Modeling and Verification of Hybrid Systems by Extending AADL

XIONG XU, Institute of Software, Chinese Academy of Sciences, China
EHSAN AHMAD*, Saudi Electronic University, Saudi Arabia
SHULING WANG*, Institute of Software, Chinese Academy of Sciences, China
XIANGYU JIN, Institute of Software, Chinese Academy of Sciences, China
BOHUA ZHAN, Huawei Technologies Co., Ltd., China
NAIJUN ZHAN, Peking University & Zhong Guan Cun Lab., China

System level design, and dependability prediction of safety-critical systems demand integration of architectural and analysis artifacts in a single development environment. Hybrid systems, with mutual dependencies and extensive interactions between the control portion and its physical environment, further intensify this need. Architecture Analysis & Design Language (AADL) is a model-based engineering language for the architectural design and analysis of embedded control systems. Core AADL has been extended with sub-languages for modeling and analysis of discrete behavior of the control portion, but not for continuous behavior of the physical environment. In a previous work, we have introduced Hybrid Annex for continuous behavior modeling as part of initial findings of an ongoing research effort on fulfilling the need for integrated modeling of the computing system along with its physical environment. In this paper, we first detail complete structure of the Hybrid Annex along with appropriate examples for each section. Then, we present formal semantics of the synchronous subset of AADL models annotated with Hybrid Annex specifications using Hybrid Communicating Sequential Processes (HCSP). Formal semantics are used to verify correctness of AADL models (with Hybrid Annex specifications) using Hybrid Hoare Logic (HHL). A case study on a realistically-scaled automatic cruise control system is provided to demonstrate modeling and verification of hybrid systems using AADL with the proposed extension.

CCS Concepts: • **Computer systems organization → Embedded and cyber-physical systems**; • **Theory of computation → Timed and hybrid models**; • **Computing methodologies** → *Modeling and simulation.*

Additional Key Words and Phrases: AADL, behavior modeling, cyber-physical system, formal semantics, HCSP, HHL, hybrid annex, hybrid system

ACM Reference Format:
Xiong Xu, Ehsan Ahmad, Shuling Wang, Xiangyu Jin, Bohua Zhan, and Naijun Zhan. 2025. Modeling and Verification of Hybrid Systems by Extending AADL. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2025), 51 pages. https://doi.org/10.1145/3737698

*Corresponding author.

Authors' addresses: Xiong Xu, National Key Laboratory of Space Integrated Information System, Institute of Software, Chinese Academy of Sciences, Beijing, China; Ehsan Ahmad, e.ahmad@seu.edu.sa, College of Computing and Informatics, Saudi Electronic University, Saudi Arabia; Shuling Wang, wangsl@ios.ac.cn, National Key Laboratory of Space Integrated Information System, Institute of Software, Chinese Academy of Sciences, Beijing, China; Xiangyu Jin, KLSS (CAS) and SKLCS, Institute of Software, Chinese Academy of Sciences, Beijing, China; Bohua Zhan, Huawei Technologies Co., Ltd., Beijing, China; Naijun Zhan, njzhan@pku.edu.cn, School of Computer Science and Key Laboratory of High Confidence Software Technology, Peking University & Zhong Guan Cun Lab., Beijing, China.

1049-331X/2025/1-ART1 $15.00
https://doi.org/10.1145/3737698

1 INTRODUCTION

Embedded systems have become ubiquitous in our daily life with significant impact on automotive, aerospace, consumer electronics, communications, medical, manufacturing, and so on. Such systems make use of computing units to monitor and control physical processes to carry out highly complex, and often critical functions. Correct design and development of these systems is challenging due to a thorough validation and verification activity required to meet expected requirements and to fulfill the quality criteria mandated by any relevant standards. Our reliance on embedded systems is truly based on their dependability, which can only be assured by behavior of the control system under the diverse states of physical environment in which it must operate.

Model-Based Engineering (MBE) has been successfully applied across different engineering disciplines for correctly developing complex embedded systems [37, 43]. MBE starts with defining an initial system model, then extensive analysis and verification are conducted based on the model so that errors can be detected and corrected at early design stages. Afterwards, model transformation techniques are applied to transform abstract formal models into more concrete models, and even into source code. *Hybrid Systems* are models of embedded systems with precise mathematical semantics, wherein continuous dynamics of the physical environment are combined with discrete transitions of the computing units to describe the overall behavior of the system in its environment [6, 17, 36, 46, 47]. Such systems interact with their external environment so as to *monitor* and *control* the *controlled variables* necessary for ensuring correct system functionality.

Hybrid system models consist of several components (with continuous and discrete dynamics) interacting with each other to perform the overall system behavior. For example in transportation systems, medical devices, and industrial plants; the motion of a vehicle, biological cell growth and body temperature, and chemical reaction are modeled by continuous dynamics. The sequence of events in control modes in a moving vehicle, reaction to a particular body temperature in medical devices, and the contacts of valves and pumps in industrial plants, are modeled by the discrete dynamics [45]. Through the use of hybrid models, rigorous analysis and verification of safety-critical embedded systems become feasible.

There are already several modeling languages and tools in the community of MBD for hybrid and cyber-physical systems. We can distinguish two types of modeling approaches: functional and architecture [25]. Functional modeling languages focus on the "what", i.e., what the system is doing, such as SCADE [28] and Simulink/Stateflow [48, 49]), while architecture languages focus on the "how", i.e., how the system provides a service and supports its functions, such as SysML and AADL. Compared with other frameworks (especially the "how" approaches), AADL is excellent in modeling system architecture. Basically, the architecture model is a central development artifact, i.e., the backbone of development process: once we have the architecture model, we can analyze the system, generate the implementation, and derive tests from it.

Architecture Analysis & Design Language (AADL) [29] is now used all over the world and extensively in many other safety-critical domains including automotive manufacturing, healthcare, avionics, and the military. For example, in spite of the fact that the AUTOSAR consortium uses the AUTOSAR modeling language, the SAE supports the exploration of AADL as a standard modeling language for the automobile sector because of its broad analysis support [59].The COMPASS [20] project focused on using AADL for safety analysis, which led the Error Model Annex [1] and the D-MILS [22] project to use AADL for security analysis. The University of Minnesota and Rockwell-Collins participated in many Defense Advanced Research Projects Agency (DARPA) projects [61] and developed a lot of AADL tools, such as AGREE and RESOLUTE. The CMU-SEI has applied the AADL technology to several projects, including SAVI [58], a project that regroups major avionics and aerospace companies and demonstrated the value of MBD for safety-critical software.

In accordance with the principles of system engineering, AADL facilitates the hierarchical assembly of large systems by dissecting them into more manageable sub-components and minimizing mismatched assumptions to guarantee a functional integration. AADL promote Architecture-Centric Virtual Integration (ACVI) for system engineering to cope with the challenges of system integration. When integrating several components or subsystems of various discrete and continuous dynamics, this method is highly helpful for hybrid systems. Although, the component and connection constructs of AADL are only sufficient for modeling the structure of a system architecture, AADL supports extensions to its core language by way of properties and annexes for behavior specification and analysis. The Behavior, and Behavior Language for Embedded Systems with Software (BLESS) annexes were introduced to use state transition mechanisms with guards and actions to model the discrete behavior of control systems [40, 42]. In order to prove correctness, BLESS incorporates a feature that generates verification conditions automatically and facilitates interactive theorem proof using first-order logic. Similarly, DEVS annex, based on discrete-event system specification was introduced for behavior simulation using DEVS-Suite simulator [5].

Although, these annexes precisely specify behavior of software components, they are not designed to model the continuous behavior of the environment with which the engineered system interacts. Hybrid system models endeavor to express behavior of the engineered system together with its environment. In order to fully understand how the behaviors in one domain influence those in the other demands an integrated approach to the modeling of the computing units and the physical environment of the respective domains. To equip AADL for hybrid system modeling and analysis, the core language needs to be extended.

Formal semantics are crucial for formal analysis and verification of safety-critical systems. For formal verification of AADL models, it is required not only to define the formal semantics of the core language but also define the formal semantics of the dedicated annex, used for continuous behavior modeling, in such a language which is designed to model and formally verify the hybrid systems.

1.1 Contributions

This paper is based on our previous works [2, 3, 66] with the extension and originality mainly reflected in the following aspects:

- Some of the preliminary results of furnishing AADL for hybrid system modeling and verification were presented in [2, 3]. This paper contains complete description of the syntax and semantics of each element of Hybrid Annex (HA) sub-language, using appropriate examples.
- We present the translation from (informal) AADL models with HA to (formal) HCSP models which can be analyzed by simulation and verification.
- Simulation of the translated HCSP models of the extended AADL models with HCSP Simulator [66] discussed along with verification through HHL prover [64, 71] —in-house developed Hybrid Hoare Logic theorem provers.
- Additionally, a more realistic case study of automatic cruise control system is used to illustrate use of the AADL with HA sub-language for modeling, simulation, and verification of complex hybrid safety-critical systems.

This paper also presents formal semantics of AADL execution and synchronous communication models using latest HCSP constructs, in complete detail. The translation algorithm is revised to be used for AADL models with multiple threads having multiple connections. Our expression of formal semantics of HA and AADL run-time, using HCSP, illustrates how safety-critical hybrid systems can be modeled and verified in a development environment in both discrete and continuous domains.

1.2 Outline

Section 2 introduces the HCSP, and AADL with its structure and behavior specification mechanism. Section 3 presents a detailed description of HA syntax and semantics, illustrating each language production with a small example modeling different aspect of a hybrid system. In Section 4, a realistically-scaled automatic cruise control system is presented to illustrate the use of the proposed HA. Section 5 describes AADL run-time mechanisms with emphasis on its execution and synchronous communication models and Section 6 introduces their formal semantics with the HCSP, i.e., translating AADL models with HA to (formal) HCSP models. Section 7 presents the simulation and verification of the translated HCSP model of the case study introduced in Section 4 with the aid of the simulator and HHL prover of HCSP. Section 8 presents a summary of the related work, and Section 9 summarizes this paper.

2 BACKGROUND

This section presents the basic AADL notions and notations and an overview of HCSP with its simulator and verification tool.

2.1 AADL

Architecture Analysis & Design Language (AADL), as an architectural description language of embedded systems [40], has successfully been applied in several safety-critical industrial case studies in domains like medical and aerospace engineering. The System Architecture Virtual Integration (SAVI) project for multi-vendor avionics systems has selected AADL as its architecture language [30]. SAVI emphasizes *virtual integration* of architectural components during system engineering, so later actual integration proceeds correctly.

Architectural modeling in AADL is realized through the component specification of both the *application software*, and the *execution platform* it is to run on. Component *Type*, and *Implementation* classifiers, corresponding to system entities, are instantiated and then connected together to form the system architecture model.

Component interface elements, called *ports*, are specified in the *features* section of a type classifier. AADL provides *data*, *event*, and *event data* ports to transmit and receive data (without queuing), control (queuing at the recipient), and combined control and data (queuing at the recipient) signals, respectively. Application software may contain *process*, *thread*, *data*, and *subprogram* components to represent shared memory space, control flow, local data sub-components, and the callable code. The execution platform is made up of computation and communication resources, primarily consisting of *processor*, *memory*, *bus*, and *device* components to represent the hardware and software responsible for thread scheduling and execution, code and data storage, physical connections among components, and interactive components like actuators and sensors.

AADL provides *system* components to model the composition of the software, and execution platform components, and *abstract* components to model interfaces without further elaboration. Open Source AADL Tool Environment (OSATE) [53] is a development environment built on Eclipse to implement AADL for modeling and analysis of real-time embedded systems. OSATE not only provides full-features text editor and a set of analysis plug-ins, but also supports domain-specific analysis plug-in development.

AADL is extendable, and additional sub-languages for modeling and analysis can be added through its *annex* mechanism. The component and connection constructs of AADL are sufficient for modeling the structure of a system architecture. However, development of dependable systems requires detailed behavior modeling, which AADL lacks. The Behavior (BA), and BLESS annexes [40, 42] were introduced to address this shortcoming. They both use state transition mechanisms with

guards and actions to model the discrete behavior of control systems. BLESS provides a declarative behavior interface specification language, an action language for defining subprogram behavior, and a state transition machine language for defining reactive behavior. To prove that all executions of BLESS programs uphold their specifications, a BLESS proof engine automatically generates verification conditions, and transforms programs having proof outlines into complete, formal proofs. Even with addition of the BA and BLESS annexes, AADL still lacks continuous behavior modeling, so nothing related to the physical portion of a hybrid system can be modeled or verified. Hybrid Annex (HA) was introduced in [3], for specifying the continuous behavior of model components, and the cyber-physical interaction. Some of the preliminary results of furnishing AADL for hybrid system modeling and verification were also presented in [2].

2.2 HCSP

Hybrid Communication Sequential Processes (HCSP) is an extension of Hoare's Communicating Sequential Processes (CSP) [38] for hybrid systems [35, 72] . In HCSP, differential equations are introduced to model continuous evolution of the physical processes (in the physical environment) along with interrupts. A hybrid system in HCSP is a parallel composition of networked sequential processes interacting through dedicated channels, or a repetition of a sub-system. Processes in parallel can only interact through communication, and no shared variables are allowed.

2.2.1 Syntax. The processes of HCSP are constructed as follows:

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

Here $\mathtt{P}$, $\mathtt{P}'$, and $\mathtt{P}_i$ represent sequential processes, whereas $\mathtt{S}$ and $\mathtt{S}'$ stand for (sub)systems, ch and ch_i are communication channels, while $ch_i \square_i$ is a communication event which can either be an input event $ch?x$ or an output event $ch!e$, B and e are boolean and arithmetic expressions respectively, and d is a non-negative real constant.

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

HCSP supports both sequential and concurrent composition. A sequentially composed process $\mathtt{P} \fatsemi \mathtt{P}'$ behaves as $\mathtt{P}$ first, and if it terminates, as $\mathtt{P}'$ afterwards. The alternative process $B \longrightarrow \mathtt{P}$ behaves as $\mathtt{P}$ only if B is true and terminates otherwise. We can then define the conditional

$$
\mathtt{if}\ B\ \mathtt{then}\ \mathtt{P}\ \mathtt{else}\ \mathtt{P}' \quad \triangleq \quad f := 0 \fatsemi B \longrightarrow (f := 1 \fatsemi \mathtt{P}) \fatsemi (f = 0 \wedge \neg B) \longrightarrow \mathtt{P}'
$$

where f is a fresh variable indicating whether the branch corresponding to B being true is taken.

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

Continuous evolution is specified as $\langle \dot{\boldsymbol{s}} = F(\boldsymbol{s}) \& B \rangle$. Real variables $\boldsymbol{s}$ evolve continuously according to differential equations F as long as the boolean expression B is true. In HCSP, continuous evolution can be preempted due to the following interrupts:

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

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

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

In order to support modularity, we extend HCSP to include procedures and modules. A procedure definition follows the syntax **proc** $p = \mathsf{P}$, where P is a sequential process. The call to p is equivalent to executing process P. A module definition follows the syntax **module** $m(\overline{T\ x}) = \{\overline{proc}\ \mathsf{P}\}$, where $\overline{T\ x}$ is a list of parameters, $\overline{proc}$ is a list of procedure definitions and P is a sequential process including the call to these procedures. The module execution $m(\overline{e})$ is equivalent to executing P by instantiating $\overline{x}$ by $\overline{e}$.

2.2.2 Simulator. A simulator for HCSP with a graphical user interface is introduced in [66], which allows us to quickly obtain the result of running an HCSP process, in order to check that its behavior is as expected. The simulator, implemented in Python, is customized for HCSP programs. 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 Ordinary Differential Equations (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. 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.

2.2.3 HHL Prover. HHL Prover is an interactive prover implemented in Isabelle/HOL. In this prover, we formalize the semantics of HCSP and the synchronization function of traces. For single process, we give the partial correctness lemmas of hybrid Hoare logic with different HCSP commands. Based on corresponding library of Ordinary Differential Equation, we give various rules of proving differential invariant for continuous evaluations. To deal with parallel process and the handshake of communications, we provide elimination rules for assertions on synchronized traces. With the help of these lemmas, we can perform formal verification of the HCSP system and prove the properties we need.

3 HYBRID ANNEX

This section presents the constructs of the Hybrid Annex (HA), an extension to AADL for hybrid system modeling and verification. HA was briefly introduced in [3], here we described each HA section in detail with its syntax, and grammar with appropriate examples. HA facilitates modeling of the physical, real-world elements, or processes, that the system must interact with to achieve its goals of monitoring and controlling one or more of those processes. For continuous behavior modeling of sensors and actuators, HA sub-clauses are used within AADL *device* component implementations. Continuous behavior modeling of physical processes/environment is achieved through specifications within AADL *abstract* component implementations. Modeling hybrid systems involves both discrete and continuous dynamics. The AADL core language does not provide continuous behavior modeling, but it does have hardware and software components for discrete dynamics modeling. To model this missing component of a hybrid system modeling—the physical process/environment—we employ the Abstract component category.

HA is expressive enough to model physical processes with complex continuous dynamics attached to AADL ports and mapped with AADL connections. In this paper, we present an updated version of HA to accommodate HHL for assertion and invariant specification for both continuous and discrete behaviors.

Below we explain the Extended Backus-Naur Form (EBNF) of the HA grammar, in which: literals are printed in **bold**; alternatives are separated by a pipe |; groupings are enclosed with parentheses ( ); square braces [ ] delimit optional elements; and the closures { }+ and { }* are used to signify one-or-more, and zero-or-more of the enclosed element, respectively. Following is the grammar of the HA sub-clause:

```
hybrid_annex ::=
    [ assertion  { Assertion }+ ]
    [ assume     { assume_declaration }+ ]
    [ ensure     { ensure_declaration }+ ]
    [ invariant    invariant_declaration ]
    [ variables  { variable_declaration }+ ]
    [ constants   constant_declarations ]
    [ channels { channel_declaration }+ ]
      behavior { behavior_declaration }+
```

Here, **assertion**, **assume**, **ensure**, **invariant**, **variables**, **constants**, **channels**, and **behavior** are the sections of an HA sub-clause, each of which is dedicated to specify particular aspect of a detailed behavior model.

3.1 Assertion Section

In addition to detailed (continuous and discrete) behavior modeling, HA also accommodates behavior constrains specification as assertions. Assertions are used to express constraints on any HA defined behavior. The **assertion** section may declare assertions either for later inclusion in the **invariant** section, thereby making a more concise invariant, or in **assume** and **ensure** sections for expressing pre- and post-conditions at the *component level* or along with an HCSP process (as assume, ensure, or invariant at the *process level*) in the **behavior** section. An assertion is a first-order logic formula enclosed between << and >>.

Following is the grammar of the **assertion** section. The italicized prefix for identifiers gives "hints" about the kind of identifier.

```
assertion_declaration ::=
    { referenced_assertion }+
  referenced_assertion ::=
    « assertion_identifier : predicate »
  predicate::=
    UQ_PredicateExpression |
    EQ_PredicateExpression |
    PredicateExpression
```

An assertion declared as a *referenced_assertion* has a unique identifier which can be used for reference in the **assume**, **ensure**, **invariant**, and **behavior** section. HA supports universal quantification specified as *UQ_PredicateExpression* and existential quantification specified as *EQ_PredicateExpression*.

3.2 Assume Section

Assertions defined in the **assume** section specify the constraints imposed by the component on its environment. These are the claims that any initial state of the component should satisfy.

3.3 Ensure Section

Behavior constraints specified in the **ensure** section must be satisfied by every terminating state of the component starting from an initial state satisfying the constraints specified in the **ensure** section, given that the behavior terminates.

3.4 Invariant Section

Behavior constraints defined in the **invariant** section which stipulates the property should be satisfied throughout the behavior execution of the component. Note that there is only one invariant, but it can be logically complex, having as many assertions as needed.

Below is the grammar of the **assume**, **ensure**, and **invariant** sections.

```
assume_declaration |
ensure_declaration |
invariant_declaration ::=
    { assertion_identifier | inline_assertion }+
  inline_assertion ::=
    « predicate »
  predicate::=
    UQ_PredicateExpression |
    EQ_PredicateExpression |
    PredicateExpression
```

Behavior constraints specification at component (in **assume**, **ensure**, and **invariant** sections) and HCSP process level forms an HHL triple, an extension of Hoare Logic for hybrid systems [44], as shown below. Here, P is an HCSP process.

$$\{assume\}\ P\ \{ensure\,;\,\lceil invariant\rceil\}$$

If behavior of the component is modeled by several sequential processes in parallel, say two, i.e., P_1 and P_2, each of which with its own local assume, ensure and invariant, i.e.,

$$\{assume_i\}\ P_i\ \{ensure_i;\lceil invariant_i\rceil\}$$

then, the corresponding HHL triple is a conjunction of the global and local assume, ensure, and invariant, as given blow.

$$\{assume_1 \wedge assume, assume_2 \wedge assume\}\ P_1 \parallel P_2\ \{ensure_1 \wedge ensure, ensure_2 \wedge ensure\,;\ \lceil invariant_1 \wedge invariant\rceil, \lceil invariant_2 \wedge invariant\rceil\}$$

Below is the assertion specification of an (overly) simplified train. It shows the **assertion**, **assume**, **ensure**, and **invariant** sections used to specify behavior constraints of an AADL abstract component `SimpleTrain`. All the variables used are declared in the **variables** section (not shown here).

Assertion with label `SBL` is a disjunction of `[s=CTCS_Properties::start]` and `(v < iSeg_v2)`, where `s` is current position of the train, `start` is a property declared in a property set `CTCS_Properties` to represent the starting position of the train, `v` is the current velocity of the train, and `iSeg_v2` is the

```
abstract implementation SimpleTrain.impl
 annex hybrid {**
  assertion
    << SBL: [s=CTCS_Properties::start] or [v < iSeg_v2] >>
    << EBL: [v < iSeg_v1] >>
    << DSPV2: [s=CTCS_Properties::start] or [(v^2)+(2*b*s)
< (nSeg_v2+(2*b*iSeg_e))] >>

  assume
    << [s=0] and [v=0] and [t=0] >>

  ensure
    << [v>=0] or [a>=0] or ([t>Temp] and [t<Tdelay]) >>

  invariant
    <<  SBL and EBL and DSPV2  >>
 ...

  behavior
    Move ::=  ' DT 1 s = v ' & ' DT 1 v = a ' & ' DT 1 t = 1 '

**};
end SimpleTrain.impl;
```

service brake limit of the current track segment `iSeg` (track of a train is subdivided into several distance segments for controlling the movement authorization). This assertion defines the service brake limit for a train. It states that the train is either not moving (its position is at the starting point) or its current velocity must be less than the service brake limit (`v2`) of the current track segment, if it is moving in segment `iSeg`.

The second assertion with label `EBL` defines an emergency brake limit for the train by describing that whenever the train is moving in a particular track segment `iSeg`, its current velocity `v` must be less than the emergency brake limit `v1`.

The third assertion with label `DSPV2` specifies that during operation the train is either at the start position or its speed is less than dynamic speed profile of the current segment. It is a disjunction of two terms: `[s=CTCS_Properties::start]`, and `[(v^2)+(2*b*s) < (nSeg_v2+(2*b*iSeg_e))]`. Here, `b` is the maximum deceleration when emergency brake is applied. Variable `iSeg_e` represents the end of the current segment `iSeg` and variable `nSeg_v2` represents the dynamic speed profile of the next segment `nSeg`.

Behavior constraint `<< [s=0] and [v=0] and [t=0] >>`, specified on line 9, in **`assume`** section describes the pre-condition that must be satisfied by the environment for the execution of the abstract component `SimpleTrain`. It states the initialization of variables `s`, `v`, and `t`. Behavior constraint `<< [v>=0] or [a>=0] or ([t>Temp] and [t<Tdelay]) >>` on line 12, in **`ensure`** section, specifies the post-condition that must be true after the execution of `SimpleTrain`. This disjunction describes that either the train is moving forward (`[v>=0] or [a>=0]`) or the sampling time did not expire (`[t>Temp] and [t<Tdelay]`).

The **`invariant`** section contains a conjunction of three assertions `SBL`, `EBL`, and `DSPV2`. It specifies that during operation the speed of the train must be less then the service `SBL` and emergency brake `EBL` limits and the dynamic speed profile of the current segment must follow a certain pattern of deceleration (based on track condition of the next segment.) This conjunction must be true

throughout the execution of `SimpleTrain` component. Process `Move` on line 19, defined in **`behavior`** section, contains time derivation of `s`, `v`, and `t` to specify train movement. Processes in the **`behavior`** section are further discussed in Section 3.7.

3.5 Variables Section

Local variables in the scope of an HA sub-clause are declared in the **`variables`** section along with their data types. Data types are assigned to variables by classifier references to the appropriate AADL (user definable) data components. Following is the grammar of the **`variables`** section:

```
variable_declaration ::=
    variable_identifier
    { , variable_identifier }* :
    data_component_classifier_reference
```

The referred external data component must either be part of the package containing the component being annotated, or must be declared within the scope of another package that has been imported using the AADL **`with`** clause. Following example shows the use of the **`variables`** section to declare different types of variables.

```
...
annex hybrid {**
  ...
  variables
    t_clk, c_clk : CTCS_Types::Time
    speed : Base_Types::Float
    counter : Base_Type::Integer
  ...
**};
```

Variables `t_clk`, and `c_clk` are of type `Time` while variable `speed` is of type Float. Variable `counter` is of type `Integer`. Data components `Float`, and `Integer` are defined in an AADL package `Base_Types` while data component `Time` in defined in another AADL package `CTCS_Types`.

3.6 Constants Section

Constants in the scope of an HA sub-clause are declared in the **`constants`** section. Adhering to standard convention: constants can only be initialized at declaration, and cannot be assigned another value afterwards. A constant must be initialized with either an integer, or a real value, and may include a description of its unit of measure. HA also supports specification of measuring units. Common constants like the mathematical ratio π and the physical gravitational attractive force g can easily be declared as `pi = 3.14159` (no units) and `g = 9.81 mpss` (*meters per second*^{2}), respectively. The grammar of the **`constants`** section is as follows:

```
constant_declarations ::=
   behavior_constant
   {, behavior_constant }*
behavior_constant ::=
   behavior_constant_identifier =
   ( integer_literal | real_literal | boolean_literal )
   [ unit_identifier ]
```

Following example shows the use of **`constants`** section to declare different types of constants.

```
...
annex hybrid {**
  ...
  constants
    pi = 3.14159, g = 9.8 mpss,
    u = 1,
    r = 18 cm
  ...
**};
```

Constants `pi`, and `u` are declared with values `3.14159`, and `1`, without any measuring units. Constants `g` is declared with value `9.8` and measuring unit `mpss` for *meters per second*^{2}, while constant `r` is declared with value `18` and measuring unit `cm` for *centimeter*.

3.7 Behavior Section

The **behavior** section of the HA sub-clause is used to specify the continuous and discrete behavior of an AADL component in terms of concurrently-executing HCSP processes. Behavior specification has process declarations, which in turn, may contain several predefined processes by different constructs (sequential, concurrent, repetitive, etc.).

The process algebra notation that models hybrid system behavior is shown below, with the ampersand sign **&** acting as a separator having no semantics.

```
behavior_declaration ::=
   behavior_identifier ::=
     [ assume ] process_declaration { & process_declaration }*
     [ ensure ] [ ; { invariant } ]
process_declaration:
   stop | skip | wait time_value | assignment
   | boolean_assignment | sequential_composition
   | concurrent_composition | choice | repetition
   | continuous_evolution | communication
```

Assertions before and after a behavior process specification, specified as *assume* and *ensure* respectively, represent pre- and post-conditions while the *invariant* represent the process invariant as explained in Section 3.4. The process invariant must be true throughout the execution of the process it is specified for.

Below, we explain each of these algebraic notation which may constitute a continuous behavior individually or in combination with other notations.

3.7.1 Stop process. The **stop** process does nothing but keeps idle for ever.

3.7.2 Skip process. The **skip** process terminates immediately having no further effect on variable values. It is used to model the successful termination of the current execution.

3.7.3 Wait process. The **wait** keeps idle for a specific time value. During this idle period, the respective process does not perform any action and the variables are unchanged. Below is the syntax of the wait process:

```
time_value ::=
 time_variable_identifier | real_literal time_unit
```

The time variable identifier can be a local variable identifier declared in the `variables` section. It can also be a constant identifier declared in the `constants` section. The time unit defines a unit of measurement of time and can be any time unit declared in the *Time_Units* enumerated property set within the project specific property set in AS5506D [40]: **ps**, **ns**, **us**, **ms**, **sec**, **min**, **hr**.

3.7.4 Assignment process. The assignment process assigns the value of an expression to a local variable declared in the `variables` section. Grammar of the assignment process is as under:

```
assignment ::=
   variable_identifier := numeric_expression
```

3.7.5 Boolean assignment process. The boolean assignment process assigns the boolean value to a local boolean type variable declared in `variables` section.

```
boolean_assignment ::=
   Boolean_variable_identifier := boolean_expression
```

3.7.6 Sequential composition. HA supports both sequential and concurrent composition. Sequential composition defines consecutively-executing processes. Below is the grammar of the sequential composition.

```
sequential_composition ::=
   { behavior_identifier { ; behavior_identifier }+ }
```

A sequentially composed process {`P ; Q`} behaves as `P` first and after its successful termination, behaves as `Q`. Behavior identifiers used in sequential composition must refer to behavior declarations.

3.7.7 Concurrent composition. A concurrently composed process {`S1 || S2`} behaves as if `S1` and `S2` run independently except that all interactions occur through communication events. Communication events between concurrently-composed behaviors, must occur along common communication channels declared in the `channels` section, connecting processes `S1` and `S2`. Below is the grammar of the concurrent composition.

```
concurrent_composition ::=
   { behavior_identifier { || behavior_identifier }+ }
```

Behaviors defined using concurrent composition may not themselves be used in either sequential or concurrent compositions. In concurrent compositions, communication channels (explained in Section 3.8) must be shared pair-wise with complementary directions—*in* communication with *out* communication. Variables used in common communication channels must have the same type, and more than two processes can take part in a particular concurrent composition. Concurrently executing processes `S1` and `S2` can neither share variables, nor input or output channels, hence

$$(\mathcal{V}_{S1} \cap \mathcal{V}_{S2} = \emptyset) \wedge (\Sigma_{in}(S1) \cap \Sigma_{in}(S2) = \Sigma_{out}(S1) \cap \Sigma_{out}(S2) = \emptyset)$$

where $\mathcal{V}$ is a set of variables while Σ_{in} and Σ_{out} are the set of input and output channels, respectively.

3.7.8 Choice process. Internal execution choice between processes P and Q, denoted as P [] Q is resolved by the process itself. Following is the grammar of the choice process:

```
choice ::=
   alternative { [] alternative }*
alternative ::=
   ( boolean_expression ) -> process_identifier
```

Choice executes a process with an alternative having true boolean expression. When more than one alternatives have true boolean expressions, the choice is resolved non-deterministically. When no alternative has true boolean expression the choice is equivalent to **skip**. The alternative process (b)->P behaves as P only if the boolean expression b is true and terminates otherwise.

3.7.9 Repetition. The repetition executes a process for a finite number of times. Syntax of the repetition is define below, where REPEAT is a reserved word:

```
repetition ::=
   REPEAT [ [ (integer_literal
   | integer_variable_identifier) ] ]
   ( process_identifier )
```

A variable identifier used in a REPEAT statement must refer to an integer value. Repetition can be of following three type:

- The statement REPEAT (P) causes process P to be repeated a finite, unspecified number of times.
- The statement REPEAT [5](P) causes process P to be repeated five times.
- The statement REPEAT [n](P) causes process P to be repeated the value of integer variable n times.

3.7.10 Continuous evolution. Behavior of a physical, controlled variable of a hybrid system is specified by continuous evolution. The semantics of continuous evolution are: a differential equation holds when its (optional) boolean expression evaluates to true, and until its (optional) interrupt occurs.

The continuous evolution statement forces values of variables declared in the variables section to follow differential equations as long as an optional boolean expression is true. The boolean expression specifies boundary condition of the variables. Continuous evolution terminates as soon as the boolean expression turns to false. Interruption of continuous evolution due to boundary condition is known as *boundary interrupt*. Continuous evolution can also be preempted due to *timed*, and *communication interrupts*, as presented in the Section 3.7.11 in relation with cyber-physical interaction modeling.

Below is the grammar of the continuous evolution:

```
continuous_evolution ::=
   'differential_expression = differential_expression'
   [ < boolean_expression > ]    [ interrupt ]
```

Differential expression: Differential expressions consist of several differentials combined using the usual multiplication (∗), addition (+), and subtraction (−) operators. Differentials may be numeric

literals, a variable optionally raised to a numeric literal power, a derivative w.r.t time, a partial derivative, or a parenthesized differential expression.

```
differential_expression ::=
   differential
   | differential { * differential }+
   | differential { + differential }+
   | differential - differential
differential ::=
   numeric_literal
   | variable_identifier [^ numeric_literal ]
   | derivative_expression
   | derivative_time
   | ( differential_expression )
```

Derivative expression: A partial derivative expression is indicated using the keyword `DE` followed by the order of the differential equation, then the dependent variable, and finally the independent variable. For example, the rate of change of variable y with respect to x, denoted $\frac{dy}{dx}$, a first order equation, is specified as `' DE 1 y x '`, while the second order equation $\frac{d^2y}{dx^2}$ is specified with `' DE 2 y x '`. Here, `'` is the delimiter at the start and end of each differential expression.

```
derivative_expression ::=
   DE order_integer_literal   dependent_variable_identifier
   independent_variable_identifier
```

Derivative time: A similar notation is defined for derivatives w.r.t. time–a frequently encountered concept in real-time systems. Here the keyword is `DT`, and the independent variable, always being time, is not needed. Thus, the rate of change of y with respect to time t, $\frac{dy}{dt}$, is specified as `' DT 1 y '`.

```
derivative_time ::=
   DT order_integer_literal variable_identifier
```

3.7.11 Communication. Communication between physical processes uses the channels declared in the `channels` section of the respective behavior specifications, while communication with other AADL components relies on the ports that are declared in the component's type. HA accommodates assignment and communication of continuous variables, as a result analog-to-digital and digital-to-analog converters can easily be modeled as communication events.

```
communication ::=
  port_communication | channel_communication
port_communication ::=
   port_identifier (?|!)
   ( [variable_identifier] )
```

```
channel_communication ::=
   channel_identifier (?|!)
   [ variable_identifier ]
```

Channel communication synchronizes involved processes and can only occur when both sender and receiver are ready and may cause one of them to wait. Port communication has semantics of AADL core language. Communication events are of two types: *input* event and *output* event. A port input event `p1?(x)` specifies that an input value is received from port `p1` and stored in a local variable `x`. A port output event `p2!(y)` specifies that an output value of variable `y` is sent out through port `p2`. A channel input event `ch1?z` specifies that an input value is received from channel `ch1` and stored in a local variable `z`. A channel output event `ch2!w` specifies that an output value of variable `w` is sent out along channel `ch2`.

Interrupts: Continuous process evolution may be terminated after a specific time, or on a communication event. These are invoked through timed, and communication interrupts, respectively. A timed interrupt preempts continuous evolution after a given amount of time whereupon the process then assumes the behavior specified by the interrupt. A process with continuous evolution, boundary interrupt, and a timed interrupt, continues its evolution if it terminates due to boundary interrupt before time value, time units. Otherwise, after a specific time value, the process behaves like the next specified process.

A communication interrupt preempts continuous evolution whenever communication takes places along any one of the named ports or channels. A process with continuous evolution, boundary interrupt, and a communication interrupt continues its evolution except that the continuous evolution is interrupted whenever any communication event takes place along any channel or port.

```
interrupt ::=
  timed_interrupt | communication_interrupt
timed_interrupt ::=
  [> time_value ]> { behavior_identifier }+
  time_value ::=
   time_variable_identifier | real_literal time_unit
communication_interrupt ::=
   [[> port_or_channel_identifier ~> process_identifier
   { , port_or_channel_identifier ~> process_identifier }*   ]]>
```

For timed interrupt, as soon the time value expires, the process behaves as specified after `]>`. For communication interrupt The communication event can either be an input or an output event. As soon as the communication event takes place, the process behaves as the next process specified right after `~>`.

Following code snippet illustrates continuous behavior specification using ordinary and partial differential equations attached to the implementation classifier of an abstract component `AlphaTest`. Behavior processes in this example are not related to each other.

```
abstract implementation AlphaTest.impl
 annex hybrid {**
 ...

  variables
   s, v, x, a, y, z, u : Base_Types::Float
   t : CTCS_Types::Time
```

```
  constants
   c = 0.0123, alpha = 19 mmsps

  behavior
   Train ::= 'DT 1 s = v' & 'DT 1 v = a' & 'DT 1 t = 1'
   WE ::= 'DT 2 y = (c^2)  * DE 2 y x'
   HE ::= 'DT 1 u - (alpha * ( (DE 2 u x) + (DE 2 u y) + (DE 2 u z) )) = 0'

**};
end AlphaTest.impl;
```

All the required variables and constants are declared in respective sections with appropriate data types, and initial values with measuring units.

Process `Train`, on line 13, specifies of motion of a (simplified) train; where `s` is the displacement, `v` is the velocity, `a` is the acceleration, and `t` is the train clock time.

On line 14, process `WE` specifies wave behavior in a one dimension space. Where `c` is a constant and is declared in **`constants`** section with value `0.0123`.

Process `HE`, on line 15, specifies temperature change in a 3D space. Where `alpha` is the thermal diffusivity of the material or substance in use, `u` is the temperature, and `x`, `y`, `z` are the dimensions. For air (the substance used in this example) at 300°K, alpha is $19mm^2/sec$, specified as `19 mmsps` in the **`constants`** section.

3.8 Channels Section

A computing unit's extensive interactions with, and strong dependence on its physical environment makes precise specification of the system's cyber-physical interaction (communication between computing units and their physical environment) an essential part of hybrid system modeling.

```
channel ::=
   channel_identifier {, channel_identifier }* :
   data_component_classifier_reference
```

HA channels only support unidirectional communication and a process may not use the same channel for both input and output communication events. Channel declarations contain AADL data component classifier references to specify type of the data sent or received along a particular channel. Extensive support for interaction and continuous evolution preemption due to timed and communication interrupts is a major contribution of the HA.

4 RUNNING EXAMPLE: AN AUTOMATICALLY CRUISE CONTROL SYSTEM

In this section, we introduce an automatic cruise control system (ACCS) as the case study to illustrate the whole procedure of modeling, simulation, and verification. The ACCS captures pictures while the car is operating in order to detect obstacles on the road. Obstacle detection and velocity control are critical functions that must operate in a real-time, deterministic fashion. There are two basic requirements for the system:

Hard The car must not collide onto the obstacle ahead;

Soft The evolution of the car should follow the driver's intent as much as possible.

This example is adapted from the self-driving car system in [25], where it is modeled only in AADL, and then extended in [66] by adding physical environment and control components modeled in Simulink/Stateflow. In this paper, we modify the architecture of the ACCS in [66] and the behavior of the AADL components in the system is now described by the HA proposed in Section 3.

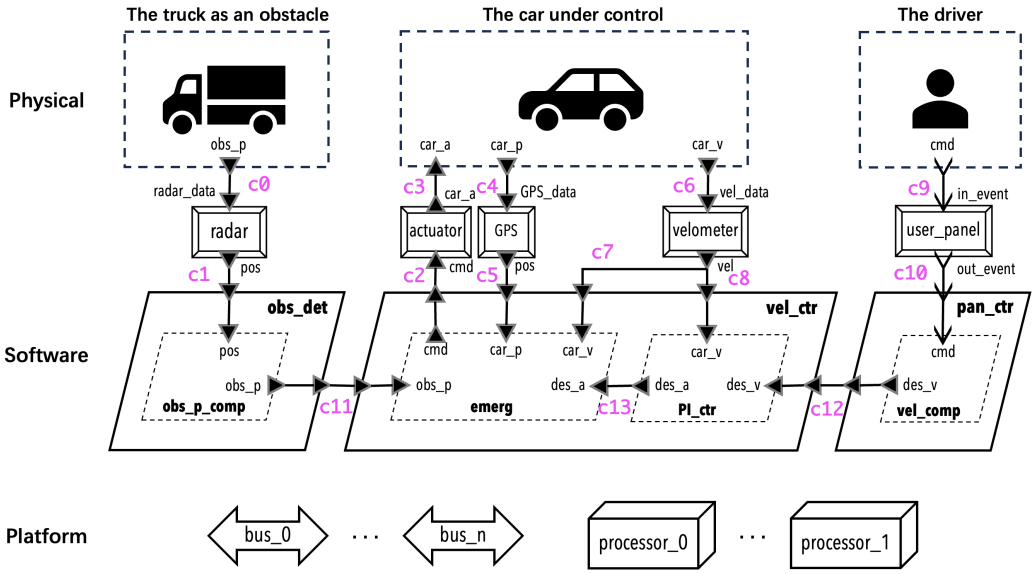


Fig. 1. The automatically cruise control system (ACCS)

4.1 Architecture and Behavior

The architecture of the ACCS decomposes to three levels, shown in Fig. 1. The physical layer contains a car, a driver controlling the car, and a truck in front of the car as an obstacle. The software level defines control of the system and it contains three processes for obstacle detection, velocity control, and panel control, and each process is composed of several threads. These processes interact with the environment (the physical layer) through devices. The platform layer consists of buses, processors, and some devices. The connections between processes and devices could be bound to buses and all the threads are bound to processors, with some scheduling policies such as FIFO (First-In-First-Out) and HPF (High-Priority-First).

The execution of the ACCS is as follows. The `car` is placed at the starting point initially and the `driver` at the wheel can accelerate and decelerate the car by the device `user_panel`. Process `pan_ctr` receives a command from the driver and the thread `vel_comp` in the process computes a desired velocity (`des_v`) and sends it to the process controlling the velocity of the car (`vel_ctr`). Meanwhile, process `obs_det` detects the `truck` by a `radar` and provides the velocity controller process (`vel_ctr`) with the real-time position of the truck (`obs_p`). Process `vel_ctr` computes a command (`cmd`) from the received obstacle (truck) position (`obs_p`), the position (`car_p` by `GPS`) and velocity (`car_v` by `velometer`) of the car, and the desired velocity (`des_v`) from the driver. Concretely, the thread `PI_ctr` in the process computes a desired acceleration (`des_a`) according to the real-time velocity of the car and the desired velocity set by the driver and sends the result to the thread `emerg` in the process. Thread `emerg` collects the real-time position and velocity of the car, the desired acceleration provided by `PI_ctr`, and the real-time position of the obstacle to work out a command (`cmd`), by some emergency control strategy. Finally, process `vel_ctr` outputs the command to the `actuator` which controls the motion of the car and the above procedure repeats.

4.2 Physical Level

The truck, car, and driver are modeled as AADL **abstract** components. The type of the truck is shown below. It is an abstract component with one output data port obs_p denoting that the position of the truck can be sensed by other AADL components.

```
abstract truck
  features
    obs_p : out data port Base_Types::Float;
end truck;
```

We assume the truck is placed at the position of 35 m ahead of the car initially (Init) and then stays motionless for 10 s (Stay) before moving forward for 10 s with velocity 2 m/s (Run). After that, it moves to another lane and will no longer be in front of the car (Away). Before the truck moves to another lane, the environment of the truck can sense the real-time position (p) of the truck at any time through the output data port obs_p. This behavior can be described by the following HA, where t and p denote the local time and the position of the truck.

```
abstract implementation truck.imp
  annex hybrid {**
    variables
      t, p : Base_Types::Float
    channels
      obs_p! : Base_Types::Float
    behavior
      Main ::= Init; Stay; Run; Away
      Init ::= t := 0; p := 35
      Stay ::= 'DT 1 t = 1' < t<10 > [[>obs_p!p ~> Stay]]>
      Run ::= 'DT 1 t = 1 & DT 1 p = 2' < t<20 > [[>obs_p!p ~> Run]]>
      Away ::= 'DT 1 t = 1' [[>obs_p!0 ~> Away]]>
  **}
end truck.imp;
```

The car moves according to the ODE $\{\dot{p} = v, \dot{v} = a\}$, where p, v, and a denote the position, velocity, and acceleration of the car, respectively, and they are set to 0 initially. During the evolution, the car can be actuated by the acceleration from the input data port car_a and its position and velocity can be sent out through the output data ports car_p and car_v, respectively. In summary, the car can be modeled by the following **abstract** component with HA:

```
abstract car
  features
    car_a : in data port Base_Types::Float;
    car_p : out data port Base_Types::Float;
    car_v : out data port Base_Types::Float;
end car;

abstract implementation car.imp
  annex hybrid {**
    variables
      p, v, a : Base_Types::Float
    channels
      car_a?, car_p!, car_v! : Base_Types::Float
    behavior
      Main ::= Init; Run
      Init ::= p := 0; v := 0; a := 0
```

```
      Run ::= 'DT 1 p = v' & 'DT 1 v = a'
        [[> car_a?a ~> Run, car_p!p ~> Run, car_v?v ~> Run ]]>
  **}
end car.imp;
```

The behavior of the `driver` is trivial: it first speeds up the car three times with time interval 0.5s in between to set a desired speed to 3m/s. After 29s, the driver continues to speed down twice in 0.5s time intervals to decrease the desired speed. The driver sends the command (acceleration or deceleration) through the output event port `cmd`, and its AADL code is shown as below:

```
abstract driver
  features
    cmd : out event port Base_Types::Integer;
end driver;

abstract implementation driver.imp
  annex hybrid {**
    constants
      acc = 1, dec = -1
    channels
      cmd! : Base_Types::Integer
    behavior
      Main ::= Up; Wait; Down
      Up ::= cmd!acc; wait 0.5; cmd!acc; wait 0.5; cmd!acc
      Wait ::= wait 29
      Down ::= cmd!dec; wait 0.5; cmd!dec
  **}
end driver.imp;
```

4.3 Software Level

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

4.3.1 Obstacle Detection. The obstacle detection process `obs_det` contains one thread: `obs_p_comp`, which acquires obstacle positions from the input data port `pos` and transfers the processed positions through the output data port `obs_p`. Concretely, the AADL code of this thread is as follows, where `x`, and `y` are temporary variables.

```
thread obs_p_comp
  features
    pos : in data port Base_Types::Float;
    obs_p : out data port Base_Types::Float;
  properties
    Dispatch_protocol => Periodic;
    Period => 97 ms;
    Deadline => 97 ms;
    Compute_execution_time => 20 ms;
    Priority => 1;
end obs_p_comp;

thread implementation obs_p_comp.imp
```

```
  annex hybrid {**
    variables
      x, y : Base_Types::Float
    channels
      pos?, obs_p! : Base_Types::Float
    behavior
      Input ::= pos?x
      Main ::= y := x
      Output ::= obs_p!y
  **}
end obs_p_comp.imp;
```

Finally, the process of obstacle detection (`obs_det`) is modeled as follows:

```
process obs_det
  features
    pos : in data port Base_Types::Float;
    obs_p : out data port Base_Types::Float;
end obs_det;

process implementation obs_det.imp
  subcomponents
    obs_p_comp : thread obs_p_comp.imp;
  connections
    c0 : pos -> obs_p_comp.pos;
    c1 : obs_p_comp.obs_p -> obs_p;
end obs_det.imp;
```

4.3.2 Velocity Control. The process `vel_ctr` for velocity control consists of two threads: `PI_ctr` and `emerg`. The thread `PI_ctr` receives the car speed at the input data port `car_v` and a desired speed at the input data port `des_v`; then it computes a desired acceleration and sent it out through port `des_a`. The type of `PI_ctr` is shown as below:

```
thread PI_ctr
  features
    car_v : in data port Base_Types::Float;
    des_v : in data port Base_Types::Float;
    des_a : out data port Base_Types::Float;
  properties
    Dispatch_protocol => Periodic;
    Period => 7 ms;
    Deadline => 7 ms;
    Compute_execution_time => 1 ms;
    Priority => 1;
end PI_ctr;
```

Concretely, it computes the difference between the desired and the real velocities of the vehicle and then sends the difference to discrete PI controller with a wind-up method (back-calculation) to calculate a desired acceleration. The computation is specified by the HA in the implementation of this thread as follows:

```
thread implementation PI_ctr.imp
  annex hybrid {**
    constants
      period = 7 ms
```

```
    variables
      Integrator_DSTATE : Base_Types::Float
      v_car, v_des, a_des : Base_Types::Float
      Sum, IntegralGain, SumFdbk, SumFdbk_0 : Base_Types::Float
    channels
      car_v?, des_v?, des_a! : Base_Types::Float
    behavior
      Init ::= Integrator_DSTATE := 0
      Input ::= car_v?v_car; des_v?v_des
      Main ::= Sum := v_des - v_car; IntegralGain := Sum;
        SumFdbk := Sum + Integrator_DSTATE;
        (SumFdbk > 3) -> (SumFdbk_0 := 3) [] (SumFdbk < -3) -> (SumFdbk_0 := -3)
        [] (-3 <= SumFdbk <= 3) -> (SumFdbk_0 := SumFdbk);
        IntegralGain := (SumFdbk_0 - SumFdbk + IntegralGain) * period
          + Integrator_DSTATE;
        Sum := Sum + IntegralGain;
        (Sum > 3) -> (a_des := 3) [] (Sum < -3) -> (a_des := -3)
        [] (-3 <= Sum <= 3) -> (a_des := Sum);
        Integrator_DSTATE := IntegralGain
      Output ::= des_a!a_des
  **}
end PI_ctr.imp;
```

The thread `emerg` receives obstacle position at port `obs_p`, car position at port `car_v`, car speed at port `car_p`, and the desired acceleration at port `des_a`, and outputs a command at port `car_a` based on all these inputs. It checks whether the desired acceleration is safe with respect to obstacle position. If so this is allowed as the final command. Otherwise, it overrides the command with a safe deceleration. Concretely, the control of `emerg` is based on the Maximum Protection Curve [66] computed as follows:

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

where car_p and obs_p are the respective current positions of the car and the obstacle, v_{max} is the maximum velocity that the car can reach and $a_{min} < 0$ is the braking deceleration of the car. 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 $\sqrt{-2a_{min} \cdot (obs_p - car_p)}$ in order to avoid the collision (provided $obs_p - car_p > 0$); otherwise, if $obs_p - car_p \leq 0$, then a collision has already happened, and the vehicle should stop ($V_{lim}(car_p) = 0$).

At each iteration, `emerg` predicts the position s_{next} and velocity v_{next} of the car at the next period based on the desired acceleration (des_a) provided by `PI_ctr` (see Fig. 1). Concretely, they can be computed by

$$\begin{aligned} v_{next} &= car_v + des_a \cdot period \\ p_{next} &= car_p + car_v \cdot period + \tfrac{1}{2} \cdot des_a \cdot period^2 \end{aligned}$$

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

If, at the next period, the velocity does not exceed the upper limit computed as above, i.e., $v_{next} \leq V_{lim}(p_{next})$, then the desired acceleration des_a is safe; if not, it continues to test if the constant velocity (no acceleration or deceleration) is safe ($car_v \leq V_{lim}(car_p + car_v \cdot period)$); otherwise, the emergency alerts and the thread `emerg` outputs the minimal deceleration ($a_{min} < 0$)

to brake the car. The above control strategy can be described by

$$a(car_p, car_v) = \begin{cases} des_a & \text{if } v_{next} \leq V_{lim}(p_{next}) \\ 0 & \text{if } car_v \leq V_{lim}(car_p + car_v \cdot period) \\ a_{min} & \text{otherwise} \end{cases}$$

In summary, the thread `emerg` is modeled as follows:

```
thread emerg
  features
    obs_p : in data port Base_Types::Float;
    car_p : in data port Base_Types::Float;
    car_v : in data port Base_Types::Float;
    des_a : in data port Base_Types::Float;
    cmd : out data port Base_Types::Float;
  properties
    Dispatch_protocol => Periodic;
    Period => 5 ms;
    Deadline => 5 ms;
    Compute_execution_time => 1 ms;
    Priority => 2;
end emerg;

thread implementation emerg.imp
  annex hybrid {**
    constants
      v_max = 10 m/s
      a_min = -3 m/s
      period = 5 ms
    variables
      p_obs, p_car, v_car, a_des, CMD : Base_Types::Float
      p, v, v_lim : Base_Types::Float
    channels
      obs_p?, car_p?, car_v?, des_a?, cmd! : Base_Types::Float
    assume
      << [p_car = 0] and [v_car = 0] >>
    invariant
      << [p_car <= p_obs] and [v_car <= v_lim] >>
    ensure
      << [p_car <= p_obs] >> -- The car will never collide into the obstacle ahead
    behavior
      Input ::= obs_p?p_obs; car_p?p_car; car_v?v_car; des_a?a_des
      Main ::= p := p_car + v_car * period + 0.5 * a_des * period^2;
        v := v_car + a_des * period; Comp_V_lim;
        (v <= v_lim) -> (CMD := a_des) []
        (v > v_lim) -> (p := p_car + v_car * period; Comp_V_lim;
          (v_car <= v_lim) -> (CMD := 0) [] (v_car > v_lim) -> (CMD := a_min)
        )
      Comp_V_lim ::=
        (p_obs - p >= v_max^2 / (-2 * a_min)) -> (v_lim := v_max) []
        (p_obs - p < v_max^2 / (-2 * a_min)) -> (
          (p_obs - p > 0) -> (v_lim := sqrt(-2 * a_min * (p_obs - p))) []
          (p_obs - p <= 0) -> (v_lim := 0)
```

```
        )
      Output ::= cmd!CMD
  **}
end emerg.imp;
```

Finally, the process of velocity control (vel_ctr) is modeled as follows:

```
process vel_ctr
  features
    obs_p : in data port Base_Types::Float;
    car_p : in data port Base_Types::Float;
    car_v0 : in data port Base_Types::Float;
    car_v1 : in data port Base_Types::Float;
    des_v : in data port Base_Types::Float;
    cmd : out data port Base_Types::Float;
end vel_ctr;

process implementation vel_ctr.imp
  subcomponents
    emerg : thread emerg.imp;
    PI_ctr : thread PI_ctr.imp;
  connections
    c0 : obs_p -> emerg.obs_p;
    c1 : emerg.cmd -> cmd;
    c2 : car_p -> emerg.car_p;
    c3 : car_v0 -> emerg.car_v;
    c4 : car_v1 -> PI_ctr.car_v;
    c5 : des_v -> PI_ctr.des_v;
    c13: PI_ctr.des_a -> emerg.des_a; -- Asynchronous
end vel_ctr.imp;
```

4.3.3 Panel Control. The process pan_ctr includes only one thread vel_comp. It receives events from event port in_event and transfers the receive events through event port out_event. The driver can control the car by triggering events acc and dec to increase and decrease the desired speed, respectively. The thread pan_ctr_th is modeled as below:

```
thread vel_comp
  features
    cmd : in event port Base_Types::Float;
    des_v : out data port Base_Types::Float;
  properties
    Dispatch_protocol => Aperiodic;
    Deadline => 100 ms;
    Compute_execution_time => 10 ms;
    Priority => 0;
end vel_comp;

thread implementation vel_comp.imp
  annex hybrid {**
    variables
      CMD, v_des : Base_Types::Float
    channels
      cmd?, des_v! : Base_Types::Float
    behavior
```

```
      Init ::= v_des := 0
      Input ::= cmd?CMD
      Main ::= (CMD = 1) -> (v_des := v_des + 1)
        [] (CMD = -1) -> (v_des := v_des - 1);
      Output ::= des_v!v_des
  **}
end vel_comp.imp;
```

Then, the process of panel control `pan_ctr` is modeled as follows:

```
process pan_ctr
  features
    cmd : in event port Base_Types::Float;
    des_v : out data port Base_Types::Float;
end pan_ctr;

process implementation pan_ctr.imp
  subcomponents
    vel_comp : thread vel_comp.imp;
  connections
    c0 : cmd -> vel_comp.cmd;
    c1 : vel_comp.des_v -> des_v;
end pan_ctr.imp;
```

4.4 Platform Level

The platform is composed of a bus, a processor, and some devices. Devices serve as the "routers" that connect the physical and software levels. All the devices in the ACCS are modeled as follows:

```
device radar
  features
    radar_data : in data port Base_Types::Float;
    pos : out data port Base_Types::Float;
  properties
    Dispatch_protocol => Periodic;
    Period => 10 ms;
end radar;

device implementation radar.imp
  annex hybrid {**
    variables
      data_radar, POS : Base_Types::Float
    channels
      radar_data?, pos! : Base_Types::Float
    behavior
      Input ::= radar_data?data_radar
      Main ::= POS := data_radar
      Output ::= pos!POS
  **}
end radar.imp;

device actuator
  features
    cmd : in data port Base_Types::Float;
```

```
    car_a : out data port Base_Types::Float;
  properties
    Dispatch_protocol => Aperiodic;
end actuator;

device implementation actuator.imp
  annex hybrid {**
    variables
      CMD, a_car : Base_Types::Float
    channels
      cmd?, car_a! : Base_Types::Float
    behavior
      Input ::= cmd?CMD
      Main ::= a_car := CMD
      Output ::= car_a!a_car
  **}
end actuator.imp;

device GPS
  features
    GPS_data : in data port Base_Types::Float;
    pos : out data port Base_Types::Float;
  properties
    Dispatch_protocol => Periodic;
    Period => 6 ms;
end GPS;

device implementation GPS.imp
  annex hybrid {**
    variables
      data_GPS, POS : Base_Types::Float
    channels
      GPS_data?, pos! : Base_Types::Float
    behavior
      Input ::= GPS_data?data_GPS
      Main ::= POS := data_GPS
      Output ::= pos!POS
  **}
end GPS.imp;

device velometer
  features
    vel_data : in data port Base_Types::Float;
    vel : out data port Base_Types::Float;
  properties
    Dispatch_protocol => Periodic;
    Period => 10 ms;
end velometer;

device implementation velometer.imp
  annex hybrid {**
    variables
```

```
      data_vel, VEL : Base_Types::Float
    channels
      vel_data?, vel! : Base_Types::Float
    behavior
      Input ::= vel_data?data_vel
      Main ::= VEL := data_vel
      Output ::= vel!VEL
  **}
end velometer.imp;

device user_panel
  features
    in_event : in event port Base_Types::Integer;
    out_event : out event port Base_Types::Integer;
  properties
    Dispatch_protocol => Aperiodic;
end user_panel;

device implementation user_panel.imp
  annex hybrid {**
    variables
      event_in, event_out : Base_Types::Integer
    channels
      in_event?, out_event! : Base_Types::Integer
    behavior
      Input ::= in_event?event_in
      Main ::= event_out := event_in
      Output ::= out_event!event_out
  **}
end user_panel.imp;
```

The connections between devices and processes are all bound to a bus, and all the threads in the processes are bound to a processor adopting HPF scheduling policy. The bus and processor are modeled as follows:

```
bus bus0
  properties
    latency => 3 ms;
end bus0;

bus bus1
  properties
    latency => 3 ms;
end bus1;

processor cpu
  properties
    scheduling_protocol => (HPF);
end cpu;
```

As it should be, we can consider multiple buses and multiple processors. Each bus has the property of latency denoting the transfer delay and each processor has its own scheduling policy,

thus we can consider different settings of buses and processors to observe the impact on the system performance caused by bus and processor.

4.5 Composite System

The whole system of the ACCS can be described by the AADL models of the physical, software, and platform levels, and they are connected together to form the following system model:

```
system ACCS
end ACCS;

system implementation ACCS.imp
  subcomponents
    -- Physical Level
    truck : abstract truck.imp;
    car : abstract car.imp;
    driver : abstract driver.imp;
    -- Software Level
    obs_det : process obs_det.imp;
    vel_ctr : process vel_ctr.imp;
    pan_ctr : process pan_ctr.imp;
    -- Platform Level
    radar : device radar.imp;
    actuator : device actuator.imp;
    GPS : device GPS.imp;
    velometer : device velometer.imp;
    user_panel : device user_panel.imp;
    bus0 : bus bus0;
    cpu : processor cpu;
  connections
    c0 : truck.obs_p -> radar.radar_data; -- Synchronous
    c1 : radar.pos -> obs_det.pos;  -- Asynchronous
    c2 : vel_ctr.cmd -> actuator.cmd; -- Synchronous
    c3 : actuator.car_a -> car.car_a; -- Synchronous
    c4 : car.car_p -> GPS.GPS_data; -- Synchronous
    c5 : GPS.pos -> vel_ctr.pos; -- Asynchronous
    c6 : car.car_v -> velometer.vel_data; -- Synchronous
    c7 : velometer.vel -> vel_ctr.car_v0; -- Asynchronous
    c8 : velometer.vel -> vel_ctr.car_v1; -- Asynchronous
    c9 : driver.cmd -> user_panel.in_event; -- Synchronous
    c10 : user_panel.out_event -> pan_ctr.cmd; -- Asynchronous
    c11 : obs_det.obs_p -> vel_ctr.obs_p; -- Asynchronous
    c12 : pan_ctr.des_v -> vel_ctr.des_v; -- Asynchronous
  properties
    actual_processor_binding => (reference (cpu)) applies to
      obs_det, velo_ctr, pan_ctr;
    actual_connection_binding => (reference (bus0)) applies to c0;
    actual_connection_binding => (reference (bus1)) applies to c1;
    actual_connection_binding => (reference (bus0)) applies to c2;
    actual_connection_binding => (reference (bus0)) applies to c3;
    actual_connection_binding => (reference (bus0)) applies to c4;
    actual_connection_binding => (reference (bus0)) applies to c5;
    actual_connection_binding => (reference (bus0)) applies to c6;
```

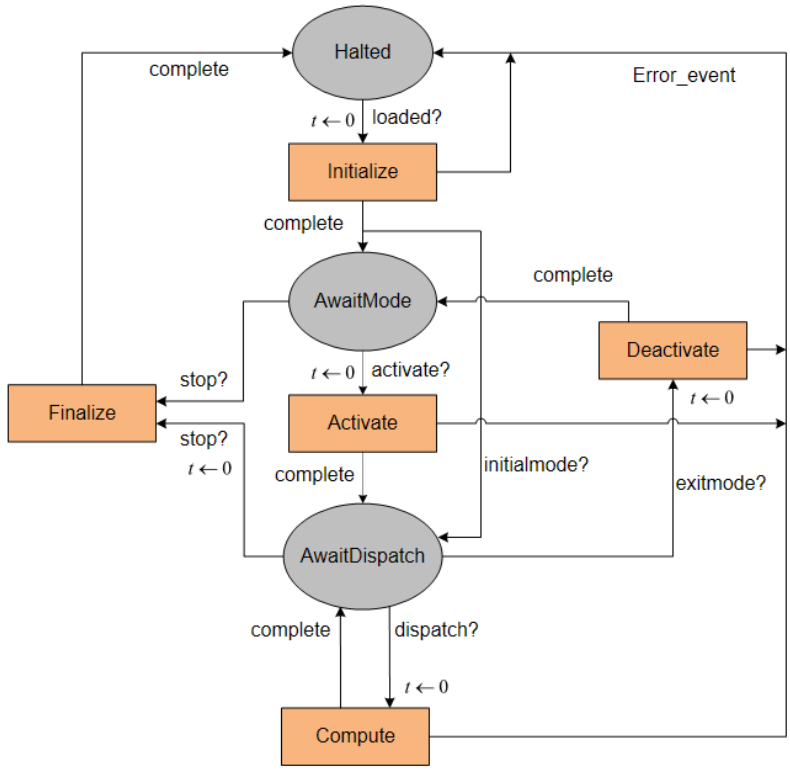


Fig. 2. Thread execution state machine

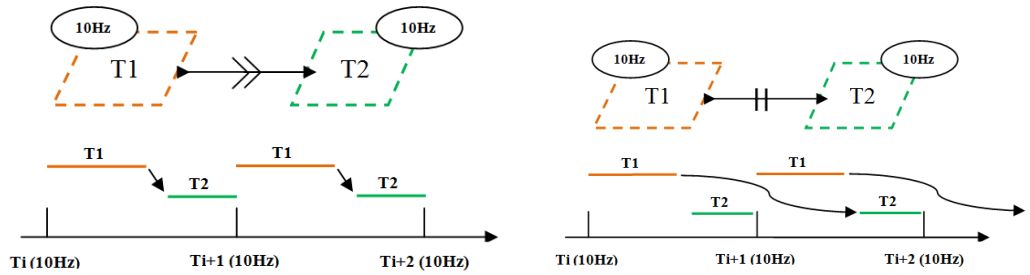
```
    actual_connection_binding => (reference (bus0)) applies to c7;
    actual_connection_binding => (reference (bus0)) applies to c8;
    actual_connection_binding => (reference (bus0)) applies to c9;
    actual_connection_binding => (reference (bus0)) applies to c10;
end ACCS.imp;
```

where the connection `c1` is bound to `bus1` and the others are bound to `bus0`.

5 AADL RUN-TIME SEMANTICS

The AADL standard [40] defines semantics for run-time services including communication, and thread creation, dispatch, suspension, and disposal using timed automata.

Thread life cycle, as depicted in Fig. 2 is same for every thread. Thread execution uses two types of timed automata states: *action states* and *rest states*. Threads in action states are forced to execute associated program code while in rest states threads do not perform any execution. *Initialize*, *Activate*, *Deactivate*, *Compute*, and *Finalize* are the action states while *Halted*, *AwaitMode*, and *AwaitDispatch* are the rest states. Active states can have properties specifying the source code entry points, computation time, and deadlines. For example, *Initialize_Entrypoint* property identifies the subprogram in the source code to be executed in the initialize state, *Initialize_Execution_Time* specifies the time a thread consumes to execute its initialization code sequence, and *Initialize_Deadline* is a property to specify maximum time allowed to complete the initialization code sequence.



(a) Immediate connection (b) Delayed connection

Fig. 3. Synchronous connections for deterministic sampling

A thread in *AwaitDispatch* state is active in current operational mode (AADL supports more than one operational modes), and is waiting for dispatch. Thread dispatch condition is type dependent. A periodic thread is dispatched after a fixed time interval specified in its *period* property. An aperiodic thread, if its predefined dispatch port is not connected, is dispatched each time it receives an event, otherwise it is dispatched each time it receives an event on the dispatch port.

A thread is initialized after the respective process is loaded into memory and is directly moved to the *AwaitDispatch* state if it is active in current process mode, otherwise it is moved to the *AwaitMode* state. Thread dispatch is controlled by *Enabled(t)* function, and *Wait_For_Dispatch* invariant in the *AwaitDispatch* state. A clock variable t is reset each time an active state is entered, and the timing assertion *assert* $t \leq$ (*state_Deadline* + *Recover_Deadline*) is placed in the active state to specify deadline violation. If this assertion in any active state is violated, the thread is moved to the *Halted* state.

5.1 Immediate Communication

Inter-thread communication in synchronous data flow situations can either be *immediate*, or *delayed*, depending on the data port connections. For an immediate connection, data values are transmitted whenever the source thread completes its execution. Meanwhile the destination thread is suspended, and the value received at the destination is the value produced by the latest completion of the source thread. For immediate connection, the threads must share a common dispatch.

Fig. 3(a) shows an immediate connection between two threads T1 and T2, where T1 is the source thread with a sampling rate of 10 Hertz (calculated from its *period* property), and T2 is the destination thread with the same sampling rate. Whenever T1 completes its execution, it immediately transmits data to T2 through its out data port.

5.2 Delayed Communication

For a delayed connection, the output is transmitted at the deadline of the source thread and is available to the destination thread at the next dispatch. The value received at the destination is that produced at the latest deadline of the source thread. For delayed connections, threads do not need to share a common dispatch. Fig. 3(b) shows a delayed connection between two threads T1 and T2, where T1 is a source thread with a sampling rate of 10 Hertz (calculated from its *period* property), and T2 is a destination thread with the same sampling rate. The threads do not share a common dispatch, their sampling rates could be different. Whenever T1 completes its execution, it waits for the deadline. Once passed, T1 transmits the value to T2 through its out data port, in the next dispatch.

To further ensure deterministic exchange of data and to limit the amount of jitter between different frequency periodic threads, AADL also considers *over* and *under-sampling* in addition to the normal synchronous case of communication for both delayed and immediate communication. Formal semantic definition for over and under-sampling is a subject of a future work, no further details are provided for these cases.

5.3 Synchronous and asynchronous connections

The immediate (Section 5.1) and delayed (Section 5.2) communication specify when threads output and complete the computation. In this section, we consider synchronous and asynchronous communications of connections, i.e., we distinguish some of the connections in AADL between synchronous and asynchronous. Concretely, if two ports are connected synchronously, their communication only happens when they are ready, which may cause synchronous waiting; however, if two ports are connected asynchronously, i.e., there is a buffer between them, then the input and output ports get and put messages from and to the buffer at proper time instants, respectively. Therefore, asynchronization will not cause any communication delay but the input port of an asynchronous connection may receive out-of-date messages. For example, in our case study (Fig. 1), the connections between the physical `car` and devices `actuator`, `GPS`, and `velometer` are synchronous, while the ones between devices `GPS` and `velometer` and process `vel_ctr` are asynchronous.

6 FROM AADL WITH HYBRID ANNEX TO HCSP

In this section, we present a translation from AADL with HA to HCSP, and we consider translation of processors, threads, devices, physical environments, connections, and buses in turn.

Remark 6.1. *The translation from the extended AADL models introduced in this paper is different from the one proposed in [66] and the different reflects mainly in the following two aspects:*

(1) *The AADL is extended with HA, a paradigm that describes the hybrid behavior of AADL components, in this paper while in [66], it is extended with Simulink/Stateflow, i.e., the hybrid behavior of a component is modeled by a Simulink/Stateflow diagram.*
(2) *The translation to HCSP in this paper is simplified significantly based on the latest HCSP constructs, especially, with the introduction of procedures.*

6.1 Processors

AADL processor components are modeled as schedulers used for scheduling threads. Detailed specification of the scheduling policies and protocols is beyond the scope of this paper as we are not aiming for schedulability analysis. There are several scheduling policies and in this section, we introduce two of them: FIFO (First-In-First-Out) and HPF (Highest-Priority-First).

6.1.1 FIFO. The FIFO scheduler is represented as a parametric module `SchedulerFIFO(sid)` in Module 1, where `sid` denotes the scheduler identifier. Each scheduler has a unique identifier. It starts by initializing (@`Initialize`) the local time (t) and thread pool (*Pool*) of the scheduler and then schedules (@`Schedule`) the threads bound to it.

In the scheduling procedure, the scheduler is `Idle` if the thread pool is empty (`len`(*Pool*) == 0) and `Busy` otherwise. In the `Idle` status, it monitors the requests from threads. Concretely, the local time t of the scheduler keeps going ($\langle \dot{t} = 1 \& \mathbf{true} \rangle$) until ($\trianglerighteq$) the communication on channel *reqProcessor* occurs. The input channel *reqProcessor*[`sid`][`_tid`]? is parameterized: `sid` and `_tid` denote the identifiers of the scheduler and the requesting thread, respectively. Notice that the identifier of the thread is prefixed with an underline, which means that parameter `_tid` will be

instantiated by pattern matching. For example, the parallel composition

$$reqProcessor[0][\texttt{\_tid}]?\|reqProcessor[0][0]!\|reqProcessor[0][1]!$$

is equivalent to

$$reqProcessor[0][0]?\|reqProcessor[0][1]?\|reqProcessor[0][0]!\|reqProcessor[0][1]!$$

Upon the communication on channel *reqProcessor* occurs, the scheduler receives from the requesting thread a *deadline* and then pushes the thread information (the thread identifier `_tid` together with the final time $t + deadline$ for scheduling the thread) to the thread pool (*Pool*, regarded as a queue) which should be empty at the very moment ($\texttt{assert}(\texttt{len}(Pool) == 0)$). After that, the scheduler gets into `Busy` status.

A `Busy` scheduler first fetches the information of the thread at the head of the thread pool ($(tid, deadline) := \texttt{head}(Pool)$) and then remove it from the pool ($Pool := \texttt{tail}(Pool)$). If the current time of the scheduler is less than the deadline of the thread ($t < deadline$), it sends the thread (identified by *tid*) a *run* signal ($run[\texttt{sid}][tid]!$) and then performs the computation (@`Compute`). If, however, the thread has expired ($t \geq deadline$), the scheduler will ignore it and then schedule the next thread (if any) from the pool (@`Schedule`).

In the `Compute` status, the local time of the scheduler keeps going until either (1) another thread is requiring the processor ($reqProcessor[\texttt{sid}][\texttt{\_tid}]?$) or (2) the thread being scheduled completes its execution ($complete\_exec[\texttt{sid}][tid]?$). In the first case, it pushes the information of the requesting thread onto the thread pool and then returns to the `Compute` status. In the second case, it simply stops (`skip`) and then starts the next schedulling (@`Schedule`, the last line of procedure `Busy`).

Remark 6.2. *Normally, the output channel* $run[\texttt{sid}][tid]!$ *in procedure* `Busy` *should be extended as* $\langle \dot{t} = 1 \& \mathbf{true} \rangle \trianglerighteq [\!] (run[\texttt{sid}][tid]! \longrightarrow \texttt{skip})$ *for that synchronous communications may wait and during the waiting period the local time of the scheduler should go forward. The reason we use the simple form is that we expect that* $run[\texttt{sid}][tid]!$ *does not wait, i.e., the* run *signal invoked by the scheduler should be dealt with immediately.*

6.1.2 HPF. The HPF scheduler is represented as a parametric module `SchedulerHPF(sid)` in Module 2. Since the frameworks of FIFO and HPF schedulers are similar, we only explain their differences. The major difference is that the HPF scheduler selects threads from thread pool by priority. Thus, it is necessary to collect the priority of each thread.

An `Idle` scheduler waits for the requests from the threads to be scheduled. Once it receives a request via input channel $reqProcessor[\texttt{sid}][\texttt{\_tid}]?$, it adds the thread information (thread identifier `_tid`, the final time $t + deadline$ for scheduling the thread, and the *priority* of the thread) to the empty pool. After that, it becomes `Busy`.

Similar to the FIFO scheduler in Module 1, a `Busy` HPF scheduler also fetches and then removes the head thread from the thread pool. Note that here we assume that the threads in *Pool* are sorted in descending order by priority, say *well-ordered*, thus the first thread in the pool always takes the highest priority. If the selected thread has a channel to run, the scheduler gets into `Compute` status.

In `Compute` status, if the scheduler receives a request from another thread, it will interrupt to deal with the (potential) `Preempt`ion. Concretely, it compares the priorities of the thread being scheduled (*prior*) and the one requiring for scheduling (*prior′*). If $prior' > prior$, i.e., the requesting thread takes the higher priority, the running thread (identified by *tid*) will be preempted ($preempt[\texttt{sid}][tid]!$) and put back to the head of the thread pool ($Pool := \texttt{push}([tid, deadline, prior], Pool)$). Since the running thread takes the highest priority, the updated *Pool* is also well-ordered. After that, the running thread switches to the newly added thread ($(tid, deadline, prior) := (\texttt{\_tid}, t + deadline', prior')$), which is then scheduled to execute ($run[\texttt{sid}][tid]!$).

Module 1. FIFO scheduler

```
module SchedulerFIFO(sid):

procedure Initialize begin
  t := 0 ⨟ Pool := [] # Initializing the the local time (t) and thread pool (Pool) of the scheduler
end

procedure Idle begin
  ⟨ṫ = 1&true⟩ ⊵ [] ( reqProcessor[sid][_tid]?deadline ⟶ assert(len(Pool) == 0)⨟
                      Pool := push(Pool, [_tid, t + deadline]) ⨟ @Busy )
end

procedure Compute begin
  ⟨ṫ = 1&true⟩ ⊵ [] ( reqProcessor[sid][_tid]?deadline ⟶ Pool := push(Pool, [_tid, t + deadline]) ⨟ @Compute,
                      complete_exec[sid][tid]? ⟶ skip )
end

procedure Busy begin
  (tid, deadline) := head(Pool) ⨟ Pool := tail(Pool)⨟
  (t < deadline) ⟶ (run[sid][tid]! ⨟ @Compute)⨟
  @Schedule⨟
end

procedure Schedule begin
  if (len(Pool) == 0) {@Idle} else {@Busy}
end

begin
  @Initialize ⨟ @Schedule
end
endmodule
```

If the thread requiring the processor does not take the higher priority than the running thread, i.e., $prior' \leq prior$, the requesting thread will then be inserted to the (well-ordered) thread pool such that the updated *Pool* is well-ordered as well. Procedure Insert implements the insertion algorithm. Concretely, we maintain two well-ordered queues (lists) *Pool* and *Pool*′. The former stores the threads taking lower priority than the requesting thread while the latter stores the others. Then, the updated *Pool* can be obtained by the concatenation of *Pool*′, the inserted thread, and the old *Pool*, i.e., $Pool' := \texttt{push}(Pool', [\texttt{\_tid}, t + deadline', prior'])$ ⨟ $Pool := \texttt{push}(Pool', Pool)$. After Preemption, the scheduler returns to the Compute status.

6.2 Threads

Based on specific properties, associated connections, and timing constraints, each thread corresponding to the state machine shown in Fig. 2 is translated into one HCSP process. AADL modal semantics are not considered here, so every thread has only one operational mode. Following parallel composition of HCSP processes Dispatch and Execution represents the HCSP process corresponding to an AADL thread:

$$\texttt{Thread} \quad ::= \quad \texttt{Dispatch} \| \texttt{Execution}$$

where Dispatch specifies dispatching the thread and Execution implements the state machine of Fig. 2. The two processes have repeating behavior and can only communicate through channels.

Module 2. HPF scheduler

```
module SchedulerHPF(sid):

procedure Initialize begin
  t := 0 ⨟ Pool := [ ]
end

procedure Idle begin
  ⟨ṫ = 1&true⟩ ⊵ [] ( reqProcessor[sid][_tid]?(deadline, prior) ⟶ assert(len(Pool) == 0)⨟
                      Pool := push(Pool, [_tid, t + deadline, prior]) ⨟ @Busy )
end

procedure Loop begin
  if (¬success ∧ len(Pool) > 0) {
    (tid'', deadline'', prior'') := head(Pool) ⨟ Pool := tail(Pool)⨟
    if (prior' > prior'') {success := true} else {Pool' := push(Pool', [tid'', deadline'', prior''])}⨟
    @Loop
  } else {Pool' := push(Pool', [_tid, t + deadline', prior']) ⨟ Pool := push(Pool', Pool)}
end

procedure Insert begin
  Pool' := [ ] ⨟ success := false ⨟ @Loop
end

procedure Preempt begin
  if (prior' > prior) {
    preempt[sid][tid]!⨟
    Pool := push([tid, deadline, prior], Pool)⨟
    (tid, deadline, prior) := (_tid, t + deadline', prior')⨟
    run[sid][tid]!
  } else {@Insert}⨟
  @Compute
end

procedure Compute begin
  ⟨ṫ = 1&true⟩ ⊵ [] ( reqProcessor[sid][_tid]?(deadline', prior') ⟶ @Preempt,
                      complete_exec[sid][tid]? ⟶ skip )
end

procedure Busy begin
  # assert(the threads in Pool has been sorted in descending order by priority)
  (tid, deadline, prior) := head(Pool) ⨟ Pool := tail(Pool)⨟
  (t < deadline) ⟶ (run[sid][tid]! ⨟ @Compute)⨟
  @Schedule
end

procedure Schedule begin
  if (len(Pool) == 0) {@Idle} else {@Busy}
end

begin
  @Initialize ⨟ @Schedule
end
endmodule
```

Module 3. Periodic dispatch

```
module Periodic_DIS(tid, period):

procedure Execute begin
  dispatch[tid]!;
  t := 0 ; ⟨ṫ = 1&t < period⟩ ⊵ [](complete_comp[tid]? ⟶ ⟨ṫ = 1&t < period⟩);
  @Execute
end

begin
  @Execute
end
endmodule
```

Module 4. Aperiodic dispatch

```
module Aperiodic_DIS(tid, inConn):

procedure Execute begin
  inputs[inConn]?event ; dispatch[tid]!event;
  t := 0 ; ⟨ṫ = 1&true⟩ ⊵ [](complete_comp[tid]? ⟶ skip);
  @Execute
end

begin
  @Execute
end
endmodule
```

6.2.1 Dispatch. Since the **properties** section of a thread specifies the `dispatch_protocol` (periodic or aperiodic), `deadline`, and `period` (if periodic) of the thread, the dispatcher for the thread can be constructed according to these information.

A periodic thread is dispatched after every fixed time interval specified in its period property. The dispatch process for a periodic thread with thread identifier `tid` and dispatch `period` is specified in Module 3. A periodic dispatcher is a recursive process which, at each iteration, dispatches its thread (identified by `tid`) by output channel *dispatch*[`tid`]! and then waits for one `period`. During the waiting period, it monitors the signal *complete_comp*, indicating the computation completes, from the thread. If the computation of the thread completes, the dispatcher will still evolve until the current period completes ($\langle \dot{t} = 1 \& t <$ `period`$\rangle$). After that, it will start the next dispatching and the procedure repeats (@`Execute`).

An aperiodic thread is triggered by an incoming event, so an aperiodic dispatcher is always associated with some input connection to the thread. As shown in Module 4, an aperiodic dispatcher takes two parameters: the identifier of the thread to be dispatched (`tid`) and an input connection (`inConn`) to the thread. An aperiodic dispatcher will be activated by the input *event* received from the input connection, i.e., it is invoked by *inputs*[`inConn`]?*event*. Once it is invoked, it dispatches the thread by sending the received *event*, i.e., *dispatch*[`tid`]!*event*. Then, it waits until the thread completes its computation (*complete_comp*[`tid`]?). After that, it waits for the next activation and the procedure repeats (@`Execution`).

6.2.2 Execution. The execution behavior of a thread is specified in Module 5, where the parameters mean that the thread of `tid` is bound to the processor of `sid`. At the beginning, the thread is

Initialized by setting the initial values of the variables including the *wcet* (Worst Case Execution Time), *deadline*, *prior*ity, and so on of the thread. All these information can be obtained from the **properties** section of the thread, where compute_execution_time denotes the *wcet* here. In addition, the state variables of the thread should also be Initialized, which is specified by Init in the **behavior** section of the HA of the thread. For example, the initialization of the thread PI_ctr of the ACCS in Section 4.3.2 can be written as follows:

procedure Initialize **begin**
  $period := 0.007$; $deadline := 0.007$; $wcet := 0.001$; $prior := 1$;
  $Integrator\_DSTATE := 0$ # state variable
end

After the initialization, the thread can perform Execution, which is a recursive procedure. The Execution of the thread starts once it is dispatched by a periodic (Module 3) or aperiodic (Module 4) dispatcher, depending on whether it is a periodic thread or not. Upon dispatched, it gets data from the input connections (@Input). Each input channel $inputs[\mathtt{inConn}_i]?in_i$ in Input corresponds to an input connection $\mathtt{inConn}_i$ to the thread, where in_i is the name of the corresponding input port.

Then, the thread requires the processor it is bound to for the computation based on the input data. Concretely, it sends the processor, which is translated to a scheduler specified in Module 1 or 2, its *deadline* and *prior*ity via channel *reqProcessor*, i.e., $reqProcessor[\mathtt{sid}][\mathtt{tid}]!(deadline, prior)$. After that, the local time t of the thread is initialized ($t := 0$) and then it waits for the *run* signal from the scheduler. If it has failed to *run* before the *deadline*, the thread will give up waiting and start the next Execution; otherwise, i.e., it receives a *run* signal before the *deadline*, the execution time of the thread is initialized ($c := 0$) and the thread begins to Run.

In procedure Run, the thread evolves until either (1) its local time (t) reaches the *deadline*, (2) its execution time (c) reaches the *wcet*, or (3) it is *preempted* by another thread with higher priority. For the third case (*preempted* is **true**), it will be put into the thread pool and keeps waiting for the next *run* signal until the *deadline* arrives. If the thread stops without being preempted, it notifies the scheduler that its execution completes ($complete\_exec[\mathtt{sid}][\mathtt{tid}]!$); if the execution time (c) of the thread reaches its *wcet*, it informs the dispatcher the computation completes ($complete\_comp[\mathtt{tid}]!$) and then Outputs the results.

Before the output, it is necessary to get the computation Results. The procedure Result is obtained from the Main process in the **behavior** section of the HA of the thread. Notice that the use of Main may introduce into other auxiliary procedures. For example, the use of Main in thread emerg (Section 4.3.2) will bring in Comp_V_lim as the auxiliary procedure. After computing the outputs out_i from the inputs and states by Result, it sends the results out through the output connections ($outputs[\mathtt{outConn}_i]!out_i$), where $\mathtt{outConn}_i$ is an output connection from the port out_i of the thread. It should be noted that if some output connection is bound to a bus, the thread should obtain the permission to the bus before outputting. Concretely, if the output connection $\mathtt{outConn}_i$ is bound to $\mathtt{bus}_j$, the output $outputs[\mathtt{outConn}_i]!out_i$ in procedure Output should be replaced with

$$\langle \dot{t} = 1 \& t < deadline \rangle \trianglerighteq [\!](reqBus[\mathtt{bus}_j][\mathtt{outConn}_i]!out_i \longrightarrow \mathtt{skip})$$

which indicates that the thread must obtain the permission to $\mathtt{bus}_j$ before the *deadline*. Upon it gets the permission (the communication on *reqBus* occurs), it sends the bus the output result (out_i). Section 6.6 introduces the behavior of buses.

Remark 6.3. *One may think that the* $outputs[\mathtt{outConn}_i]!out_i$ *above should be modified as*

$$outputs[\mathtt{tid}][\mathtt{outport}_i]!out_i$$

Module 5. Thread execution

```
module EXE(tid, sid):

procedure Initialize begin
  wcet := ··· ; deadline := ··· ; prior := ··· ;   # Constant variables
  s⃗ := ···   # The vector of the state variables of the thread
end

procedure Input begin
  inputs[inConn_1]?in_1 ; ··· ; inputs[inConn_n]?in_n
end

procedure Result begin
  # Compute the outputs from the inputs and states
  out_1 := f_1(in_1, ··· , in_n, s⃗) ; ··· ; out_m := f_m(in_1, ··· , in_n, s⃗);
  s⃗ := f(in_1, ··· , in_n, s⃗)   # Update the states of the thread
end

procedure Output begin   # Without buses
  @Result;
  outputs[outConn_1]!out_1 ; ··· ; outputs[outConn_m]!out_m
end

procedure Run begin
  preempted := false;
  ⟨ṫ = 1, ċ = 1 & c < wcet ∧ t < deadline⟩ ⊵ [](preempt[sid][tid]? ⟶ preempted := true);
  if (preempted) {⟨ṫ = 1 & t < deadline⟩ ⊵ [](run[sid][tid]? ⟶ @Run)}
  else {complete_exec[sid][tid]! ; (c ≥ wcet) ⟶ (complete_comp[tid]! ; @Output)}
end

procedure Execute begin
  dispatch[tid]?;   # We use dispatch[tid]?event for aperiodic dispatch
  @Input;   # Get inputs from data ports
  reqProcessor[sid][tid]!(deadline, prior);   # We remove prior if sid adopts the FIFO policy
  # t and c are the local and execution times of the thread, respectively
  t := 0 ; ⟨ṫ = 1 & t < deadline⟩ ⊵ [](run[sid][tid]? ⟶ c := 0 ; @Run);
  @Execute
end

begin
  @Initialize ; @Execute
end
endmodule
```

because the latter clarifies the fact that out_i *is sent through the output port* $\texttt{outport}_i$ *of the thread* tid*. However, we prefer the former for the simple reason that there may be multiple connections, say* $\texttt{outConn}_{i,j}$ *for* $1 \leq j \leq k$*, from one output port, say* $\texttt{outport}_i$*. For this case, we can use*

$$outputs[\texttt{outConn}_{i,1}]!out_i \,;\, \cdots \,;\, outputs[\texttt{outConn}_{i,k}]!out_i$$

to express that out_i *is broadcast via the* i*-th output port. For the same reason, we use* $inputs[\texttt{inConn}]?$ *instead of* $inputs[\texttt{tid}][\texttt{inport}]?$ *in this paper.*

In Module 5, the *immediate* connection (Fig. 3(a)) is considered. However, for the *delayed* connection (Fig. 3(b)), the thread sends the computation results out and completes the computation after

Module 6. Aperiodic device

```
module device_name():

procedure Initialize begin
  s⃗ := ···    # The vector of the state variables of the device
end

procedure Input begin
  inputs[inConn_1]?in_1 ⨟ ··· ⨟ inputs[inConn_n]?in_n
end

procedure Compute begin
  # Compute the outputs and update the states
  out_1 := f_1(in_1, ··· , in_n, s⃗) ⨟ ··· ⨟ out_m := f_m(in_1, ··· , in_n, s⃗) ⨟ s⃗ := f(in_1, ··· , in_n, s⃗)
end

procedure Output begin
  outputs[outConn_1]!out_1 ⨟ ··· ⨟ outputs[outConn_m]!out_m
end

procedure Execute begin
  @Input ⨟ @Compute ⨟ @Output ⨟ @Execute
end

begin
  @Initialize ⨟ @Execute
end
endmodule
```

it reaches the deadline. Concretely, the sequential composition *complete_comp*[tid]! ⨟ @Output in procedure Run of Module 5 should be modified as

$$\langle \dot{t} = 1 \& t < deadline \rangle \fatsemi @\texttt{Output} \fatsemi complete\_comp[\texttt{tid}]!$$

for the delay connection.

6.3 Devices

Similar to threads (Section 6.2), a device gets inputs at its input ports and outputs the results through its output ports after performing some computation. The major difference between the behaviors of threads and devices is that a device is not bound to any processor. There are various kinds of devices in the real world and it is impossible to consider all of them in this paper. In addition, devices are not our concerns, so we only address two kinds of devices: *periodic* and *aperiodic*.

The behavior of an aperiodic device is specified in Module 6. After the initialization of the states (@Initialize), the device begins to Execute, which is a recursive procedure. At each iteration of the recursion, it gets inputs from the input connections (@Input), computes the outputs and updates its states according to the inputs and current states (@Compute), and finally outputs the computation results (@Output). The procedure Compute is obtained from the Main process in the **behavior** section of the HA of the device. The HCSP model of a periodic device is shown in Module 7. The only difference between periodic and aperiodic devices is that a periodic device will wait for one period before the next execution.

Module 7. Periodic device

```
module device_name(period):
⋮
procedure Execute begin
  @Input ; @Compute ; @Output ; wait(period) ; @Execute
end
⋮
endmodule
```

Module 8. The HCSP model of the car

```
module car():

procedure Initialize begin
  p := 0 ; v := 0 ; a := 0    # Position (p), velocity (v), and acceleration (a)
end

procedure Execute begin
  ⟨ṗ = v, v̇ = a & true⟩ ⊵ [] ( inputs[c3]?a ⟶ @Execute,
                              outputs[c4]!p ⟶ @Execute,
                              outputs[c6]!v ⟶ @Execute )
end

begin
  @Initialize ; @Execute
end
endmodule
```

6.4 Physical Environments

The environment includes continuous and discrete processes interacting with the software level (through devices potentially). Different from threads and devices, an environment is more flexible and hence there is no pattern (such as Module 5 for threads) describing the behavior of environments. On the other hand, this flexibility leaves more space for the designer to model the environment. To illustrate this, we show how to translate the `car` in the physical level (Section 4.2) of the ACCS. Concretely, the HCSP model of the `car` is shown in Module 8 and it is obtained from the `Main` process in the **`behavior`** section of the HA of the `car`. The `Main` process is a sequential composition of two sub-processes `Init` and `Run`, which are translated to two procedures in Module 8, and the communications through the ports of `car` are unified into the form $input[\texttt{inConn}_i]in_i$ and $output[\texttt{outConn}_i]out_i$. From the architecture of the ACCS (Fig. 1), `c3` is the input connection to port `car_a`, and `c4` and `c6` are the output connections from ports `car_p` and `car_v`, respectively.

6.5 Port Connections

A port is an interface for the directional transfer of data, events, or both into or out of an AADL component [29]. Port connections are pathways for such directional transfers between components. Data ports are interfaces for state data transmission among components without queuing. Event ports are interfaces for the communication of events that may be queued. Event data ports are interfaces for message transmission with queuing, and these interfaces enable the queuing of the data associated with an event.

Module 9. Synchronous connection

```
module SyncConn(conn):

procedure Execute begin
    outputs[conn]?msg ⨟ inputs[conn]!msg ⨟ @Execute
end

begin
   @Execute⨟
end
endmodule
```

The connections between ports can be classified as either synchronous or asynchronous, and accordingly, we present synchronous and asynchronous communication models using HCSP. Similarly to CSP [38] and other classic process calculi [50], HCSP adopts the handshake communication, upon which more complex communication paradigms can be formally defined [51]. Ports connected synchronously will communicate in a synchronized manner: if one party is not ready, the other should wait. The synchronous communication between ports can be represented naturally based on HCSP's handshake communication, as illustrated in Module 9. In this module, *outputs*[conn] is a channel denoting the output port of the source component of connection conn and *inputs*[conn] is a channel representing the input port of the target component of conn (see, for instance, Modules 5). During each iteration, it receives a message (*msg*) from the output port of the source of the connection, i.e., *outputs*[conn]?*msg*, and then sends the message to the input port of the target of the connection, i.e., *inputs*[conn]!*msg*. If the target is performing some execution, i.e., its *inputs*[conn]? is not ready, then the *inputs*[conn]!*msg* should wait, thereby reflecting the semantics of synchronous communication.

Compared to synchronous connections, the asynchronous case presents greater complexity, as it requires us to model asynchronous communication between ports using HCSP's synchronous communication mechanism. To formalize the semantics of asynchronous communication between data ports, we employ the external choice construct ($\square$) within HCSP, as illustrated in Module 10. Intuitively, an asynchronous connection between data ports serves as a data buffer with a capacity of one, which can also be regarded as a *data* variable. Initially, it receives the initial data from the output port of the connection's source (@Execute). Subsequently, a recursive process (@Execute) is established, offering two external choices that allow the source and the target of the connection to write and read the data asynchronously whenever they are triggered. Specifically, when the source intends to output data, i.e., *outputs*[conn]!, the branch prefixed with *outputs*[conn]?*data* is triggered (left), meaning the data will be stored in the *data* buffer. Conversely, when the target wishes to input a value, i.e., *inputs*[conn]?, the branch prefixed with *inputs*[conn]!*data* is triggered (right), allowing the target to retrieve the data stored in the buffer. The mechanism of external choice ensures two key aspects: (1) the source can transmit data directly through the output port at the end of execution, without needing to wait for the target to be ready to receive; and (2) if the source generates output data while the target is in the midst of execution, this data will be stored in the data buffer without affecting (or interrupting) the target's execution.

For the asynchronous communication between event ports, in addition to coordinating the asynchronous input and output between ports using external choice, a queue is also needed to buffer the events being transmitted, as illustrated in Module 11. Initially, the *Queue* is empty (@Initialize). Subsequently, it enters a recursive procedure where it appends incoming events to the end of the queue and takes the head events from it (@Execute). Specifically, during each iteration, if the queue

Module 10. Asynchronous data connection

```
module Async_Data_Conn(conn):

procedure Input begin
    outputs[conn]?data
end

procedure Execute begin
    (outputs[conn]?data ⟶ @Execute) [] (inputs[conn]!data ⟶ @Execute)
end

begin
   @Input ; @Execute
end
endmodule
```

Module 11. Asynchronous event connection

```
module Async_Event_Conn(conn):

procedure Initialize begin
    Queue := []
end

procedure Execute begin
   if (len(Queue) == 0) {
      outputs[conn]?event ; Queue := [event]
   } else {
      [] ( outputs[conn]?event ⟶ Queue := push(Queue, event),
           inputs[conn]!head(Queue) ⟶ Queue := tail(Queue) )
   } ; @Execute
end

begin
   @Initialize ; @Execute
end
endmodule
```

is empty (len(*Queue*) == 0), it awaits an output from the connection source and then places the received event into the now-empty queue (*outputs*[conn]?*event* ; *Queue* := [*event*]); alternatively, if the queue is not empty, it has two options: it can either receive an event from the connection source (*outputs*[conn]?*event*) and append it to the queue (*Queue* := push(*Queue*, *event*)), or it can send the head event in the queue to the connection target (*inputs*[conn]!head(*Queue*)) and then remove it from the queue (*Queue* := tail(*Queue*)). For example, the component vel_comp in the ACCS (Fig. 1) is an aperiodic thread, triggered by the cmd event produced from the user_panel device. The semantics of the connection (c10) between the two event ports are defined by Async_Event_Conn(c10). Specifically, when the user_panel generates a cmd, the connection, i.e., Async_Event_Conn(c10), queues it. On the vel_comp thread side, its aperiodic dispatch mechanism (refer to Module 4) monitors the connection's queue: if the queue is non-empty, *inputs*[c10]?*event* is triggered, and the received *event* activates the vel_comp thread, i.e., *dispatch*[vel_comp]!*event*.

Module 12. The behavior of bus

```
module Bus(bid, d):

procedure Execute begin
  reqBus[bid][_conn]?msg ⟶ wait(d) ⨟ outputs[_conn]!msg ⨟ @Execute
end

begin
  @Execute
end
endmodule
```

The asynchronous communication semantics between event data ports are similar to those between event ports, with the only modification being the extension of the *event* variable in Module 11 to a pair (*event*, *data*), which represents an *event* carrying *data*.

In addition, we can model more complex communication paradigms such as one-to-many paradigm using HCSP. For example, in the ACCS (Fig. 1), device `velometer` transmits the sensed vehicle speed data via its output port to the input ports of threads `emerg` and `PI_ctr` by connections `c7` and `c8`, respectively. This one-to-many communication semantics can be defined by the parallel composition the semantics of `c7` and `c8`, i.e., $\texttt{Async\_Data\_Conn(c7)} \| \texttt{Async\_Data\_Conn(c8)}$. For more examples of the communication semantics between ports introduced above, readers may refer to the case study (Section 7), where the process `Conn` in Section 7.1 defines the HCSP semantics for all connections in the case study of ACCS (Fig. 1).

6.6 Buses

The connections introduced in Section 6.5 may be bound to buses. If a connection is bound to a bus, the source of the connection should request the permission for using the bus before sending messages via the connection. The behavior of a bus with identifier `bid` and latency `d` is specified in Module 12. Assume that there are n connections, say $\texttt{conn}_1, \cdots, \texttt{conn}_n$, bound to the bus. Then, the behavior of the bus is shown as follows:

$$\mu \texttt{X}.\ [] \begin{pmatrix} \textit{reqBus}[\texttt{bid}][\texttt{conn}_1]?\textit{msg}_1 & \longrightarrow & \texttt{wait(d)} \fatsemi \textit{outputs}[\texttt{conn}_1]!\textit{msg}_1 \fatsemi \texttt{X}, \\ \vdots & \vdots & \vdots \\ \textit{reqBus}[\texttt{bid}][\texttt{conn}_n]?\textit{msg}_n & \longrightarrow & \texttt{wait(d)} \fatsemi \textit{outputs}[\texttt{conn}_n]!\textit{msg}_n \fatsemi \texttt{X} \end{pmatrix}$$

At each iteration, it provides n external choices receiving the requests for the bus. If the i-th connection gets the permission to the bus, i.e., the i-th branch is selected by receiving a message $\textit{msg}_i$ from the source of the connection ($\textit{reqBus}[\texttt{bid}][\texttt{conn}_i]?\textit{msg}_i$), then it `wait`s for `d` time units before sending $\textit{msg}_i$ out ($\textit{outputs}[\texttt{conn}_i]!\textit{msg}_i$). The above process can be abstracted to Module 12, where `_conn` can match any connection bound to the bus.

Remark 6.4. *The sequential composition* `wait(d)` ⨟ *outputs*`[_conn]`!*msg in Module 12 models the latency of the bus transferring messages. During the transfer period (*`wait(d)`*), any request for the bus will wait until the current message on the bus is sent out successfully (outputs*`[_conn]`!*msg).*

6.7 Restrictions on AADL

In this work, we only take part of AADL components including processes, threads, processors, buses, devices, abstract components (used to model physical environment), and port connections

between components, into account. Other components have not been considered, for instance, memories and processors with more scheduling policies. The reasons for the restriction include:

(1) We concern more on the abstract and formal model of AADL. The design details of AADL will lead to complex and redundant formal models, which prevents us from verifying the key properties (like safety and latency) of AADL models;
(2) The HA proposed in this paper is used to describe the hybrid and physical behaviors of components, i.e., we focus more on behavioral components like threads. The components regarding resource (for instance, memories) are not the concern of this paper. In our future work, we will consider the effect of resource constraints on AADL models.

For port communication, this paper introduces the communication semantics of basic data, event, and event-data ports using HCSP. More interesting characteristics of AADL ports, which represent different communication models, can also be captured using HCSP. All in all, the paper's emphasis is on extending AADL to hybrid systems and developing analysis and verification methods based on this extended framework. Therefore, to maintain focus on this motivation, we present only the HCSP formal semantics for the commonly used components and ports. The semantics of other components and ports with interesting properties [52] can likewise be defined using HCSP but are not included in this paper.

7 CASE STUDY

The architecture and behavior of the ACCS have been illustrated in Section 4. In this section, we transform it to a formal model of HCSP by the translation algorithm introduced in Section 6. Then, we introduce the simulation and verification for the translated HCSP model of the ACCS.

7.1 Translation

The physical level consists of a `truck` as a mobile obstacle, a `car` behind the truck, and a `driver` controlling the car. All of the three physical objects have been modeled by AADL with hybrid annex in Section 4.2 and their translated HCSP models are specified in Section 6.4. In summary, the physical level can be described as follows:

$$\texttt{Physical} \triangleq \texttt{truck}() \| \texttt{car}() \| \texttt{driver}()$$

The software level contains several AADL processes, and each of the processes is composed of several threads. According to Section 6.2, the HCSP model of a `Thread` consists of two parts: a periodically `Dispatch`er and the `Execution` of the thread. Concretely,

$$\begin{aligned} \texttt{obs\_p\_comp} &\triangleq \texttt{Periodic\_DIS}(0, 0.097) \| \texttt{EXE}(0, 0) \\ \texttt{emerg} &\triangleq \texttt{Periodic\_DIS}(1, 0.005) \| \texttt{EXE}(1, 0) \\ \texttt{PI\_ctr} &\triangleq \texttt{Periodic\_DIS}(2, 0.007) \| \texttt{EXE}(2, 0) \\ \texttt{vel\_comp} &\triangleq \texttt{Aperiodic\_DIS}(3, \texttt{c10}) \| \texttt{EXE}(3, 0) \end{aligned}$$

where the identifiers of threads `obs_p_comp`, `emerg`, `PI_ctr`, and `vel_comp` are 0, 1, 2, and 3, respectively, and they are all bound to the processor whose identifier is 0. Then, the HCSP models of the AADL processes can be represented by

$$\begin{aligned} \texttt{obs\_det} &\triangleq \texttt{obs\_p\_comp} \\ \texttt{vel\_ctr} &\triangleq \texttt{emerg} \| \texttt{PI\_ctr} \\ \texttt{pan\_ctr} &\triangleq \texttt{vel\_comp} \end{aligned}$$

The parallel composition of these processes forms the HCSP model of the software level:

$$\texttt{Software} \triangleq \texttt{obs\_det} \| \texttt{vel\_ctr} \| \texttt{pan\_ctr}$$

The platform level includes the devices connecting the physical and software levels, some buses and processors. These components form the following parallel composition:

$$\begin{aligned}\texttt{Platform} \triangleq\ & \texttt{radar}(0.01)\|\texttt{actuator}()\|\texttt{GPS}(0.006)\|\texttt{velometer}(0.01)\|\texttt{user\_panel}()\\ & \|\texttt{SchedulerHPF}(0)\|\texttt{bus}(0,0.003)\|\texttt{bus}(1,0.003)\end{aligned}$$

Note that this platform contains two buses (identifiers 0 and 1) with latency 3 ms.

Do not forget that the connections between the AADL components should also be translated to HCSP models. For this case study, we have the following connections:

$$\begin{aligned}\texttt{Conn} \triangleq\ & \texttt{Sync\_Conn}(\texttt{c0})\|\texttt{Async\_Data\_Conn}(\texttt{c1})\|\texttt{Sync\_Conn}(\texttt{c2})\|\texttt{Sync\_Conn}(\texttt{c3})\\ & \|\texttt{Sync\_Conn}(\texttt{c4})\|\texttt{Async\_Data\_Conn}(\texttt{c5})\|\texttt{Sync\_Conn}(\texttt{c6})\|\texttt{Async\_Data\_Conn}(\texttt{c7})\\ & \|\texttt{Async\_Data\_Conn}(\texttt{c8})\|\texttt{Sync\_Conn}(\texttt{c9})\|\texttt{Async\_Event\_Conn}(\texttt{c10})\\ & \|\texttt{Async\_Data\_Conn}(\texttt{c11})\|\texttt{Async\_Data\_Conn}(\texttt{c12})\|\texttt{Async\_Data\_Conn}(\texttt{c13})\end{aligned}$$

Finally, the translated HCSP model of the ACCS can be obtained as follows:

$$\texttt{ACCS} \triangleq \texttt{Physical}\|\texttt{Software}\|\texttt{Platform}\|\texttt{Conn}$$

If **assume**, **invariant**, and **ensure** sections are specified for the above components in the original AADL model, then through translation, we will obtain an HCSP specification to be proved, with the following form: $\texttt{ACCS\_Spec} \triangleq \{\texttt{assume}\}\texttt{ACCS}\{\texttt{ensure}; \lceil\texttt{invariant}\rceil\}$.

7.2 Simulation

In [66], we developed an HCSP simulator by which we can simulate the HCSP programs translated from the original AADL model with hybrid annex and then observe the execution result of the original model intuitively. Hence, with the aid of the HCSP simulator, we can observe the impact on systems under different configurations.

In this section, the whole translated HCSP model of the ACCS in Fig. 1 is used to run the simulation, where we consider the impact of different bus configurations on the behavior of the car. Concretely, we consider three configurations for buses: (1) no bus; (2) two buses with latency 3 ms: the connection between device `radar` to obstacle detection process `obs_det` is bound to a specific bus, and the other connections between the devices and the processes are all bound to the other bus; (3) three buses with latency 5 ms: the connection between device `radar` and obstacle detection process `obs_det` and the connection between velocity control process `vel_ctr` to device `actuator` are bound to two separate buses, and the other connections between the devices and the processes are bound to the left bus.

The simulation results of the three bus configurations are shown in Fig. 4. The left shows the velocity evolutions of the car under different bus configurations, and we can see that under the configuration of two buses with latency 3 ms (blue line), there is a clear delay for the change of the velocity of the car: the truck appears in front of the car as an obstacle at 10 s (see the right of Fig. 4), but the car begins to decelerate about 4 s after the truck appears. However, under the other two configurations, the car can decelerate in about 2 s. From these results, we can observe that the configuration that three buses with latency 5 ms may be a better choice because it causes less delay, but on the other hand, it may cause larger fluctuations: the highest velocity of the car under this configuration is up to 5 m/s, higher than the two other configurations.

7.3 Verification

The motivation of translating AADL with hybrid annex to HCSP is to verify the informal AADL models. In this section, we introduce verification of HCSP models by theorem proving. Verification using HHL prover is based on Hybrid Hoare Logic (HHL) in the proof assistant Isabelle.

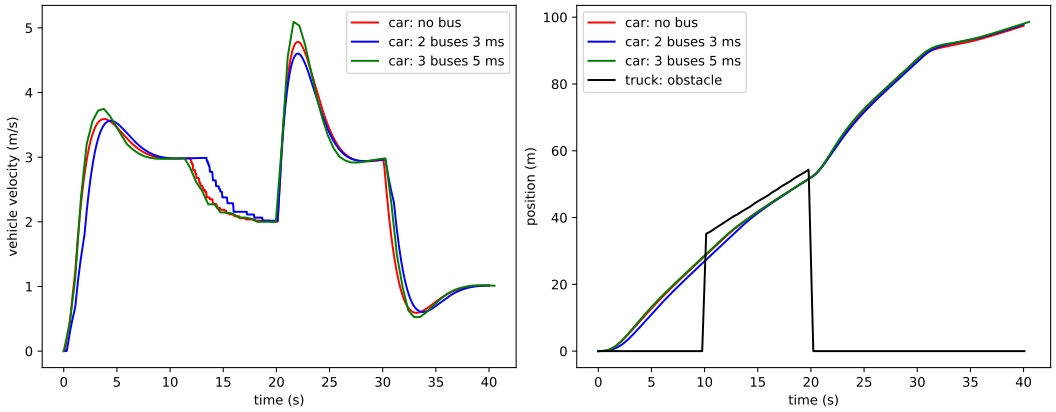


Fig. 4. The velocity and position of the car under different bus settings

The HHL Prover is an interactive theorem prover for verifying HCSP. We give the big step semantics of the HCSP process and define the relation of the form $(c, s) \Rightarrow (s', tr)$, which means that executing process c starting from state s leads to the final state s', where tr is the list of resulting tracks. The track list contains two types of track blocks: continuous modules and communication modules. We use the extended Hoare triple $\{P\}c\{Q\}$ to describe the partial correctness of the process. Assertions P and Q are predicates of the state and trajectory of the process before it starts and after it ends, respectively. During the proof, we first verify the individual behavior of the process to get its Hoare triple, and then use the synchronization rule of the trajectories to combine the assertions which have communication handshakes with each other, and finally get the complete assertion to verify the property of the parallel process.

However, the original generated HCSP model of the ACCS is so complicated which covers all the components of ACCS including `Physical`, `Software`, `Platform` and `Conn`, and would be very costly to verify thoroughly. Thus, we consider verifying the safety of the controlling strategy of the ACCS specified in `emerg` component, which is the main safety-critical part of the whole system. Concretely, we consider part of the HCSP model of the ACCS that corresponds to the controlling strategy with two main components: a controller (`Ctrl`) and a physical car (`Car`), as shown in Module 13, where the controller adopts the controlling algorithm specified in the translated `emerg` HCSP process. Furthermore, the assertion part specified in the thread `emerg` is translated to HCSP model directly to form a complete specification to be proved, which has the following form:

$$\{p = 0 \wedge v = 0\} \quad \texttt{Car(v0,p0)} \| \texttt{Ctrl} \quad \{p \leq \texttt{p\_obs}; \lceil p \leq \texttt{p\_obs} \wedge v \leq v_{\text{lim}} \rceil\} \tag{1}$$

where, the pre-condition, post-condition and invariant correspond to the assertions specified in **`assume`**, **`ensure`**, and **`invariant`** sections of thread `emerg` of the ACCS respectively. Next we use the HHL prover to prove the above safety specification.

For the component `Car`, we prove the following triple:

$$\{emp\} \quad \texttt{Car(v0,p0)} \quad \{\exists a'\ ps.\ car\_end\_state(\texttt{v0}, \texttt{p0}, a', ps) \wedge car\_block(\texttt{v0}, \texttt{p0}, a', ps)\}$$

and for `Ctrl`, we have:

$$\{emp\} \quad \texttt{Ctrl} \quad \{\exists v'\ p'\ cs.\ ctrl\_end\_state(v', p', cs) \wedge ctrl\_block(v', p', cs)\}$$

the predicates *car_end_state* and *ctrl_end_state* calculate the end state for each process, *car_block* and *ctrl_block* records the corresponding trace block, where the existence variables in the assertions represent unknown values during the communications to be determined.

By the synchronization rule, we get

$$\{loop\_inv(\texttt{v0},\texttt{p0}) \wedge emp\}\ \texttt{Car(v0,p0)}\|\texttt{Ctrl}\ \{\exists n.\ tot\_end\_state(\texttt{v0},\texttt{p0},n) \wedge tot\_block(\texttt{v0},\texttt{p0},n)\}$$

where loop_inv shows the safety property we want to prove $\texttt{loop\_inv}(v,p) := p \leq \texttt{p\_obs} \wedge v \leq v_{\text{lim}}$. The definition of post-condition implies the following formulas:

$$s \models tot\_end\_state(v,p,0) \longleftrightarrow v_s = v \wedge p_s = p$$

$$s \models tot\_end\_state(v,p,n{+}1) \longleftrightarrow s \models tot\_end\_state(v{+}a{\cdot}\texttt{period}, p{+}a{\cdot}\texttt{period}{+}\frac{1}{2}{\cdot}a{\cdot}\texttt{period}^2, n)$$

where a is the acceleration computed by InOut. Moreover, we can prove that

$$loop\_inv(v,p) \longrightarrow loop\_inv(v + a \cdot \texttt{period}, p + a \cdot \texttt{period} + \frac{1}{2} \cdot a \cdot \texttt{period}^2)$$

Overall, we can obtain that when the initial value v0 and p0 satisfy the safety property, the end state of this process after each repetition will be safe and furthermore *loop_inv* is preserved. This indicates that the safety specification (1) holds for the controlling strategy of the ACCS system.

Remark 7.1. *The verification of HCSP processes in HHL prover is performed in Isabelle/HOL, which is an interactive theorem prover and conducts proof by choosing appropriate lemmas at each step, without exhaustive checking of reachable sets of the system to be modeled. The effort of the proof in theorem proving is usually measured by the lines of proof scripts and the automation it supports. For HHL prover, it consists of a set of lemmas that correspond to the inference rules of HHL and the axioms of the assertion logic, and it provides some automation support on using these lemmas [60]. For the above case study, when the whole proof is constructed, the runtime of the file containing the model and the proof costs only a few seconds.*

8 RELATED WORK

The works most closely related to ours, in terms of providing AADL with behavior modeling support, are the behavior (BA) [40], and BLESS [42], and DEVS [5] annexes. Both BA and BLESS use state transition systems to model the discrete behavior of control systems while DEVS annex is based on Discrete-Event System Specification specifically targeted for DEVS-Suite simulation [4]. Our proposed Hybrid Annex (HA) is focused on modeling the cyber-physical interaction, and continuous behavior of physical process to be monitored and controlled by the control system.

Another work closely related to ours is MARS [70], an integrated tool-chain for modeling, analysis, verification and code generation of cyber-physical systems. With MARS, one can build a graphical model of a system to be developed with AADL⊕S/S, a combination of AADL and Simulink/Stateflow [48, 49], and then conduct extensive simulation [66]. As pointed out in [66], the integration of different paradigms may result in extra burden. Particularly, it requires to compute a type classifier for each of Simulink/Stateflow diagram, which remains challenging theoretically. Besides, like Simulink/Stateflow, the reliability of a system developed with AADL⊕S/S is not guaranteed, if it only relies on simulation, because of the incompleteness of simulation. In order to formally verify AADL⊕S/S graphical models, AADL⊕S/S is translated to HCSP automatically [67]. Using Hybrid Hoare Logic [44] and its theorem prover [64], the translated HCSP formal model can be verified. The correctness of the translation is proved with Higher-order UTP (HUTP) [65] theoretically. Finally, the notion of approximate bisimulation is proposed so that one can discretize a given HCSP process correctly in the sense of approximate bisimulation [68]. Based on which, sets of refinement rules are provided through which one can refine an HCSP process into a piece of SystemC [68] or ANSI-C code [63].

Module 13. The parallel composition of Car and Ctrl

```
module Car(v0, p0):
begin
  v := v0 ⨟ p := p0⨟
  car_v!v ⨟ car_p!p ⨟ car_a?a⨟
  (⟨ṗ = v, v̇ = a&true⟩ ⊵ [](car_v!v ⟶ car_p!p ⨟ car_a?a))*
end
endmodule

module Ctrl(period, p_obs, a_des, v_max, a_min):
procedure Comp_V_lim begin
  if (p_obs − p′ ≥ v_max²/(−2 · a_min)) {v_lim := v_max} else {
    if (p_obs − p′ > 0) {v_lim := √(−2 · a_min · (p_obs − p′))} else {v_lim := 0}
  }
end
procedure InOut begin
  car_v!v ⨟ car_p!p⨟
  p′ := p + v · period + ½ · a_des · period² ⨟ v′ := v + a_des · period ⨟ @Comp_V_lim⨟
  if (v′ ≤ v_lim) {a := a_des} else {p′ := p + v · period ⨟ @Comp_V_lim⨟
    if (v ≤ v_lim) {a := 0} else {a := a_min}
  }⨟
  car_a!a⨟
end
begin
  @InOut ⨟ (wait(period) ⨟ @InOut)*
end
endmodule

system
  Car‖Ctrl
endsystem
```

In [11], Banerjee, et al. discussed the modeling of Body Area Networks (BANs—networks of medical devices attached to a human body) with AADL using the region of impact, and the region of interest based on impacting, impacted, and monitored parameters. Boundaries of these regions are determined by both the cyber properties (*e.g.*, sensing range and communication range) and the physical properties (*e.g.*, extent of heat dissipation) of the medical devices. Each medical device with its regions is termed a Local Cyber-Physical System (LCPS), and a collection of LCPSs constitute a Global Cyber-Physical System (GCPS). The BAN-CPS annex has been proposed to specify the physical dynamics of human tissues, which are normally determined by differential equations based on Pennes' bio-heat equation.

Our proposed HA is more expressive in specifying the primitives of hybrid system models, *e.g.*, variables with data types, constants with measuring units, and behavior with complex boolean expressions. It also provides extensive support for cyber-physical interaction modeling through use of timed and communication interrupts—an essential element of hybrid system modeling not provided for to such an extent by related efforts. Exclusive support for behavior constraints and the definition of component invariants with BLESS Assertions is a novel feature of our HA.

Formalization of AADL has been explored a lot. Yang et al [69] have formalized BA by translating it into Time Abstract State Machine (TASM). Process algebra interpretation of AADL models is presented in [62]. They have translated AADL models to process algebra ACSR and Real-Time Calculus (RTC) for performance evaluation using VERSA and RTC Toolbox respectively. COMPASS

tool-set used a variant of AADL called SLIM and SuSMv model checker for safety, dependability and performance evaluation [16]. In [19], a tool called AADL2BIP based on BIP (Behavior Interaction Priority) for safety property verification has been introduced. [39] reports on an ongoing effort to capture AADL using Coq and delivers the mechanization of a significantly large subset of AADL along with verification capabilities. [12] proposes a statistical model checking-based framework that can perform quantitative evaluation of uncertainty-aware hybrid AADL designs against various performance queries. Uncertain hybrid AADL proposed in [12] is an extension of AADL and our Hybrid Annex together with other existing annexes by introducing Uncertain Annex there.

Considerable amount of efforts are made to formalize AADL, but most of them are focused on control systems with discrete behavior. To our best knowledge, expression of continuous time modeling based on an AADL annex sub-language has not been explored before.

There have been a number of modeling languages proposed for formalizing hybrid systems. The most popular is hybrid automata [6, 47], with real-time temporal logics interpreted on their behaviors as specification languages. However, analogous to state machines, hybrid automata provides little support for structured description and composition. We refer to [24], and [18], for extensive review on languages and tools introduced for specification and analysis of hybrid systems.

Platzer proposed *hybrid programs* [54, 56] and the related first-order dynamic logic, *differential dynamic logic* (d$\mathcal{L}$) for the compositional modeling and deductive verification by reducing properties of hybrid systems to properties of their parts [55]. In differential dynamic logic, the correctness of a transition behavior can be expressed as formulas based on the operational models of the hybrid system. Discrete transitions are specified as instantaneous assignments of values to state variables which, can be combined to handle simultaneous assignments. However, in his work, parallelism and communication were not well handled, that occur ubiquitously in AADL models.

Zélus [13–15] extends Lustre, a synchronous dataflow language, to support the description of continuous behavior and it implements a type system based on which some erroneous behaviors can be checked statically. There are extended Event-B framework based on refinement and proof for modeling and verifying hybrid systems, for instance, core hybrid Event-B [8–10] and Event-B hybridation [27]. In addition, PTIDES [26] is a programming model for distributed real-time embedded systems.

Some approaches on the co-modeling and co-simulation of systems have been proposed to design a system by considering its different viewpoints. Metropolis [7, 23] is a platform-based design environment for heterogeneous systems, that provides simulation, verification, and code synthesis by transforming all models to a unified meta-model language. It focuses on the discrete setting for system behaviors. Ptolemy [57] combines different models of computation in terms of actors and provides co-modeling and co-simulation for the combined models. Functional Mock-up Interface (FMI 3.0) [41] is an industrial standard maintained by the Modelica Association that enables the exchange and co-simulation of dynamic component models. It couples different simulation tools at system level by coordinating and synchronizing their respective executions. However, Ptolemy supports very limited facilities to model continuous dynamics [21], and furthermore, both Ptolemy and FMI are not designed for hardware architecture modeling and analysis as specified by AADL. Reference [31] gives a survey on the state-of-the-art techniques for co-simulation and reference [32] provides some introductory material for co-simulation of continuous systems. Other recent works on co-simulation include [33], [34], and so on.

9 CONCLUSION AND FUTURE WORK

AADL, augmented with the HA sublanguage, can model continuous behavior of the physical process to be monitored and controlled by the control system. Translation of HA semantics into HCSP, of synchronous subset of AADL annotated with HA supports modeling, simulation, and

formal verification of hybrid systems. The application of the HA for modeling, and translation into HCSP for verification, is illustrated using an example hybrid system. The Hybrid Communicating Sequential Processes (HCSP) translation of an AADL model using HA was verified using an in-house developed HHL prover. Being a step towards formalization of continuous behavior and cyber-physical interaction modeling and verification using AADL, this study has opened new domains for research in AADL.

Our future work includes enhancement of the current approach to cover asynchronous subset of AADL which is based on aperiodic thread with event-driven communication models and the development of a plug-in to Open-Source AADL Tool Environment (OSATE), the development environment for AADL modeling, for automatic translation of AADL models (with HA specifications) to HCSP processes and verification using the automatic theorem prover HHLPy [60].

ACKNOWLEDGMENTS

This work has been partially funded by the National Key R&D Program of China under grant No. 2022YFA1005100 and 2022YFA1005101, the NSFC under grant No. 62192732, 62402479, and 62032024, the CAS Project for Young Scientists in Basic Research under grant No. YSBR-040, and the Major Project of ISCAS (ISCAS-ZD-202302).

REFERENCES

[1] AADL-EMV2. [n.d.]. *AADL Error Model Annex v2.* https://saemobilus.sae.org/content/as5506/1

[2] Ehsan Ahmad, Yunwei Dong, Shuling Wang, Naijun Zhan, and Liang Zou. 2014. Adding Formal Meanings to AADL with Hybrid Annex. In *FACS 2014 (LNCS, Vol. 8997)*, Ivan Lanese and Eric Madelaine (Eds.). Springer, 228–247. https://doi.org/10.1007/978-3-319-15317-9_15

[3] Ehsan Ahmad, Brian R. Larson, Stephen C. Barrett, Naijun Zhan, and Yunwei Dong. 2014. Hybrid annex: an AADL extension for continuous behavior and cyber-physical interaction modeling. In *HILT 2014*, Michael B. Feldman and S. Tucker Taft (Eds.). ACM, 29–38. https://doi.org/10.1145/2663171.2663178

[4] Ehsan Ahmad and Hessam S. Sarjoughian. 2023. An Environment for Developing Simulatable AADL-DEVS Models. *Simul. Model. Pract. Theory* 123 (2023), 102690. https://doi.org/10.1016/J.SIMPAT.2022.102690

[5] Ehsan M. Ahmad and Hessam S. Sarjoughian. 2019. A Behavior Annex For AADL Using The DEVS Formalism. In *SpringSim 2019*, Alberto A. Del Barrio, Christopher J. Lynch, Fernando J. Barros, Xiaolin Hu, and Andrea D'Ambrogio (Eds.). IEEE, 1–12. https://doi.org/10.23919/SPRINGSIM.2019.8732894

[6] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. 1992. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In *Hybrid Systems (LNCS, Vol. 736)*, Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel (Eds.). Springer, 209–229. https://doi.org/10.1007/3-540-57318-6_30

[7] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto L. Sangiovanni-Vincentelli. 2003. Metropolis: An Integrated Electronic System Design Environment. *Computer* 36, 4 (2003), 45–52. https://doi.org/10.1109/MC.2003.1193228

[8] Richard Banach. 2024. Core Hybrid Event-B III: Fundamentals of a reasoning framework. *Sci. Comput. Program.* 231 (2024), 103002. https://doi.org/10.1016/J.SCICO.2023.103002

[9] Richard Banach, Michael J. Butler, Shengchao Qin, Nitika Verma, and Huibiao Zhu. 2015. Core Hybrid Event-B I: Single Hybrid Event-B machines. *Sci. Comput. Program.* 105 (2015), 92–123. https://doi.org/10.1016/J.SCICO.2015.02.003

[10] Richard Banach, Michael J. Butler, Shengchao Qin, and Huibiao Zhu. 2017. Core Hybrid Event-B II: Multiple cooperating Hybrid Event-B machines. *Sci. Comput. Program.* 139 (2017), 1–35. https://doi.org/10.1016/J.SCICO.2016.12.003

[11] Ayan Banerjee, Sailesh Kandula, Tridib Mukherjee, and Sandeep K. S. Gupta. 2012. BAND-AiDe: A Tool for Cyber-Physical Oriented Analysis and Design of Body Area Networks and Devices. *ACM Trans. Embed. Comput. Syst.* 11, S2 (2012), 49:1–49:29. https://doi.org/10.1145/2331147.2331159

[12] Yongxiang Bao, Mingsong Chen, Qi Zhu, Tongquan Wei, Frédéric Mallet, and Tingliang Zhou. 2017. Quantitative Performance Evaluation of Uncertainty-Aware Hybrid AADL Designs Using Statistical Model Checking. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 36, 12 (2017), 1989–2002. https://doi.org/10.1109/TCAD.2017.2681076

[13] Albert Benveniste, Timothy Bourke, Benoît Caillaud, Jean-Louis Colaço, Cédric Pasteur, and Marc Pouzet. 2018. Building a Hybrid Systems Modeler on Synchronous Languages Principles. *Proc. IEEE* 106, 9 (2018), 1568–1592. https://doi.org/10.1109/JPROC.2018.2858016

[14] Timothy Bourke, Francois Carcenac, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. 2017. A Synchronous Look at the Simulink Standard Library. *ACM Trans. Embed. Comput. Syst.* 16, 5s (2017), 176:1–176:24. https://doi.org/10.1145/3126516

[15] Timothy Bourke and Marc Pouzet. 2013. Zélus: a synchronous language with ODEs. In *HSCC 2013*, Calin Belta and Franjo Ivancic (Eds.). ACM, 113–118. https://doi.org/10.1145/2461328.2461348

[16] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. 2011. Safety, Dependability and Performance Analysis of Extended AADL Models. *Comput. J.* 54, 5 (2011), 754–775. https://doi.org/10.1093/COMJNL/BXQ024

[17] Michael Stephen Branicky. 1995. *Studies in Hybrid Systems: Modeling, Analysis, and Control.* Massachusetts Institute of Technology, Cambridge, MA, USA.

[18] Luca P. Carloni, Roberto Passerone, Alessandro Pinto, and Alberto L. Sangiovanni-Vincentelli. 2006. Languages and Tools for Hybrid Systems Design. *Found. Trends Electron. Des. Autom.* 1, 1/2 (2006). https://doi.org/10.1561/1000000001

[19] Mohamed Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis. 2008. Translating AADL into BIP - Application to the Verification of Real-Time Systems. In *MODELS 2008 (LNCS, Vol. 5421)*, Michel R. V. Chaudron (Ed.). Springer, 5–19. https://doi.org/10.1007/978-3-642-01648-6_2

[20] COMPASS. [n.d.]. *COMPASS.* http://www.compass-toolset.org/

[21] Fabio Cremona, Marten Lohstroh, David Broman, Edward A. Lee, Michael Masin, and Stavros Tripakis. 2019. Hybrid co-simulation: it's about time. *Softw. Syst. Model.* 18, 3 (2019), 1655–1679. https://doi.org/10.1007/S10270-017-0633-6

[22] D-MILS. [n.d.]. *Distributed MILS.* http://www.d-mils.org

[23] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu. 2007. A Next-Generation Design Framework for Platform-based Design. In *DVCon 2007.*

[24] Jennifer M. Davoren and Anil Nerode. 2000. Logics for hybrid systems. *Proc. IEEE* 88, 7 (2000), 985–1010. https://doi.org/10.1109/5.871305

[25] Julien Delange. 2017. *AADL in Practice.* Reblochon Development Company.

[26] Patricia Derler, Thomas Huining Feng, Edward A Lee, Slobodan Matic, Hiren D Patel, Yang Zhao, and Jia Zou. 2008. PTIDES: A programming model for distributed real-time embedded systems. *University of California, Berkeley, EECS Technical Report. EECS-2008-72* (2008).

[27] Guillaume Dupont, Yamine Aït Ameur, Neeraj Kumar Singh, and Marc Pantel. 2021. Event-B Hybridation: A Proof and Refinement-based Framework for Modelling Hybrid Systems. *ACM Trans. Embed. Comput. Syst.* 20, 4 (2021), 35:1–35:37. https://doi.org/10.1145/3448270

[28] Esterel Technologies. [n.d.]. *SCADE suite.* http://www.esterel-technologies.com/products/scade-suite

[29] Peter Feiler and David Gluch. 2012. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language.* Addison-Wesley.

[30] Peter Feiler, Jörgen Hansson, Dionisio de Niz, and Lutz Wrage. 2009. *System Architecture Virtual Integration: An industrial Case Study.* Technical Report CMU/SEI-2009-TR-017. SEI, CMU.

[31] Cláudio Gomes, Casper Thule, David Broman, Peter Gorm Larsen, and Hans Vangheluwe. 2018. Co-Simulation: A Survey. *ACM Comput. Surv.* 51, 3 (2018), 49:1–49:33. https://doi.org/10.1145/3179993

[32] Cláudio Gomes, Casper Thule, Peter Gorm Larsen, Joachim Denil, and Hans Vangheluwe. 2018. Co-simulation of Continuous Systems: A Tutorial. *CoRR* abs/1809.08463 (2018). arXiv:1809.08463

[33] Simon Thrane Hansen, Casper Thule, Cláudio Gomes, Kenneth Lausdahl, Frederik Palludan Madsen, Giuseppe Abbiati, and Peter Gorm Larsen. 2024. Co-simulation at different levels of expertise with Maestro2. *J. Syst. Softw.* 209 (2024), 111905. https://doi.org/10.1016/J.JSS.2023.111905

[34] Simon Thrane Hansen, Casper Thule, Cláudio Gomes, Jaco van de Pol, Maurizio Palmieri, Emin Oguz Inci, Frederik Palludan Madsen, Jesus Alfonso, José Ángel Castellanos, and José Manuel Rodriguez-Fortun. 2022. Verification and synthesis of co-simulation algorithms subject to algebraic loops and adaptive steps. *Int. J. Softw. Tools Technol. Transf.* 24, 6 (2022), 999–1024. https://doi.org/10.1007/S10009-022-00686-8

[35] Jifeng He. 1994. *From CSP to hybrid systems.* Prentice Hall International (UK) Ltd., GBR, 171–189.

[36] Thomas A. Henzinger. 1996. The Theory of Hybrid Automata. In *LICS 1996.* IEEE Computer Society, 278–292. https://doi.org/10.1109/LICS.1996.561342

[37] Thomas A. Henzinger and Joseph Sifakis. 2006. The Embedded Systems Design Challenge. In *FM 2006 (LNCS, Vol. 4085)*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer, 1–15. https://doi.org/10.1007/11813040_1

[38] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (1978), 666–677. https://doi.org/10.1145/359576.359585

[39] Jérôme Hugues, Lutz Wrage, John Hatcliff, and Danielle Stewart. 2022. Mechanization of a Large DSML: An Experiment with AADL and Coq. In *MEMOCODE 2022.* IEEE, 1–9. https://doi.org/10.1109/MEMOCODE57689.2022.9954589

[40] SAE International. 2022. *SAE AS5506 Rev. D Architecture Analysis and Design Language (AADL).* SAE International.

[41] Andreas Junghanns, Cláudio Gomes, Christian Schulze, Klaus Schuch, Pierre R., Matthias Blaesken, Irina Zacharias, Andreas Pillekeit, Karl Wernersson, Torsten Sommer, Christian Bertsch, Torsten Blochwitz, and Masoud Najafi. 2021. The Functional Mock-up Interface 3.0 - New Features Enabling New Applications. In *Proceedings of 14th Modelica Conference 2021*.

[42] Brian R. Larson, Patrice Chalin, and John Hatcliff. 2013. BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software. In *NASA Formal Methods 2013 (LNCS, Vol. 7871)*, Guillaume Brat, Neha Rungta, and Arnaud Venet (Eds.). Springer, 276–290. https://doi.org/10.1007/978-3-642-38088-4_19

[43] Edward A. Lee. 2000. What's Ahead for Embedded Software? *Computer* 33, 9 (2000), 18–26. https://doi.org/10.1109/2.868693

[44] Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. 2010. A Calculus for Hybrid CSP. In *APLAS 2010 (LNCS, Vol. 6461)*, Kazunori Ueda (Ed.). Springer, 1–15. https://doi.org/10.1007/978-3-642-17164-2_1

[45] John Lygeros. 2004. Lecture Notes on Hybrid Systems. In *Notes for an ENSIETA workshop*.

[46] Oded Maler, Zohar Manna, and Amir Pnueli. 1991. From Timed to Hybrid Systems. In *Real-Time: Theory in Practice, REX Workshop (LNCS, Vol. 600)*, J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg (Eds.). Springer, 447–484. https://doi.org/10.1007/BFB0032003

[47] Zohar Manna and Amir Pnueli. 1992. Verifying Hybrid Systems. In *Hybrid Systems (LNCS, Vol. 736)*, Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel (Eds.). Springer, 4–35. https://doi.org/10.1007/3-540-57318-6_22

[48] MathWorks Inc. 2013. *Simulink User's Guide*. `http://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf`.

[49] MathWorks Inc. 2013. *Stateflow User's Guide*. `http://www.mathworks.com/help/pdf_doc/stateflow/sf_ug.pdf`.

[50] Robin Milner. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92. Springer. https://doi.org/10.1007/3-540-10235-3

[51] Robin Milner. 1989. *Communication and concurrency*. Prentice Hall.

[52] Hana Mkaouar, Bechir Zalila, Jérôme Hugues, and Mohamed Jmaiel. 2020. A formal approach to AADL model-based software engineering. *Int. J. Softw. Tools Technol. Transf.* 22, 2 (2020), 219–247. https://doi.org/10.1007/S10009-019-00513-7

[53] OSATE. 2024. Open Source AADL Tool Environment, Version 2.14.0. https://www.osate.org. (accessed on March 24, 2024).

[54] André Platzer. 2007. Differential Dynamic Logic for Verifying Parametric Hybrid Systems. In *TABLEAUX 2007 (LNCS, Vol. 4548)*, Nicola Olivetti (Ed.). Springer, 216–232. https://doi.org/10.1007/978-3-540-73099-6_17

[55] André Platzer. 2008. Differential Dynamic Logic for Hybrid Systems. *J. Autom. Reason.* 41, 2 (2008), 143–189. https://doi.org/10.1007/S10817-008-9103-8

[56] André Platzer. 2010. *Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics*. Springer. https://doi.org/10.1007/978-3-642-14509-4

[57] Claudius Ptolemaeus (Ed.). 2014. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org. http://ptolemy.org/books/Systems

[58] SAVI. [n.d.]. *System Architecture Virtual Integration*. http://savi.avsi.aero/

[59] Oliver Scheid. 2015. *AUTOSAR Compendium - Part 1: Application & RTE*. Bruchsal: CreateSpace Independent Publishing Platform.

[60] Huanhuan Sheng, Alexander Bentkamp, and Bohua Zhan. 2023. HHLPy: Practical Verification of Hybrid Systems Using Hoare Logic. In *FM 2023 (LNCS, Vol. 14000)*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.). Springer, 160–178. https://doi.org/10.1007/978-3-031-27481-7_11

[61] SMACCM. [n.d.]. *Secure Mathematically-Assured Composition of Control Models*. http://loonwerks.com/projects/smaccm.html

[62] Oleg Sokolsky, Insup Lee, and Duncan Clarke. 2009. Process-Algebraic Interpretation of AADL Models. In *Reliable Software Technologies - Ada-Europe 2009 (LNCS, Vol. 5570)*, Fabrice Kordon and Yvon Kermarrec (Eds.). Springer, 222–236. https://doi.org/10.1007/978-3-642-01924-1_16

[63] Shuling Wang, Zekun Ji, Xiong Xu, Bohua Zhan, Qiang Gao, and Naijun Zhan. 2024. Formally Verified C Code Generation from Hybrid Communicating Sequential Processes. In *ICCPS 2024*. IEEE, 123–134. https://doi.org/10.1109/ICCPS61052.2024.00018

[64] Shuling Wang, Naijun Zhan, and Liang Zou. 2015. An Improved HHL Prover: An Interactive Theorem Prover for Hybrid Systems. In *ICFEM 2015 (LNCS, Vol. 9407)*, Michael J. Butler, Sylvain Conchon, and Fatiha Zaïdi (Eds.). Springer, 382–399. https://doi.org/10.1007/978-3-319-25423-4_25

[65] Xiong Xu, Jean-Pierre Talpin, Shuling Wang, Bohua Zhan, and Naijun Zhan. 2023. Semantics Foundation for Cyber-physical Systems Using Higher-order UTP. *ACM Trans. Softw. Eng. Methodol.* 32, 1 (2023), 9:1–9:48. https://doi.org/10.1145/3517192

[66] Xiong Xu, Shuling Wang, Bohua Zhan, Xiangyu Jin, Jean-Pierre Talpin, and Naijun Zhan. 2022. Unified graphical co-modeling, analysis and verification of cyber-physical systems by combining AADL and Simulink/Stateflow. *Theor. Comput. Sci.* 903 (2022), 1–25. https://doi.org/10.1016/J.TCS.2021.11.008

[67] Xiong Xu, Bohua Zhan, Shuling Wang, Jean-Pierre Talpin, and Naijun Zhan. 2023. A denotational semantics of Simulink with higher-order UTP. *J. Log. Algebraic Methods Program.* 130 (2023), 100809. https://doi.org/10.1016/J.JLAMP.2022.100809

[68] Gaogao Yan, Li Jiao, Shuling Wang, Lingtai Wang, and Naijun Zhan. 2020. Automatically Generating SystemC Code from HCSP Formal Models. *ACM Trans. Softw. Eng. Methodol.* 29, 1 (2020), 4:1–4:39. https://doi.org/10.1145/3360002

[69] Zhibin Yang, Kai Hu, Dianfu Ma, and Lei Pi. 2009. Towards a formal semantics for the AADL behavior annex. In *DATE 2009*, Luca Benini, Giovanni De Micheli, Bashir M. Al-Hashimi, and Wolfgang Müller (Eds.). IEEE, 1166–1171. https://doi.org/10.1109/DATE.2009.5090839

[70] Bohua Zhan, Xiong Xu, Qiang Gao, Zekun Ji, Xiangyu Jin, Shuling Wang, and Naijun Zhan. 2024. Mars 2.0: A Toolchain for Modeling, Analysis, Verification and Code Generation of Cyber-Physical Systems. *CoRR* abs/2403.03035 (2024). https://doi.org/10.48550/ARXIV.2403.03035 arXiv:2403.03035

[71] Naijun Zhan, Bohua Zhan, Shuling Wang, Dimitar P. Guelev, and Xiangyu Jin. 2023. A Generalized Hybrid Hoare Logic. *CoRR* abs/2303.15020 (2023). https://doi.org/10.48550/ARXIV.2303.15020 arXiv:2303.15020

[72] Chaochen Zhou, Ji Wang, and Anders P. Ravn. 1995. A Formal Description of Hybrid Systems. In *Hybrid Systems III: Verification and Control 1995 (LNCS, Vol. 1066)*, Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag (Eds.). Springer, 511–530. https://doi.org/10.1007/BFB0020972