. The automatic cruise controller [23] is a switched system with six modes of operation and eleven transitions between them, which include PI controller, acceleration, two service brakes and two emergency brakes. Stability of the controller has been verified using KeYmaera X in [32]. Since HHLPy currently only supports verifying safety properties, we verify instead a rephrasing of the stability condition using safety properties.

We modeled the system using Stateflow charts, shown in Fig. 8. The Stateflow model is slightly different from the original model, as Stateflow models generally have deterministic behavior. Despite these modifications, the model captures the essential behaviors of the original model. The Stateflow model was translated into an HCSP process using the algorithm in Section 3.3.

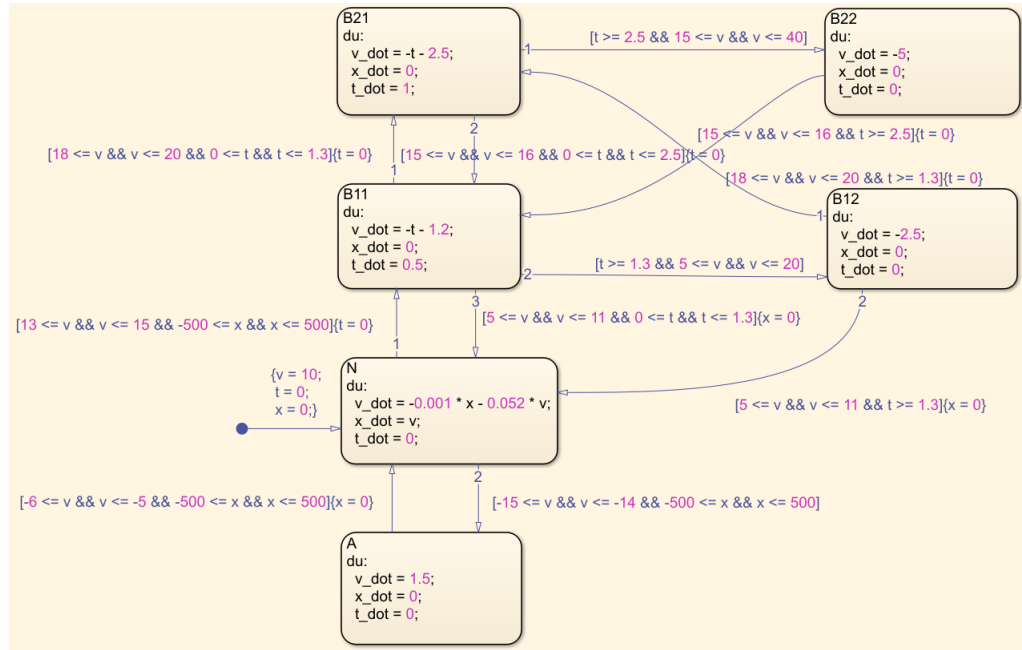


Fig. 8. Stateflow Model of Automatic Cruise Control

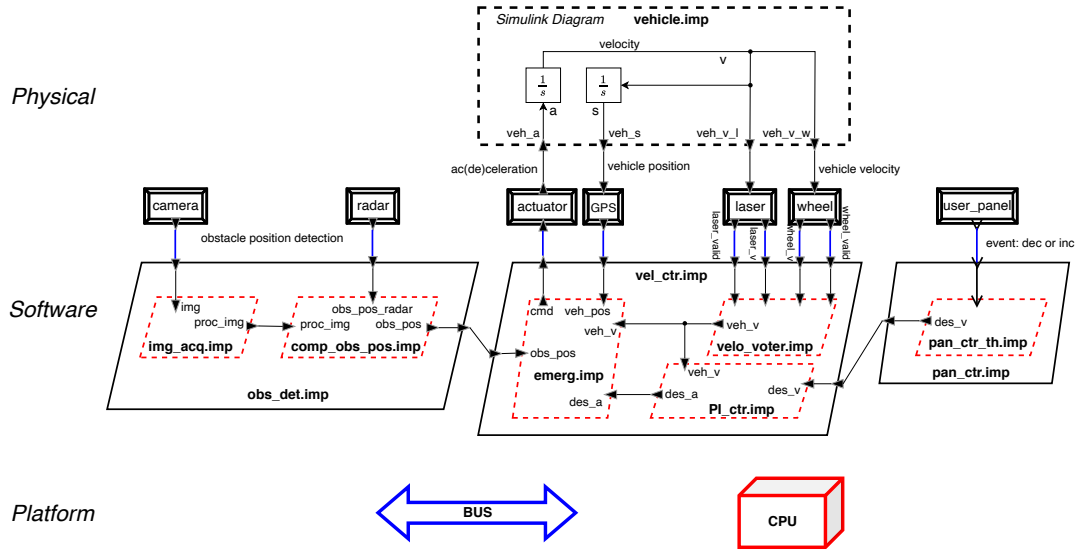


Fig. 9. Automatic Cruise Control System (borrowed from [35])

We then verified the following safety property of the HCSP process: given that initial velocity satisfies $v_0^2 < \delta^2$ with δ sufficiently small ($\delta > 0 \wedge \delta \leq 11 \wedge \delta < \epsilon - 0.1$) and initial mode is PI controller (state N), then velocity satisfies $v^2 < \epsilon^2$ throughout for all $\epsilon > 0.1$. This safety property is similar but weaker than stability (which impose conditions for all $\epsilon > 0$). The verification steps are similar to that in [32] for stability, concerning the PI controller only. Invariants for the system are that the system always stays in state N , together with a polynomial constraint based on the Lyapunov function of state N , which is verified by differential invariant rule. Due to the many states actually present in the system, a total of 1288 verification conditions are generated, all of which are checked correct using the Z3 solver.

6.4 The Whole Design of Automatic Cruise Control System

For the last case study, we adopt the realistically-scaled Automatic Cruise Control System (ACCS for short) from [35] to explain the whole procedures. The translation from AADL \oplus S/S to HCSP and the verification for the generated HCSP model were addressed in [35]. In this case, we add a new part: from HCSP to C (Sect. 6.4.3).

The ACCS is modeled using AADL \oplus S/S in [35] and decomposes into three layers, as shown in Fig. 9. The physical layer is the physical vehicle described by a Simulink diagram. The software level defines control of the system. It contains three processes for obstacle detection, velocity control, and panel control, and each process is composed of several threads. These processes interact with the environment through devices. The platform layer consists of bus and processor. The connections in the software level can be bound to buses in the platform level. All threads are bound to the single processor, with HPF scheduling policy.

The execution of ACCS is as follows. A vehicle is placed at the starting point initially and the driver can accelerate (inc) and decelerate (dec) the vehicle by the user_panel. Thread pan_ctr_th deals with the commands from the driver and then sends desired velocities (des_v) to the discrete PI controller (thread PI_ctr). Meanwhile, process obs_det detects the obstacles ahead by a camera and a radar and provides the velocity controller process (vel_ctr) with the real-time position of obstacle (obs_pos). Thread velo_voter in process vel_ctr monitors the velocity of the vehicle

using laser and another device located on one wheel of the vehicle and produces the real-time velocity of the vehicle (veh_v) to the discrete PI controller PI_ctr and the emergency control thread ($emerg$). Based on the real-time velocity of vehicle and the desired velocity received, PI_ctr computes a desired acceleration (des_a) which will be sent to $emerg$. Finally, $emerg$ collects the real-time position (veh_pos by GPS) and velocity of the vehicle, the desired velocity set by the driver, and the real-time position of the obstacle to work out a command, by some emergency control strategy, which will update the acceleration of the vehicle through actuator. The vehicle moves according to the new acceleration and the above procedure repeats.

6.4.1 Translation from AADL \oplus S/S to HCSP. In [35], the AADL \oplus S/S model of the ACCS of Fig. 9 is stored in one JSON file, which was taken as an input by the translator to generate files of HCSP modules as outputs. In this work, however, the AADL \oplus S/S model of the ACCS is composed of three parts: (1) AADL files describing the three levels of architecture; (2) Simulink/Stateflow diagrams of XML format modeling the behaviours of threads, devices, and physical environment; and (3) a configuration file (introduced at the beginning of Sect. 3.1) of JSON format. The translator takes these files as inputs and generates files of HCSP modules of TXT format as outputs, where the translations for Simulink and Stateflow diagrams adopt the new algorithms introduced in Sect. 3.2 and 3.3. Readers could refer to [35] for more details of the translation procedures. Notice that the architecture of the ACCS could be changed (such as adding a bus) to obtain variants of the model by modifying the AADL files directly.

6.4.2 Verification for HCSP. A generated HCSP model should be verified against some desired properties before being translated to executable code. Since the HCSP model of the ACCS of Fig 9 is extremely complex, it was abstracted as a simple parallel composition of a plant and a controller in [35] and then the safety property (the vehicle will never collide onto the moving obstacle ahead and its velocity will never exceed the upper limit) could be verified using HHL in Isabelle/HOL.

6.4.3 Translation from HCSP to C. The above obtained HCSP files can be fed into the translator introduced in Sect. 5 to generate C code. This part is not included in the previous work of [35]. The communication between processes is implemented using pthreads and the correctness of the code generation is proved based on approximate bisimulation, i.e., an HCSP model and the C code generated from it are in some approximate bisimulation relation. The HCSP model is specified by a formal language and therefore verifiable. Thus, the C code generated from HCSP files is more reliable than from the graphical AADL \oplus S/S model directly (Sect. 2.1). The generated C code is of 3500–4000 lines, longer than the C code (about 2500 lines) generated by [39] (Sect. 2.1). One major reason is that the generated C code in this paper is approximately bisimilar to the original AADL \oplus S/S model of ACCS, which means the detailed behaviours of ACCS are reflected in the C code and vice versa, while it is not the case for [39] where no such bisimulation can be guaranteed.

6.4.4 Comparison. We consider the following scenario. At the beginning, the driver pushes the inc button three times with time interval 0.5s in between to set a desired speed to 3m/s. After 30s, the driver pushes the dec button twice in 0.5s time intervals to decrease the desired speed. On the obstacle side, we assume that the obstacle appears at time 10s and position 35m, then moves ahead with velocity 2m/s, before finally moving away at time 20s and position 55m.

We adopt the parameters setting of Table 1 in [35] for the threads and devices in the ACCS of Fig. 9, except that the deadline of thread pan_ctr_th is shrunk to 50ms in this paper. We compare our results with the work [39] (Sect. 2.1) and the simulation of the HCSP model of ACCS [35]. It should be noted that bus latency is not considered in [39]. Thus, to make it fair, we only consider the case with on bus latency, i.e., we ignore the bus component in Fig. 9. Examples involving bus latency can be found in [35].

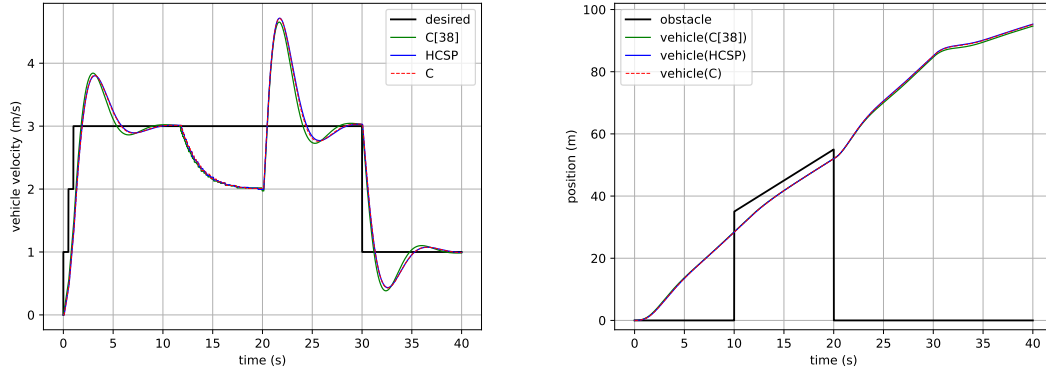


Fig. 10. Comparison of execution results

The left of Fig. 10 shows the evolutions of the vehicle speed, where the black line denotes the desired velocity set by the driver, and the red, green, and blue lines denote the results of our work, the work of [39] (Sect. 2.1), and the simulation of the HCSP model of ACCS [35], respectively. We can see that the execution of the generated C code (red line) is almost the same with the simulation of its HCSP model (blue line).

From all the three results, we can see that the vehicle accelerates to the desired speed (3m/s) in 10s. The fluctuation during [2s, 10s] reflects feature of PI controllers. It then decelerates to avoid the collision onto the obstacle ahead. After the obstacle moves away (at 20s), the vehicle accelerates again to the desired speed. At 30s, the driver pushes the dec button to adjust the desired velocity to 1m/s and we can see that the vehicle decelerates to 1m/s in about 6s under the PI controller. The positions of the vehicle and of the obstacle with respect to time are shown on the right of Fig. 10.

7 CONCLUSION

This paper presents Mars 2.0, an integrated toolchain for formal design of CPSs, supporting to take physicality, software and architecture into account uniformly. Mars 2.0 covers the whole design process of CPSs, from graphical modeling in AADL \oplus S/S at the beginning, to formal modeling in HCSP for simulation and verification, and to final code implementation with correctness guarantee. It realizes the automatic transformation from AADL \oplus S/S to HCSP, and from HCSP to SystemC/C, both of which have formal correctness guarantees by proving the semantic consistency between corresponding source and target models. Therefore, a more reliable CPS can be developed by using Mars 2.0 compared with existing tools. We also demonstrate its use on several case studies of varying complexity.

As future work, we will try to integrate more functionalities into Mars 2.0, such as modeling and verification of hybrid systems containing delay or stochastic differential equations. We are also considering to implement a more friendly graphical user interface for the co-modeling of AADL \oplus S/S, which currently only supports the textual descriptions of models. Finally, we intend to apply the toolchain to real-world case studies on a larger scale.

REFERENCES

- [1] 2013. *SysML V 1.4 Beta Specification*. <http://www.omg.org/spec/SysML>.
- [2] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto L. Sangiovanni-Vincentelli. 2003. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer* 36, 4 (2003), 45–52.

- [3] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. 2011. A hybrid synchronous language with hierarchical automata: static typing and translation to synchronous code. In *EMSOFT 2011*. ACM, 137–148.
- [4] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. 2012. Non-Standard Semantics of Hybrid Systems Modelers. *J. Comput. System Sci.* 78 (May 2012), 877–910.
- [5] Rose Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. 2018. VeriPhy: verified controller executables from verified cyber-physical system models. In *PLDI 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 617–630.
- [6] Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. 2015. A Synchronous-Based Code Generator for Explicit Hybrid Systems Languages. In *CC 2015*, Björn Franke (Ed.). Springer, 69–88.
- [7] Timothy Bourke and Marc Pouzet. 2013. Zélus: a synchronous language with ODEs. In *HSCC 2013*, Calin Belta and Franjo Ivancic (Eds.). ACM, 113–118.
- [8] Mingshuai Chen, Xiao Han, Tao Tang, Shuling Wang, Mengfei Yang, Naijun Zhan, Hengjun Zhao, and Liang Zou. 2017. MARS: A Toolchain for Modelling, Analysis and Verification of Hybrid Systems. In *Provably Correct Systems*, Michael G. Hinchey, Jonathan P. Bowen, and Ernst-Rüdiger Olderog (Eds.). Springer, 39–58.
- [9] Fabio Cremona, Marten Lohstroh, David Broman, Edward A. Lee, Michael Masin, and Stavros Tripakis. 2019. Hybrid Co-simulation: It’s about time. *Softw. Syst. Model.* 18, 3 (2019), 1655–1679.
- [10] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu. 2007. A Next-Generation Design Framework for Platform-based Design. In *DVCon 2007*.
- [11] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS 2008*, C. R. Ramakrishnan and Jakob Rehof (Eds.). 337–340.
- [12] François Xavier Dormoy. 2008. SCADE 6: a model based solution for safety critical software development. In *ERTS 2008*.
- [13] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völpl, and André Platzer. 2015. KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems. In *CADE (LNCS, Vol. 9195)*. Springer, 527–538.
- [14] Luis Garcia, Stefan Mitsch, and André Platzer. 2019. HyPLC: hybrid programmable logic controller program translation for verification. In *ICCPs 2019*. ACM, 47–56.
- [15] Panhua Guo, Bohua Zhan, Xiong Xu, Shuling Wang, and Wenhui Sun. 2022. Translating a large subset of stateflow to hybrid CSP with code optimization. *J. Syst. Archit.* 130 (2022), 102665.
- [16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320. <https://doi.org/10.1109/5.97300>
- [17] Jifeng He. 1994. From CSP to Hybrid Systems. In *A Classical Mind, Essays in Honour of C.A.R. Hoare*. Prentice Hall International (UK) Ltd., 171–189.
- [18] Xiaoqing Jin, Jyotirmoy V. Deshmukh, James Kapinski, Koichi Ueda, and Kenneth R. Butts. 2014. Powertrain control verification benchmark. In *HSCC’14*, Martin Fränzle and John Lygeros (Eds.). ACM, 253–262.
- [19] Andreas Junghanns, Cláudio Gomes, Christian Schulze, Klaus Schuch, Pierre R., Matthias Blaesken, Irina Zacharias, Andreas Pillekeit, Karl Wernersson, Torsten Sommer, Christian Bertsch, Torsten Blochwitz, and Masoud Najafi. 2021. The Functional Mock-up Interface 3.0 - New Features Enabling New Applications. In *Proceedings of 14th Modelica Conference 2021*. to appear.
- [20] Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. 2010. A Calculus for Hybrid CSP. In *APLAS 2010*. 1–15.
- [21] MathWorks Inc. 2013. *Simulink User’s Guide*. http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf.
- [22] MathWorks Inc. 2013. *Stateflow User’s Guide*. http://www.mathworks.com/help/pdf_doc/stateflow/sf Ug.pdf.
- [23] Jens Oehlerking. 2011. *Decomposition of stability proofs for hybrid systems*. Ph.D. Dissertation. Carl von Ossietzky University of Oldenburg.
- [24] OSATE. 2017. . <https://osate.org>.
- [25] André Platzer. 2018. *Logical Foundations of Cyber-Physical Systems*. Springer, Cham.
- [26] André Platzer and Yong Kiam Tan. 2020. Differential Equation Invariance Axiomatization. *J. ACM* 67, 1 (2020), 6:1–6:66.
- [27] Claudius Ptolemaeus (Ed.). 2014. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org. <http://ptolemy.org/books/Systems>
- [28] Rational Rose. 2017. . <http://www-03.ibm.com/software/products/en/rosemod>.
- [29] SAE International Standards. 2017. Architecture Analysis & Design Language (AADL), Revision C. (2017).
- [30] Bran Selic and Sebastien Gerard. 2013. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. The MK/OMG Press.
- [31] Huanhuan Sheng, Alexander Bentkamp, and Bohua Zhan. 2023. HHLPy: Practical Verification of Hybrid Systems Using Hoare Logic. In *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14000)*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.). Springer, 160–178. https://doi.org/10.1007/978-3-031-27481-7_11
- [32] Yong Kiam Tan, Stefan Mitsch, and André Platzer. 2022. Verifying Switched System Stability With Logic. In *HSCC ’22*. ACM, 2:1–2:11.
- [33] TargetLink. 2017. . <https://www.dspace.com/en/inc/home/products/sw/pgcs/targetli.cfm>.
- [34] Shuling Wang, Naijun Zhan, and Liang Zou. 2015. An Improved HHL Prover: An Interactive Theorem Prover for Hybrid Systems. In *ICFEM 2015*. Springer, 382–399.
- [35] Xiong Xu, Shuling Wang, Bohua Zhan, Xiangyu Jin, Jean-Pierre Talpin, and Naijun Zhan. 2022. Unified graphical co-modeling, analysis and verification of cyber-physical systems by combining AADL and Simulink/Stateflow. *Theor. Comput. Sci.* 903 (2022), 1–25.

- [36] Xiong Xu, Bohua Zhan, Shuling Wang, Jean-Pierre Talpin, and Naijun Zhan. 2023. A denotational semantics of Simulink with higher-order UTP. *Journal of Logical and Algebraic Methods in Programming* 130 (2023).
- [37] Gaogao Yan, Li Jiao, Shuling Wang, Lingtai Wang, and Naijun Zhan. 2020. Automatically Generating SystemC Code from HCSP Formal Models. *ACM Trans. Softw. Eng. Methodol.* 29, 1 (2020), 4:1–4:39.
- [38] Bohua Zhan, Bin Gu, Xiong Xu, Xiangyu Jin, Shuling Wang, Bai Xue, Xiaofeng Li, Yao Chen, Mengfei Yang, and Naijun Zhan. 2021. Brief Industry Paper: Modeling and Verification of Descent Guidance Control of Mars Lander. In *RTAS 2021*. IEEE, 457–460.
- [39] Haolan Zhan, Qianqian Lin, Shuling Wang, Jean-Pierre Talpin, Xiong Xu, and Naijun Zhan. 2019. Unified Graphical Co-modelling of Cyber-Physical Systems Using AADL and Simulink/Stateflow. In *UTP 2019 (Lecture Notes in Computer Science, Vol. 11885)*, Pedro Ribeiro and Augusto Sampaio (Eds.). Springer, 109–129.
- [40] Naijun Zhan, Shuling Wang, and Hengjun Zhao (Eds.). 2017. *Formal Verification of Simulink/Stateflow Diagrams, A Deductive Approach*. Springer.
- [41] Hengjun Zhao, Mengfei Yang, Naijun Zhan, Bin Gu, Liang Zou, and Yao Chen. 2014. Formal Verification of a Descent Guidance Control Program of a Lunar Lander. In *FM 2014*. 733–748.
- [42] Chaochen Zhou and Michael R. Hansen. 2004. *Duration Calculus - A Formal Approach to Real-Time Systems*. Springer.
- [43] Chaochen Zhou, Ji Wang, and Ravn Anders P. 1996. A Formal Description of Hybrid Systems. In *Hybrid Systems (LNCS, Vol. 1066)*. 511–530.
- [44] Liang Zou, Naijun Zhan, Shuling Wang, and Martin Fränzle. 2015. Formal Verification of Simulink/Stateflow Diagrams. In *ATVA 2015*. Springer, 464–481.
- [45] Liang Zou, Naijun Zhan, Shuling Wang, Martin Fränzle, and Shengchao Qin. 2013. Verifying Simulink diagrams via a Hybrid Hoare Logic Prover. In *EMSOFT 2013*, Rolf Ernst and Oleg Sokolsky (Eds.). IEEE, 9:1–9:10.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009