Adding Formal Meanings to AADL with Hybrid Annex

Ehsan Ahmad[1,2], Yunwei Dong[1], Shuling Wang[2], Naijun Zhan[2⋆], and Liang Zou[2]

[1] School of Computer Science, Northwestern Polytechnical University
[2] State Key Lab. of Comput. Sci. Inst. of Software, Chinese Academy of Sciences

Abstract. AADL is a Model-Based Engineering language for architectural analysis and specification of real-time embedded systems with stringent performance requirements (e.g. fault-tolerance, security, safety-critical etc.). However, core AADL lacks of a mechanism for modeling continuous evolution of physical processes which are controlled by digital controllers. In our previous work, we have introduced Hybrid Annex—an AADL extension for continuous behavior and cyber-physical interaction modeling based on Hybrid Communicating Sequential Processes (HCSP). In this paper, we present formal semantics of the synchronous subset of AADL models annotated with Hybrid Annex specifications using HCSP. The semantics are then used to verify correctness of AADL models (with Hybrid Annex specifications) using an in-house developed theorem prover — Hybrid Hoare Logic (HHL) prover.

Keywords: AADL, Formal Semantics, HCSP, Hybrid Annex, Hybrid Systems

1 Introduction

Embedded Systems (ESs) make use of computer units to control physical processes so that the behavior of the controlled processes meets expected requirements. They have become ubiquitous in our daily life, e.g. automotive, aerospace, consumer electronics, communications, medical, manufacturing and so on. ESs are used to carry out highly complex and often critical functions such as to monitor and control industrial plants, complex transportation equipment, communication infrastructure, etc. The development process of ESs is widely recognized as a highly complex and challenging task. A thorough validation and verification activity is necessary to enhance the quality of ESs and, in particular, to fulfill the quality criteria mandated by the relevant standards. How to design correct embedded systems is a grand challenge for computer science and control theory. Model-Based Engineering (MBE) is considered as an effective way of developing correct complex ESs, and has been successfully applied in industry [8, 10]. In the framework of MBE, a model of the system to be developed is defined at the beginning; then extensive analysis and verification are conducted based on the model so that errors can be detected and corrected at early stages of design of the system. Afterwards, model transformation techniques are applied to transform abstract formal models into more concrete models, even into source code. Hybrid Systems (HSs) are

⋆ The corresponding author: No. 4, South Fourth Street, Zhong Guan Cun, Beijing, 100190, P.R. China.

mathematical models with precise mathematical semantics for ESs, wherein continuous physical dynamics are combined with discrete transitions. Based on HSs, rigorous analysis and verification of ESs become feasible.

Architectural Analysis & Design Language (AADL) is an SAE International standard and is an ADL for ESs [14]. It is based on architectural-centric MBE approach. It has been introduced to cope with embedded system design challenges by minimizing model inconsistency, decreasing mismatched assumptions of different stakeholders and supporting dependability predictions through analyzable architecture development. However, core AADL only provides support for structural modeling of embedded computing units and nothing related to detailed behavior of the software and physical processes which are controlled by the software can be modeled. So, as a result not only the reliability prediction, performance analysis and verification of AADL models are not precise enough, but also the cost is very high. To address these issues, Behavior Annex (BA) and BLESS annex are introduced for more precise behavior modeling using state transition systems with guards and actions [13, 15]. Both BA and BLESS annex are intended to model discrete behavior of a control system. However, in practice, it is quite common that a control system contains continuous behavior, in particular, the behavior of controlled physical processes. In [18], we have introduced a lightweight language extension to AADL called the Hybrid Annex (HA) for continuous behavior and cyber-physical interaction modeling.

In addition, formal semantics are especially important for safety-critical systems and are the basis for formal analysis and verification of these systems. Although considerable amount of literature is available on formalization of AADL for performance and dependability analysis but the majority of the literature is focused on discrete behavior (behavior of the computing units) only and formalization of the continuous part of hybrid systems and the cyber-physical interaction (communication between the computing unit and the physical processes) is not addressed at all.

Hence, in order to use AADL for modeling and verification of hybrid system, it is required not only to define the formal semantics of the core language but also define the formal semantics of the dedicated annex (HA) used for continuous behavior modeling, in such a language which is designed to model and formally verify the HSs.

1.1 A Running Example

Throughout this paper, we use the Water Level Control System (WLCS) [9] as a running example to explain the motivation, to illustrate how to apply the HA extension to model HSs, as well as the use of proposed formal semantics for verification. As depicted in Fig. 1, WLCS consists of two main parts, the *water tank* and the *controller*. Continuous change of water level h in the water tank is described by

$$\begin{cases} \dot{h} = v \cdot Q_{max} - \pi \cdot r^2 \cdot \sqrt{2 \cdot g \cdot h} \\ \dot{v} = 0 \qquad v \in \{0, 1\} \end{cases}$$

where $Q_{max} = 0.007 m^3 s^{-1}$, $\pi = 3.14$, $r = 0.0254 m$, and $g = 9.8 m s^{-2}$. $v \cdot Q_{max}$ is the water inflow Q_{in} into the tank, which takes the value 0 or Q_{max} depending on if the valve v is close or open. $\pi \cdot r^2 \cdot \sqrt{2 \cdot g \cdot h}$ stands for water outflow Q_{out}, which follows

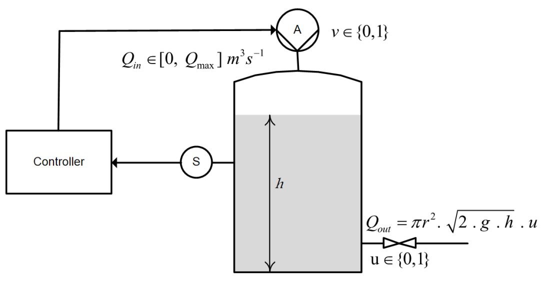
A
$v \in \{0,1\}$
$Q_{in} \in [0, Q_{max}]\ m^3 s^{-1}$
Controller
S
h
$Q_{out} = \pi r^2 . \sqrt{2 . g . h} . u$
$u \in \{0,1\}$

Fig. 1. WLCS diagram - a classical hybrid system

Torricelli's law.[1] *S*, *v*, and *A* represent the sensor, inflow valve and the actuator used to control inflow valve (*v*) respectively. The main goal of WLCS is to maintain water level *h* between a specified limit which is $0.30m$ to $0.60m$, by controlling the inflow valve *v* to be close or open. The control command is computed by the controller based on the water level observed by sensor *S* and the predefined control strategy. The command is then sent to actuator *A* to control the inflow valve *v* appropriately.

Core AADL with HA can be used to model the structural architecture of the controller, continuous behavior of the water tank and the cyber-physical interaction between them. Assurance of the correct system behavior and the certification of real-time and dependability related properties demand a system level formal verification approach which is not addressed at all in the existing literature on formalization of AADL. In this paper, formal semantics for AADL models with HA specifications are proposed to fill this gap.

Contributions: In this paper, we propose formal semantics of AADL execution model with synchronous communication and HA using HCSP. The contribution of the paper is twofold. Firstly, we illustrate the use of HA to model the continuous behavior and the communication between the controller and the physical process. Continuous system behavior specified using HA can easily be attached to predefined AADL components. Secondly, formal semantics of AADL execution model and synchronous communication mechanism based on a language (HCSP) suitable for hybrid systems modeling and analysis is presented. Formal semantics are then used to verify correctness of AADL model annotated with HA specifications using an in-house developed theorem prover known as HHL prover [17, 20].

Outline: Section 2 introduces HCSP, HHL and AADL with its execution model and synchronous communication semantics. Section 3 presents the continuous behavior and cyber-physical interaction modeling using HA and Section 4 discusses the formal semantics of AADL execution model along with synchronous communication. Section 5 illustrates verification of the case study using HHL prover. Section 6 presents a summary of the related work. Section 7 concludes this paper and discusses the future work.

[1] Normally, $Q_{out} = \pi \cdot r^2 \cdot \sqrt{2 \cdot g \cdot h} \cdot u$. But for simplicity, we take $u = 1$ here.

2 Preliminaries

This section presents an overview of HCSP by highlighting primitive language constructs. The specification logic HHL for reasoning about HCSP behavior is then introduced briefly. Basic AADL notions and notations are also presented with emphases on execution model and synchronous data communication semantics.

2.1 Overview of HCSP

HCSP is an extension of Hoare's Communicating Sequential Processes (CSP) for modeling HSs [6, 9] . In HCSP, differential equations are introduced to model continuous evolution of the physical processes along with interrupts, so both discrete and continuous behaviors are still modeled as *processes*. A hybrid system in HCSP is a parallel composition of networked sequential processes interacting through dedicated channels, or a repetition of a sub-system. Note that processes in parallel can only interact through communication and no shared variables are allowed. The set of variables is denoted by $\mathcal{V} = \{x, y, z, ...\}$ and the set of channels is denoted by $\Sigma = \{ch_1, ch_2, ch_3, ...\}$. The processes of HCSP are constructed as follows:

$$
\begin{aligned}
P ::= {} & \textbf{skip} \mid x := e \mid wait\ d \mid ch?x \mid ch!e \mid P;Q \mid B \rightarrow P \mid P \sqcup Q \mid \ \talloblong_{i \in I}\,(ch_i* \rightarrow Q_i) \mid P^* \\
& \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq_d Q \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq \talloblong_{i \in I}(ch_i* \rightarrow Q_i) \\
S ::= {} & P \mid S^* \mid S \| S
\end{aligned}
$$

Here P, Q, and Q_i represent sequential processes, whereas S stands for a (sub)system; $ch, ch_i \in \Sigma$ are communication channels, while ch_i* is a communication event which can be either an input event $ch?x$ or an output event $ch!e$; B and e are boolean and arithmetic expressions respectively, and d is a non-negative real constant.

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

HCSP supports both sequential and concurrent composition. A sequentially composed process $P;Q$ behaves as P first, and if it terminates, as Q afterwards. The alternative process $B \rightarrow P$ behaves as P only if B is true and terminates otherwise. Internal choice between processes P and Q, denoted as $P \sqcup Q$ is resolved by the process itself. Communication controlled external choice $\talloblong_{i \in I}(ch_i* \rightarrow Q_i)$ specifies that as soon as one of the communications ch_i* takes place, the process starts behaving as respective process Q_i. The repetition P^* executes P for an arbitrary finite number of times, and the choice of the number of times is non-deterministic.

Continuous evolution is specified as $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$. Real variables s evolve continuously according to differential equations $\mathcal{F}$ as long as the boolean expression B is true. B defines the domain of s. Interruption of continuous evolution due to B (as soon as it becomes false) is known as *Boundary Interrupt*. Continuous evolution can also be preempted due to the following interrupts:

– *Timeout Interrupt*: $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq_d Q$ behaves like $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$, if the continuous evolution terminates before d time units. Otherwise, after d time units of evolution according to $\mathcal{F}$, it moves on to execute Q.
– *Communication Interrupt*: $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq [\!]_{i \in I}(ch_i * \rightarrow Q_i)$ behaves like $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$, except that the continuous evolution is preempted whenever one of the communications $ch_i *$ takes place, which is followed by respective Q_i.

Finally, S defines a HCSP system on the top level. A parallel composition $S_1 \| S_2$ behaves as if S_1 and S_2 run independently except that they need to synchronize along the common communication channels.

Ehsan Ahmad, Brian R. Larson, Stephen C. Barrett, Naijun Zhan, Yunwei Dong, *Hybrid Annex: An AADL extention for continuous behavior and cyber-physical interaction modeling*, accepted for publication, HILT'14, 2014.

2.2 Overview of Hybrid Hoare Logic (HHL)

In [11], we have extended Hoare Logic to hybrid systems, by adding history formulas to describe continuous properties that hold throughout the whole execution of HCSP processes. The history formulas are defined by Duration Calculus (DC), which is a real arithmetic extension of Interval Temporal Logic (ITL) for specifying and reasoning about real-time systems. The mainly used assertion $\lceil S \rceil$, where S is a state formula, means that S holds everywhere inside the considered interval.

In HHL, the specification for a sequential process P is of the form $\{Pre\}\ P\ \{Post; HF\}$, where $Pre, Post$ represent pre-/post-conditions, expressed by first-order logic, to specify properties of variables held at starting and termination of the execution of P, and HF is a history formula, expressed by DC, to record the execution history of P, including real-time and continuous properties. The specification for a parallel process is then defined by assigning to each sequential component the respective pre-/post-conditions and history formula, that is

$$\{Pre_1, Pre_2\}\ P_1 \| P_2\ \{Post_1, Post_2; HF_1, HF_2\}$$

In HHL, each HCSP construct is axiomatized by a set of axioms and inference rules. Based on the inference system, we have implemented an interactive theorem prover for HHL in proof assistant Isabelle/HOL. The tool can be downloaded from lcs.ios.ac.cn/∼znj/HHLProver. For further details on HCSP, HHL and HHL prover, we refer to [17].

The WLCS can be modeled, using HCSP, as a parallel composition of the water tank and the controller, whose specification is given as follows:

$$\{Pre_1, Pre_2\}\ Watertank \| Controller \{0.30 \leq h \leq 0.60, True; \lceil 0.30 \leq h \leq 0.60 \rceil, True\}.$$

As shown by the postcondition and history formula corresponding to *Watertank*, the water level h will always be kept in the range $[0.30, 0.60]$. The details for the modeling and verification of the WLCS system are described in the rest of the paper.

2.3 Overview of AADL

AADL contains components for both the application software, and the execution hardware of an embedded system, and supports textual, graphical and XML Metadata Interchange (XMI) specification formats. Components with *type* and *implementation* classifiers are instantiated and connected together to structure the system architecture. AADL core language constructs are categorized into application software, execution platform and composite components. The *system* component represents a composite entity containing software, execution platform or system components.

Components and Connections Execution platform category represents computation and communication resources including *processor*, *memory*, *bus* and *device* components. A processor component represents the hardware and software responsible for thread scheduling and execution. Properties can be assigned to a processor component to specify scheduling policies, high-level operating system services and communication protocols. A memory component is used to represent storage entities for data and code. A device component can model a physical entity in the external environment: a plant or the software simulation of the plant. It can also be used as an interactive component like sensor or actuator. A bus component represents the physical connections among execution platform components.

Application software category consists of *process*, *data*, *subprogram*, *thread*, and *thread group* components. A process component represents the protected address space, which is bound to a memory component. A data component can be used to abstract data type, local data or parameter of a subprogram. A subprogram models the executable code which is called, with parameters, by thread and other subprograms. Thread is the only schedulable component with execution semantics to model system execution behavior. A thread represents sequential flow of the execution and the associated semantic automation describes life cycle of the thread.

A component type declaration defines interface elements and may contain *Features*. Features contain communication ports. AADL supports *data*, *event* and *event data* ports to transmit and receive data, control, and control and data respectively. Port communication is typed and directional. An *in* port receives data/control and an *out* port sends data/control while an *in out* port can send and receive data/control. Communication is realized through *connections* between ports, parameters and access to shared data.

This paper is focused on formalizing execution model semantics of AADL with synchronous communication in which threads are communicating through data ports. Due to the page limitation, these two aspects are briefly discussed in the rest of this section. We refer to AADL standard document AS5506-B [14] for further details.

Execution Model AADL structure model (hierarchical composition of the components) does not contain explicit information about the execution model, instead it is specified by the execution control automaton and properties at model and project level. AADL execution model deals with execution control automaton, thread dispatch strategy, scheduling and execution and fault handling. Our focus in this paper is on the execution model and (synchronous) communication formalism and the formalization of thread fault handling and modal semantics will be subject of a later paper.

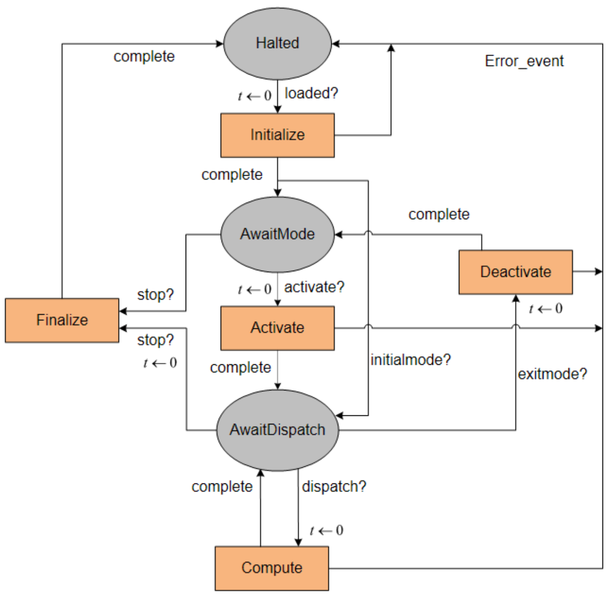

Fig. 2. Thread execution state machine

Thread execution life cycle, as depicted in Fig. 2 is same for every thread. Thread execution life cycle consists of two types of 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.

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 dependant. 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 dispatch port.

A thread is initialized after the respective process is loaded into memory and is directly moved to *AwaitDispatch* state if it is active in current process mode otherwise it is moved to *AwaitMode* state. Thread dispatch is controlled by *Enabled(t)* function and *Wait_For_Dispatch* invariant in *AwaitDispatch* state. The 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 assertion in any active state is violated, thread is moved to the *Halted* state.

Synchronous Communication Inter-thread communication in synchronous data flow communication pattern can either be *immediate* or *delayed* depending on data port con-

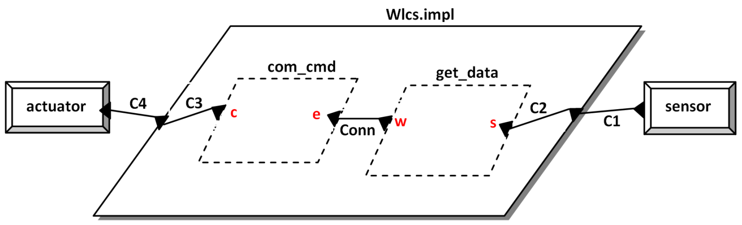

Fig. 3. AADL model of WLCS using graphical notations

nections. For an immediate connection, data is transmitted whenever the source thread completes its execution and meanwhile destination thread is suspended. The value received at destination is the value produced at the latest completion of source thread. For immediate connection, threads must share a common dispatch. For a delayed connection the output is transmitted at the deadline of the source thread so it is available to the destination thread at the next dispatch. The value received at destination is the value produced at the latest deadline of the source thread. For delayed connection, threads do not need to share a common dispatch.

2.4 WLCS Discrete Behavior Modeling

Depicted using the AADL graphical notations, Fig. 3 shows the architecture of the controller of the running example (WLCS), while the detailed behavior of the water tank is presented in Listing 1 and is discussed in the next section. The continuous state of the water tank, i.e. the water level *h*, is measured by a *sensor* and the output is sent to the controller process *Wlcs.impl* through connection *C*1, which contains two periodic threads *get_data* and *com_cmd*. Threads are connected through an immediate connection *Conn*. Thread *get_data* samples sensor data through its in data port *s* along connection *C2* on every dispatch and sends computed data to *com_cmd* thread through out data port *w* along connection *Conn*. Control command according to control laws, is computed by *com_cmd* thread and is sent to *actuator* through out data port *c* using connections *C3* and *C4*.

In relation to execution model and synchronous communication mechanisms discussed earlier, threads *get_data* and *com_cmd* share same execution life cycle presented in Fig. 2 and the immediate connection *Conn* between them is as discussed in Section 2.3. Formal semantics of this discrete behavior of the controller component (consisting of threads and communication between them) are presented in Section 4 in detail.

3 Hybrid Annex

Hybrid Annex (HA) has been proposed to equip AADL for hybrid system modeling and analysis [18]. An HA specification can be attached to either the implementation classifier of an AADL device component or to an abstract component to model the continuous behavior of the interactive components (i.e. sensors and actuators) or to model the behavior of a physical process respectively. An HA specification may contain six sections:

Listing 1. AADL WaterTank Component Specification using Hybrid Annex

```
abstract WaterTank
  features
    cc: in data port WLCS::ValveStatus;
    wl: out data port WLCS::WaterLevel;
 end WaterTank;
abstract implementation WaterTank.impl
annex hybrid {**
  variables
    t :  WLCS::Time   -- value of current time
    v :  WLCS::ValveStatus -- inflow valve status
    h :  WLCS::WaterLevel   -- current water level
  constants
    Qmax = 0.007 cmps -- maximum water inflow
    g    = 9.8 mpss -- gravitational force
    pi   = 3.14
    r    = 0.0254 m -- radius of outflow valve
    u    = 1 -- outflow valve status, permanently open
    period = 0.01 sec -- sampling period
  behavior
    Plant ::=  t := 0 &
          'DT 1 h =(v*Qmax)-(pi*(r^2)*1.414*(g^0.5)*(h^0.5)*u)' &
          'DT 1 v = 0' <(t<period)> [[> wl!(h)]]> GetCmd
    GetCmd ::= cc?(v)
    WaterTank ::= repeat (Plant)
**};
end WaterTank.impl;
```

assert, **invariant**, **variables**, **constants**, **channels**, and **behavior** to specify predicates, predicates that must hold throughout continuous behavior model, local variables, constants, communication channels and continuous behavior respectively.

Listing 1 presents the complete textual AADL model with HA specifications of the water tank of WLCS–the running example. The type classifier declares the interface of the `WaterTank` component with two data ports. The out data port `wl` is used to send the current water level, while the in data port `cc` is used to receive the control command. The `WaterTank` is connected to the sensor and actuator using ports `wl` and `cc` with appropriate connections. The `WaterLevel` and `ValveStatus` refer to AADL Data Model Annex components in package `WLCS` that specifies the details of the range and measuring units of the data types used in this model.

The implementation of the `WaterTank` component in Listing 1 is specified using three HA sections: **variables**, **constants**, and **behavior**. Below we explain each of these sections in the context of the running example. The formal syntax, grammar and details on each section of HA are presented in [18].

3.1 Variables Section

Local variables in the scope of current HA subclause are declared in **`variables`** section. A variable may either be discrete or continuous depending on the component to which HA specification is attached and must have a data type specified by AADL component classifier reference. In Listing 1, the **`variables`** section contains `t`, `v`, and `h` to specify the current time, status of the inflow valve and the current water level. Variable `v` can take value either 0 or 1 to represent the *close* and *open* status of the inflow valve.

3.2 Constants Section

In the standard way, **`constants`** section is used to define constants that are used in modeling continuous behavior of the physical process. Constants are only initialized once at declaration by either integer or real value, along with appropriate measuring unit specification.

The **`constants`** section in Listing 1 contains six constants. Constant `Qmax` with value `0.007` specifies the maximum water inflow through valve *v*. Measuring unit of the water inflow m^3s^{-1} is specified as `cmps`. Constant `g` is the gravitational force with `9.8` and measuring unit ms^{-2} specified as `mpss`. Constants `pi` and `u` represent the value of π and status of the outflow valve without any measuring unit specification. The constant `period` with value `0.01` and measuring unit seconds specified as `sec` represents the sampling period of the controller. Radius of the outflow valve `r` takes value `0.0254` and is measured in meters specified as `m`.

3.3 Behavior Section

The `behavior` section in HA contains parallel composition of networked sequential processes to specify the continuous behavior and cyber-physical interaction between AADL components and the physical processes. Each behavior specification is represented as a HCSP process explained in section 2.1. Specification of a continuous evolution consists of differential expression along with boolean conditions followed by one or more communication events denoting interrupts. Differential expression contains differential equations specified using the keyword **`DE`** followed by the order of the differential equation and the dependant and independent variables. Keeping in view the extensive use of time derivation in real-time modeling, separate notation is defined for time derivation specification consisting of the keyword **`DT`** followed by the order and the dependant variable.

The **`behavior`** section in Listing 1 shows HA specification for the water tank of the running example. Continuous evolution of water level h is modeled using time derivation `'DT 1 h = (v*Qmax)-(pi*(r^2)*1.414*(g^0.5)*(h^0.5)*u)'` with boundary condition `t < period` as process `Plant`. Here, `(v*Qmax)` is the total water inflow if the value of `v` is 1 and `(pi*(r^2)*1.414*(g^0.5)*(h^0.5)*u)'` is the total water outflow at a particular time `t`. `'DT 1 v = 0'` represents the rate of change of variable `v` with respect to time, which is 0 in this case. The continuous evolution of the water level `h` is preempted by the communication event on out data port `wl`. This communication interrupt is modeled using `[[> wl!(h) ]]>` followed by the process

Algorithm 1: Translation of an AADL model into HCSP processes

```
Require: AADL instance model
Ensure: Generate HCSP processes for periodic threads and connections
 1: for all $tr \in Tr$ do
 2:     generate an activation process $ACT_{tr}$ (Section 4.1)
 3:     generate a dispatch process $DIS_{tr}$ (Section 4.2)
 4:     generate a compute process $COM_{tr}$ (Section 4.3)
 5: end for
 6: for all $c_{tr} \in CON_{tr}$ do
 7:     if $tr$ is source in $c_{tr}$ then
 8:         update $COM_{tr}$ for $c_{tr}$ (Section 4.4)
 9:     else
10:         update $DIS_{tr}$ for $c_{tr}$ (Section 4.4)
11:     end if
12: end for
```

`GetCmd`. Process `GetCmd` contains the communication event `cc?(v)` used to get the control command from the controller on port `cc`. The repeating continuous behavior of the water tank is modeled by the `WaterTank` process where every iteration starts by resetting the time clock `t:=0`.

HA is expressive enough to model physical processes with complex continuous dynamics attached to AADL ports and mapped with AADL connections.[2]

4 Formal Semantics

4.1 Formalization of Synchronous Subset of AADL

Algorithm 1 lists the main steps followed for defining formal semantics of the AADL execution model with synchronous communications. Here, Tr is a set of threads in an AADL model, and for every $tr \in Tr$, processes ACT_{tr}, DIS_{tr}, and COM_{tr} are generated to specify activation, dispatch and computation behavior of the thread. Based on specific properties, associated connections and timing constraints, each active thread corresponding to state machine shown in Fig. 2 is translated into one HCSP process.

Separate activation and dispatch processes are defined for each thread to specify the activation and dispatch behavior of the thread. AADL modal semantics is not considered here, so every thread has only one operational mode. Following parallel composition of *ACT*, *DIS* and *COM* processes represents the HCSP process corresponding to an AADL periodic thread.

$$ThrdName(period, deadline, bcet, wcet) \triangleq ACT^{*}||DIS^{*}||COM^{*}$$

[2] The details of the AADL textual model for the Controller component and all proof files (discussed in Section 5) related to the running example are available at https://github.com/ehah/FACS2014

Here, *period*, *deadline*, *bcet*, and *wcet* are process parameters to represent *Period*, *Deadline*, minimum and maximum *Compute_Execution_Time* properties respectively. Processes *ACT*, *DIS* and *COM* have repeating behavior and can only communicate through common channels.

Process *ACT* is used to specify the behavior of a thread which had already been initialized, activated and is ready for dispatch. Behavior of *ACT* process, as shown below, is quite simple. It only signalizes process *DIS* via output communication event *complete_act*!, which shows the execution completion of source code contained in a file associated with *Activate_Entrypoint_Source_Text* property of a thread.

$$ACT \triangleq \ complete\_act!$$

Below we explain dispatch (*DIS*) and compute (*COM*) processes in detail.

4.2 Dispatch Process

A periodic thread is dispatched after every fixed time interval specified in its *Period* property. The dispatch process for a periodic thread is specified as:

$$\begin{aligned} DIS \triangleq\ & complete\_act?; wait\ period; dispatch!dis; GetData(data); trans!data;\\ & complete\_comp? \end{aligned}$$

At the start, process *DIS* is ready to receive activation completion event from process *ACT*. Then it keeps idle for the period of the thread, after which it is ready to send dispatch event (*dis*) over channel *dispatch*. Thread execution completion event is received across channel *complete_comp* and after which the process is repeated again. Periodic thread inputs value from *in* data ports at dispatch time and outputs values to out data ports at completion time. Process *GetData* shows getting *data* from all *in* data ports at dispatch time. The data is then sent along channel *trans* to compute process (*COM*) which is explained below.

4.3 Compute Process

Compute process itself is a parallel composition of *Ready*, *Running*, and *AwaitResource* processes with clock variables *c* and *t* to be initialized at the start, as depicted in Fig. 4. continuous evolution of variable *t* represents the total amount of time since the dispatch event received from the dispatch process while the continuous evolution of variable *c* specifies execution time in the current dispatch. Therefore, the clock *t* is always progressing in all the sub-states (represented by $\delta t = 1$), while *c* is only progressing in *Running* sub-state (represented by $\delta c = 1$).

Process *COM* below specifies behavior of a compute process. After the dispatch event, it first receives data x_2 along channel *trans*, then communications along tri_k for $k = 1, 2, 3$ are performed, not only to coordinate the execution order between the four parallel sub-processes, but also to transmit x_2 to processes *Ready*, *Running*, and

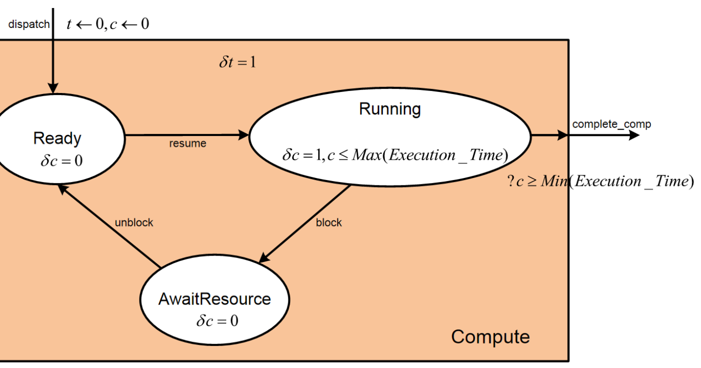

Fig. 4. Thread execution and actions in compute state

AwaitResource. The boolean variable *isReady* indicates whether the ready state is enabled.

$$\begin{aligned}
COM \triangleq\ & (dispatch?\mathrm{x}; trans?x_2; tri_1!x_2; tri_2!x_2; tri_3!x_2) \\
& \|(tri_1?y_1; t := 0; isReady := 1; (isReady \rightarrow Ready)^*) \\
& \|(tri_2?y_2; c := 0; Running^*)\|(tri_3?y_3; AwaitResource^*)
\end{aligned}$$

Thread execution in *Compute* state (see Fig. 4) is controlled by the scheduler modeled using AADL *processor* component. Detailed specification of the scheduling policies and protocols is beyond the scope of this paper as we are not aiming for schedulability analysis. Although, in this paper, we use a simplified static scheduler with predefined unique thread priorities assigned at design level, proposed semantics can be enhanced to model dynamic scheduling by adding a separate process to specify the respective behavior. Whenever the executing thread completes its execution or is blocked due to required resources, the processor is allocated to the highest priority thread in the *Ready* state (modeled as *Ready* process). Execution of the thread can only be interrupted due to required resources blocked by any other thread. This waiting for resource behavior is specified by *AwaitResource* process.

Process *Ready* maintains a continuous variable *t* to model the deadline of the thread. Continuous evolution of time *t* starts once a dispatch event is received from *DIS* process on *dispatch* channel by the parent process *COM*. After process *Ready* receives an event from the scheduler via *run* channel, indicating that current thread is ready to run, it then sends current value of time *t* via *resume* channel to process *Running*. After this communication, it is ready to accept the new value of *t* via *unblock* channel sent by *AwaitResouce* process. If the new value of *t* is equal to the *deadline*, the ready state is disabled, as a result the thread will need re-initialization and re-activation. The *Ready* process is specified as below:

$$\begin{aligned}
Ready \triangleq\ & \langle \dot{t} = 1 \& t < deadline \rangle \trianglerighteq (run? \rightarrow (resume!t; unblock?t)); \\
& t = deadline \rightarrow isReady := 0
\end{aligned}$$

Process *Running* maintains variable *ct* to record current time *t* sent by process *Ready* via *resume* channel, and variable *c* (defined in *COM* process) to record execution time. The boundary condition for continuous evolution is to check if *c* is less than or equal to WCET and *ct* is less than or equal to the deadline. Continuous evolution can be interrupted by an event from the scheduler along *res_busy* channel, indicating that the shared resource is blocked, then the *Running* process will send the current value of *ct* along *block* channel to *AwaitResource*. Continuous evolution can also be interrupted by an event to the scheduler along *complete_exec* channel, indicating the execution completion of the source code specified in a file associated with *Compute_Entrypoint_Source_Text* property of the thread in the dispatch. The *Running* process will then signal the successful execution completion along *complete_comp*! to process *DIS*. To ensure determinism, it is checked that the thread must execute at least till BCET otherwise it has to wait for $(bcet - c)$ time units. Process $SetData(x_r)$ represents the computation of new values x_r based on the received data y_2 (as shown in the parent *COM* process) and then sending new values to the out data ports. As an illustration, it is instantiated and explained in [19] in detail for our running example. The *Running* process is specified as below:

$$\begin{aligned} Running \triangleq\ & resume?ct; \langle \dot{c} = 1, \dot{ct} = 1 \& c \leq wcet \wedge ct \leq deadline \rangle \trianglerighteq \\ & ((res\_busy? \to block!ct)\ [\!]\ (complete\_exec! \to (SetData(x_r); \\ & complete\_comp! \to c < bcet \to wait\ (bcet - c)))) \end{aligned}$$

When an executing thread accesses a shared data component locked by any other thread, it is blocked. Such waiting behavior is specified by *AwaitResource* process. It receives current time via *block* channel from process *Running* and stores it in variable *act*. After it receives an event via *res_free* channel from the scheduler, indicating that the required resource is available, the current value of *act* is sent via *unblock* channel to *Ready* process. Below is the specification of *AwaitResource* process.

$$AwaitResource \triangleq block?act; \langle \dot{act} = 1 \& act \leq deadline \rangle \trianglerighteq (res\_free? \to unblock!act);$$

The lock/un-clock mechanism of shared resources depends on the implementation strategies and does not affect analysis at architecture level so it is not discussed here.

4.4 Connection Process

The connection between two threads or between a thread and a device has an ultimate *source* and an ultimate *destination*. Synchronous communication in AADL is realized through periodic thread with data ports. Based on communication semantics explained in Section 2.3, the behavior of a connection is specified by

$$Conn_{tr} \triangleq StC?x_c; CtD!x_c.$$

StC is a communication channel between the source and the connection process ($Conn_{tr}$ in this case). $StC?x_c$ shows input communication event ready to occur when the respective source thread completes its execution and is ready to send dispatch event, and moreover, receives the data x_c from the source state. The dispatch event together with the data x_c is sent across channel *CtD* to destination thread to start its execution.

For every connection $c_{tr} \in CON_{tr}$ in which thread *tr* is a source thread, the *Running* process is updated based on connection type: *immediate* or *delayed*. In case of immediate connection it sends the complete event on execution completion together with data x_r to connection process $Conn_{tr}$ via *StC* as defined below.

$$\begin{aligned} Running_i \triangleq\, & resume?ct; \langle \dot{c} = 1, \dot{ct} = 1 \& c \leq wcet \wedge ct \leq deadline \rangle \trianglerighteq \\ & ...\ (SetData(x_r); complete\_comp! \rightarrow \\ & c < bcet \rightarrow wait(bcet - c); StC!x_r)\ ... \end{aligned}$$

The behavior of the *Running* process, in case of a source thread with delayed connection, is specified below, in which the completion event *complete_comp*! together with data x_r is sent after the deadline.

$$\begin{aligned} Running_d \triangleq\, & resume?ct; \langle \dot{c} = 1, \dot{ct} = 1 \& c \leq wcet \wedge ct \leq deadline \rangle \trianglerighteq ... \\ & (ct = deadline \rightarrow StC!\, x_r; complete\_comp!)... \end{aligned}$$

For every connection $c_{tr} \in CON_{tr}$ in which *tr* is a destination thread, the respective *DIS* process is updated to wait for complete event with the data from the connection process $Conn_{tr}$ via *CtD* channel. As a result, process *DIS* does not need to specify the period of the thread. Instead, after the complete event with data z is received from channel *CtD*, the dispatch *dis* event is sent across channel *dispatch* which is received by respective *COM* process. The behavior of modified *DIS* process of a destination thread is specified as follows:

$$DIS \triangleq complete\_act?; CtD?z; dispatch!dis; GetData(data); trans!data; complete\_comp?$$

4.5 WLCS Hybrid System Modeling

The structure of the running example (WLCS) has been simplified to focus tightly on the elements needed to present a description of hybrid behavior of the system using plant and the controller while the internal behavior of the sensor and actuator is not discussed. So, connections $C1$ and $C2$ in Fig. 3 are specified as channel *wl* and connections $C3$ and $C4$ are mapped to channel *cc* resulted in a cyber-physical interaction supported by HA. HA uses ports to communicate with other AADL component and channels for internal process communication. Both of these communication mechanisms are mapped as channel communications for verification in terms of HCSP. The hybrid system of the running example (WLCS as whole), as specified below, is modeled as parallel composition of the *WaterTank* and the *Controller*.

$$\begin{aligned} WLCS &\triangleq Watertank \| Controller \\ Watertank &\triangleq (t := 0; Plant)^* \\ Plant &\triangleq \langle \dot{h} = v \cdot Q_{max} - \pi \cdot r^2 \cdot \sqrt{2 \cdot g \cdot h}, \dot{v} = 0 \& t < 0.01 \rangle \\ &\quad \trianglerighteq wl!\, h \rightarrow cc?v; \\ Controller &\triangleq get\_data \| Conn \| com\_cmd \end{aligned}$$

Behavior of the *WaterTank* is specified in Listing 1 and the behavior of *Controller* is obtained by applying the translation approach explained in Algorithm 1 to the AADL model of Fig. 3. The process Controller is composed of three subprocesses: *get_data*, *Conn*, and *com_cmd* executing in parallel. These processes specify behavior of thread *get_data*, immediate connection *Conn* and thread *com_cmd* respectively.

The complete HCSP model of the running example, along with parameters (obtained from respective AADL properties) and details of the subprocesses *get_data*, *Conn*, and *com_cmd*, are presented in [19]. As HA is based on HCSP so each notation of HA automatically corresponds to a respective HCSP notation.

5 Verification using HHL prover

In this section, we show how to use HHL prover to formally verify an AADL model with HA specification through the running example WLCS.

The main goal of WLCS is to maintain water level *h* between a specified limit which is $0.30m$ to $0.60m$, by controlling the inflow valve *v* to be close or open. The control algorithm of the system is designed as follows: every $0.01sec$, the controller samples the value of h, and when *h* is greater than $0.59m$, it assigns value 0 to *v*, while when *h* is less than $0.31m$, *v* assigned value 1. We can investigate the safety of the WLCS system from two aspects:

- it is deadlock-free, under the assumption that the scheduler is well-behaved
- the property $0.30 \leq h \leq 0.60$ always holds for the WLCS system

The deadlock-freedom can be checked by some existing CSP checkers, like the known CSP tool FDR. Here we focus on the verification of the second property, which is mostly related to the hybrid behavior of the system. Thus, we abstract away various communications for synchronizing AADL components, and obtain a simplified controller for the WLCS system with the same control behavior as the *Controller* in the translated HCSP model:

$$\mathit{Controller} = (\text{wait}\,\mathit{period}; \mathit{wl}?x; x \leq 0.31 \rightarrow y := 1; x \geq 0.59 \rightarrow y := 0; \mathit{cc}!y)^*$$

where *period* is $0.01sec$ as mentioned above. The resulting model for the WLCS system covers the continuous plant, the controller containing the corresponding control algorithm, and the interactions between them.

By applying the HHL prover, we have proved the following specification for the WLCS system:

$$\{t = 0 \wedge h = 0.31 \wedge v = 1, y = 0 \vee y = 1\}\ \mathit{WLCS}$$
$$\{0.30 \leq h \leq 0.60, \mathit{True}; \lceil 0.30 \leq h \leq 0.60 \rceil, \mathit{True}\}$$

indicating that from the initial state when *t* is $0sec$, *h* is $0.31m$ and *v* is *open*, throughout the execution of *WLCS*, the safety requirement $0.30 \leq h \leq 0.60$ always holds for the water tank.

6 Related Work

Formalization of AADL has been explored a lot. Here we summerize some of the important work. Yang et al [16] have formalized BA by translating it into Time Abstract State Machine (TASM). Process algebra interpretation of AADL models is presented in [12]. They have translated AADL models to process algebra ACSR and Real-Time Calculus (RTC) for performance evaluation using VERSA and RTC Toolbox respectively. COMPASS toolset used a variant of AADL called SLIM and SuSMv model checker for safety, dependability and performance evaluation [5]. In [7], a tool called AADL2BIP based on BIP (Behavior Interaction Priority) for safety property verification has been introduced.

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, formalization of the continuous time modeling based on a dedicated annex has not been explored before. The proposed formal semantics based on language purely designed for hybrid system is novel and the first step to enhance AADL modeling and formal analysis capabilities for systems with both discrete and continuous dynamics.

There have been a number of modeling languages proposed for formalizing hybrid systems. The most popular is hybrid automata [1, 2], 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. As an alternative approach, Platzer [3] proposed hybrid programs and the related differential dynamic logic for the compositional modeling and deductive verification of hybrid systems. However, in his work, parallelism and communication were not well handled, that occur ubiquitously in AADL models.

Based on HA for continuous behavior modeling and AADL core for discrete modeling, our approach of defining formal semantics for verification is scalable and can be used to verify complex HSs.

7 Conclusion and Future Work

The AADL with Hybrid Annex can model continuous behavior of the physical process to be monitored and controlled by the control system. Formal semantics, based on HCSP, of synchronous subset of AADL annotated with Hybrid Annex are presented to furnish AADL for modeling and verification of hybrid systems. The application of the Hybrid Annex for modeling and formal semantics for verification is illustrated through a benchmark of hybrid system, i.e., the example of water level control system. AADL model is verified using an in-house developed theorem prover called HHL prover. Being the first step towards formalization of continuous behavior and cyber-physical interaction modeling and verification using AADL, this study has opened new horizon 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 Hybrid Annex specifications) to HCSP processes and verification using HHL prover.

Acknowledgements. The work has been partly supported by the National Basic Research Program of China under Grant No. 2014CB340700, by Natural Science Foundation of China under Grant No. NSFC-91118007 and NSFC-6110006, by the CAS/SAFEA International Partnership Program for Creative Research Teams, and by the National Infrastructure Software Plan under Grant No.2012ZX01041-002-003.

References

1. Alur, R., Courcoubetis, C., Henzinger, T. and Ho, P.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Hybrid Systems, LNCS 736, pp. 209-229, 1992.
2. Manna, Z. and Pnueli, A.: Verifying hybrid systems. In Hybrid Systems, LNCS 736, pp. 4-35, 1993.
3. Platzer, A.: Differential dynamic logic for hybrid systems. Journal of Automated Reasoning, 41(2):143-189, 2008.
4. Ayan, B., Sailesh, K., Tridib, M., Gupta, S.K.S.: Band-aide: A tool for cyber-physical oriented analysis and design of body area networks and devices. ACM Transantions on Embedded Computing Systems, 11(2):1-49, 2012.
5. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V., Noll, T., Roveri, M., Wimmer, R.: A model checker for AADL. In Proc. of CAV 2010, LNCS 6174, pp.562–565, 2010.
6. Zhou, C., Wang, J., Ravn, A.: A formal description of hybrid systems. In Proc. of Hybrid Systems III, LNCS 1066, pp.511–530, 1996.
7. Chkouri, M., Robert, A., Bozga, M., Sifakis, J.: Translating AADL into BIP - application to the verification of real-time systems. In Proc. of MODELS 2008, LNCS 5421, pp. 5-19, 2009.
8. Henzinger, T., Sifakis, J.: The embedded systems design challenge. In Proc. of FM 2006, LNCS 4085, pp.1-15, 2006.
9. He, J.:From CSP to hybrid systems. A classical mind, pp.171–189. Prentice Hall, 1994.
10. Lee, E.: What's ahead for embedded software? IEEE Computer, 33(9):18-26, 2000.
11. Liu, J., Lv, J., Quan, Z., Zhan, N., Zhao, H., Zhou, C., Zou, L.: A calculus for hybrid CSP. In Proc. of APLAS 2010, LNCS 6461, pp.1–15, 2010.
12. Sokolsky, O., Lee, I., Clarke, D.: Process-algebraic interpretation of AADL models. In Proc. of Ada-Europe 2009, LNCS 5570, pp.222-236, 2009.
13. SAE Internatinal, Architecture Analysis & Design Language (AADL) Annex Volume 2: Annex D: Behavior Annex, SAE International Standards, 2011.
14. SAE International, Aarchitecture Analysis & Design Language (AADL), revision: B, SAE International Standards, 2012.
15. Larsson, B., Chalin, P., Hatcliff, J.: BLESS: Formal specification and verification of behaviors for embedded systems with software. In Proc. of NFM 2013, LNCS 7871, pp.276–290, 2013.
16. Yang, Z., Kai, H., Ma, D., Lei, P.: Towards a formal semantics for the AADL behavior annex. In Proc. of DATE 2009, pp. 1166–1171, 2009.
17. Zhan, N., Wang, S., Zhao, H.: Formal modelling, analysis and verification of hybrid systems. In the book of Theories of Programming, LNCS 8050, pp. 207–281, 2013.

18. Ahmad, E., Larson, B., Barrett, S., Zhan, N., Dong, Y.: Hybrid Annex: An AADL extention for continuous behavior and cyber-physical interaction modeling, accepted for publication, HILT'14, 2014.
19. Ahmad, E., Dong, Y., Wang, S., Zhan, N., Zou, L.: Adding formal meanings to AADL with hybrid annex, Tech. Report ISCAS-SKLCS-14-09, State key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190. China, 2014.
20. Zou, L., Lv, J., Wang, S., Zhan, N., Tang, T., Yuan, L., Liu, Y.: Verifying Chinese train control system under a combined scenario by theorem proving. In Proc. of VSTTE 2013, LNCS 8164, pp. 262-280, 2013.