Case Study: Modeling, Simulation, Verification, and Code Generation of an Automatic Cruise Control System

Xiong Xu[1 [0000−0003−4236−9992]], Shuling Wang[3 [0000−0002−2798−2660]], Zekun Ji[1,2 [0009−0000−0302−1737]], Qiang Gao[1,2 [0009−0000−4663−8622]], Xiangyu Jin[1,2 [0000−0001−6176−2242]], Bohua Zhan[1,4 [0000−0001−5377−9351]], and Naijun Zhan[1,5 [0000−0003−3298−3817]⋆]

[1] KLSS (CAS) and SKLCS, Institute of Software, Chinese Academy of Sciences, Beijing, China
{xux,wangsl,jizk,gaoqiang,jinxy,bzhan,znj}@ios.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
[3] National Key Laboratory of Space Integrated Information System, Institute of Software, Chinese Academy of Sciences, Beijing, China
[4] Huawei Technologies Co., Ltd., Beijing, China
[5] School of Computer Science, Peking University, Beijing, China

Abstract. Cyber-Physical Systems (CPSs) seamlessly integrate computational and physical capabilities, that expand the capabilities of physical world through communication, computation, and control. As pointed out by Jones, any failure of these systems may cause severe catastrophes, especially for safety-critical applications. The Model Based Design (MBD) approach can control complexity of systems by appropriate abstraction/refinement and composition/decomposition, and meanwhile provides techniques based on modeling, analysis, and verification in order to detect and correct errors in the early stage of design. However, the MBD for CPSs faces challenges, for it is difficult to consider all the three aspects, including physical processes, software functionality, and system architecture, uniformly in CPS design. To address this issue, we proposed a unified graphical framework combing AADL and Simulink/Stateflow, called AADL⊕S/S, for modeling all the three aspects of CPSs, and we also presented the translation from informal AADL⊕S/S to formal Hybrid Communicating Sequential Processes (HCSP) and finally to C code. In this paper, we introduce a case study of a realistically-scaled automatic cruise control system. The goal of this case study is to show the whole procedure of modeling, simulation, verification, and code generation of the MBD methodology, illustrate the combined framework of AADL and Simulink/Stateflow to cover all the three aspects of CPS design, as well as the capability of analyzing the combined model, and finally demonstrate the practicability of our MBD approach for complex CPSs.

Keywords: CPS · MBD · AADL · Simulink/Stateflow · HCSP · Simulation · Verification · Code Generation.

⋆ Corresponding author.

1 Introduction

Cyber-physical systems (CPSs) tightly couple hardware and software to sense and actuate a physical environment. CPSs are ubiquitous in our daily life such as aerospace, high-speed train, and automotive, and are expected to achieve desired behaviors, especially for safety-critical ones. CPS can be dated back to Jones's computer-based systems [17], later his DSoS (*dependable systems of systems*) [14]. As stressed by Jones [17, 4], dependability and reliability are essential for these systems. To attack these issues, model based design (MBD), has become a popular development approach in the CPS community. In the MBD approach, a system is usually modeled as different levels of abstraction for different purposes. However, the MBD of CPSs is not such easy. The key reason for the difficulty is that when modeling CPSs, it is paramount to take multiple perspectives e.g. their software functionalities, physical environment, hardware platform and system architecture into account uniformly, as shown in Fig. 1.

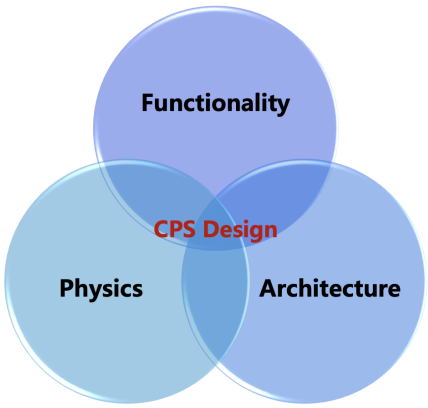

Fig. 1. The three perspectives of CPS design

Unfortunately, most of existing design methodologies and workflows do not support all these design aspects uniformly. For example, AADL [11, 23] features strong capabilities for describing the architecture of a system due to the pragmatic effectiveness of combining software and hardware component models. However, the core of AADL only supports modeling of embedded system hardware and abstraction of its relevant discrete behavior, and does not support the description of the continuous physical processes to be controlled and its combination with software. By contrast, Simulink/Stateflow [19, 20], the de-facto industry standard for model-based analysis and design of embedded systems, is best-suited for modeling and analyzing continuous physical processes, discrete computations and their combination, but it cannot naturally model system architectures and hardware platforms.

To address the above issue, we presented AADL⊕S/S [30, 27], a combination of AADL and Simulink/Stateflow (S/S), that provides a unified graphical modeling formalism to represent all three perspectives of CPS design in Fig. 1. An overview of AADL⊕S/S is given in Fig. 2. Using AADL⊕S/S, a CPS 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 modelled either through AADL behavioral annexes or Simulink/Stateflow diagrams.

Physical layer The physics of the CPS and its interaction with the hardware/software platform are modeled by Simulink/Stateflow diagrams.

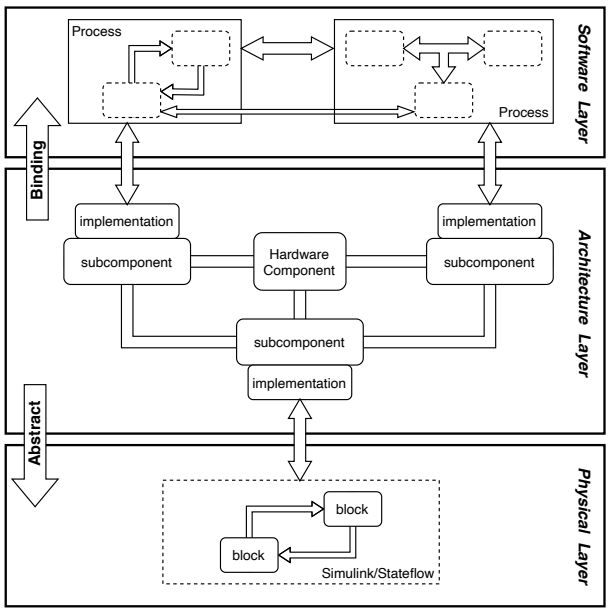

Fig. 2. An overview of AADL⊕S/S

We have developed a toolchain, called Mars [7, 29], for modeling, analysis, verification, and code generation of CPSs. With the aid of the toolchain, this paper will introduce a realistically-scaled example of an automatic cruise control system to illustrate the whole procedure of the MBD approach for CPSs, including modeling, simulation, verification, and code generation. Concretely, the graphical model of the system is constructed using AADL⊕S/S and then it is translated to a formal HCSP (Hybrid Communicating Sequential Processes) model, based on which the simulation and verification can be performed. From

the verified HCSP model, we can continue to generate the C code. Since the correctness of the translation is guaranteed, the generated C code is more reliable. The case study is an extension of our previous work in [27], including:

- The translation of Simulink/Stateflow is simplified and improved.
- The verification is enhanced by a more automated HHL prover.
- The C code generation from the translated HCSP model is considered.

Paper Organization Section 2 introduces briefly the basic notions in our formalism, including the informal aspect including AADL and Simulink/Stateflow, the formal aspect HCSP, and the framework of our toolchain Mars. The automatic cruise control system as the case study is illustrated in detail in Section 3, followed with the simulation, verification, and code generation of this complex system in Section 4 and 5. After reviewing related works in Section 6, we conclude in Section 7.

2 Background

2.1 AADL

AADL is an Architecture Analysis & Design Language used to model embedded real-time systems as assembly of software components mapped onto execution platforms [11, 23, 10]. An AADL specification is composed of software, hardware, and composite systems. However, at present, there is no suitable AADL component describing the continuous behavior of physical environments. Thus, in our setting, we choose to represent the physical environments as `system` components.

Software in AADL consists of `data`, `subprogram`, `threads`, and `processes`. A `data` component represents a data type. A `subprogram` component represents executable code that can be called, with parameters provided by threads and other subprograms. A `thread` component represents the fundamental unit for executing a sequential flow of control behavior. A `process` component, which is closely affiliated to a `processor` component, refers to a software instance responsible for executing threads. It usually contains multiple `thread` components, whose execution is managed by a scheduler.

The hardware side represents computation and communication resources including `processor`, `memory`, `bus` and `device` components. A `processor` component represents the hardware and software responsible for scheduling and executing task threads. A `memory` component is used to represent storage entities for data and code. A `device` component models a component interacting with the environment, such as sensor or actuator. A `bus` component represents a physical connection among execution platform components. Finally, a `system` is a top-level component consisting of a hierarchy of software and hardware components.

Communication among different components is realized by `connections` via ports, parameters and access to shared data. Especially, port connection represents transmission of data, event, or data/event between different threads, or between threads and hardware components such as processors or devices.

2.2 Simulink/Stateflow

Simulink [19] is an environment for model-based design of dynamical systems, and has become a de-facto standard in the embedded systems industry. Generally, a Simulink diagram contains a set of blocks, subsystems, and wires, where blocks and subsystems cooperate by exchanging data flows through connected wires. Simulink provides an extensive library of pre-defined blocks for building and managing such block diagrams, and also a rich set of fixed-step and variable-step ODE solvers for simulating dynamical systems. Blocks can be grouped into subsystems, such as normal, triggered, and enabled subsystems, to establish a hierarchical structure on Simulink diagrams. A hierarchical Simulink model is thus composed of blocks, subsystems, and lines between them. After a Simulink model is built, it is ready for simulation. Each step of simulation corresponds to one sample time of the overall diagram.

Stateflow [20] is a toolbox adding facilities for modeling and simulating reactive systems by means of hierarchical statecharts. They can be defined as Simulink blocks, fed with Simulink inputs and producing Simulink outputs. It extends Simulink's scope to event-driven and hybrid forms of embedded control. A Stateflow diagram is composed of transitions, states and junctions. Each transition connects a source state to a destination state. It supports construction of *flow charts* using connective junctions and transitions, which can be used between states to specify decision logics to form transition networks. The Stateflow states can be composed to form hierarchical diagrams: *Or diagram*, for which the states are mutually exclusive and only one state becomes active at a time, and *And diagram*, for which the states are parallel and all of them become active simultaneously.

2.3 HCSP

HCSP is an extension of Hoare's Communicating Sequential Processes (CSP) for hybrid systems [32, 15] . In HCSP, differential equations are introduced to model continuous evolution of the physical processes (in the physical environment) along with interrupts. A hybrid system in HCSP is a parallel composition of networked sequential processes interacting through dedicated channels, or a repetition of a sub-system. Processes in parallel can only interact through communication, and no shared variables are allowed.

Syntax The processes of HCSP are constructed as follows:

$$
\begin{aligned}
\mathsf{P} ::= {} & \mathsf{skip} \mid x := e \mid \mathsf{wait}(d) \mid ch?x \mid ch!e \mid B \to \mathsf{P} \mid \langle \dot{\boldsymbol{s}} = F(\boldsymbol{s}) \& B \rangle \\
& \mid [\!]_{i \in I}(ch_i \Box_i \longrightarrow \mathsf{P}_i) \mid \langle \dot{\boldsymbol{s}} = F(\boldsymbol{s}) \& B \rangle \trianglerighteq [\!]_{i \in I}(ch_i \Box_i \longrightarrow \mathsf{P}_i) \\
& \mid \mathsf{X} \mid \mu \mathsf{X}.\mathsf{P} \mid \mathsf{P} \fatsemi \mathsf{P}' \mid \mathsf{P} \sqcup \mathsf{P}' \\
\mathsf{S} ::= {} & \mathsf{P} \mid \mathsf{S} \| \mathsf{S}'
\end{aligned}
$$

where P, P', and P_i represent sequential processes, whereas S and S' stand for (sub)systems, ch and ch_i are communication channels, while $ch_i \Box_i$ is a communication event which can either be an input event $ch?x$ or an output event

$ch!e$, B and e are boolean and arithmetic expressions respectively, and d is a non-negative real constant.

Process skip terminates immediately without updating variables, and process $x := e$ assigns the value of expression e to variable x and then terminates. Process wait(d) keeps idle for d time units without any change to respective variables. Interaction between processes is based on two types of communication events: $ch!e$ sends the value of e along channel ch and $ch?x$ assigns the value received along channel ch to variable x. Communication takes place when both the source, and the destination processes are ready.

HCSP supports both sequential and concurrent composition. A sequentially composed process $\mathsf{P}\fatsemi\mathsf{P}'$ behaves as P first, and if it terminates, as P' afterwards. The process $B \to \mathsf{P}$ behaves as P only if B is true and terminates otherwise.

Internal choice between processes P and P', denoted as $\mathsf{P} \sqcup \mathsf{P}'$, is resolved by the process itself. Communication controlled external choice $[]_{i\in I}(ch_i\Box_i \longrightarrow \mathsf{P}_i)$ specifies that as soon as one of the communications $ch_i\Box_i$ takes place, the process starts behaving as respective process P_i.

Continuous evolution is specified as $\langle \dot{\boldsymbol{s}} = F(\boldsymbol{s})\&B\rangle$. Real variables $\boldsymbol{s}$ evolve continuously according to differential equations F as long as the boolean expression B is true. B defines the domain of $\boldsymbol{s}$ and we require B to be open. For example, $x > 0$ and $x \neq 0$ are open while $x \geq 0$ and $x = 0$ are not. In HCSP, continuous evolution can be preempted due to the following interrupts:

Boundary Violation Process $\langle \dot{\boldsymbol{s}} = F(\boldsymbol{s})\&B\rangle$ evolves until the boundary condition B becomes false.

Communication Process $\langle \dot{\boldsymbol{s}} = F(\boldsymbol{s})\&B\rangle \trianglerighteq []_{i\in I}(ch_i\Box_i \longrightarrow \mathsf{P}_i)$ behaves like $\langle \dot{\boldsymbol{s}} = F(\boldsymbol{s})\&B\rangle$, except that the continuous evolution is preempted whenever one of the communications $ch_i\Box_i$ takes place, followed by respective P_i.

The recursion $\mu\mathsf{X}.\mathsf{P}$ means that the execution of P can be repeated by replacing each occurrence of X with $\mu\mathsf{X}.\mathsf{P}$ itself during executing P, i.e., $\mu\mathsf{X}.\mathsf{P}$ behaves like $\mathsf{P}[\mu\mathsf{X}.\mathsf{P}]$. Finally, S defines an HCSP system on the top level. A parallel composition $\mathsf{S}\|\mathsf{S}'$ behaves as if S and S' run independently except that they need to synchronize along the common communication channels.

2.4 From AADL⊕S/S to HCSP

The translation of AADL⊕S/S models to HCSP is introduced in [27] and we give a brief overview here. The combined model is implemented in two ways: First, the physical environment of the system can be described by a continuous Simulink diagram; Second, the behavior of threads in AADL can be modeled by discrete Simulink/Stateflow diagrams. The translation procedures for Simulink and Stateflow are described in [29]. The connections between threads can be translated into buffer and bus processes for coordinating asynchronous communication between components. Processors are translated into scheduler processes managing the execution of threads according to the specified scheduling policies. In addition, devices can be modeled directly by HCSP processes for producing

simulated signals, or modeled as Simulink blocks like `signalBuilder` and then translated into HCSP processes. Finally, these separated HCSP processes can be integrated in parallel to form the whole model of the system.

2.5 The Mars Toolchain

We developed a toolchain, called Mars 2.0 [29], for modeling, analysis, verification, and code generation of CPSs. As shown in Fig. 3, Mars 2.0 (right) integrates Mars 1.0 [7] (left) with significant extensions and improvements, allowing the design of CPSs using the combination of AADL and Simulink/Stateflow (AADL⊕S/S). The toolchain automatically translates AADL⊕S/S models into HCSP; the translated HCSP can then be simulated using the HCSP simulator [27], and to complement incomplete simulation, it can be verified using the Hybrid Hoare Logic prover [33] in Isabelle/HOL, as well as the more automated HHLPy prover [25]. Finally, implementations in SystemC [28] or C [26] can be automatically generated from the verified HCSP processes. On the other side, the toolchain can translate AADL⊕S/S models directly into C code [30], but the correctness of the translation cannot be guaranteed.

The architecture of the previous version of this tool, i.e., Mars 1.0, 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. In summary, Mars 2.0 extends Mar 1.0 mainly from the following aspects:

- The co-modeling with AADL;
- The HCSP simulator and improved HHL provers;
- The code generation from HCSP models.

In what follows, we illustrate the whole procedure of Mars 2.0 by the case study of an automatically cruise control system introduced in the next section.

3 An Automatic Cruise Control System

In this section, we introduce an automatic cruise control system (ACCS for short) as the case study to illustrate the whole procedure of modelling, simulation, verification, and code generation by our toolchain Mars introduced in Section 2.5. This example is adapted from the self-driving car system in [10], where it is modelled only in AADL, and then extended in [27] by adding physical environment and control components modelled in Simulink/Stateflow.

The architecture of the ACCS decomposes to three levels, shown in Fig. 4. The physical layer contains the physical vehicle. The software level defines control of the system and 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 (the physical layer) through devices. The platform layer consists of a bus, a processor, and some devices. The connections

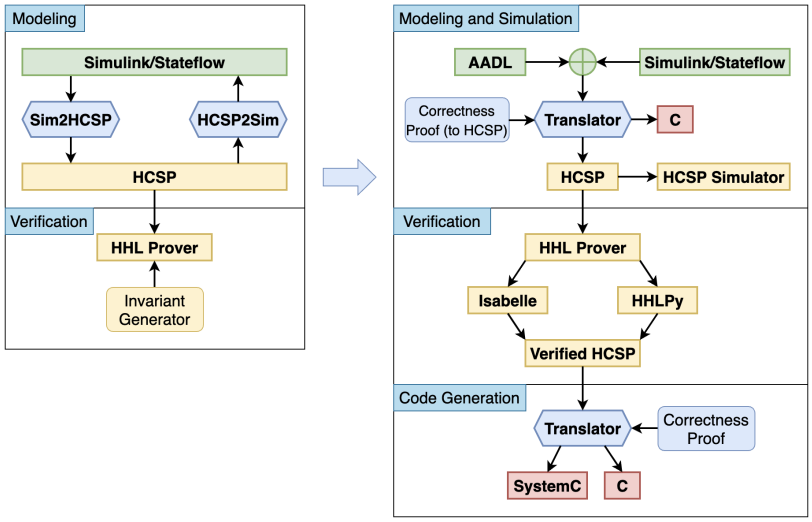

Fig. 3. Mars 1.0 (left, [7]) and 2.0 (right)

between processes and devices could be bound to the bus and all the threads are bound to the processor, with HPF (High-Priority-First) scheduling policy.

The execution of the 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.

3.1 Physical Level

The physical level is represented by an AADL *system component* which contains a Simulink diagram (shown in Fig. 4) describing how the vehicle moves. Specifically, it receives an ac(de)celeration a from the input port `veh_a`, based on which the velocity v and position s of the vehicle evolve. The evolution can easily be

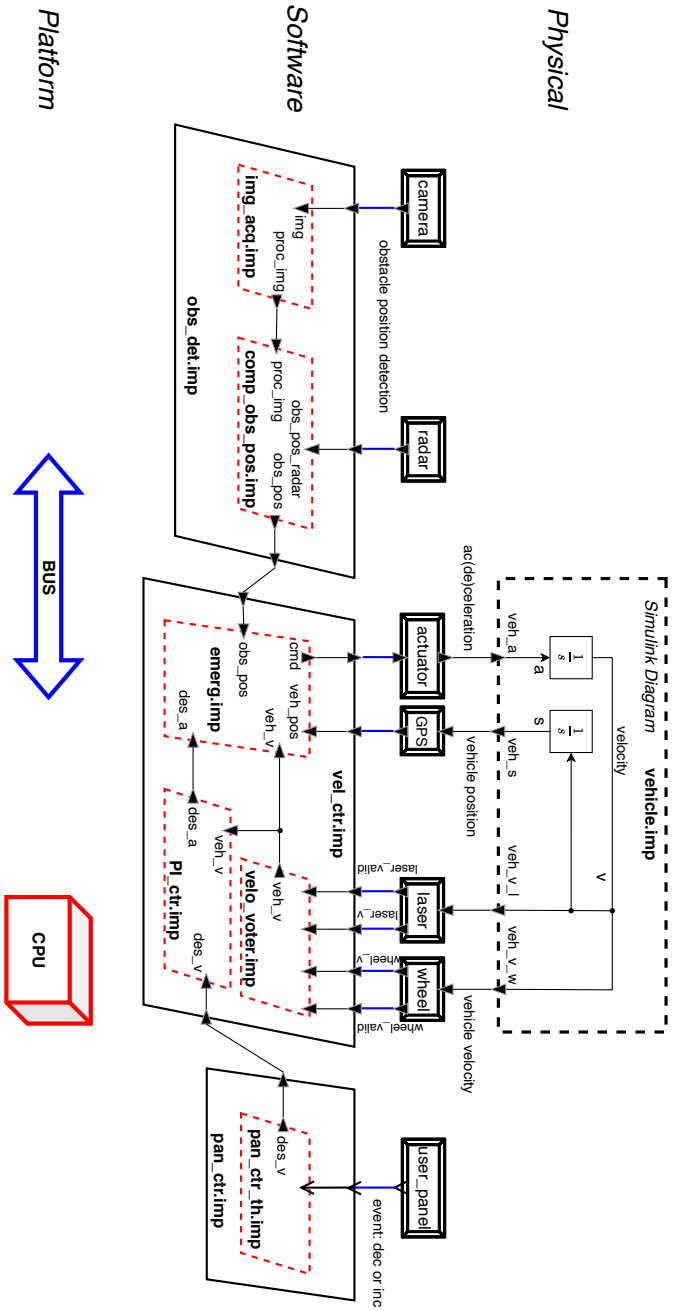

Fig. 4. Automatic Cruise Control System

described by a set of ODEs $\{\dot{s} = v, \dot{v} = a\}$. During the evolution, the vehicle can interact with its environment (composed of several devices) at any time: it can be actuated by the `actuator` and its position and velocity can be sensed by the `GPS` and two speedometers (`laser` and `wheel`), respectively.

3.2 Software Level

The software level is composed of three AADL processes: `obs_det` for detecting the obstacle in front of the vehicle, `vel_ctr` for controlling the vehicle, and `pan_ctr` for dealing with the commands from the user panel. These three processes are connected to make the evolution of the vehicle follow the driver's intent as much as possible, and in the meantime the vehicle should not collide into the obstacle ahead.

Obstacle Detection The obstacle detection process `obs_det` is composed of two threads: `img_acq` for dealing with the obstacle image from the `camera` and `comp_obs_pos` for computing the position of the obstacle according to the image from the `camera` and the signal from the `radar`. Concretely, the thread `img_acq` acquires from a camera raw images (`img`) of the road ahead and then sends the processed images (`proc_img`) to the thread `comp_obs_pos` which also receives obstacle information detected by a `radar`. The thread `comp_obs_pos` then outputs the final position of the obstacle (`obs_pos`). The image processing of `img_acq` may cause some delay, so its behavior is simply abstracted as a Simulink diagram containing only a `unit delay` (Fig. 5), because the detail of the image processing is not a concern in this case study. The behavior of `comp_obs_pos` is also described by a discrete Simulink diagram (Fig. 6) which combines the two inputs in a conservative way: it takes the minimal of the *valid* (≥ 0) obstacle positions detected as the final value and then sends it out.

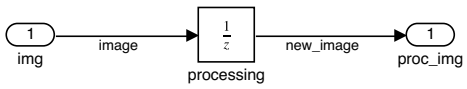

Fig. 5. The behavior of thread `img_acq`

Velocity Control The process `vel_ctr` for velocity control consists of three threads. The thread `vel_voter` is a voter procedure combining velocity information received from the `wheel` and `laser` of the vehicle. Concretely, if the `wheel` is valid ($\texttt{wheel\_valid} > 0$) but the `laser` is not ($\texttt{laser\_valid} \leq 0$), then the velocity information from the `wheel` will be sent out; if, symmetrically, the `laser` is valid but the `wheel` is not, i.e., $\texttt{laser\_valid} > 0 \wedge \texttt{wheel\_valid} \leq 0$, then the velocity sensed by the `laser` will be taken; otherwise, i.e., both are valid

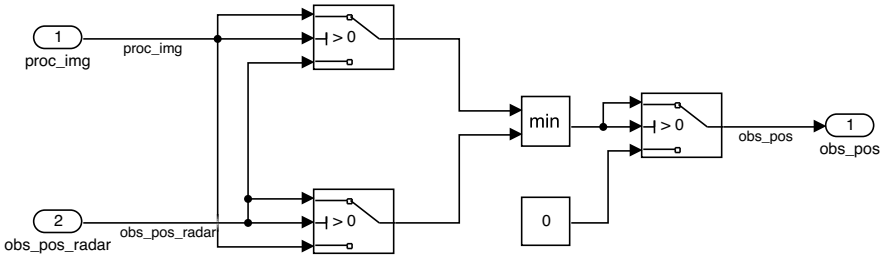

Fig. 6. The behavior of thread `comp_obs_pos`

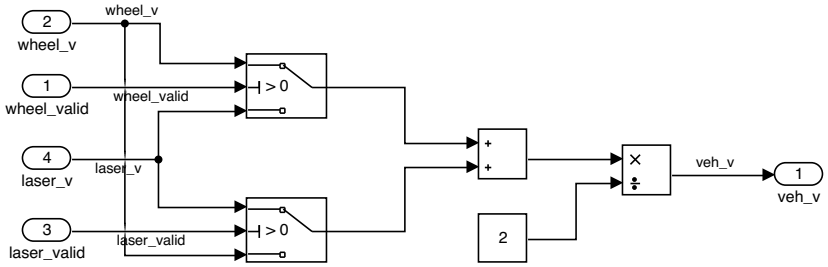

Fig. 7. The behavior of thread `velo_voter`

or invalid, the mean of the sensed velocity values will be regarded as the final velocity of the vehicle. The Simulink diagram in Fig. 7 models this behavior.

The thread `PI_ctr` receives the vehicle speed (`veh_v`) produced by `vel_voter` and a desired speed (`des_v`) from the user panel and then computes a desired acceleration (`des_a`). Concretely, it computes the difference between the desired and the real velocities of the vehicle and then sends the difference to discrete PI controller with a wind-up method (back-calculation) to calculate a desired acceleration. The Simulink diagram in Fig. 8 models this behavior.

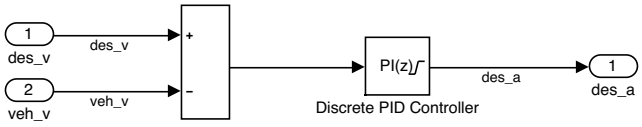

Fig. 8. The behavior of thread `PI_ctr`

The thread `emerg` receives obstacle position (`obs_pos`) from `obs_det`, vehicle position (`veh_pos`) from GPS, vehicle speed (`veh_v`) from `vel_voter` and the desired acceleration (`des_a`) from `PI_ctr`, and computes a command (`veh_a`) to the actuator based on all these inputs. It checks whether the acceleration output by `PI_ctr` is safe with respect to obstacle position. If so this is allowed as the final command. Otherwise, it overrides the command with a safe deceleration.

Concretely, the control of `emerg` is based on the Maximum Protection Curve [27] computed as follows:

$$V_{lim}(s) = \begin{cases} v_{max} & \text{if } s_{obs} - s \geq \frac{v_{max}^2}{(-2a_{min})} \\ \sqrt{-2a_{min} \cdot (s_{obs} - s)} & \text{if } 0 < s_{obs} - s < \frac{v_{max}^2}{(-2a_{min})} \\ 0 & \text{otherwise} \end{cases}$$

where s and s_{obs} are the respective current positions of the vehicle and the obstacle, v_{max} is the maximum velocity that the vehicle can reach and $a_{min} < 0$ is the braking deceleration of the vehicle. If the obstacle is out of the safe distance $(-v_{max}^2/2a_{min})$ of the vehicle, the upper limit velocity of the vehicle can be the maximum v_{max}; if not, the velocity should not exceed $\sqrt{-2a_{min} \cdot (s_{obs} - s)}$ in order to avoid the collision (provided $s_{obs} - s > 0$); otherwise, if $s_{obs} - s \leq 0$, then a collision has already happened, and the vehicle should stop ($V_{lim} = 0$).

At each iteration, `emerg` predicts the position s_{next} and velocity v_{next} of the vehicle at the next period based on the desired acceleration (a_{des}) provided by `PI_ctr` (see Fig. 4). Concretely, they can be computed by

$$\begin{aligned} v_{next} &= v + a_{des} \cdot \mathit{period} \\ s_{next} &= s + v \cdot \mathit{period} + \tfrac{1}{2} \cdot a_{des} \cdot \mathit{period}^2 \end{aligned}$$

where *period* is the period of the thread `emerg`.

If, at the next period, the velocity does not exceed the upper limit computed as above, i.e., $v_{next} \leq V_{lim}(s_{next})$, then the desired acceleration a_{des} is safe; if not, it continues to test if the constant velocity (no acceleration or deceleration) is safe ($v \leq V_{lim}(s + v \cdot \mathit{period})$); otherwise, the emergency alerts and the thread `emerg` outputs the minimal deceleration ($a_{min} < 0$) to brake the vehicle. The above control strategy can be summarized as

$$a(s, v) = \begin{cases} a_{des} & \text{if } v_{next} \leq V_{lim}(s_{next}) \\ 0 & \text{if } v \leq V_{lim}(s + v \cdot \mathit{period}) \\ a_{min} & \text{otherwise} \end{cases}$$

and it is modeled by the Stateflow diagram shown in Fig. 9.

Panel Control The process `pan_ctr` includes only one thread `pan_ctr_th`. It receives events from device `user_panel`. The driver can control `user_panel` by triggering events `inc` and `dec` to increase and decrease the desired speed, respectively. The behavior of the thread `pan_ctr_th` is modeled by a Stateflow diagram shown in Fig. 10.

3.3 Platform Level

The hardware platform is composed of a bus, a processor, and some devices such as `camera`, `radar`, and `actuator`, connecting the physical and software levels. The behavior of each device is described directly by HCSP using *HCSP*

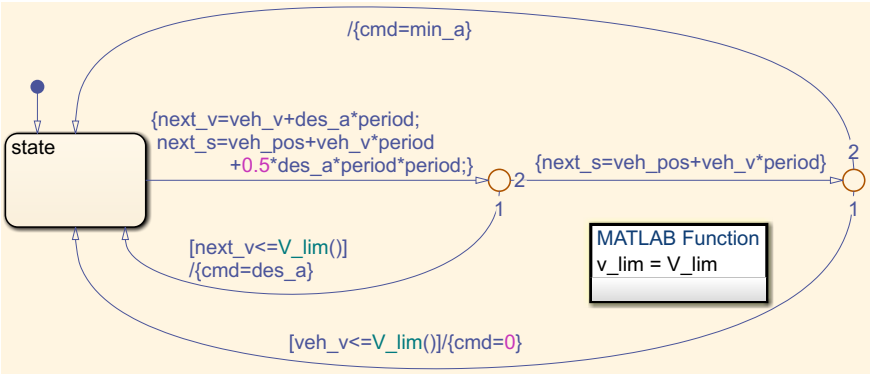

Fig. 9. The behavior of thread `emerg`

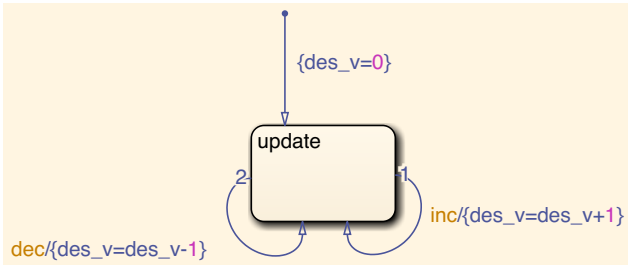

Fig. 10. The behavior of thread `pan_ctr_th`

Annex. In the ACCS architecture (Fig. 4), the connections between devices and processes are all bound to a bus, and all the threads in the processes are bound to a processor adopting HPF (High-Priority-First) scheduling policy. Each bus has the property of latency denoting the transfer delay, thus we can consider different settings of number of buses and their latency to observe the impact on the system performance caused by bus.

3.4 Restrictions on AADL

We only take part of AADL components including processes, threads, processors, buses, devices, abstract components (used to model physical environment), and the connections between components, into account. Other components have not been considered, for instance, memories and processors with more scheduling policies. The reasons for the restrictions include:

1. We concern more on the abstract and formal model of AADL. The design details of AADL will lead to complex and redundant formal models, which prevents us from verifying the key properties (like safety and latency) of AADL models;

2. The Simulink/Stateflow diagrams are used to describe the hybrid and physical behaviors of components, i.e., we focus more on behavioral components like threads. The components regarding resource (for instance, memories) are not the concern. In our future work, we will consider the effect of resource constraints on AADL models.

4 Simulation and Verification

The ACCS introduced in Section 3 is an (informal) graphical combined model of AADL and Simulink/Stateflow (AADL $\oplus$ S/S). In order to analyze it in a formal manner, we translate it to a formal HCSP model. The translation from AADL $\oplus$ S/S to HCSP has been presented in [27] and integrated into our Mars toolchain introduced in Section 2.5. The correctness of the translation has also been proved [27]. Given the translated HCSP model of the ACCS, we can perform simulation and verification using the toolchain Mars.

4.1 Simulation

We set up a scenario where there is a mobile obstacle in front of the vehicle and where the driver also sets a desired speed for the vehicle. 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. In this scenario, we assume the `camera` fails to work and thus only `radar` can detect the obstacle. Concretely, the behavior of the failed `camera` can be simply described by the HCSP program: $\mu X.\texttt{out}!-1;\mathsf{wait}(0.2);X$, i.e., it outputs -1 (invalid) through its output port every 0.2s (the period of the `camera`), and the behavior of the `radar` can be modeled by $\mu X.x := 0;(10 \le t < 20 \to x := 2t+15);\texttt{out}!x;\mathsf{wait}(0.01);X$, where t denotes the local time of the `radar` and is set 0 initially, and x represents the detected position of the obstacle.

On the vehicle side, we assume that at the beginning the vehicle is at rest at the initial position 0m and the driver pushes the `inc` button (to increase the speed) three times with time interval 0.5s in between to set a desired speed to 3m/s. After 29s, the driver pushes the `dec` button (to decrease the speed) twice in 0.5s time intervals to decrease the desired speed. In summary, the behavior of the driver (`user_panel`) can be modeled by the following HCSP program:

$$\begin{aligned}&\texttt{out}!\texttt{"inc"};\mathsf{wait}(0.5);\texttt{out}!\texttt{"inc"};\mathsf{wait}(0.5)\texttt{out}!\texttt{"inc"};\\&\mathsf{wait}(29);\\&\texttt{out}!\texttt{"dec"};\mathsf{wait}(0.5);\texttt{out}!\texttt{"dec"}\end{aligned}$$

Other devices (`actuator`, `GPS`, `laser`, and `wheel`) are treated as "routers" which transfer periodically the received messages at the input ports directly through the output ports. For example, the behavior of the `GPS` can be described as $\mu X.\texttt{in}?x;\texttt{out}!x;\mathsf{wait}(0.006);X$, where x denotes the sensed vehicle position and the period of the `GPS` is 0.006s.

Impact of Bus We first set the latency of the bus in ACCS (Fig. 4) to 3ms, and the simulation results are shown in Fig. 11, from which we can see that the vehicle nearly hits the moving obstacle ahead. The reason for this dangerous situation is the competition for bus permission. The competition is so intense that the `radar` can hardly transfer the sensed obstacle position to the process `obs_det` in time. Actually, the delay of the transferring is up to 5s in this case, which is absolutely intolerable in the real world applications.

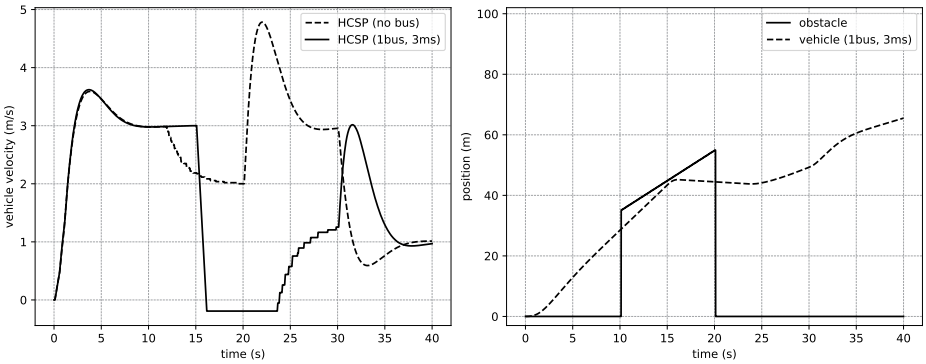

Fig. 11. One bus with the latency 3ms

The above can be seen as a design error: the allocation of bus capacity is insufficient for the given latency. To correct this problem, we set an extra bus with the same latency (3ms) for the `radar`. The connection between the device `radar` and the process `obs_det` is bound to this dedicated bus. The simulation result of the vehicle velocity in this case is shown in Fig. 12 (dashed line), which is similar to the case not involving buses (solid line). The minor gap between them is due to the latency of the buses.

Based on the setting of two buses, we further increase the bus latency to 5ms to test the performance of the system. The result is that the vehicle never starts. By examining the logs of simulation, we can find that the thread `emerg` cannot obtain bus permission in order to transfer the acceleration command to actuator, causing the vehicle keeping motionless. The reason is the lack of throughput of the bus. To resolve it, we hence add another bus to the architecture and bind the connection between the thread `emerg` and the `actuator` to this bus, and the simulation result returns to normal according to Fig. 12 (dotted line).

4.2 Verification

One of the motivations of translating the AADL⊕S/S model of ACCS to HCSP is to verify the informal graphical model of ACCS. In this case study, we verify the safety property of the control strategy of the thread `emerg` (**Velocity Control** in Section 3.2) using trace-based Hybrid Hoare Logic.

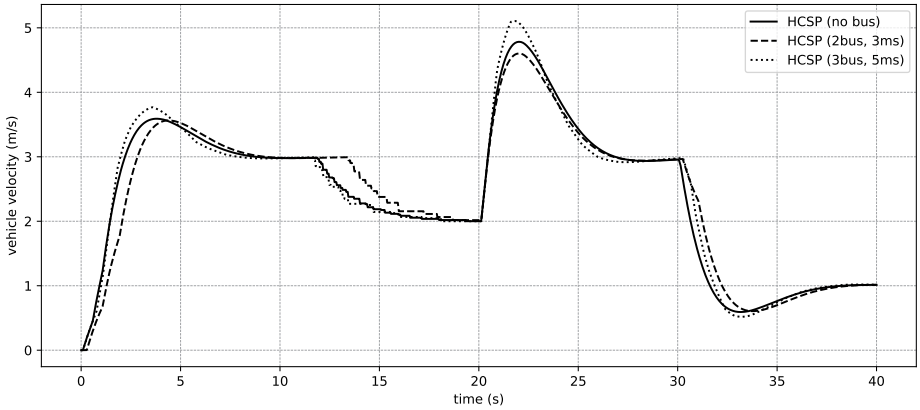

Fig. 12. Vehicle velocity under different bus settings

First, we use automated HHL prover: HHLPar to generate an assertion R on trace and state satisfying the following lemma.

Theorem 1.

$$\forall \bar{s_0}.\ p(\bar{s_0}) \longrightarrow \models \{\bar{s} = \bar{s_0} \wedge tr = \epsilon\}\, \texttt{Plant} \| \texttt{Control}\, \{R(\bar{s_0})\}$$

where p represents the initial condition on the starting state $\bar{s_0}$ and R maps the the starting state $\bar{s_0}$ to the assertion on the state $\bar{s}$ and trace tr after the termination of the process describing the whole behaviour this parallel system.

The process `Plant` and `Control` are defined in following:

$$\begin{aligned}
\texttt{Plant} \triangleq\ & \texttt{ch1!}v; \texttt{ch2!}s; (\texttt{ch3?}a; \langle \dot{s} = v, \dot{v} = a \,\&\, \texttt{true}\rangle \trianglerighteq [][\texttt{ch1!}v \rightarrow \texttt{ch2!}s])^* \\
\texttt{Control} \triangleq\ & \texttt{ch1?}v; \texttt{ch2?}s; (s_{next} := s + v \cdot period + \tfrac{1}{2} \cdot a_{des} \cdot period^2; \\
& v_{next} := v + a_{des} \cdot period; \\
& (\texttt{if}\ -2 \cdot a_{min} \cdot (s_{obs} - s_{next}) \geq v_{max}^2\ \texttt{then}\ V_{lim} := v_{max}\ \texttt{else} \\
& \texttt{if}\ s_{obs} - s_{next} > 0\ \texttt{then}\ V_{lim} := \sqrt{-2 \cdot a_{min} \cdot (s_{obs} - s_{next})} \\
& \texttt{else}\ V_{lim} := 0); \\
& (\texttt{if}\ v_{next} \leq V_{lim}\ \texttt{then}\ a := a_{des}\ \texttt{else} \\
& s_{next} := s + v \cdot T; \\
& (\texttt{if}\ -2 \cdot a_{min} \cdot (s_{obs} - s_{next}) \geq v_{max}^2\ \texttt{then}\ V_{lim} := v_{max}\ \texttt{else} \\
& \texttt{if}\ s_{obs} - s_{next} > 0\ \texttt{then}\ V_{lim} := -2 \cdot a_{min} \cdot (s_{obs} - s_{next}) \\
& \texttt{else}\ V_{lim} := 0); \\
& \texttt{if}\ v \leq V_{lim}\ \texttt{then}\ a := 0\ \texttt{else}\ a := a_{min}); \\
& ch3!a; \texttt{wait}\ period; ch1?v; ch2?s)^*
\end{aligned}$$

The generating procedure of R to satisfy the theorem through HHLPar includes the following three steps:

Lemma 1.

$$\models \{\bar{s} = \bar{s_1} \wedge tr = \epsilon\}\, \texttt{Plant}\, \{R_1(\bar{s_1})\}$$

Lemma 2.

$$\models \{\bar{s} = \bar{s_2} \wedge tr = \epsilon\}\, \texttt{Control}\, \{R_2(\bar{s_2})\}$$

Based on the structure of a single process we automatically generate its corresponding assertions using the specification rules. For example:

$$\frac{\mathsf{paramODEsol}(\overrightarrow{x} = \overrightarrow{e}, B, \overrightarrow{p}, e) \quad \mathsf{lipschitz}(\overrightarrow{x} = \overrightarrow{e}) \quad \mathsf{spec\_of}(c, Q)}{\mathsf{spec\_of}(\langle \overrightarrow{x} = \overrightarrow{e} \,\&\, B\rangle; c, \mathsf{wait}(s = s0[\overrightarrow{x} \mapsto \overrightarrow{p}\,(s_0, t)], e, \{d \Rightarrow Q[\overrightarrow{x} := \overrightarrow{p}\,(s_0, d)]\}))}$$

Then we apply the synchronization rules between R_1 and R_2 to compute the R on parallel process.

Lemma 3.

$$\forall\ \bar{s_1}\ \bar{s_2}\ ,\ p(\bar{s_1} \uplus \bar{s_2}) \longrightarrow R_1(\bar{s_1}) \| R_2(\bar{s_2}) \Longrightarrow R(\bar{s_1} \uplus \bar{s_2})$$

The assertion R computed by HHLPar contains state changes and generated trajectories of the process and is strong enough to inference the safety property we need which can be concluded as $\mathtt{Safe} \triangleq s \leq s_{obs} \wedge v \leq V_{lim}(s)$, and we have

Theorem 2.

$$R(\bar{s_0}, \bar{s}, tr) \longrightarrow \mathtt{Safe}(\bar{s})$$

which can be proved by Isabelle/HOL. Combining the two theorems, we can obtain complete verification of the security property of this system.

5 Code Generation

In our previous work [30], combined models of AADL $\oplus$ S/S are translated to C code directly. The translation is done by the following steps: (1) the AADL part is translated to C following the execution semantics of AADL; (2) the Simulink/Stateflow part is translated using the existing C code generation facility in Matlab; (3) the architecture part is implemented by a library in C that includes thread scheduling protocols and so on; and (4) the combined model is translated by integrating the C codes generated from the above three parts. However, the interaction (communication) between components is implemented by shared variables (without pthreads) and the correctness of the translation has not been proved.

In our latest work [26], we generate the C code of AADL $\oplus$ S/S models from their HCSP models. Since HCSP are verifiable and the correctness of the translation can be guaranteed, the generated code, especially the code for the controller, satisfies the safety requirement and therefore is reliable. 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 generated C code is of 3500–4000 lines. 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.

In Fig. 13, we compare our results with the simulation of the HCSP model of the ACCS and the execution of the C code generated directly from the AADL $\oplus$ S/S model of the ACCS ([30]). The left of Fig. 13 shows the execution results of the vehicle speed, where the black line denotes the desired velocity set by the driver. We can see that the execution result of the generated C code is almost the same as the simulation of its HCSP model. Specifically, the average relative error (ARE) between the time series

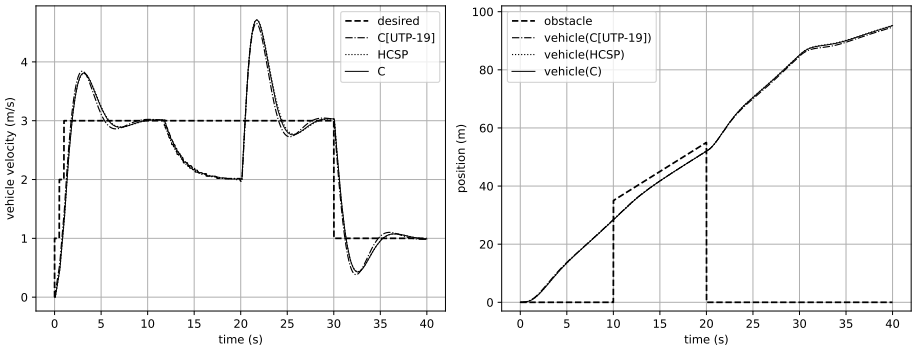

Fig. 13. Comparison of execution results where UTP-19 denotes the work of [30]

of the velocity generated from the HCSP model and its C code is 0.138% with the variance 4.686×10^{-5}. Besides, we can also observe that there is a negligible difference between the results of the C code generated by [30] and the latest C code generated: the ARE is 0.182% with the variance 4.232×10^{-3}.

From both 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. 13.

6 Related Work

There exist a huge amount of work on the modeling, analysis, verification and code generation of CPSs. Several unified frameworks have been proposed for designing CPSs. The Metropolis design framework [2, 9] 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 [22] 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) [18] 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 [8], 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 [24] 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 [6] extends the synchronous language Lustre [13] with ODEs and zero-crossing

events for designing and implementing hybrid systems and has also been implemented in SCADE 6. It supports analysis of hybrid models by type systems and semantics, and handles the detection of zero-crossing events [6, 3]. Differential dynamic logic [21] is developed for reasoning about behaviors of hybrid dynamic models, and based on which the KeYmaera X prover [12] is implemented for safety analysis of dynamic systems. VeriPhy [5] automatically transforms verified formal models of CPSs modelled in differential dynamic logic [21, 12] to controller implementations that preserve safety properties of original models.

7 Conclusion

In this paper, we introduced a realistically-scaled automatic cruise control system as the case study to illustrate the whole procedure of modeling, simulation, verification, and code generation of the MBD for CPS design. We can learn from the results in Section 4 that the impact on system behavior and perform can be foreseen and the defect of system design can be checked by analyzing the model of system at the very early stage, reducing the development cost significantly, reflecting the original intention of the MBD methodology.

This case study demonstrates the practicability of the MBD tool, Mars, in dealing with complex CPSs. The toochain Mars was first proposed in [7] with limited capability, and later it is significantly improved based on our recent works including [27, 25, 31], and so on. Given the case study described in this paper, we will summarize the toolchain Mars into a tool paper and make it public to the MBD and CPS communities.

However, there are some limitations to our approach. First, AADL provides plenty of components and functions, while we only consider its core functionalities, which limits the practicality of our framework for more case-studying realistic CPSs. Second, at present, our verifier only scales to small HCSP models and some sequential models (no communication is involved). Thus, as future work, we will try to integrate more functionalities into our approach, such as considering more AADL components and Simulink/Stateflow blocks, modeling and verification of hybrid systems containing delay or stochastic differential equations. Correspondingly, we will improve verification efficiency and scalability by extending Jones's *rely/guarantee* [16] to CPS such that our approach can be applied to real-world case studies on a larger scale.

Acknowledgment

This work is partly funded by the NSFC under grant No. 62192732, the National Key R&D Program of China under grants No. 2022YFA1005101 and 2022YFA1005103, the NSFC under grant No. 62032024, and the CAS Project for Young Scientists in Basic Research under grant No. YSBR-040.

References

1. SysML 1.6 Beta Specification (2019), http://www.omg.org/spec/SysML
2. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: An integrated electronic system design environment. Computer **36**(4), 45–52 (2003). https://doi.org/10.1109/MC.2003.1193228

3. Benveniste, A., Bourke, T., Caillaud, B., Pouzet, M.: Non-standard semantics of hybrid systems modelers. J. Comput. Syst. Sci. **78**(3), 877–910 (2012). https://doi.org/10.1016/J.JCSS.2011.08.009
4. Besnard, D., Jones, C.: Designing dependable systems needs interdisciplinarity. Safety Critical System's Club Newsletter **13**(3), 6–9 (2004), https://hal.science/hal-00724103
5. Bohrer, R., Tan, Y.K., Mitsch, S., Myreen, M.O., Platzer, A.: Veriphy: verified controller executables from verified cyber-physical system models. In: PLDI 2018. pp. 617–630. ACM (2018). https://doi.org/10.1145/3192366.3192406
6. Bourke, T., Pouzet, M.: Zélus: a synchronous language with ODEs. In: HSCC 2013. pp. 113–118. ACM (2013). https://doi.org/10.1145/2461328.2461348
7. Chen, M., Han, X., Tang, T., Wang, S., Yang, M., Zhan, N., Zhao, H., Zou, L.: MARS: A toolchain for modelling, analysis and verification of hybrid systems. In: Provably Correct Systems, pp. 39–58. NASA Monographs in Systems and Software Engineering, Springer (2017). https://doi.org/10.1007/978-3-319-48628-4_3
8. Cremona, F., Lohstroh, M., Broman, D., Lee, E.A., Masin, M., Tripakis, S.: Hybrid co-simulation: It's about time. Softw. Syst. Model. **18**(3), 1655–1679 (2019). https://doi.org/10.1007/S10270-017-0633-6
9. Davare, A., Densmore, D., Meyerowitz, T., Pinto, A., Sangiovanni-Vincentelli, A., Yang, G., Zeng, H., Zhu, Q.: A next-generation design framework for platform-based design. In: DVCon 2007. Citeseer (February 2007)
10. Delange, J.: AADL in Practice. Reblochon Development Company (2017)
11. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley Professional (2012)
12. Fulton, N., Mitsch, S., Quesel, J., Völp, M., Platzer, A.: Keymaera X: an axiomatic tactical theorem prover for hybrid systems. In: CADE 2015. LNCS, vol. 9195, pp. 527–538. Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_36
13. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE **79**(9), 1305–1320 (1991). https://doi.org/10.1109/5.97300
14. Hayes, I.J., Jackson, M.A., Jones, C.B.: Determining the specification of a control system from that of its environment. In: FME 2003. LNCS, vol. 2805, pp. 154–169. Springer (2003). https://doi.org/10.1007/978-3-540-45236-2_10
15. He, J.: From CSP to hybrid systems, p. 171–189. Prentice Hall International (UK) Ltd., GBR (1994)
16. Jones, C.B.: Specification and verification. IEEE Transactions on Software Engineering **SE-10**(2), 126–127 (1984). https://doi.org/10.1109/TSE.1984.5010214
17. Jones, C.B.: Dependability of computer-based systems. In: Sampaio, A. (ed.) SBSE 2000. pp. 16–20. SBC (2000). https://doi.org/10.5753/SBES.2000.25917
18. Junghanns, A., Gomes, C., Schulze, C., Schuch, K., R., P., Blaesken, M., Zacharias, I., Pillekeit, A., Wernersson, K., Sommer, T., Bertsch, C., Blochwitz, T., Najafi, M.: The functional mock-up interface 3.0 - new features enabling new applications. In: Proceedings of 14th Modelica Conference 2021 (2021)
19. MathWorks Inc.: Simulink User's Guide (2013), `http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf`
20. MathWorks Inc.: Stateflow User's Guide (2013), `http://www.mathworks.com/help/pdf_doc/stateflow/sf_ug.pdf`
21. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer (2018). https://doi.org/10.1007/978-3-319-63588-0

22. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation Using Ptolemy II. Ptolemy.org (2014), http://ptolemy.org/books/Systems
23. SAE International Standards: Architecture analysis & design language (AADL), Revision C (2017)
24. Selic, B., Gerard, S.: Modeling and Analysis for Real-time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems. The MK/OMG Press (2013)
25. Sheng, H., Bentkamp, A., Zhan, B.: HHLPy: Practical verification of hybrid systems using Hoare logic. In: FM 2023. LNCS, vol. 14000, pp. 160–178. Springer (2023). https://doi.org/10.1007/978-3-031-27481-7_11
26. Wang, S., Ji, Z., Xu, X., Zhan, B., Gao, Q., Zhan, N.: Formally verified C code generation from hybrid communicating sequential processes. In: ICCPS 2024. pp. 123–134. IEEE (2024), https://doi.org/10.1109/ICCPS61052.2024.00018
27. Xu, X., Wang, S., Zhan, B., Jin, X., Talpin, J., Zhan, N.: Unified graphical co-modeling, analysis and verification of cyber-physical systems by combining AADL and Simulink/Stateflow. Theor. Comput. Sci. **903**, 1–25 (2022). https://doi.org/10.1016/J.TCS.2021.11.008
28. Yan, G., Jiao, L., Wang, S., Wang, L., Zhan, N.: Automatically generating SystemC code from HCSP formal models. ACM Trans. Softw. Eng. Methodol. **29**(1), 4:1–4:39 (2020). https://doi.org/10.1145/3360002
29. Zhan, B., Xu, X., Gao, Q., Ji, Z., Jin, X., Wang, S., Zhan, N.: Mars 2.0: A toolchain for modeling, analysis, verification and code generation of cyber-physical systems. arXiv **abs/2403.03035** (2024)
30. Zhan, H., Lin, Q., Wang, S., Talpin, J.P., Xu, X., Zhan, N.: Unified graphical co-modelling of cyber-physical systems using AADL and Simulink/Stateflow. In: UTP. LNCS, vol. 11885, pp. 109–129 (2019). https://doi.org/10.1007/978-3-030-31038-7_6
31. Zhan, N., Zhan, B., Wang, S., Guelev, D.P., Jin, X.: A generalized hybrid Hoare logic. CoRR **abs/2303.15020** (2023)
32. Zhou, C., Wang, J., Ravn, A.P.: A formal description of hybrid systems. In: Hybrid Systems III: Verification and Control. LNCS, vol. 1066, pp. 511–530. Springer (1995). https://doi.org/10.1007/BFB0020972
33. Zou, L., Zhan, N., Wang, S., Fränzle, M., Qin, S.: Verifying Simulink diagrams via a hybrid Hoare logic prover. In: EMSOFT. pp. 1–9. IEEE (2013). https://doi.org/10.1109/EMSOFT.2013.6658587