Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Unified graphical co-modeling, analysis and verification of cyber-physical systems by combining AADL and Simulink/Stateflow

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

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

A R T I C L E I N F O

Article history:
Received 19 July 2021
Received in revised form 8 October 2021
Accepted 16 November 2021
Available online xxxx

Keywords:
Simulink/Stateflow
AADL
HCSP
Formal semantics
Simulation and verification

A B S T R A C T

The efficient design of safety-critical cyber-physical systems (CPS) requires the co-modeling and -verification of its physics, architecture and functionalities. Existing co-modeling formalisms do not account for these three design aspects into account uniformly. AADL is a precise formalism for modeling architecture and prototyping real-time hardware platforms, but it delegates co-modeling physical and software behaviors to so-called annexes. By contrast, Simulink/Stateflow (S/S) is strong for modeling interacting physical and software behaviors, but weak for modeling architecture and hardware platforms. To address this issue, this paper considers the combination of AADL and S/S to co-model CPSs and presents a method to uniformly analyze and verify this combination. AADL⊕S/S provides a unified graphical co-modeling environment for CPS design and supports simulation through C code generation. We present a formal semantics of AADL⊕S/S by translating it to Hybrid Communicating Sequential Processes (HCSP), which yields a deductive verification framework for AADL⊕S/S models based on Hybrid Hoare Logic (HHL). We also prove the correctness of the translation of AADL⊕S/S to HCSP. The effectiveness of our approach is illustrated by the realistically-scaled case study of an automatic cruise control system.

1. Introduction

Cyber-physical systems (CPS) tightly couple hardware and software to sense and actuate a physical environment. To correctly model them, it is paramount to take the three perspectives of its software functionality, physical environment and hardware platform and system architecture into account, uniformly. Unfortunately, according to the commonly accepted design paradigm of "separation of concerns", most of existing design methodologies and workflows do not support all three design aspects uniformly. For example, the Architecture Analysis & Design Language (AADL) [1] features strong capabilities for describing the architecture of a system due to the pragmatic (and practice-inspired) 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

* Corresponding author at: SKLCS, Institute of Software, Chinese Academy of Sciences, Beijing, China.
E-mail address: znj@ios.ac.cn (N. Zhan).

https://doi.org/10.1016/j.tcs.2021.11.008

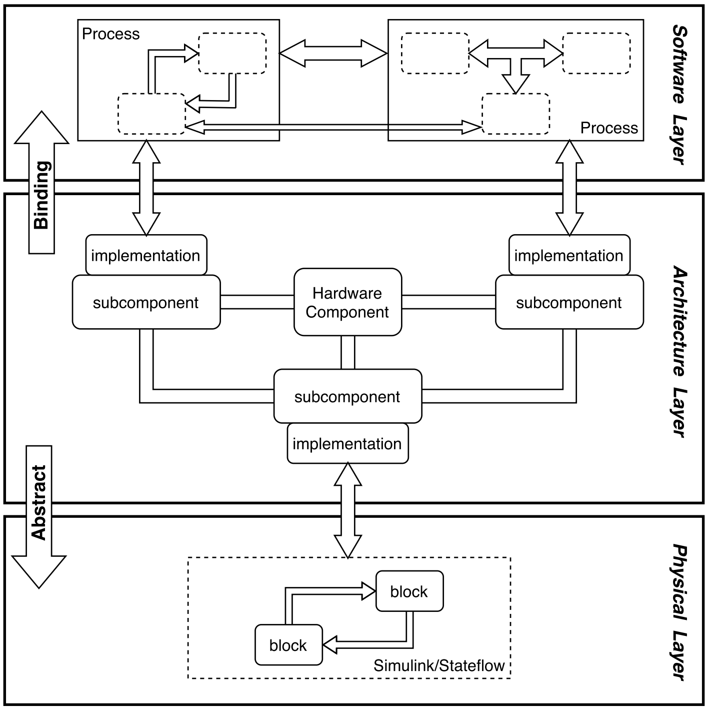


Fig. 1. An overview of AADL⊕S/S (from [4]).

be controlled and its combination with software. By contrast, Simulink/Stateflow (S/S) [2,3], developed by Mathworks, is the de-facto industry standard for model-based analysis and design of embedded systems. It is best-suited for modeling and analyzing continuous physical processes, discrete computations and their combination. However, S/S cannot naturally model system architectures and hardware platforms.

To address the above issue, we first present a combination of AADL and S/S, named AADL⊕S/S, that provides a unified graphical modeling formalism to represent all three perspectives of CPS design. An overview of AADL⊕S/S is given in Fig. 1. Using AADL⊕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.

In order to simulate AADL⊕S/S models, we also present a way to translate AADL⊕S/S to C code, allowing co-simulation of the combined models. It relies on the Real Time Workshop (RTW) toolbox of Matlab, which permits code generation from S/S diagrams. The translation of the combined model amounts to coordinating code generated from AADL and S/S through port communications specified in the architecture layer. The result of the simulation is then displayed visually for analysis.

However, guaranteeing the reliability of a safety-critical CPS developed using AADL ⊕ S/S remains challenging, as simulation-based techniques are inherently incomplete, and therefore cannot ensure reliability of safety-critical CPS rigorously. To address this problem, we further develop an HCSP-based deductive verification approach for AADL⊕S/S, including

- First, a formal semantics of AADL in terms of transition systems, including thread dispatch, scheduling, execution, and bus connections with latency.
- Second, a translation from graphical AADL⊕S/S models to Hybrid Communicating Sequential Processes (HCSP) [5,6]. Compared with other formalisms such as hybrid automata [7] and hybrid programs [8], HCSP provides a compositional way to model complex CPSs due to its rich set of algebraic operators. The correctness of the translation is proved by building a weak bisimulation relation between the transition semantics of the source and target models. The translated HCSP model can be formally verified using HHL [9–11] and the results are preserved by the original AADL⊕S/S model.
- Additionally, we develop a simulator for HCSP, so that the translated HCSP model can also be simulated after translation, and the correctness of the translation can be tested by comparing the simulation results before and after translation. Furthermore, one can even design a CPS model starting from HCSP, as it also supports simulation and verification.

In summary, the main contributions of this paper comprise:

1. A combination of AADL and S/S;
2. A simulation tool for AADL⊕S/S;
3. An HCSP-based analysis and verification approach for AADL⊕S/S;
4. An application of the above workflow to modeling and analyzing the realistically-scaled case study of an automatic cruise control system.

These contributions deliver a framework for co-designing complex safety-critical CPS using AADL⊕S/S, simulating the resulting model and revising it until the desired and formally specified properties are satisfied. Afterwards, the whole model, or at least its safety-critical parts, can be formally verified using HHL. Finally, correct SystemC code can be automatically generated from the HCSP models based on [12]. Some preliminary results of contributions 1 and 2 were reported in [4]. Compared with our previous work [4], the novelty of this paper concerns the following aspects:

1. A formal semantics for AADL (as well as its combination with S/S);
2. The translation from AADL⊕S/S to HCSP and its proof of correctness (in particular, a translation of buses in AADL);
3. A combination framework where threads are modeled by discrete Simulink diagrams (and a reformulation of threads translation).
4. Devices in AADL are modeled by Simulink or HCSP to support the simulation of an entire CPS;
5. An HCSP simulator is developed to support the simulation of AADL⊕S/S;
6. We demonstrate how to verify the translated HCSP model using HHL in Isabelle/HOL.

Paper organization. Sect. 2 provides an overview of AADL, S/S, and HCSP. Sect. 3 depicts the combined framework composed of AADL and S/S. Sect. 4 describes the simulation of AADL⊕S/S models by translation to C. Sect. 5 defines a formal semantics for AADL and AADL⊕S/S using timed transition systems. Sect. 6 presents the translation of AADL and AADL⊕S/S to HCSP. The correctness of translation is proved in Sect. 7. A simulation tool for HCSP is introduced in Sect. 8. A case study of a fully-functional automatic cruise control system is presented in Sect. 9. Finally, we review related work in Sect. 10 and conclude in Sect. 11.

2. Preliminaries

In this section, we first provide an overview of the AADL standard and of Simulink/Stateflow (S/S). Then, we briefly introduce HCSP, a formal specification language for modeling embedded and hybrid systems.

2.1. AADL

AADL is an architecture description language used to model embedded real-time systems as assembly of software components mapped onto execution platforms [1,13,14]. An AADL specification is composed of software, hardware, and composite systems.

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 through *connections* via ports, parameters and access to shared data.

In this paper, we focus on modeling thread scheduling and execution, devices, and bus connections with latency. In Sect. 5, we will describe these aspects of AADL in more detail and define a formal semantics.

2.2. Simulink/Stateflow

Simulink [2] is an environment for model-based design of dynamical systems, and has become a de-facto standard in the embedded systems industry. A Simulink model 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. Stateflow [3] 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.

2.3. HCSP

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 discrete computation. The standard syntax of HCSP is as follows [11,15]:

$$\begin{aligned} P \ &::= \ \mathrm{skip} \mid x := e \mid ch?x \mid ch!e \mid P;\, Q \ \mid B \rightarrow P \mid P \sqcap Q \mid P^* \mid [\!]_{i\in I}(io_i \longrightarrow Q_i) \mid \\ &\quad\ \langle F(\dot{\mathbf{s}},\mathbf{s}) = 0 \& B\rangle \mid \langle F(\dot{\mathbf{s}},\mathbf{s}) = 0 \& B\rangle \trianglerighteq [\!]_{i\in I}(io_i \longrightarrow Q_i) \\ S \ &::= \ P_1 \| P_2 \| \ldots \| P_n \text{ for some } n \geq 1 \end{aligned}$$

where x (resp. $\mathbf{s}$) stands for a variable (resp. a vector of variables), B and e are Boolean and arithmetic expressions, ch is a channel name, io_i stands for a communication event (i.e., either $ch_i?x$ or $ch_i!e$), P, Q, Q_i and P_i are sequential process terms, and S stands for an HCSP process term. The informal meanings of the individual constructors are as follows:

- skip, assignment $x := e$, input $ch?x$, output $ch!e$ sequential composition $P;\, Q$ and internal choice $P \sqcap Q$ can be understood as usual.
- External choice $[\!]_{i\in I}(io_i \longrightarrow Q_i)$ waits for any of communication along io_i to occur and triggers execution of the corresponding Q_i.
- $B \rightarrow P$ behaves as P if B is true, and otherwise terminates. We can then define the conditional statement `if` B `then` P `else` Q as $f := 0;\, B \rightarrow (f := 1;\, P);\, (f = 0 \wedge \neg B) \rightarrow Q$, where f is a fresh variable indicating whether the branch corresponding to B being true is taken.
- Repetition P^* means executing P for an arbitrarily finite number of times.
- $\langle F(\dot{\mathbf{s}},\mathbf{s}) = 0 \& B\rangle$ is the continuous evolution statement. It forces the vector $\mathbf{s}$ of real variables to obey the differential equation F as long as the domain B holds, and terminates when B turns false. For instance, wait d is a special case defined as $t := 0;\, \langle \dot{t} = 1 \& t < d\rangle$. The communication interrupt $\langle F(\dot{\mathbf{s}},\mathbf{s}) = 0 \& B\rangle \trianglerighteq [\!]_{i\in I}(io_i \longrightarrow Q_i)$ behaves like $\langle F(\dot{\mathbf{s}},\mathbf{s}) = 0 \& B\rangle$, except that the continuous evolution is preempted as soon as one of the communications io_i takes place, and the execution of the respective Q_i follows. These two statements are the main extensions of CSP for describing continuous behavior.
- For $n \geq 2$, $P_1 \| P_2 \| \ldots \| P_n$ represents the parallel composition of $P_1, P_2, \ldots, P_n$, which run independently except all communications along the common channels are synchronized.

Compared to the standard HCSP syntax, we make use of an extended language including data structures such as lists and operations on lists, arrays of channels, while loops, and module definitions. A simulator for the extended HCSP language is implemented (Sect. 8) and will be used in the case study.

3. General framework of AADL⊕S/S

We proposed a co-modeling framework for cyber-physical systems combining AADL and S/S [4], called AADL⊕S/S, in which a cyber-physical system can be characterized from the software, hardware, and physics perspectives uniformly, as shown in Fig. 1. In this framework, AADL is used to define the overall architecture of the system, including connections between the software, hardware, and physical components. The software components define the discrete behavior of the system, either as behavior annex within AADL, or S/S diagrams. The physical components define the continuous plants of the system as S/S diagrams.

The architecture layer, described as AADL system composite components, specifies the types of components, and (part of) their implementation (an abstraction of their actual implementation), as well as their composition. It usually consists of a central processor unit classifier with several subcomponent devices (like sensor, controller, and actuator etc). Each of these classifiers has its own type and implementation. For software functionality and physical processes, the architecture layer usually needs their *abstractions*, i.e., the *type classifiers* of these software and physical components. The type classifier of a component declares the set of input and output ports, specifies the contract of its behavior, that are accessible from outside. By contrast, the implementation classifier of a component binds its type classifier with a concrete implementation in the software and physical layers.

Computing type classifier for S/S diagrams. When combining S/S with AADL, we need to provide an abstraction for each S/S diagram, i.e., its type classifier, so that it can be assembled with other components to form the whole system at the architecture layer, while the diagram itself will be used as the implementation classifier of the component. Normally, the type classifier of a component consists of two parts: *port declaration* and *constraints*.

The port declaration declares a set of ports used to input and output data between the component and other ones. However, S/S diagrams can be hierarchical, and hence its external ports can sometimes not be extracted directly. For example, consider the triggered subsystems in a Simulink diagram, they do not have any input and output ports, but are triggered by events. Therefore, we need to analyze the whole system in detail in order to obtain all external ports, particular, event ports. Moreover, this often gets worse when Stateflow models are additionally considered.

To address this problem, we exploit the tool `ss2hcsp`, a component in our toolkit MARS[1] [15,16], which can translate an S/S diagram into a formal HCSP process. By applying `ss2hcsp`, all external ports of an S/S diagram can now be translated, and exposed, by a set of channels in the corresponding HCSP model, which is stored in a separate file.

The reminder of the specification defines the properties of the component. We can adopt two approaches to generate the constraints for a given S/S diagram. The first one uses Daikon [17]. The basic idea is to simulate the given S/S diagram, and then run Daikon to generate a candidate invariant which is satisfied by all simulation runs. The efficiency of this approach is much higher, but the generated invariant (approximation) can only be linear. Moreover, it may not be a true invariant, which could be invalidated by conducting more simulations.

Alternatively, we can generate invariants directly from the S/S diagram, or the translated HCSP process, by using techniques for invariant generation for hybrid systems [18]. This approach can generate more expressive and semantically correct invariants, but the efficiency is normally low.

4. Co-simulation of AADL⊕S/S

In this section, we describe the co-simulation of AADL⊕S/S models by generating simulation code in C, denoted by *AADL⊕S/S2C*, as an extension of our previous work [4]. The C code generation is divided into three parts:

(1) for the AADL part, we use *AADL2C* translator to generate C code following the execution semantics of AADL.
(2) for the S/S part, we use the existing code generation facility in Matlab, to produce C code that can simulate this part of the model step-by-step;
(3) for the architecture part, we implement a library in C that includes thread scheduling protocols, interaction between components and combination of AADL and S/S.

To realize co-simulation, the three parts are integrated together to form an executable C code that simulates the combined model.

4.1. Translating AADL to C

For each thread in the AADL model, we create a corresponding `Thread` object containing its component properties. We use the thread `emerg.imp` from the Cruise Control System (CCS) case study to clarify the mapping rules. `emerg.imp` serves as an emergency control computing the acceleration of a self-driving car in real time. The full case study is described in Sect. 9. The description of `emerg.imp` in AADL is as follows.

```
thread implementation emerg.impl
properties
  Dispatch_Protocol => Periodic;
  Priority => 2;  // highest
  Deadline => 5ms;
  Period => 5ms;
  Execution_Time => 1ms...1ms;
annex Simulink{** ./Examples/AADL/CCS/Simulink/emerg_imp.slx **};
end emerg.impl;
```

The implementation block consists of two parts: properties and Simulink annex. Properties of a thread that are relevant to the simulation include: dispatch protocol, priority, deadline, period, and minimum/maximum execution times. After translation to C, the corresponding `Thread` object `emerg_imp` is given as follows.

```
Thread *emerg_imp = (Thread *)malloc(sizeof(Thread));
emerg_imp->tid = 2;
emerg_imp->threadName = "emerg_imp";
emerg_imp->period = 5;
emerg_imp->priority = 2;
emerg_imp->deadline = 5;
emerg_imp->state = "INITIAL";
emerg_imp->dispatch_protocol = "Periodic";
emerg_imp->maxExecutionTime = 1;
emerg_imp->minExecutionTime = 1;
```

Here, the *state* field stores the status of the thread during simulation, and takes one of five values as defined in the AADL standard: `Initial`, `Ready`, `Running`, `Complete` and `Finish`.

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

4.2. 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 the `Thread` object `emerg_imp` can be defined by the three function pointers:

```
emerg_imp->initialize = emerg_imp_initialize;
emerg_imp->compute = emerg_imp_step;
emerg_imp->finalize = emerg_imp_finalize;
```

where `emerg_imp_initialize`, `emerg_imp_step` and `emerg_imp_finalize` are all functions included in the C file `emerg_imp.c` generated by the tool of Matlab.

4.3. Co-simulation

The above C code is combined together through a function implementing the thread scheduling protocol. In particular, the Highest Priority First (HPF) protocol is implemented in our case study. The communication between components is implemented by shared variables in the context of C code. We set the step size of the Matlab simulation to agree with that of AADL simulation, in this case 1 ms. At each step of the overall simulation, first, the C code denoting the physical environment (such as `vehicle_imp_step()` describing the dynamics of the vehicle) executes one step, updating some shared variables; then, determining a thread to be executed according to HPF and executing the behavior of the thread (such as `emerg_imp->compute()`), which takes into account the period, deadline, and execution time of each thread. The output of the model can then be visualized (in our case using Python's plotting library), serving as a visual check that properties of the model are satisfied for the given initial state.

5. An operational semantics of AADL⊕S/S

In this section, we describe a formal semantics for AADL and for AADL⊕S/S based on timed transition systems that communicate with each other. The main purpose is to describe the semantics, including its many subtleties, in a more familiar language, before presenting the translation to HCSP in Sect. 6. The transition rule has the form $(s,\sigma) \xrightarrow{c,e} (s',\sigma')$, where s, s' are AADL states and σ, σ' are valuations that map variables to values, c condition and e communication event. It means that, starting from s and σ, if c holds, then taking event e leads to s' and σ'.

For AADL, we focus on thread dispatch and execution, scheduling, and bus connections. We first describe each of these aspects in turn, and finally consider the combination AADL⊕S/S.

5.1. Thread

A thread includes a set of ports, properties and its behavior. Ports are used to transfer event and data between threads, processors and devices. Event (resp. event data) ports send events (resp. events with data) that may be queued when the receiver is not ready. The arrival of events can trigger a dispatch of a thread. Data ports send data, where only the latest value is kept on the receiving side. In the semantics, we define each data port as a variable, and each event or event data port as a queue of unprocessed events. They are shared by threads which are their input and output sides, respectively.

The properties of a thread include: dispatch_protocol, which can be *periodic*, *aperiodic*, *sporadic*, *timed* and *hybrid* (we only consider *periodic* and *aperiodic* cases); period for a periodic thread; priority that determines the execution order during scheduling; deadline for the length of the life cycle of a thread; and execution_time for the range of the accumulative time that a thread requires the processor during each dispatch. Usually a minimum and a maximum execution time are specified. For simplicity we will only consider the maximum execution time.

The behavior of a thread can be described by two processes: thread dispatch and execution. The semantics is presented in Fig. 2.

Thread dispatch. After a thread is initialized, it enters the awaiting dispatch (*waitD*) state. Depending on its dispatch protocol, it can be dispatched periodically or aperiodically (by the arrival of events). Given a thread i, if the thread is periodic with period d_i, it can stay at $waitD_i$ state for less than d_i time (rule D1), and at time d_i, sends a dispatch signal to thread i (rule D2). The variable dt_i is introduced to record the elapsed time, with initial value 0. For an aperiodic thread, it is triggered by an incoming event. We use gc to represent a global clock. Let cn_{ki} denote the queue of events arriving at thread i from thread k. We consider cn_{ki} as a variable shared by threads i and k, and write $cn_{ki}(t)$ to denote the queue stored by cn_{ki} at time t. If it is empty, then the thread needs to wait (rule D3); as soon as it turns not empty, it triggers a dispatch and at the same time sends the triggering event to the thread (rule D4), meanwhile the event is removed from the queue. For both cases, at the $disp_i$ state, it goes to $waitD_i$ state directly, waiting for the next dispatch (rule D5).

$$(D1)\ (waitD_i, \sigma) \xrightarrow{dt_i+d\le d_i} (waitD_i, \sigma[dt_i \mapsto dt_i + d]) \quad (D2)\ (waitD_i, \sigma) \xrightarrow{dt_i=d_i, dis_i!} (disp_i, \sigma)$$
$$(D3)\ (waitD_i, \sigma) \xrightarrow{\forall t\in[0,d).cn_{ki}(\sigma(gc)+t)=\emptyset} (waitD_i, \sigma[gc \mapsto gc + t])$$
$$(D4)\ (waitD_i, \sigma) \xrightarrow{cn_{ki}(\sigma(gc))\neq\emptyset, dis_i!(top(cn_{ki}))} (disp_i, \sigma[cn_{ki} \mapsto pop(cn_{ki})])$$
$$(D5)\ (disp_i, \sigma) \rightarrow (waitD_i, \sigma[dt_i \mapsto 0])$$

$$(E1)\ (wait_i, \sigma) \xrightarrow{dis_i?} (ready_i, \sigma[in_i \mapsto cn_{ki}, t_i \mapsto 0, en_i \mapsto 0, sr_i \mapsto 0])$$
$$(E1')\ (wait_i, \sigma) \xrightarrow{dis_i?} (ready_i, \sigma[in_i \mapsto top(cn_{ki}), cn_{ki} \mapsto pop(cn_{ki}), t_i \mapsto 0, en_i \mapsto 0, sr_i \mapsto 0])$$
$$(E2)\ (wait_i, \sigma) \xrightarrow{dis_i?e} (ready_i, \sigma[in_i \mapsto e, t_i \mapsto 0, en_i \mapsto 0, sr_i \mapsto 0])$$
$$(E3)\ (ready_i, \sigma) \xrightarrow{sr_i=0, reqProcessor_i!} (ready_i, \sigma[sr_i \mapsto 1])$$
$$(E4)\ (ready_i, \sigma) \xrightarrow{sr_i=1\wedge t_i+d<DL_i} (ready_i, \sigma[t_i \mapsto t_i + d]) \quad (E5)\ (ready_i, \sigma) \xrightarrow{t_i\ge DL_i, exit_i!} (wait_i, \sigma)$$
$$(E6)\ (ready_i, \sigma) \xrightarrow{t_i\le DL_i, run_i?} (running_i, \sigma)$$
$$(E7)\ (running_i, \sigma) \xrightarrow{t_i<DL_i\wedge en_i=0} (running_i, \sigma[c_i \mapsto 0])$$
$$(E8)\ (running_i, \sigma) \xrightarrow{en_i=1\wedge c_i+d<Max_i\wedge t_i+d<DL_i} (running_i, \sigma[c_i \mapsto c_i + d, t_i \mapsto t_i + d])$$
$$(E9)\ (running_i, \sigma) \xrightarrow{en_i=1\wedge t_i<DL_i\wedge c_i<Max_i, preempt_i?} (ready_i, \sigma)$$
$$(E10)\ (running_i, \sigma) \xrightarrow{t_i=DL_i\wedge c_i<Max_i} (error_i, \sigma) \quad (E11)\ (error_i, \sigma) \xrightarrow{e} (wait_i, \sigma) \quad e \in \{free_i!, preempt_i?\}$$
$$(E12)\ (running_i, \sigma) \xrightarrow{en_i=1\wedge c_i<Max_i\wedge t_i<DL_i} (running_i, \sigma[c_i \mapsto Max_i, t_i \mapsto t_i + (Max_i - c_i)])$$
$$(E13)\ (running_i, \sigma) \xrightarrow{en_i=1\wedge c_i=Max_i, reqResource_i!} (complete_i, \sigma[cn_{ik} \mapsto push(cn_{ik}, out_i)])$$
$$(E14)\ (complete_i, \sigma) \xrightarrow{e} (wait_i, \sigma) \quad e \in \{free_i!, preempt_i?\}$$
$$(E15)\ (running_i, \sigma) \xrightarrow{en_i=1\wedge c_i=Max_i, reqResource_i!} (complete_i, \sigma[cn_{ik} \mapsto out_i])$$
$$(E16)\ (running_i, \sigma) \xrightarrow{en_i=1\wedge c_i=Max_i, block_i?} (block_i, \sigma)$$
$$(E17)\ (block_i, \sigma) \xrightarrow{e} (await_i, \sigma) \quad e \in \{free_i!, preempt_i?\}$$
$$(E18)\ (await_i, \sigma) \xrightarrow{t_i+d<DL_i} (await_i, \sigma[t_i \mapsto t_i + d]) \quad (E19)\ (await_i, \sigma) \xrightarrow{t_i<DL_i, unblock_i?} (ready_i, \sigma)$$
$$(E20)\ (await_i, \sigma) \xrightarrow{t_i=DL_i} (wait_i, \sigma)$$

Fig. 2. Semantics of thread dispatch and execution.

Thread execution. After a thread is dispatched, it goes to the execution process. In the following semantics, we assume the input and output time of ports of threads are by default the dispatch time and the completion time respectively. Assume the input port of thread i is in_i, and the output port is out_i. The cases for multiple ports can be considered similarly.

The thread stays at *wait* state initially. When the thread is dispatched, it goes to *ready* state (rules E1, E1′, E2). The input value is assigned, the elapsed time of thread i from dispatching, recorded by t_i, is initialized to 0; and the variable en_i, which denotes whether the computation of the thread is done or not, is set to 0 for the first entrance; and the variable sr_i, indicating whether it has sent a request for execution to the scheduler or not, is set to 0. If the thread is periodic, and if in_i is a data port, the input value is obtained from the connection cn_{ki} (rule E1). The case for input event or event/data port can be defined similarly (rule E1′). If the thread is aperiodic, the thread is dispatched with the corresponding triggering event received (rule E2). At the first moment after entering the ready state, the thread sends a request to the scheduler (rule E3). After the thread sends the request, if the processor is not available, it will stay in ready state for some time (rule E4), where DL_i denotes the deadline of thread i. If the elapsed time exceeds the deadline at *ready* state, the thread notifies the scheduler by sending *exit* signal and goes back to *wait* state (rule E5).

If the thread is scheduled to execute within the deadline, it enters the running state (rule E6). When it is the first time to enter the running state from the ready state in this dispatch (implied by $en = 0$), the execution time c_i is set to 0 (rule E7). At the running state, the thread will execute the behavior defined by S/S diagram. We assume the discrete computation will be finished in zero time as soon as entering the running state and will not be preempted. We leave this question to the combined semantics of AADL and S/S, where variable en_i is set to 1 after the computation is done.

According to the AADL standard, the thread completes execution at any time between the minimal and maximal execution time. In order to fix a deterministic behavior, we force the thread complete at the maximal execution time. After the computation is done, the thread can stay at *running* state for some d time (rule E8). During this process, the thread may be preempted by another ready thread (rule E9). If the elapsed time reaches the deadline first before the maximum execution time is met, the thread goes to the error state (rule E10), then it gives up the processor, by notifying the scheduler or gets preempted just at this time, and goes to awaiting dispatch state directly (rule E11). If the thread reaches the maximum execution time before the deadline, it executes successfully (rule E12).

It only remains to output the result. If the receiver is a thread in another processor, or a device, the communication is realised by a shared bus. Thus the thread has to apply for the bus resource. If the resource application is successful, it goes to *complete* state, and outputs to the bus by adding to the corresponding queue (rule E13) or updating the variable (rule E15). At *complete* state, it gives up the processor and goes to the awaiting dispatch state (rule E14). Otherwise, if the resource is being used by other thread, it will be blocked (rule E16), and then gives up the processor and goes to the await

$$(S1)\ (waitS, \sigma) \xrightarrow{reqProcessor_i?} (preempt, \sigma[rdy \mapsto i])$$

$$(S2)\ (preempt, \sigma) \xrightarrow{idle=1, run_i!} (waitS, \sigma[run\_now \mapsto i, idle \mapsto 0]) \quad i = \sigma(rdy)$$

$$(S3)\ (preempt, \sigma) \xrightarrow{idle=0, canPreempt(i,\ run\_now), preempt_{run\_now}!, run_i!} (waitS, \sigma[run\_now \mapsto i])$$

$$(S4)\ (preempt, \sigma) \xrightarrow{idle=0, \neg canPreempt(i,\ run\_now)} (waitS, \sigma[Pool \mapsto Pool \cup \{i\}])$$

$$(S5)\ (waitS, \sigma) \xrightarrow{free_i?} (sche, \sigma) \quad (S6)\ (sche, \sigma) \xrightarrow{Pool \neq \emptyset, run_j!} (waitS, \sigma[run\_now \mapsto j, Pool \mapsto Pool \backslash \{r\}])$$

$$(S7)\ (sche, \sigma) \xrightarrow{Pool = \emptyset} (waitS, \sigma[idle \mapsto 1]) \quad (S8)\ (waitS, \sigma) \xrightarrow{Pool \neq \emptyset, exit_i?} (waitS, \sigma[Pool \mapsto Pool \backslash \{i\}])$$

Fig. 3. Semantics of scheduler.

state (rule E17). At the await state, it waits to be unblocked, and as soon as it is unblocked before the deadline, it goes to the ready state again (rule E18, E19). Otherwise, the resource application fails and it goes to the awaiting dispatch state (rule E20).

5.2. Process, processor and scheduler

A process includes a set of ports, port connections, properties and threads. One important *property* defined in *processor* is *schedu_protocol*, according to which the execution of all threads on a processor is coordinated by the scheduler. There are various scheduling protocols, including First In First Out (FIFO), Rate Monotonic Scheduling (RMS), Deadline Monotonic Scheduling (DMS), Highest Priority First (HPF), and so on.

Scheduler. There are three states of *scheduler*: *waitS*, *preempt*, and *sche*, responsible for waiting for ready threads, trying to preempt current running thread, and scheduling threads respectively. The semantics is given in Fig. 3. Initially, the scheduler stays at *waitS* state, and the processor is idle, represented by $idle = 1$.

When the scheduler receives a request from thread i, it goes to *preempt* state, where rdy records the new ready thread (rule S1). If the processor is idle, the new ready thread is scheduled to execute directly (rule S2), where *run_now* denotes the current running thread. If the processor is busy, but if the incoming ready thread i has higher priority than the running thread, i becomes the new running thread, and the previous running thread is preempted (rule S3). Otherwise, it is added to the waiting ready set, represented by *Pool* (rule S4), where function *canPreempt(i, run_now)* represents that i will preempt r according to the scheduling protocol.

When the current running thread completes, then the scheduler will receive a *free* signal from the thread, and go to *sche* state (rule S5). At *sche* state, it will choose one thread from the ready set to execute if the current ready set is not empty (rule S6), where j is defined by $choose(Pool)$, choosing the next running thread according to the scheduling protocol. Otherwise, it goes to *waitS* state and the processor becomes idle (rule S7). If the thread fails to be scheduled before the deadline, the scheduler will be notified and the thread will be deleted from the ready set (rule S8).

5.3. Connection

Port connection. For the sampled port connection, we model them as a variable or a queue, depending on the type of the destination port. For instance, given a port connection cn: **port** th1.a → th2.b, if th2.b is a data port, then we model cn as a variable cn_{12}; otherwise, we model cn as a queue cn_{12}, indicating that thread 1 is the outgoing side and thread 2 is the incoming side.

Bus connection. Bus connection represents communication between processors, memory and devices by accessing a shared bus. The exact semantics of bus behavior when there are multiple users is not specified by the AADL standard. Indeed there are many variations, for example the difference between serial and parallel bus. We choose a basic semantics based on a simplified version of the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol [19,20], where the bus blocks all other users during a transmission, and each transmission takes time specified by the latency property in AADL. The semantics of bus is shown in Fig. 4. At the beginning, the bus stays at wait state. If it receives a resource application, it goes to *res* state, obtains the input from the input connection, and records the time that the thread occupies the bus (rule B1, B1′). Assume a predefined latency, denoted by L_b, is needed for the thread occupying the resource, then the bus will stay at *res* state for less than L_b time (rule B2). During this time, it will block other thread who attempts to apply for the resource (rule B3). When the latency time is passed, the bus goes back to the wait state and transfers the output through the corresponding output connection (rule B4, B4′). At the wait state, the bus may also unblock some thread that was previously blocked (rule B5).

5.4. Combination of AADL and S/S

In our framework, there are two ways by which S/S diagrams are integrated with AADL. First, the physical environment of a system can be described by a continuous Simulink diagram, which can be represented by an `abstract` component

$$(B1)\ (waitB, \sigma) \xrightarrow{reqResource_i?} (res, \sigma[t_b \mapsto 0, in \mapsto top(cn_i), cn_i \mapsto pop(cn_i)])$$
$$(B1')\ (waitB, \sigma) \xrightarrow{reqResource_i?} (res, \sigma[t_b \mapsto 0, in \mapsto cn_i])$$
$$(B2)\ (res, \sigma) \xrightarrow{t_b+d \le L_b} (res, \sigma[t_b \mapsto t_b + d]) \quad (B3)\ (res, \sigma) \xrightarrow{block_j?} (res, \sigma)$$
$$(B4)\ (res, \sigma) \xrightarrow{t_b = L_b} (waitB, \sigma[cn_j \mapsto push(cn_j, out)])$$
$$(B4')\ (res, \sigma) \xrightarrow{t_b = L_b} (waitB, \sigma[cn_j \mapsto out]) \quad (B5)\ (waitB, \sigma) \xrightarrow{unblock_i!} (waitB, \sigma)$$

Fig. 4. Semantics of bus.

of AADL. Second, the behavior of threads in AADL can be described by discrete S/S diagrams. In what follows, we introduce the operational semantics of these two combinations.

Abstract type classifier. AADL allows to use abstract type classifiers for physical components, which acts as an interface for integrating continuous models described in S/S. Such component cannot be scheduled, but rather executes continuously. The type classifier for physical components has the following form:

```
abstract phy
  features
    a: in data port;
    b: out data port;
end phy
```

The implementation of `phy` will be defined as a continuous S/S diagram with the same name *phy*, with input a and output b. Assume the connections to a and b are cn_a and cn_b respectively. We define them as the corresponding channels cn_a and cn_b. According to the semantics of S/S diagram, the semantics of `phy` implementation in S/S is composed of the following rules.

At any time, the continuous evolution is ready to output the value or receive the input value:

$$(s_1, \sigma) \xrightarrow{cn_b!\sigma(b)} (s_1, \sigma) \quad (s_1, \sigma) \xrightarrow{cn_a?c} (s_1, \sigma[a \mapsto c])$$

Suppose the solution of the continuous evolution of *phy* with initial value σ is p defined over the time interval $[0, \infty)$, then for any $d > 0$, we have $(s_1, \sigma) \xrightarrow{d} (s_1, \sigma[v \mapsto p(d)])$, with v the continuous variable of *phy*.

On the AADL side, the connections to a and b are defined by the corresponding communications. Both communications can occur immediately whenever needed on the AADL side.

Thread behavior implementation. In this paper, we focus exclusively on using S/S diagrams to define computational behavior of threads (and so omitting the case of behavioral annexes). For this case, we need to introduce new channels to transfer the values between AADL thread and S/S diagram. Assume thread i has an input port a and an output port b, then define channels as and bs for transmission of input and output respectively:

$$(s_1, \sigma) \xrightarrow{as?c} (s_2, \sigma[a \mapsto c]) \quad (s_2, \sigma[a \mapsto c]) \to (s_3, \sigma') \quad (s_3, \sigma') \xrightarrow{bs!\sigma'(b)} (s_1, \sigma')$$

Especially, the second transition corresponds to the discrete computation of the S/S diagram. On the AADL side, the rule (E7) is changed to:

$$(running_i, \sigma) \xrightarrow{t < DL_i \wedge en_i = 0, as!\sigma(a), bs?f} (running_i, \sigma[c \mapsto 0, b \mapsto f, en_i \mapsto 1])$$

where en_i is changed to 1, representing that the computation is finished.

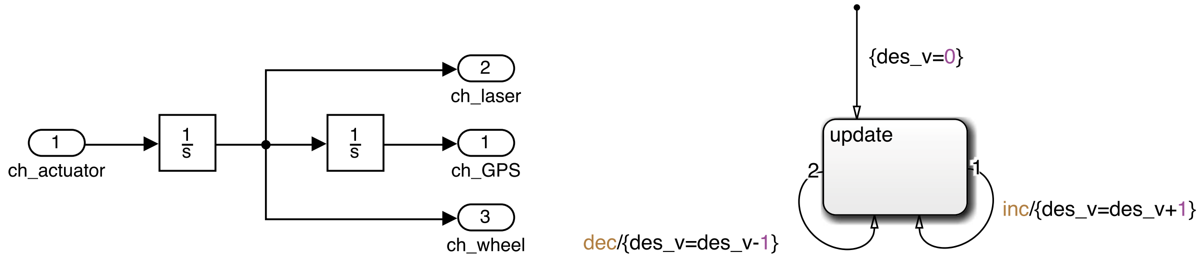
Finally, we define the semantics of the combined AADL and S/S by the parallel composition of respective transition systems in both cases, in which the communication events are synchronized.

6. An HCSP-based denotional semantics of AADL⊕S/S

In this section, we present a translation of AADL⊕S/S to HCSP, which defines a denotational semantics of AADL⊕S/S. First, we review the existing translation from S/S diagrams and give some examples. Next, we consider translation of threads, scheduler, and connections in turn. Finally, we describe the translation of combined AADL⊕S/S models.

6.1. From S/S to HCSP

Existing work by Zou et al. [21,22] define how to translate S/S components into HCSP processes. Inputs and outputs of S/S diagrams are translated into HCSP communication channels to support interaction with the other translated components. To give a specific example from our case study, consider the Simulink diagram of Fig. 5(a) modeling the physical behavior of the vehicle (`vehicle.imp`) described in Fig. 10. The following is the translated HCSP process of “programming language”



(a) `vehicle.imp` (b) `pan_ctr_th.imp`

Fig. 5. Examples of S/S diagrams.

style which can be executed by the HCSP simulator (Sect. 8) for simulation. Concretely, in the following HCSP code, the notation `$` denotes the external choice operator $[\!]$, the statement `<x_dot = y & B> |> [] (ch!x --> P, dh?y --> Q)` denotes an ODE with communication interruption: $\langle \dot{x} = y \& B\rangle \trianglerighteq [\!](ch!x \longrightarrow P, dh?y \longrightarrow Q)$, and the double-star `P**` denote the repetition P^*. In what follows, we use this "programming language" instead of the mathematical notation in Sect. 2.3 to describe HCSP processes.

```
vehicle_imp ::=
  v := 0; s := 0;
  sent_laser := 0; sent_wheel := 0; sent_GPS := 0;
  while sent_laser == 0 || sent_wheel == 0 || sent_GPS == 0 do
    ch_laser!v --> sent_laser := 1
    $ ch_wheel!v --> sent_wheel := 1
    $ ch_GPS!s --> sent_GPS := 1
  endwhile;
  ch_actuator?a;
  (<s_dot = v, v_dot = a & true> |> [](
     ch_laser!v --> skip, ch_wheel!v --> skip,
     ch_GPS!s --> skip,ch_actuator?a --> skip)
  )**
```

The velocity `v` and the position `s` of the vehicle are initialized to 0. The process first outputs all initialized values and then receives an acceleration `a` which can start the evolution of the ODE. During the continuous evolution, it is always ready to receive a new acceleration and output the current velocity or position of the vehicle. Observe that the input (acceleration) and the three outputs (one for position and the other two for velocity) are translated into communication channels `ch_actuator`, `ch_GPS`, `ch_laser` and `ch_wheel`, respectively.

Next, consider the user panel control thread (`pan_ctr_th.imp`) of the system in Fig. 10, which is modeled using a Stateflow diagram. The diagram is given in Fig. 5(b) and the translated HCSP is as follows. It first initializes the desired velocity `des_v` to 0 and then monitors events via input channels. If any event arrives, the corresponding discrete computation is performed and the execution result is delivered.

```
panel_ctr_th_imp ::=
  des_v := 0;  # Initialization
  (inputs?event;
   event == "inc" -> des_v := des_v + 1;  # Discrete Computation
   event == "dec" -> des_v := des_v - 1;
   outputs!des_v
  )**
```

From the above example, we can see that the translation of an S/S diagram defining the behavior of an AADL thread is divided into four parts: initialization of variables, input, discrete computation, and output. These will be spliced into the code for the thread as will be described in Sect. 6.3.

6.2. Translation of the scheduler

We now examine the translation of the HPF scheduler, shown in Fig. 6(a), which will be used in the case study. Similar translation principles apply to most, if not all, scheduling policies specified in the AADL standard.

Note that in Fig. 6(a), parameterized channel operations like `free[sid][_tid]` are syntactic sugar supported by our HCSP simulator (Sect. 8) that will be instantiated before simulation. The parameter `sid` will be instantiated by the `sid` of `SCHEDULE_HPF`, while `_tid` will be instantiated by pattern matching. For example, for the following parallel composition

```
free[0][_tid]?prior || free[0][0]!2 || free[0][1]!3
```

the left channel operation will be instantiated as `free[0][0]?prior` and `free[0][1]?prior` by pattern matching, resulting in

```
free[0][0]?prior || free[0][0]!2 || free[0][1]?prior || free[0][1]!3
```

However, if we treat `_tid` as an input, say `free[sid]?(tid, prior)`, then the above parallel composition will become

```
module SCHEDULE_HPF(sid) ::=
  Pool := [];
  run_now := -1; run_prior := -1;
  (
  reqProcessor[sid][_tid]?prior -->
    if run_prior > prior then
      Pool := put(Pool, [prior, _tid])
    else
      run_now != -1 ->
        preempt[sid][run_now]!;
      run_now := _tid;
      run_prior := prior;
      run[sid][run_now]!
    endif
  $ free[sid][_tid]? -->
      assert(_tid == run_now);
      if len(Pool) > 0 then
        (run_prior, run_now) :=
           get_highest(Pool);
        Pool := delete(Pool, run_now);
        run[sid][run_now]!
      else
        run_prior := -1; run_now := -1
      endif
  $ exit[sid][_tid]? -->
      Pool := delete(Pool, _tid)
  )**
```

(a) Translation of the HPF scheduler

```
module DIS_aperiodic(
    send, out_port, recv, in_port) ::=
  queue := [];
  (if len(queue) == 0 then
     outputs[send][out_port]?event;
     queue := push(queue, event)
   else  # len(queue) > 0
     outputs[send][out_port]?event -->
       queue := push(queue, event)
     $ dis[recs][in_port]!head(queue) -->
       queue := tail(queue)
   endif)**
```

(b) Dispatch for aperiodic threads

```
module DIS_periodic(tid, period) ::=
  (wait(period);
   dis[tid]!)**
```

(c) Dispatch for periodic threads

Fig. 6. Translation of scheduler and dispatching threads.

```
free[0]?(tid, prior) || free[0]!(1, 2) || free[0]!(2, 3)
```

from which we can see that the channel `free[0]` is shared by more than two processes, violating the assumption in HCSP that each channel can only connect two parallel processes [15].

The translation of HPF provides three pieces of information: the list of threads that are currently ready (`Pool`), the ID (`run_now`) and priority (`run_prior`) of the thread that is currently running (−1 if no thread is running). When the scheduler receives a request from a thread, it compares the priority of the thread with the priority of the running thread. If the running thread has higher priority, then the new thread is inserted into the ready pool. Otherwise, the running thread is preempted, and the new thread starts running. When a thread releases the processor, the scheduler chooses the thread with the highest priority in the ready pool to run. When a non-running thread signals that it is no longer ready, the scheduler simply removes it from `Pool`.

6.3. Translation of threads

In this section, we introduce translation of threads with periodic and aperiodic dispatch protocols. Each thread is translated into two HCSP processes. One process, with name prefixed by `DIS`, is used to dispatch the thread, while the second, with name prefixed by `EXE`, models execution of the thread.

6.3.1. Thread dispatch

The translation of dispatching an aperiodic thread is shown in Fig. 6(b). It describes the behavior that the source thread (`send`) sends events via the port `out_port` to the port `in_port` on the target thread (`recv`). Once `recv` receives an event, it can be dispatched. Inside the dispatching process, `queue` contains the list of events to be processed. If the event queue is empty, then the process monitors channel `outputs[send][out_port]`, and pushes any received event onto the queue. If there are events in the queue, the process either gets a new event through `outputs[send][out_port]` as in the previous case, or dispatch the thread by sending the head event of the queue along `dis[recv][in_port]`. The dispatch for periodic threads is much simpler, shown in Fig. 6(c).

6.3.2. Thread body

Next, we consider translation of thread body. The body of a thread consists of discrete computation followed optionally by outputting the result along a shared resource. The discrete computation is described by an S/S model, so the translation comes from Section 6.1. We will represent this code as {`Discrete Computation`} in the following.

Next, we consider translation of output. If no resource is required, the translation is given in Fig. 7(a). Otherwise, it is given in Fig. 7(b). We will represent this code as {`Output`} in the following. For the first case, the thread simply outputs using the given channel, and then gives up the processor, by sending a `free` signal, or exactly at this moment, receiving the `preempt` signal. For the second case, the thread requests the resource by either successfully sending the `reqResource` signal or receiving a `block` signal. These two channels are implemented by the translation of bus component (see Sect. 6.4), which guarantees that at any time one of these two channels is ready for communication. If the thread is able to send the `reqResource` signal, it obtained access to the bus, so it can proceed to send the outputs, and gives up the processor at the end as before. Otherwise, it gives up the processor and transitions to the await state (to be explained below).

```
outputs[tid][out_port1]!out1;
outputs[tid][out_port2]!out2;
...
(free[tid]! --> state := "wait"
 $ preempt[tid]? --> state := "wait")
```

(a) Output: no resource required

```
reqResource[tid]! -->
  outputs[tid][out_port1]!out1;
  outputs[tid][out_port2]!out2;
  ...
  (free[tid]! --> state := "wait"
   $ preempt[tid]? --> state := "wait")
$ block[tid]? -->
  # Resource request failed
  (free[tid]! --> state := "await"
   $ preempt[tid]? --> state := "await")
```

(b) Output: resource required

```
1  module EXE(tid, prior, Max, DL) ::=
2    {Initialization}
3    state := "wait";
4    (if state == "wait" then
5       dis[tid]?;
6       {Input}
7       t := 0; en := 0; state := "ready"
8     elif state == "ready" then
9       reqProcessor[tid]!prior;
10      <t_dot = 1 & t < DL> |> []
11        (run[tid]? --> state := "running");
12      t == DL && state == "ready" ->
13        (exit[tid]! --> state := "wait"
14         $ run[tid]? --> state := "running")
15    elif state == "running" then
16      en == 0 ->
17      (c := 0;
18       {Discrete Computation};
19       en := 1);
20      en == 1 -> (
21      <t_dot = 1, c_dot = 1 & c < Max
22       && t < DL> |> [] (
23        preempt[tid]? --> state := "ready")
24      state == "running" ->
25        # c == Max or t == DL
26        if c == Max then
27          # t <= DL
28          {Output}
29        else
30          # c < Max && t == DL
31          preempt[tid]? --> state := "wait"
32          $ free[tid]! --> state := "wait"
33        endif
34      );
35    else  # state == "await"
36      <t_dot = 1 & t < DL> |> []
37        (unblock[tid]? --> state := "ready");
38      t == DL -> state := "wait"
39    endif
40   )**
```

(c) Thread execution

Fig. 7. Translation of thread execution.

6.3.3. *Thread execution*

The translation for thread execution is shown in Fig. 7(c). It is expressed as the HCSP process `EXE`, which is structured as a state machine. Variable `state` represents the current state of the thread (one of `wait`, `ready`, `running` and `await`). Variables `t`, `c` and `en` are introduced, with the same meaning as in Sect. 5.

The thread is initially at `wait` state, waiting for dispatching. First we consider the case of periodic dispatching. Once it receives the dispatching signal from `DIS_periodic`, it takes inputs from the input ports, resets `t` and `entered`, and then enters the `ready` state. The `{Input}` can be modeled by a sequence of input channel operations that get data and events from all input ports:

```
inputs[tid][in_port1]?in1; inputs[tid][in_port2]?in2;...
```

In the `ready` state, the thread first sends its request to run to the scheduler (line 9). Then it waits for the permission from the scheduler inside an interrupt construct for at most `DL` (deadline) time units (line 10-11). If the scheduler sends the `run` signal within the deadline, the thread enters the `running` state. If the deadline has passed with the thread still in `ready` state, the thread sends the `exit` signal and returns to the `wait` state. The external choice on line 14 ensures that if the scheduler sends the run signal exactly when `t` reaches the deadline, the thread will still enter the `running` state.

In the `running` state, the implementation is divided into whether it is entered into for the first time during the current dispatch. For the first entry, the computation time `c` is set to `0`. Then, the thread performs discrete computation and set variable `en` to `1`. As explained in Sect. 5, we choose to model the thread as completing the discrete computation immediately.

After the discrete computation, the thread begins to wait for a duration of its maximum execution time, inside the interrupt construct on line 21-23. The waiting stops either when the maximum execution time is reached (`c == Max`), when the deadline is reached (`t == DL`), or preempted by the scheduler. If the interrupt construct finishes with the thread still in `running` state, then it must be the case that one of the two boundary conditions is reached. If the maximum execution time has been reached, the thread proceeds to produce output (Sect. 6.3.2). Otherwise, the deadline is reached and the thread goes to `wait` state.

In the `await` state, the thread uses an interrupt construct to wait for the resource to unblock, until the deadline is reached (line 36-37).

The above defines the execution model for periodic threads. For aperiodic threads, the only modification is replacing the dispatching statement `dis[tid]?` with an external non-deterministic choice, such as

```
dis[tid][in_event_port1]?event --> skip $ dis[tid][in_event_port2]?event --> skip $ ...
```

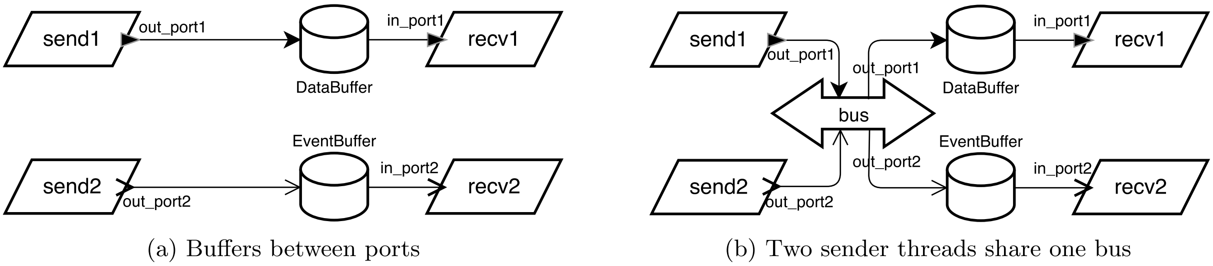


Fig. 8. Connections are translated into buffers (with and without buses).

because an aperiodic thread is dispatched by event, and there may be several different kinds of such events. In addition, the {`Input`} of an aperiodic thread can be depicted by the following sequence of input channel operations that get data from the input data ports:

```
inputs[tid][in_data_port]?in1; inputs[tid][in_data_port2]?in2;...
```

6.4. Translation of ports and connections

Each port connection can be formalized as a pair of HCSP communications in which the flow of data or control is directional. In this way, the interfaces and connections defined in the AADL file can be realized through connections between ports.

First, we consider the case that connections are not bound to buses. If two threads are bound to the same processor, then they can communicate with each other without buses. In this case, connections can be translated into a buffer on the receiver port to store temporarily data or event, see Fig. 8(a). Consider a connection from the data port `out_port` on the thread `send` to the data port `in_port` on the thread `recv`. The translation is shown in Fig. 9(a). According to the `EXE` defined above, `out_port` and `in_port` can be translated into the respective channel operations `outputs[send][out_port]!` and `inputs[recv][in_port]?`. According to [13], data ports are interfaces for data transmission among components without queuing and the transmission is asynchronous. So, there should be a buffer on the receiver port to coordinate the asynchronous transmission, which can be modeled by an ODE with communication interrupt, as shown in Fig. 9(a).

In contrast to data ports, event ports are interfaces for the communication of events that may be queued [13]. For the event ports `out_port2` on `send2` and `in_port2` on `recv2` in Fig. 8(a), if the receiver thread is aperiodic, then the connection can be represented by a dispatching process `DIS_aperiodic(send2, out_port2, recv2, in_port2)` introduced in Sect. 6.3.1. If the receiver thread is periodic, then connection can be translated into a simplified `DIS_aperiodic`, i.e., an `EventBuffer`, shown in Fig. 9(b).

Then, we consider the case that connections are bound to one or more buses., i.e., buses can be shared among different components. In order to illustrate the translation intuitively, we introduce an example of two send threads sharing one bus, showed in Fig. 8(b), where the `out_port1` on `send1` is a data port while the `out_port2` on `send2` is an event port. The bus process is shown in Fig. 9(c). Before sending an output, the senders try to request the permission for using the bus via channels `reqBus[send1]` and `reqBus[send2]`. If one gets the permission, it sends the data to the bus via channel `outputs[sendi][out_porti]` immediately. Therefore, the `DataBuffer` on `in_port1` of `recv1` and the `EventBuffer` on `in_port2` of `recv2` should be instantiated as `DataBuffer(bus, out_port1, recv1, in_port1)` and `EventBuffer(bus, out_port2, recv2, in_port2)`, respectively, where `out_port1` and `out_port2` are corresponding output ports on the bus. The transmission may generate latency. During the transmission period, any other thread requesting the bus permission will be blocked (`BLOCK`). The blocking code is shown in Fig. 9(d) and 9(e). When the bus becomes idle, it can unblock the blocked threads by sending the signal `unblock[send]` to the corresponding thread.

6.5. Translation of the combined model to HCSP

As stated in Sect. 5.4, the combination of AADL and S/S is implemented in two ways: First, the physical environment, represented by an `abstract` component, is described by a continuous Simulink diagram; Second, the behavior of `threads` in AADL can be modeled by discrete Simulink diagrams. The interaction between components is translated to HCSP using parallel composition and communications.

With the aid of Simulink, the continuous dynamics of the physical world can be described. The physical environment, such as the temperature, evolves forever following some ODEs and communicates passively with control programs, i.e., it can be observed by sensors and affected by actuators but it will never send or require data/event actively. Due to these special characteristics, as far as we know, there is no component of AADL that can model physical environments in a natural manner. Therefore, we choose `abstract` components specified by continuous Simulink diagrams to represent physical environments, which can be translated into HCSP processes using existing work [21,22]. For example, the physical

```
module DataBuffer(send, out_port,
    recv, in_port, init_value) ::=
  data := init_value; (
  <data_dot = 0 & true> |> []
    (outputs[send][out_port]?data --> skip,
     inputs[recv][in_port]!data --> skip)
  )**
```

(a) Translation of connections between data ports (not bound to buses)

```
module EventBuffer(
    send, out_port, recv, in_port) ::=
  queue := [];
  (if len(queue) == 0 then
     outputs[send][out_port]?event;
     queue := push(queue, event)
   else  # len(queue) > 0
     outputs[send][out_port]?event -->
     queue := push(queue, event)
   $ inputs[recv][in_port]!head(queue) -->
     queue := tail(queue)
   endif
  )**
```

(b) Translation of connections between event ports (not bound to buses). The receiver thread is periodic.

```
module BUS(bus_id, send1, out_port1, send2,
     out_port2) ::= (
  reqBus[send1]? -->
  outputs[send1][out_port1]?data;
  BLOCK2;
  outputs[bus_id][out_port1]!data
$ unblock[send1]! --> skip
$ reqBus[send2]? -->
  outputs[send2][out_port2]?event;
  BLOCK1;
  outputs[bus_id][out_port2]!event
$ unblock[send2]! --> skip
)**
```

(c) Translation of connections between ports (bound to buses).

```
BLOCK1 ::=
  t := 0;
  while t < latency do
    <t_dot = 1 & t < latency> |> []
      (block[send1]! --> skip)
  endwhile
```

(d) Implementation of BLOCK1

```
BLOCK2 ::=
  t := 0;
  while t < latency do
    <t_dot = 1 & t < latency> |> []
      (block[send2]! --> skip)
  endwhile
```

(e) Implementation of BLOCK2

Fig. 9. Translation of connections.

component phy in Sect. 5.4 described by an open continuous Simulink diagram will be translated into a loop of an ODE with communication interruption in the form

$$\left(\langle F(\dot{b},a)=0\&\mathbf{true}\rangle \trianglerighteq [\!] \begin{pmatrix} cn_b!b \longrightarrow \text{skip}, \\ cn_a?a \longrightarrow \text{skip} \end{pmatrix}\right)^*$$

It will be composed with the HCSP processes of other components by parallel composition.

The behavior of a thread in AADL can also be modeled by an open discrete S/S diagram which gets data and events from input ports and outputs the computation results to output ports. The input and output ports of the S/S diagram should be linked to the corresponding data and event ports on the thread. As introduced in Sect. 6.3, a `thread` in AADL can be translated into an automaton with templates labeled `Input`, `Discrete Computation` and `Output` (see Fig. 7(c)). An open S/S diagram describing the thread behavior can be translated into an HCSP process composed of three parts `Input`, `Discrete Computation` and `Output` as well, which can then be embedded into the corresponding templates of the thread's automaton. In this way, every thread can be translated into an individual HCSP process.

The connections between threads can be translated into buffer and bus processes for coordinating asynchronous communication between components (see Sect. 6.4). 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.

7. Correctness of translation

We now consider the correctness of translation from AADL and AADL⊕S/S to HCSP. The correctness of translation from S/S to HCSP that we rely on is proved in existing work [15]. Hence we only examine the AADL part. The correctness of translation of AADL⊕S/S then follows from the correctness of translation of AADL and S/S, plus the correct interaction between them via parallel composition.

Before presenting the proof, we first introduce some related notions. Let $\langle \mathcal{S}, A, \rightarrow\rangle$ be a labeled transition system, where $\mathcal{S}$ is a set of configurations, A a set of actions (including communication action, time progress, and the silent action τ), $\rightarrow\ \subseteq (\mathcal{S}\times A\times\mathcal{S})$ is the transition relation. We write $s \xrightarrow{a} t$ to represent a transition, $s \Rightarrow t$ a transition path consisting of an arbitrary number of τ transitions, and $s \overset{a}{\Rightarrow} t$ a path $s \Rightarrow s_1 \xrightarrow{a} t_1 \Rightarrow t$ for $a \in A\backslash\{\tau\}$. The semantics of AADL defined in Sect. 5 is a transition system, where each configuration has the form (s,σ), with s an AADL execution state and σ a variable valuation, and A a set of timed and communication events. The semantics of HCSP can also be defined as a transition system, see [15] for details, where a configuration has the form (P, σ), with P a HCSP process to be executed and σ a valuation. With the semantics of both AADL and HCSP defined in terms of transition systems, we can compare them using weak bisimulation. For each configuration co, we introduce $co.val$ to return the corresponding valuation. The notion of weak bisimulation is given below.

Definition 1 *(Weak bisimulation).* Let $\mathcal{T}_i = \langle \mathcal{S}_i, A_i, \rightarrow_i \rangle$ be two transition systems for $i = 1, 2$. A relation $R \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a weak bisimulation, iff it is symmetric, and moreover, for all $(s,t) \in R$, $s.val =_m t.val$, i.e. they are equivalent under a projection m from variables of T_1 to T_2, and if $s \xRightarrow{a}_1 s'$ holds, then there exists a path $t \xRightarrow{a}_2 t'$ such that $(s', t') \in R$.

We can then prove the following result indicating the equivalence between AADL and its translated HCSP model.

Theorem 1 *(Weak bisimulation). Let M be a given AADL model and H its translated HCSP process. Assume σ is an initial valuation, then there exists a weak bisimulation relation R such that $((M, \sigma), (H, \sigma)) \in R$.*

Proof. We prove the theorem according to the type of AADL construct M. If M is a thread i, then its transition semantics consists of two parts: thread dispatch and execution. We list the proof respectively for them below.

Thread dispatch. The HCSP processes of thread dispatch are given in Fig. 6(b, c) respectively. Let σ be an initial state, then M starts at $(waitD_i, \sigma)$, next we prove that it satisfies the weak bisimulation relation with the corresponding HCSP process (DIS, σ), depending on whether thread i is dispatched periodically or aperiodically. If thread i is periodic,

- Transition D1 corresponds to the execution of `wait(period)` of `DIS_periodic`, where `period` $\leftrightarrow d_i$.
- Transition D2 corresponds to the execution of `dis[tid]!`, then AADL side goes to $(disp_i, \sigma[dt_i \mapsto d_i])$, and meanwhile, the HCSP side goes to the end of the first repetition, i.e. $(\epsilon; \mathtt{DIS\_periodic}, \sigma')$, satisfying that σ' and $\sigma[dt_i \mapsto d_i])$ are equivalent except for local variables of each side (e.g. dt_i for AADL). Notice that both the communications can occur immediately from transition E1 of AADL side and HCSP process `EXE`.
- Transition D4 corresponds to the start of the next repetition, i.e. $(\mathtt{DIS\_periodic}, \sigma')$. All the above relations are symmetric.

If thread i is aperiodic,

- Transition D3 defines the waiting for incoming event, when the corresponding input event queue cn_{ki} is empty, and goes to $(\mathtt{waitD}_i, \sigma[gc \mapsto gc + d])$. The projection is `queue` $\leftrightarrow cn_{ki}$, with initial value $[\cdot]$ in `DIS_aperiodic`. We denote the repetition part by `body*`. Correspondingly, `body*` has an execution when *queue* is empty (it will wait till some incoming event is received, decided by thread k which outputs event to `queue`), resulting in $(body; body^*, \sigma[gc \mapsto gc + d])$.
- When transition D4 occurs, the queue cn_{ki} is not empty, correspondingly, *queue* is not empty in `body`, and for both sides, the output $dis_i!$ is ready to occur, resulting in $(disp_i, \sigma[cn_{ki} \mapsto pop(cn_{ki})])$ and $(\epsilon; body^*, \sigma[queue \mapsto tail(queue)])$ respectively.
- When transition D5 occurs, it goes to $(\mathtt{waitD}_i), \sigma\sigma[cn_{ki} \mapsto pop(cn_{ki})])$, and meanwhile, the HCSP side goes to next repetition, i.e. $(body^*, \sigma[queue \mapsto tail(queue)])$ again.

For most of the above relations, they are symmetric. There is one exception that, for `DIS_aperiodic`, at the beginning of execution of *body*, there exists a transition corresponding to the receiving of event when *queue* is empty, i.e. $(body; body^*, \sigma)$ goes to $(\epsilon, body^*; \sigma[queue \mapsto push(queue, ev)])$. This occurs when thread k sends ev to thread i, i.e. thread k executes output (line 28 of HCSP process `EXE`). At the AADL side, there exists a transition E13 for thread k that writes the output event out_k to the shared queue cn_{ki}. Notice that HCSP uses synchronized communication while AADL uses shared variable to manage the event queue.

Thread execution. The HCSP process of thread execution, i.e. `EXE`, is given in Fig. 7(c). We use $\mathtt{EXE}_i$ to represent the execution of thread i. The AADL side starts at $(\mathtt{wait}_i, \sigma_e)$, and the HCSP side starts at $(\mathtt{EXE}_i, \sigma_e')$, where σ_e and σ_e' are resulting from the thread dispatch and they are equivalent under a projection from the proof for thread dispatch.

- When E1 executes, $dis_i?$ occurs, resulting in $ready_i$ state and the assignment of some local variables. This transition corresponds to the execution of lines 3-7 in `EXE`, where the projection between variables is

  $$dis_i? \leftrightarrow dis[i]?, t_i \leftrightarrow t, en_i \leftrightarrow en, in_i \leftrightarrow data$$

  where `data` corresponds to the value resulting from the communication between `EXE` and `DataBuffer` in Fig. 9(a). sr_i for AADL side is local and not present in HCSP side.

- When $E1'$ executes, the assignments of in_i and cn_{ki} correspond to the communication between `EXE` and lines 10-11 of `EventBuffer`, with $in_i \leftrightarrow$ `head(queue)`, and the other mapping is the same as E1.
- When E2 executes, $dis_i?e$ occurs by receiving a triggering event, corresponding to line 5 of `EXE`, and the other mapping is the same.
- Transition E3 corresponds to the execution of line 8-9 of `EXE`.
- Transition E4 corresponds to the continuous evolution within the domain defined at line 10 of `EXE`, with the mapping $d \leftrightarrow t$, $DL_i \leftrightarrow DL$.
- Transition E5 corresponds to lines 12-13, with mapping $exit_i! \leftrightarrow exit[i]!$.
- Transition E6 corresponds to the continuous interrupt by communication at line 11, and the external choice at line 14, of `EXE`, with mapping $run_i? \leftrightarrow run[i]?$.
- Transition E7 corresponds to lines 15-17, with mapping $c_i \mapsto c$.
- Transition E8 corresponds to lines 20-22, which executes the continuous evolution for time d by preserving the domain, and the variable mapping is obvious.
- Transition E9 corresponds to the interrupt at line 23, before the continuous evolution terminates.
- Transition E10 is an internal action, and together with E11, they correspond to the external choice at lines 31-32.
- Transition E12 corresponds to line 21-22, when c_i reaches Max_i.
- Transitions E13 and E15 correspond to the cases when resource is needed for output, which is defined at line 1-2 of Fig. 7(a), together with the lines 2-3 of Fig. 9(c) if the resource is bus.
- Transition E14 corresponds to lines 4-5 and lines 5-6 of Fig. 7(a, b) depending on whether the resource is needed or not.
- Transition E16 corresponds to line 7 of Fig. 7(b), and followed by this, E17 corresponds to the external choice of lines 9-10.
- Transition E18 corresponds to the continuous evolution of line 36 of `EXE`, and E19 and E20 correspond to line 37-38 respectively.

All the above relations are symmetric, except for one case: in `EXE` line 18-19, the discrete computation is taken and the variable en is set to 1. This transition corresponds to the execution of S/S diagram, which are defined as a sequence of transitions at the bottom of Sect. 5.4.

Scheduler. The HCSP process of scheduler is given by `SCHEDULE_HPF` in Fig. 6(a). The initial state $waitS$ of scheduler corresponds to the start of the external choice on lines 5, 15, and 25, after executing the initialization at lines 2-3 of `SCHEDULE_HPF`. The variables $Pool$ and run_now are common at both sides, while the others are local. In particular, $idle = 1$ is implied by $run\_now = -1$ and $run\_prior = -1$; rdy corresponds to $\_tid$, etc.

- Transition S1 corresponds to line 5, the execution of communication action, with the mapping $rdy \leftrightarrow prior$.
- Transition S2 corresponds to lines 11-13, where $idle$ is local to the AADL side. $idle$ is set to 0, corresponds to that `run_prior` is set to the priority of the requesting thread. Transition S3 corresponds to lines 9-13.
- Transition S4 corresponds to line 7, where thread i is pushed into the waiting $Pool$.
- Transition S5 corresponds to line 15, and transitions S6 and S7 correspond to lines 18-21 and 23 respectively.
- Transition S8 corresponds to the communication action at lines 25-26.

When the above transition goes back to $waitS$ state, it corresponds to another repetition of the body at lines 5-26. All the above relations are symmetric.

Connections. The HCSP process for the bus connection is given by process `BUS` in Fig. 9(c). The initial state $waitB$ corresponds to the starting of `BUS`.

- Transitions B1 and $B1'$ correspond to the prefix events and receiving data/events of lines 7-8 and 2-3, where $data \leftrightarrow in$ and $event \leftrightarrow in$ respectively.
- State res corresponds to the `BLOCK` process on lines 4 and 9. The variable mapping is $\{t_b \leftrightarrow t, L_b \leftrightarrow \texttt{latency}\}$. Starting from this state, transitions B2 and B3 correspond to the interrupt process on lines 4-5 of `BLOCK1` and `BLOCK2` processes respectively.
- Transitions B4 and $B4'$ correspond to the output data and event on lines 5 and 10 respectively.
- Transition B5 corresponds to lines 6 and 11.

All the above relations are symmetric.

By now, all the types of AADL constructs are proved, and Theorem 1 holds. □

Theorem 2 *(Correctness of translation). The translation from AADL⊕S/S to HCSP is correct.*

Proof. By Theorem 1 and the correctness of S/S to HCSP [15], we can prove the correctness of the translation from AADL to HCSP and the translation from S/S to HCSP respectively. Based on this fact, we need to prove that the translation of the combined AADL⊕S/S to HCSP is correct. We consider the two combinations defined in Sect. 5.4. For the translation, both of them are defined as parallel composition of the corresponding HCSP processes after the channel communications among them are set. For the case of continuous evolution, the transitions $(s_1, \sigma) \xrightarrow{cn_a?c} (s_1, \sigma[a \mapsto c])$ and $(s_1, \sigma) \xrightarrow{cn_b!\sigma(b)} (s_1, \sigma)$ correspond to the interrupts to the continuous evolution at HCSP side, i.e. $\langle F(\dot{b}, a) = 0\&\mathbf{true}\rangle \trianglerighteq [\!](cn_b!b \longrightarrow \text{skip}, cn_a?a \longrightarrow \text{skip})$, both of which are ready to occur any time when the AADL side asks. At the semantics side, the two transitions are always enabled whenever needed, corresponding to the HCSP side that the repetition guarantees that the interrupt can occur anytime whenever needed. For the case of discrete computation, the transitions $(s_1, \sigma) \xrightarrow{as?c} (s_2, \sigma[a \mapsto c])$, $(s_2, \sigma[a \mapsto c]) \rightarrow (s_3, \sigma')$ and $(s_3, \sigma') \xrightarrow{bs!\sigma'(b)} (s_1, \sigma')$ correspond to the `Input`, `Discrete Computation` and `Output` parts of the translated HCSP of S/S diagram at HCSP side respectively. For both cases, the relations are symmetric. Furthermore, they are combined with the AADL sides through synchronized communication and parallel composition, thus consistent. The correctness of AADL⊕S/S to HCSP is proved. □

8. A simulation tool for HCSP

In order to test the correctness of the translation from AADL⊕S/S into HCSP given in the previous section, in this section, we describe a new simulator for HCSP with a graphical user interface. Additionally, this allows us to quickly obtain the result of running an HCSP process, in order to check that its behavior is as expected.

While there are non-deterministic elements in HCSP, they are not used often. In particular, the result of translation described in this paper is essentially deterministic. Our aim in the simulator is to compute an execution path of the process and visualize it in a graphical interface. The computation follows closely the small-step operational semantics of HCSP.

The *configuration* of a single process is given by a triple (P, pos, st), where P is the process itself, and is unchanged during the simulation, *pos* is a *program point* in P, and *st* is the state of the process, as a mapping from variable names to values (which can be numbers, strings or lists). Note that according to the semantics of HCSP, the states of processes in parallel are independent from each other.

A program point is a tuple of integers specifying the current location of execution, in the abstract syntax tree of the process. We use this concept rather than modifying the process (as in small-step semantics) for easier visualization. For each construct in HCSP, there is a corresponding definition of moving to the next program point in the construct. Their derivation from small-step semantics is routine, and we omit the details here (the reader can refer to [15]).

8.1. Abstract procedure

The abstract procedure for simulating a parallel of n processes is given in Algorithm 1 and 2.

Algorithm 1 Perform internal steps of a process.

procedure EXEC_PROCESS(pos)
Require: *pos* is the starting position
Ensure: *pos* is at a position where no more internal steps can be performed, *comm* and *delay* specify expected communications and delay.
    while true **do**
        if *pos* at $\langle \varphi \,\&\, B\rangle$ **then**
            $comm \leftarrow []$, $delay \leftarrow$ time φ stay in B
        else if *pos* at $\langle \varphi \,\&\, B\rangle(c_1 \to P_1, \cdots, c_n \to P_n)$ **then**
            $comm \leftarrow \{c_1, \ldots, c_n\}$, $delay \leftarrow$ time φ stay in B
        else if *pos* at $c_1 \to P_1 \,\$\, \cdots \,\$\, c_n \to P_n$ **then**
            $comm \leftarrow \{c_1, \ldots, c_n\}$, $delay \leftarrow \infty$
        else if *pos* at c **then**
            $comm \leftarrow c$, $delay \leftarrow \infty$
        else
            $pos \leftarrow$ perform internal step
        end if
    end while
end procedure

At each iteration, perform *exec_process* on each process, until no more internal steps can be performed. Then *exec_process* returns a list *comm* of communications the process can perform, and a *delay* recording the number of time units the process can wait (which may be infinite). The cases are:

- If the next step is a communication, then *comm* contains the communication, and *delay* is infinite.
- If the next step is an ODE, compute the amount of time before the ODE reaches the boundary (which may be infinite), and set that to *delay*. The *comm* list is empty.

- If the next step is an ODE with communication interrupt, the *delay* is computed as in the previous case. In addition, *comm* is the list of possible interrupts.
- If the next step is an external choice, then *delay* is infinite, and *comm* is the list of communications.

Algorithm 2 Simulate a parallel of processes.

```
procedure EXEC_PARALLEL(hps)
Require: hps is a list of processes in parallel.
    step ← 0
    while step < number of steps do
        for hp in hps do
            hp.comm, hp.delay ← exec_process(hp)
        end for
        if ∃hp_1 hp_2 c. c! ∈ hp_1.comm ∧ c? ∈ hp_2.comm then
            perform communication c
        else if min(hp.delay) ≠ ∞ then
            perform delay min(hp.delay)
        else
            break                                          ▷ deadlock
        end if
    end while
end procedure
```

After all available internal steps are performed, we first check whether there is a matching communication. If there is, perform the communication. If no matching communication is available, we next find the minimum of the delay among all processes. If the minimum is finite (meaning at least one process has a non-infinite delay), then that amount of delay is performed. Here an ODE solver is used to compute numerical solutions to ODEs. If all delays are infinite, we declare that the process has reached a deadlock.

8.2. Implementation

The above procedure is implemented in Python. In addition to real numbers, the state of the system may contain strings and lists. Operations on lists as stack, queue, or priority queue are supported. 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.

9. Case study

In this section, we model an automatic cruise control system using AADL⊕S/S. Then, we use the above framework to translate the model and its several variants into HCSP processes. The resulting HCSP model and its variants are analyzed, simulated and verified.

9.1. A cruise control system

The case study is adapted from the self-driving car system in [14], where it is modeled only in AADL. We extend the model by adding environment and control components modeled in S/S. The architecture is divided into three levels, shown in Fig. 10. The top level is the continuous plant, i.e., the physical vehicle, of the system described by a Simulink diagram. The vehicle receives an acceleration command from the `actuator` and then evolves following an ODE. It outputs the current position to `GPS` whenever required. There are two speed sensors, one is located on the `wheel` and the other uses `laser` technology. If one of them fails, the other can still work to guarantee that the control system gets the real-time speed of the vehicle.

The middle level defines the control of the system. First, data is obtained from sensors, then computation is performed and finally control command is sent to actuators. The process `obs_det.imp` for obstacle detection contains two threads: `img_acq.imp` and `comp_obs_pos.imp`. The thread `img_acq.imp` acquires from a camera raw images of the road ahead and then sends the processed images to the thread `comp_obs_pos.imp` which also receives obstacle information detected by a radar. `comp_obs_pos.imp` then outputs the final position of the obstacle. The image processing of `img_acq.imp` may cause some delay, so its behavior is abstracted as a unit delay (Fig. 11(a)) because the detail of the image processing is not a concern in this case study. The behavior of `comp_obs_pos.imp` is also described by a discrete Simulink diagram (Fig. 11(b)) which combines the two inputs in a conservative way.

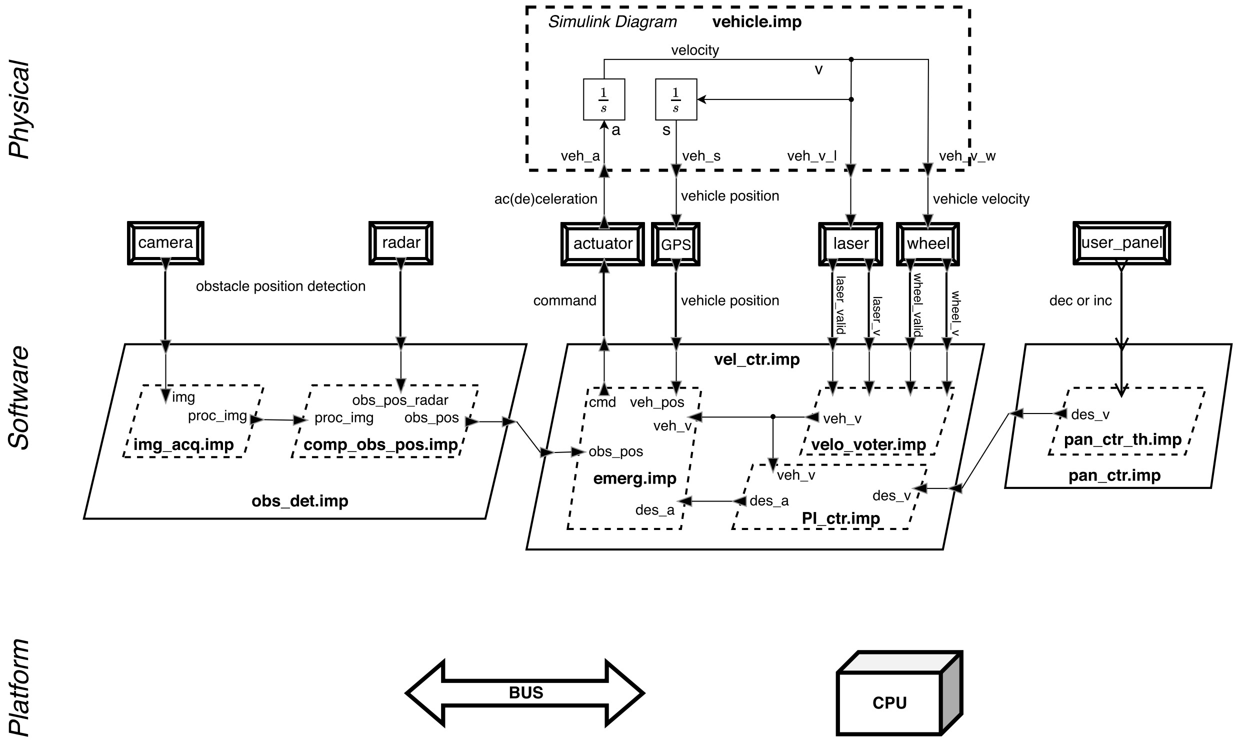


Fig. 10. A cruise control system.

The process `vel_ctr.imp` for velocity control consists of three threads. `vel_voter.imp` is a voter process receiving and combining speed information received from `wheel` and `laser`. Its behavior is modeled by a discrete Simulink diagram (Fig. 11(e)). `PI_ctr.imp` receives the vehicle speed produced by `vel_voter.imp` and a desired speed from the user panel and then computes a desired acceleration. Its behavior is modeled by a discrete PI controller with a wind-up method (back-calculation) (Fig. 11(f)). `emerg.imp` is modeled by a Stateflow diagram (Fig. 11(c)) which receives obstacle position from `obs_det.imp`, vehicle position from `GPS`, vehicle speed from `vel_voter.imp` and the desired acceleration from `PI_ctr.imp`, and computes a command to the `actuator` based on all these inputs. It checks whether the acceleration output by `PI_ctr.imp` 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. `emerg.imp` is the key of the CCS and the detail of its control strategy is specified and verified in Sect. 9.4.

Process `pan_ctr.imp` includes only one thread `pan_ctr_th.imp`. It receives events from device `user_panel`. The driver can control `user_panel` by triggering an event `inc` or `dec` to increase or decrease the desired speed. The behavior of `pan_ctr_th.imp` is modeled by a Stateflow diagram (Fig. 11(d)).

The bottom level of the architecture is the platform consisting of a bus and a processor. All threads are bound to the processor. The scheduling policy of the processor is HPF as introduced in Sect. 6.2. The bus has a latency which is set to 1 ms or 3 ms in the following experiments.

9.2. Translation to HCSP

We now describe the exact settings of parameters used for translation in the experiments. The parameters setting for threads and devices are shown in Table 1. Here MaxET is short for "Maximum Execution Time".

For the devices like `camera`, `radar` and `user_panel`, we use HCSP directly to describe their behaviors for testing. For the following experiments, we vary the number of buses and bus latency. The length of HCSP code in all variants is roughly similar, at about 970 lines.

The AADL⊕S/S model of the CCS of Fig. 10 is stored in a JSON format. The translator takes a JSON file as an input and generates files of HCSP modules as outputs.[2] One could change the configuration of the CCS (such as adding a bus) to obtain variants of the model by modifying the JSON files directly.

[2] All the JSON files and the corresponding programs are stored in the directory `mars/Examples/AADL/CCS/TCS/` of the repository of our toolkit MARS (https://gitee.com/bhzhan/mars.git).

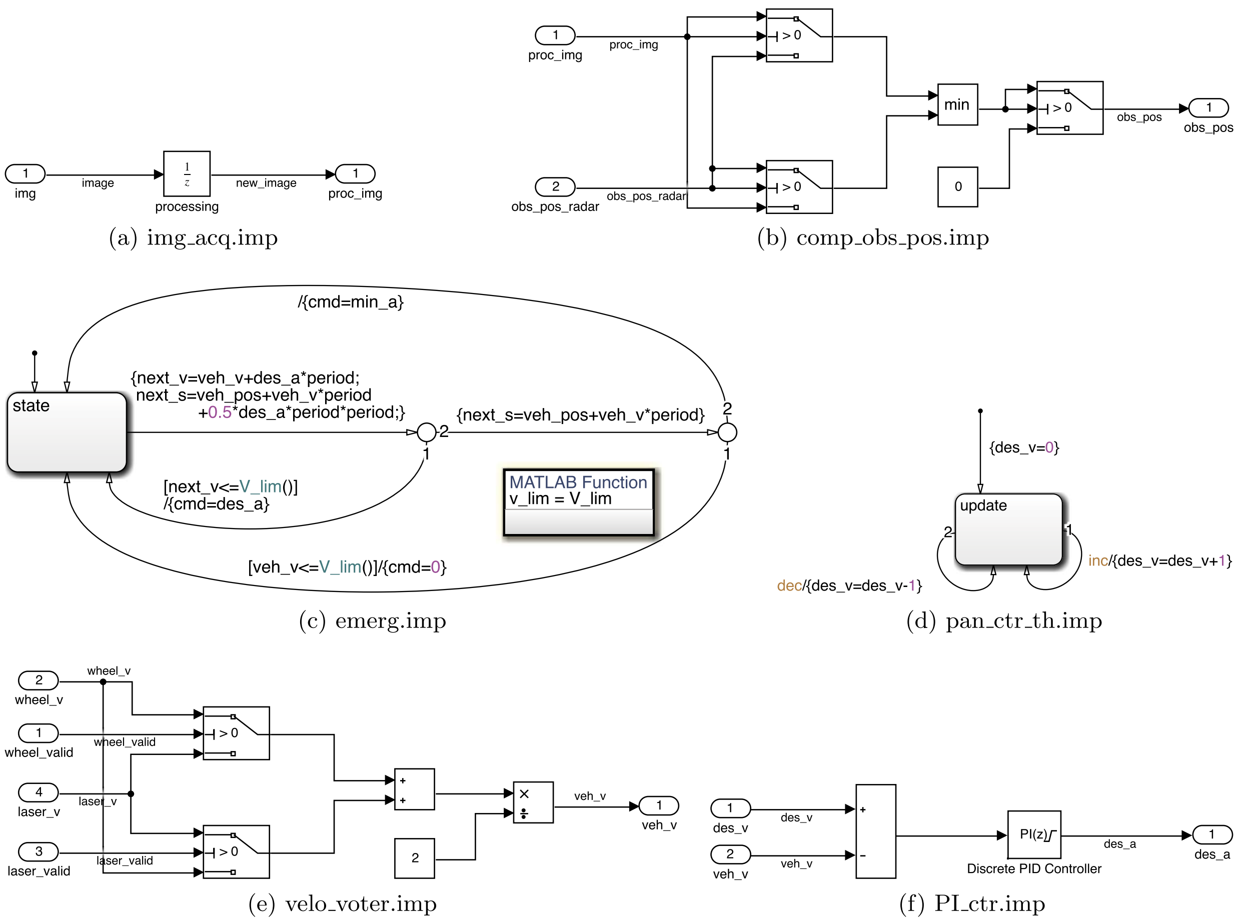


Fig. 11. The Simulink/Stateflow diagrams describing the behaviors of the threads.

Table 1
Parameters of threads and devices in Fig. 10.

thread	priority	period	MaxET	deadline	device	period
`img_acq.imp`	1	45 ms	10 ms	45 ms	`camera`	200 ms
`comp_obs_pos.imp`	1	97 ms	20 ms	97 ms	`radar`	10 ms
`emerg.imp`	2	5 ms	1 ms	5 ms	`actuator`	2 ms
`PI_ctr.imp`	1	7 ms	1 ms	7 ms	`GPS`	6 ms
`vel_voter.imp`	1	8 ms	1 ms	8 ms	`wheel`	10 ms
`pan_ctr_th.imp`	0	–	10 ms	100 ms	`laser`	10 ms
					`user_panel`	–

9.3. 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. In this scenario, `camera` fails to work and thus only `radar` can detect the obstacle. We assume that the obstacle appears at time 10 s and position 35 m, then moves ahead with velocity 2 m/s, before finally moving away at time 20 s and position 55 m. This information is represented by simulated signals received by the `radar`. At the beginning of the simulation, the vehicle is at rest at position 0 m and the driver pushes the `inc` button three times with time interval 0.5 s in between to set a desired speed to 3 m/s. After 30 s, the driver pushes the `dec` button twice in 0.5 s time intervals to decrease the desired speed. We simulate this scenario for 40 s. The results are presented below.

9.3.1. Comparison with AADL⊕S/S2C

First, we test a simplified scenario without bus latency, in order to compare the results with simulation using our tool *AADL⊕S/S2C* (which does not handle bus latency) introduced in Sect. 4. The left of Fig. 12 shows the simulation results of the vehicle speed, where the black line denotes the desired velocity set by the driver, and the dotted and dashed lines

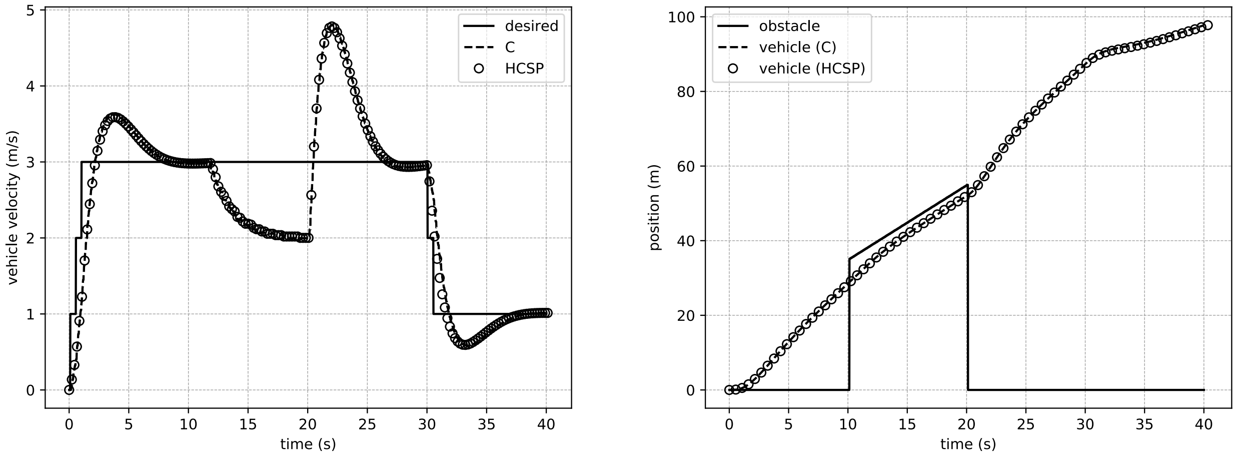


Fig. 12. Comparison of simulations results from HCSP simulator and *AADL⊕S/S2C*.

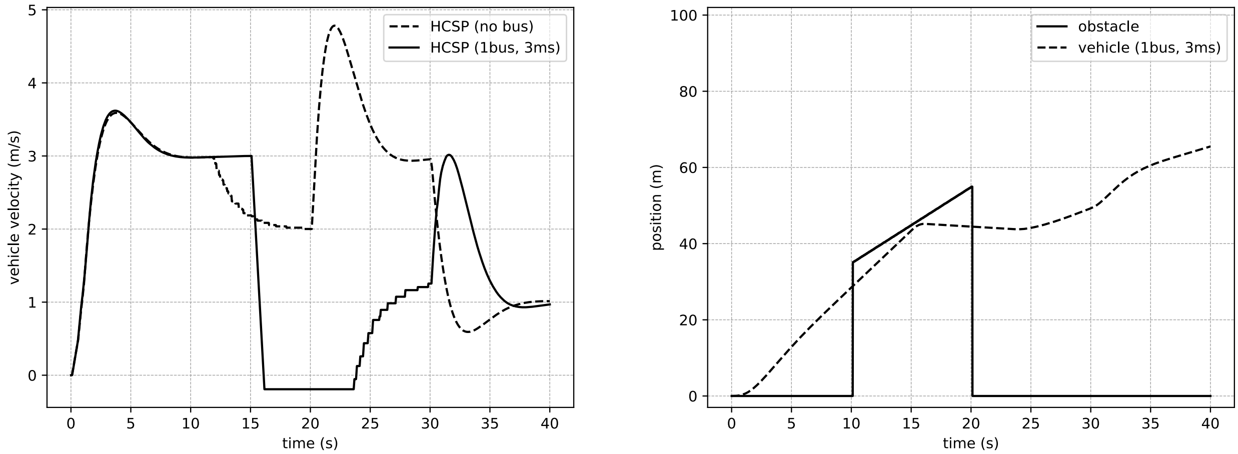


Fig. 13. HCSP simulation results (one bus with the latency 3 ms).

denote simulation results from translation to HCSP and translation to C, respectively. We can see that the two simulation results are very similar. The right figure shows the positions of the vehicle and of the obstacle with respect to time. The vehicle accelerates to the desired speed (3 m/s) in 10 s, and the acceleration is under the control of `PI_ctr.imp`. When `radar` detects an obstacle ahead (10 s), the vehicle still keeps a stable speed for about 2 s because the distance to the obstacle is safe. Then, `emerg.imp` takes control of the speed in order to avoid a collision. When the obstacle moves away at 20 s, `PI_ctr.imp` takes control back and the speed bounces back quickly. After 10 s, `PI_ctr.imp` adjusts the speed to the new desired value, set by the driver.

9.3.2. Analysis of the bus' impact

In order to observe the impact on the system performance caused by bus latency, we restore bus latency to the model, and consider different settings of number of buses and their latency.

From the CCS architecture (Fig. 10), the connections between devices and processes are all bound to one bus, and all the threads in the processes are bound to one processor. We first set the bus latency to 3 ms, and the simulation results are shown in Fig. 13, 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 `radar` can hardly transfer the obstacle position to the process `obs_det.imp` in time. Actually, the delay of the transferring is up to 5 s in this case, which is absolutely intolerable in the real world applications.

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 (3 ms) for `radar`. The connection between `radar` and `obs_det.imp` is bound to this dedicated bus. The simulation result of the vehicle velocity in this case is shown in Fig. 14 (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 5 ms 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.imp`

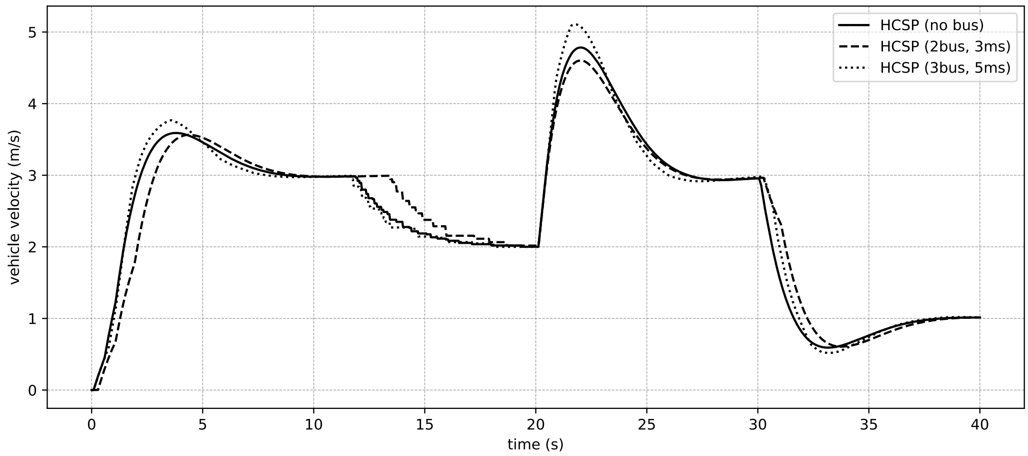


Fig. 14. Vehicle velocity under different bus settings.

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 `emerg.imp` and `actuator` to this bus, and the simulation result returns to normal according to Fig. 14 (dotted line).

9.4. Verification

One of the motivations of translating AADL⊕S/S to HCSP is to verify the informal AADL⊕S/S graphical models. In this case study, we verify the safety property of the simplified CCS using HHL in Isabelle/HOL. Since the original generated HCSP specification of the CCS would be very costly and complicated to thoroughly verify (about 970 lines), we consider an abstract model of the CCS with two main components: a controller (`Control`) and a physical plant (`Plant`), as shown in Fig. 15, and assume the system architecture as trusted computing base.

The process `Plant` models the motion of the vehicle. For initialization, it sends an initial velocity and position and receives the initial acceleration computed by `Control`. It then repeatedly evolve according to an ODE which is interrupted by sending velocity and position, then receiving the updated acceleration data. The process `Control` provides control to the vehicle acceleration based on its velocity and position. The thread `emerg.imp` in the CCS (Fig. 10) takes charge of the control and its behavior is described by a Stateflow diagram of Fig. 11 (c). The control is based on the concept of Maximum Protection Curve (MPC) 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, `Control` 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.imp` (see Fig. 10). Concretely, they can be computed by

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

where *period* is the communication period between `Control` and `Plant`.

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, `Control` continue to test if the constant velocity (no acceleration or deceleration) is safe ($v \leq v_{lim}(s + v \cdot period)$); otherwise, the emergency alerts and `Control` outputs the minimal deceleration ($a_{min} < 0$) to brake the vehicle. The above control strategy can be summarized as

```
module Plant(init_v, init_s):
output v, s, a;
begin
  v := init_v; s := init_pos;
  P2C!v; P2C!s; C2P?a;
  (<s_dot = v, v_dot = s & true> |> [] (P2C!v --> (P2C!s; C2P?a)))**
end
endmodule
```

```
module Control(v_max, a_min, a_des, period, s_obs):
# Compute Maximum Protection Curve (v_lim)
procedure MPC begin
  if s_obs <= 0 then
    v_lim := v_max
  else
    distance := s_obs - s_next;
    if distance > v_max * v_max / (-2 * a_min) then
      v_lim := v_max
    elif distance >= 0 then
      v_lim := sqrt(-2 * a_min * distance)
    else
      v_lim := 0
    endif
  endif
end
# Main process
begin
  (
    P2C?v; P2C?s;
    v_next := v+a_des*period;
    s_next := s+v*period+0.5*a_des*period*period;
    @MPC;
    if v_next <= v_lim then a := a_des
    else  # check if it will be safe when a := 0
      s_next := s+v*period;
      @MPC;
      if v <= v_lim then a := 0 else a := a_min endif
    endif;
    C2P!a;
    wait(period)
  )**
end
endmodule
```

Fig. 15. HCSP processes of `Plant` and `Control`.

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

The safety property can be implied by the loop invariant `loop_inv`: $s \le s_{obs} \wedge v \le v_{lim}(s)$, and we can prove that

$$\texttt{loop\_inv}(s,v) \rightarrow \texttt{loop\_inv}(s',v')$$

where $v' = v + a(s,v) \cdot period$ and $s' = s + v \cdot period + \frac{1}{2} \cdot a(s,v) \cdot period^2$.

According to this loop invariant, the trace assertion of the system which records communications and the system state in continuous time can be proved using HHL, as shown below:

$$\{v = v_0 \wedge s = s_0 \wedge a = a_0 \wedge \texttt{emp}\}$$
$$\texttt{Plant}$$
$$\{\exists a'\ ps.\ \texttt{plant\_end\_state}(v_0, s_0, a', ps) \wedge \texttt{plant\_block}(v_0, s_0, a_0, a', ps)\}$$

The above Hoare triple means that if the plant process starts with velocity v_0, position s_0, acceleration a_0 and an empty trace, it will results in an end state satifying `plant_end_state` and a total trace block satisfying `plant_block`, which specifies the exact behavior of `Plant`, including communications and ODE evolution, when receiving the list ps of acceleration inputs from `Control`.

The Hoare triple below means that if the control process starts with velocity v_0, position s_0, acceleration a_0 and an empty trace, it will result in an end state satisfying `control_end_state` and a total trace block satisfying `control_block` which specifies the exact behavior `Control`, including communications and waiting intervals, on receiving the list cs of pairs of velocity and position from `Plant`.

$$\{v = v_0 \wedge s = s_0 \wedge a = a_0 \wedge \texttt{emp}\}$$
$$\texttt{Control}$$
$$\{\exists v'\, s'\, cs.\ \texttt{control\_end\_state}(v', s', cs) \wedge \texttt{control\_block}(v_0, s_0, a_0, v', s', cs)\}$$

The last Hoare triple means that if the parallel process starts with a parallel state satisfying the loop invariant, it will result in a total trace block satisfying `tot_block` which specifies the exact behavior of two parallel process and declares that it satisfies $\texttt{loop\_inv}(s, v)$ after every iteration.

$$\{(v = v_0 \wedge s = s_0 \wedge a = a_0 \wedge \texttt{loop\_inv}(s_0, v_0)) \uplus (v = v'_0 \wedge s = s'_0 \wedge a = a'_0)\}$$
$$\texttt{Plant} \parallel \texttt{Control}$$
$$\{\exists n.\ \texttt{tot\_block}(s_0, v_0, n)\}$$

10. Related work

Analysis and formal semantics of either AADL or S/S have been explored extensively in existing literature. AADL Inspector is a model processing framework of AADL that encompasses various analysis features, especially including schedulability analysis and dynamic simulation. Cheddar [23] is an open-source real time scheduling tool integrated to AADL Inspector, which implements most classical scheduling simulation algorithms. For formalization of AADL, most work translate it into other formal languages and frameworks. Chkouri et al. translated AADL to the BIP language, and applied it to a model of a flight control system [24]. Hu et al. considered the translation of AADL to Timed Abstract State Machines [25]. Ölveczky et al. presents a formal semantics for AADL in rewriting logic, so the result is executable in Real-Time Maude [26]. The work by Jahier et al. translates AADL into a non-deterministic synchronous model, so the results can be integrated with translation of software components [27].

S/S has a built-in design verifier, which however only supports the verification of discrete behavior. S/S has also been translated to various frameworks for formal verification [28–31,21,22]. In [32], the authors introduce a first-order synchronous dataflow programming language extended with resettable ODEs and hierarchical automata, called Zélus, for which a dedicated type system and causality analysis ensure absence of some bad behaviors such as loops. In the later work [33], a large set of blocks from the Simulink standard library is programmed in Zélus. In contrast to the above cited work, we consider the analysis and formal semantics of combined AADL and S/S models, and therefore able to model, simulate, and verify architecture, functionality and physics of cyber-physical systems at the same time.

Several unified frameworks have also been proposed for modeling and analyzing cyber-physical systems. The most popular is Ptolemy [34], an actor-based framework for the design of heterogeneous systems. It supports different models of computation, which integrate computing, networking, and physical dynamics. It provides the model transformation facility for the analysis and verification of actor models. Functional Mock-up Interface (FMI) [35], a standard maintained by the Modelica Association, is designed to enable the exchange and co-simulation of dynamic component models using a combination of XML files for model description and compiled C-code for simulation. However, Ptolemy supports very limited facilities to model continuous behaviors [36], and both Ptolemy and Modelica are not designed for hardware architecture analysis.

11. Conclusion

This paper presented a combination of AADL and S/S named AADL⊕S/S, and developed a simulation tool for it. Moreover, to verify AADL⊕S/S models, we defined an operational semantics and an HCSP-based denotational semantics for AADL⊕S/S, and proved that there exists a weak bisimulation between the transition system of any AADL⊕S/S model and the transition system of the translated HCSP process. This makes all AADL⊕S/S models can be verified with HHL. In addition, we also developed a simulator for HCSP for testing the correctness of the translation by comparing the simulation results before and after translating, and providing the possibility that one can design a CPS starting with HCSP. We illustrated the framework by considering the case study of a realistically-scaled automatic cruise control system.

There are, however, still some limitations to our approach. First, AADL provides a plenty of components and functions, while we only consider its core functionalities, which limits the practicality of our framework for case-studying realistic CPS models. Second, at present, our verifier only scales to small HCSP models, as means of model abstraction and modular verification would be needed to tackle large-scale models.

To address these limitations, we are considering the following two main directions as future work. First, we would like to further expand the subset of AADL under consideration, by taking into account other dispatch protocols, more complex timing configurations for input and outputs, immediate and delayed timing properties, and other types of hardware components such as memory. Secondly, we will explore ways to verify complex HCSP models using HHL [15]. This requires combining the deductive analysis of different aspects of formal models such as control laws, bus and end-to-end latency, scheduling policies, and so on, into account uniformly.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: None.

Acknowledgements

This research is partly supported by NSFC under grant No. 61625206, 62032024, 61972385 and 61732001, and the Opening Project of Shanghai Trusted Industrial Control Platform. The collaborative work is also partly funded by Inria. We also would like to thank the anonymous reviewers, whose criticisms and suggestions did improve the presentation of our work very much.

References

[1] P.H. Feiler, D.P. Gluch, Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, Addison-Wesley Professional, 2012.
[2] MathWorks Inc., Simulink user's guide, http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf, 2013.
[3] MathWorks Inc., Stateflow user's guide, http://www.mathworks.com/help/pdf_doc/stateflow/sf_ug.pdf, 2013.
[4] H. Zhan, Q. Lin, S. Wang, J.-P. Talpin, X. Xu, N. Zhan, Unified graphical co-modelling of cyber-physical systems using AADL and Simulink/Stateflow, in: UTP, in: LNCS, vol. 11885, 2019, pp. 109–129.
[5] J. He, From CSP to hybrid systems, in: A Classical Mind, Essays in Honour of C.A.R. Hoare, Prentice Hall International (UK) Ltd., 1994, pp. 171–189.
[6] C. Zhou, J. Wang, A.P. Ravn, A formal description of hybrid systems, in: Hybrid Systems, in: LNCS, vol. 1066, 1996, pp. 511–530.
[7] T.A. Henzinger, The theory of hybrid automata, in: LICS, 1996, pp. 278–292.
[8] A. Platzer, Logical Foundations of Cyber-Physical Systems, Springer, 2018.
[9] J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, L. Zou, A calculus for hybrid CSP, in: APLAS, 2010, pp. 1–15.
[10] L. Zou, J. Lv, S. Wang, N. Zhan, T. Tang, L. Yuan, Y. Liu, Verifying Chinese train control system under a combined scenario by theorem proving, in: VSTTE, in: LNCS, vol. 8164, 2013, pp. 262–280.
[11] S. Wang, N. Zhan, L. Zou, An improved HHL prover: an interactive theorem prover for hybrid systems, in: ICFEM, in: LNCS, vol. 9407, Springer, 2015, pp. 382–399.
[12] G. Yan, L. Jiao, S. Wang, L. Wang, N. Zhan, Automatically generating SystemC code from HCSP formal models, ACM Trans. Softw. Eng. Methodol. 29 (1) (2020) 4.
[13] SAE International Standards, Architecture Analysis & Design Language (AADL), Revision C, 2017.
[14] J. Delange, AADL in Practice, Reblochon Development Company, 2017.
[15] N. Zhan, S. Wang, H. Zhao, Formal Verification of Simulink/Stateflow Diagrams: A Deductive Approach, Springer, 2017.
[16] M. Chen, X. Han, T. Tang, S. Wang, M. Yang, N. Zhan, H. Zhao, L. Zou, MARS: a toolchain for modelling, analysis and verification of hybrid systems, in: Provably Correct Systems, Springer, 2017, pp. 39–58.
[17] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, C. Xiao, The Daikon system for dynamic detection of likely invariants, Sci. Comput. Program. 69 (1–3) (2007) 35–45.
[18] J. Liu, N. Zhan, H. Zhao, Computing semi-algebraic invariants for polynomial dynamical systems, in: EMSOFT, ACM, 2011, pp. 97–106.
[19] A.S. Tanenbaum, D. Wetherall, Computer Networks, 5th edition, Pearson, 2011.
[20] S.S. Lam, A carrier sense multiple access protocol for local networks, Comput. Netw. 4 (1980) 21–32.
[21] L. Zou, N. Zhan, S. Wang, M. Fränzle, S. Qin, Verifying Simulink diagrams via a hybrid Hoare logic prover, in: EMSOFT, IEEE, 2013, pp. 1–9.
[22] L. Zou, N. Zhan, S. Wang, M. Fränzle, Formal verification of Simulink/Stateflow diagrams, in: ATVA, in: LNCS, vol. 9364, Springer, 2015, pp. 464–481.
[23] F. Singhoff, J. Legrand, L. Nana, L. Marcé, Cheddar: a flexible real time scheduling framework, ACM SIGAda Ada Lett. 24 (4) (2004) 1–8.
[24] M. Chkouri, A. Robert, M. Bozga, J. Sifakis, Translating AADL into BIP — application to the verification of real-time systems, in: MODELS, in: LNCS, vol. 5421, Springer, 2008, pp. 5–19.
[25] K. Hu, T. Zhang, Z. Yang, W.-T. Tsai, Exploring AADL verification tool through model transformation, J. Syst. Archit. 61 (3–4) (2015) 141–156.
[26] P. Ölveczky, A. Boronat, J. Meseguer, Formal semantics and analysis of behavioral AADL models in real-time Maude, in: FMOODS/FORTE, in: LNCS, vol. 6117, Springer, 2010, pp. 47–62.
[27] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, D. Lesens, Virtual execution of AADL models via a translation into synchronous programs, in: EMSOFT, ACM, 2007, pp. 134–143.
[28] B. Meenakshi, A. Bhatnagar, S. Roy, Tool for translating Simulink models into input language of a model checker, in: ICFEM, in: LNCS, vol. 4260, Springer, 2006, pp. 606–620.
[29] J. Barnat, J. Beran, L. Brim, T. Kratochvila, P. Rockai, Tool chain to support automated formal verification of avionics Simulink designs, in: FMICS, in: LNCS, vol. 7437, Springer, 2012, pp. 78–92.
[30] P. Filipovikj, N. Mahmud, R. Marinescu, C. Seceleanu, O. Ljungkrantz, H. Lönn, Simulink to UPPAAL statistical model checker: analyzing automotive industrial systems, in: FM, 2016, pp. 748–756.
[31] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, F. Maraninchi, Defining and translating a "safe" subset of Simulink/Stateflow into Lustre, in: EMSOFT, IEEE, 2004, pp. 259–268.
[32] T. Bourke, M. Pouzet, Zélus: a synchronous language with ODEs, in: HSCC, ACM, 2013, pp. 113–118.
[33] T. Bourke, F. Carcenac, J. Colaço, B. Pagano, C. Pasteur, M. Pouzet, A synchronous look at the simulink standard library, ACM Trans. Embed. Comput. Syst. 16 (5s) (2017) 176.
[34] J. Eker, J. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, Taming heterogeneity — the Ptolemy approach, Proc. IEEE 91 (1) (2003) 127–144.
[35] T. Blochwitz, M. Otter, et al., Functional mock-up interface 2.0: the standard for tool independent exchange of simulation models, in: International Modelica Conference, 2012.
[36] F. Cremona, M. Lohstroh, D. Broman, E.A. Lee, M. Masin, S. Tripakis, Hybrid co-simulation: it's about time, Softw. Syst. Model. 18 (3) (2019) 1655–1679.

Sponsor names

Do not correct this page. Please mark corrections to sponsor names and grant numbers in the main text.

NSFC, *country*=China, *grants*=61625206, 62032024, 61972385, 61732001

Highlights

- Framework combining AADL and Simulink/Stateflow.
- Formal semantics for the combination with HCSP.
- Justification of the translation from the combination to HCSP.
- Analysis and verification of the combination.
- Application of the approach to a real-world example of automatic cruise control.