# Modeling and Formal Analysis of Virtually Synchronous Cyber-Physical Systems in AADL

No Institute Given

**Abstract.** This paper presents a language and analysis tool to model and formally verify virtually synchronous cyber-physical systems—with complex control programs, continuous behaviors, and bounded clock skews, network delays, and execution times—in the avionics modeling standard AADL. We leverage the Hybrid PALS equivalence, so that it is sufficient to model and verify the simpler underlying synchronous designs. We define a sublanguage of the avionics modeling standard AADL, called HybridSynchAADL, for modeling such designs in AADL. HybridSynchAADL models are given a formal semantics and analyzed using Maude combined with SMT solving, which allows us to represent advanced control programs and communication features in Maude, while capturing timing uncertainties and continuous behaviors symbolically with SMT solving. We demonstrate the effectiveness of our HybridSynchAADL Analyzer tool on networks of thermostats and on sets of autonomous drones that collaborate to achieve common goals.

## 1 Introduction

Many cyber-physical systems (CPSs) are *virtually synchronous*. They should logically behave as if they were synchronous—in each iteration of the system, all components, in lockstep, read inputs and perform transitions which generate outputs for the next iteration—but have to be realized in a distributed setting, with clock skews and message passing communication. Examples of virtually synchronous CPSs include avionics and automotive systems, networked medical devices, and other distributed control systems, such as the steam-boiler benchmark [1]. The underlying infrastructure of such critical systems often guarantees bounds on clock skews, network delays, and execution times.

Virtually synchronous CPSs are notoriously hard to design and to model check, because of the state space explosion caused by asynchronous communication. Motivated by an avionics application developed at Rockwell Collins, the *PALS* ("physically asynchronous, logically synchronous") formal pattern was developed to reduce the difficulty of modeling and verifying distributed real-time systems when the infrastructure provides bounds on network delays, clock skews, and execution times [38, 39, 47]. The key point of PALS is that a synchronous design *SD*—where all components execute in lockstep and there is no asynchronous message passing, clock skews, or execution times—is stuttering bisimilar to, and therefore satisfies the same properties as, the corresponding asynchronous distributed "implementation" $PALS(SD)$.

PALS abstracts from the time when an event takes place, as long as it happens in a certain time interval. However, many virtually synchronous CPSs are networks of *hybrid* systems with continuous behaviors. In such systems, we can no longer abstract from the time when a controller reads a continuous value or sends an actuator command. *Hybrid PALS* [8] extends PALS to such virtually synchronous distributed hybrid systems, with the usual equivalence between synchronous and asynchronous models. Synchronous Hybrid PALS models, however, must take into account sensing and actuating times, which depend on imprecise local clocks. Although [8] showed that simple synchronous Hybrid PALS models, and their reachability problem, can be encoded as SMT problems, it is very difficult to directly encode complex virtually synchronous CPSs—with advanced control programs, data types, hierarchical structure, etc.—as SMT problems. Furthermore, the SMT encodings in [8] are not directly applicable to modeling/programming languages with advanced features, including communication, generic expressions and data types, actions, and so on.

This paper therefore defines the HybridSynchAADL language for modeling synchronous Hybrid PALS models using the avionics modeling standard AADL [24] (Section 3). Providing a formal semantics to such models—with control programs written in AADL's expressive Behavior Annex, with continuous behaviors, and having to cover all possible sensing and actuation times based on imprecise clocks—is challenging. We exploit the fact that the rewriting-logic-based language and tool Maude [21] recently has been integrated with SMT solving [13, 44] to *symbolically* encode continuous behaviors with all possible sensing/actuating times, and provide in Section 4 a Maude-with-SMT semantics for HybridSynchAADL. Section 4 also explains how Maude combined with the SMT solver Yices2 [23] can be used to symbolically analyze bounded reachability properties of HybridSynchAADL models with polynomial continuous dynamics. Section 6 defines an intuitive property specification language for HybridSynchAADL models and shows how the modeling and analysis of such models have been integrated into the OSATE tool environment for AADL.

As usual in symbolic approaches [14], symbolic analysis with Maude-with-SMT quickly encounters the *symbolic state-space explosion* problem. The number of symbolic states generated by symbolic analysis can grow exponentially in the size of models. We address this problem in two ways to make the symbolic analysis feasible (Section 4). First, we propose a *modular encoding* to eliminate the interleavings of different components due to interactions between environments and controllers, by symbolically executing each component *independently*. Second, our semantics is optimized with a novel state-space reduction method which merges symbolic states for Maude-with-SMT to significantly improve the performance of symbolic analysis.

We have applied this HybridSynchAADL *Analyzer* tool to model and verify a network of thermostats and a distributed collection of drones that communicate to reach the "same" location, or fly in formation, without crashing into each other. Section 8 evaluates the effectiveness of the HybridSynchAADL Analyzer tool by addressing the following questions:

1. How effective is our method and tool in finding (injected) bugs, compared to randomized simulations?
2. How effective is our new state merging technique?
3. How effective is the Hybrid PALS methodology in verifying virtually synchronous CPSs with continuous behaviors? We address this question by comparing the analysis of synchronous Hybrid PALS models with the analysis of *much simplified* versions of the corresponding distributed systems.

The HybridSynchAADL Analyzer is one of few, if any, tools—certainly in an AADL context—that can perform model-checking verification of the important class of virtually synchronous CPSs with typical CPS features such as control programs, continuous behaviors, and nondeterministic but bounded communication delays, clock skews, and execution times. This is made possible by:

1. The *Hybrid PALS equivalence* [8], which reduces the verification problem for a virtually synchronous CPS to that of verifying its synchronous design—albeit having to consider clock skews and sensing and actuation times.
2. The integration of Maude with SMT solving. Maude is suitable to model and analyze complex control programs with user-defined data types, distributed objects, asynchronous communication, and so on, whereas SMT solving allows us to analyze symbolic representations and constraints needed to cover all possible clock skews and sampling and actuation times, and to analyze continuous behaviors. In contrast, most formal frameworks are strong at analyzing either discrete or continuous behaviors, but rarely both.
3. The symbolic state-space reduction methods, which are highly optimized and make the symbolic analysis of HybridSynchAADL models feasible.

This paper is organized as follows. Section 2 explains some background on Hybrid PALS, AADL, and Maude. Section 3 presents the HybridSynchAADL language, Section 4 presents its semantics, and Section 5 presents a state-space reduction method. Section 6 presents the HybridSynchAADL Analyzer tool. Section 7 presents case studies on virtually synchronous CPSs for controlling distributed drones. Section 8 shows the experimental results. Section 9 discusses the related work, and Section 10 presents some concluding remarks.

## 2 Preliminaries

### 2.1 PALS

When the infrastructure guarantees bounds on the clock skews, the network delays, and the execution times, the PALS ("physically asynchronous, logically synchronous") pattern [38,39,47] reduces the problems of designing and verifying virtually synchronous distributed real-time systems (where all components have the same period) to the much easier problems of designing and verifying their underlying synchronous designs. Formally, given a synchronous system design $SD$, bounds $\Gamma$ on the clock skews, network delays, and execution times, and a

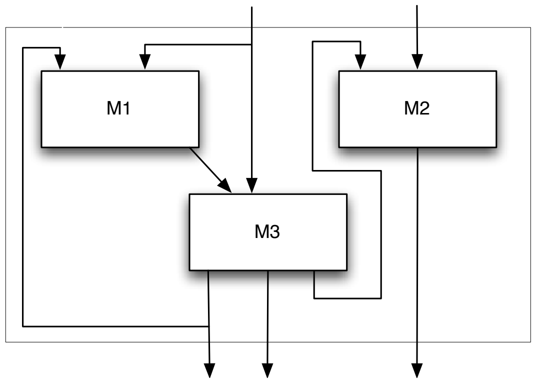


**Fig. 1.** A machine ensemble.

period $p$ of each round,[1], the PALS transformation gives us the corresponding asynchronous distributed real-time system $PALS(SD, \Gamma, p)$ so that $SD$ and the distributed system $PALS(SD, \Gamma, p)$ are stuttering bisimilar, and therefore satisfy the same CTL$^*$ formulas not involving the "next" operator.

PALS is formalized and verified in [38], with the synchronous design $SD$ formalized as the synchronous composition of an *ensemble* of state machines with input and output ports. In each iteration (i.e., at the beginning of each "period"), each machine performs a transition based on its current state and its inputs, proceeds to the next state, and generates outputs. An ensemble has a *synchronous semantics*: the transitions of all machines are performed simultaneously, and output to other machines becomes an input at the *next* iteration. PALS was extended to the multi-rate setting in [5]. For simplicity of exposition, this paper considers the case where all components have the same period.

## 2.2 Hybrid PALS

*Hybrid PALS* [8] extends PALS to virtually synchronous distributed CPSs whose environments exhibit continuous behaviors. As mentioned in the introduction, Hybrid PALS does not abstract from the times when physical values is sampled or when actuator commands are sent/received. Since the times when these events take place depend on the local controller's imprecise local clock, both the sensing and actuating times, and the local clock (skews) must be taken into account in the Hybrid PALS *synchronous* models.

In Hybrid PALS, the *physical environment* $E_M$ of a machine $M$ has real-valued physical parameters $\vec{x} = (x_1, \ldots, x_l)$. The continuous behaviors of $\vec{x}$ are modeled by a set of ordinary differential equations (ODEs) over $\vec{x}$ that specify different *trajectories* on $\vec{x}$. $E_M$ also defines *which* trajectory the environment follows, as a function of the last *control command* received by $E_M$.

[1] Given performance bounds $\Gamma$, PALS can find the shortest period $p$ that allows all nodes to read the messages in the correct "rounds."

The local clock of a machine $M$ can be represented as a function $c_M : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, where $c_M(t)$ denotes the value of the local clock at time $t$, so that $\forall t \in \mathbb{R}_{\geq 0},\ |c_M(t) - t| < \epsilon$, with $\epsilon > 0$ the maximal clock skew [38]. In its $i$th iteration, a controller $M$ samples the values of its environment at time $c_M(i \cdot p) + t_s$, where $t_s$ is the *sampling time*, and then executes a transition (based on the sampled values, the values received from other controllers, and the controller's own state). As a result, the new control command is received by the environment at time $c_M(i \cdot p) + t_a$, where $t_a$ is the *actuating time*. We refer to [8] for the formal definition of the Hybrid PALS models, and for a proof of the bisimulation result relating synchronous and asynchronous models.

The paper [8] also shows how some synchronous Hybrid PALS models with "finite-state machine" controllers can be encoded as SMT formulas, and how the dReal solver for nonlinear theories over the reals [28] can analyze their safety problems up to a given precision $\delta > 0$.

## 2.3 AADL

The *Architecture Analysis & Design Language* (AADL) [24] is an industrial modeling standard used in avionics, aerospace, automotive, medical devices, and robotics to describe an embedded real-time system as an assembly of software components mapped onto an execution platform. In AADL, a component *type* specifies the component's *interface* (e.g., ports) and *properties* (e.g., periods), and a component *implementation* specifies its internal structure as a set of *subcomponents* and a set of *connections* linking their ports. An AADL construct may have *properties* describing its parameters, declared in *property sets*. The OSATE modeling environment provides a set of Eclipse plug-ins for AADL.

An AADL model describes a system of hardware and software components. This paper focuses on the software components, since we use AADL to specify *synchronous designs*. Hardware components include: *processor* components that schedule and execute threads, *memory* components, *device* components, and *bus* components that interconnect processors, memory, and devices. Software components include *threads* that model the application software to be executed; *process* components defining protected memory that can be accessed by its thread and data subcomponents; and *data* components representing data types. *System* components are the top-level components.

A port is either a *data* port, an *event* port, or an *event data* port. Event ports and event data ports support queuing of, respectively, "events" and message data, while *data* ports only keep the latest data. *Modes* represent the operational states of components. A component can have mode-specific property values, subcomponents, etc. Mode transitions are triggered by events.

Thread behavior is modeled as a guarded transition system with local variables using the *Behavior Annex* sublanguage [26]. The actions performed when a transition is applied may update local variables, call methods, and/or generate new outputs. Actions are built from basic actions using sequencing, conditionals, and finite loops. When a thread is activated, an enabled transition is applied; if the resulting state is not a *complete* state, another transition is applied, until a complete state is reached. The *dispatch protocol* of a thread determines when the thread is executed. For example, a *periodic* thread is activated at fixed time intervals, and an *aperiodic* thread is activated when it receives an event.

### 2.4 Maude with SMT Solving

Maude [21] is a language and tool for formally specifying and analyzing concurrent systems in rewriting logic. System states are specified as elements of algebraic data types, and transitions are specified using rewrite rules. Formally, a *rewrite theory* [36] is a triple $\mathcal{R} = (\Sigma, E, R)$, where

1. $(\Sigma, E)$ is an equational theory with $\Sigma$ a signature (declaring sorts, subsorts, and function symbols) and $E$ a set of equations; and
2. $R$ is a set of rewrite rules $l :\ t \longrightarrow t'$ **if** $cond$, where $l$ is a label, $t$ and $t'$ are terms, and $cond$ is a conjunction of equations and rewrites.[2]

A *rewrite* $t \longrightarrow^* t'$ holds if $t'$ is reachable from $t$ using the rewrite rules in $\mathcal{R}$.

A class declaration `class C |` $att_1 : s_1, \dots, att_n : s_n$ declares a class $C$ with attributes $att_1, \dots, att_n$ of sorts $s_1, \dots, s_n$. An *object* $o$ of class $C$ is a term `<` $o : C$ `|` $att_1 : v_1, \dots, att_n : v_n$ `>` of sort `Object`, where $v_i$ is the current value of $att_i$. A `subclass` inherits the attributes and rewrite rules of its superclasses. A *configuration*, a multiset of objects and messages, has sort `Configuration`, where multiset union is denoted by a juxtaposition operator.

In rewriting modulo SMT [13,44], constrained terms are used to *symbolically* represent (possibly infinite) sets of system states. A *constrained term* is a pair $\phi \parallel t$ of a constraint $\phi(x_1, \dots, x_n)$ and a term $t(x_1, \dots, x_n)$ over SMT variables $x_1, \dots, x_n$, representing the set of all instances of the pattern $t$ such that $\phi$ holds. That is, $[\![\phi \parallel t]\!] = \{t(a_1, \dots, a_n) \mid \mathcal{T} \models \phi(a_1, \dots, a_n)\}$, where $\mathcal{T}$ denotes the underlying SMT theory.

A *symbolic rewrite* $\phi_t \parallel t \rightsquigarrow^* \phi_u \parallel u$ on constrained terms symbolically represents a (possibly infinite) set of system transitions sequences. For a symbolic rewrite $\phi_t \parallel t \rightsquigarrow^* \phi_u \parallel u$, there is a "concrete" rewrite $t' \longrightarrow^* u'$ with $t' \in [\![\phi_t \parallel t]\!]$ and $u' \in [\![\phi_u \parallel u]\!]$, and vice versa for each $t' \longrightarrow^* u'$ with $t' \in [\![\phi_t \parallel t]\!]$.

In addition to its explicit-state analysis methods for concrete states (ground terms), Maude also provides SMT solving and *symbolic reachability analysis* for constrained terms, using connections to Yices2 [23] and CVC4 [16].

## 3 The HybridSynchAADL Modeling Language

This section introduces the HybridSynchAADL modeling language for specifying virtually synchronous CPSs in AADL. HybridSynchAADL can specify environments with continuous dynamics, synchronous designs of distributed

[2] For decidability we assume the following conditions [37]: (i) $E$ can be decomposed into a disjoint union $E_0 \cup B$ with $B$ a set of structural axioms (such as associativity, commutativity, and identity); (ii) $E_0$ is sort-decreasing, confluent, terminating, and coherent modulo $B$; and (iii) $R$ is coherent with $E_0$ modulo $B$.

controllers, and complex interactions between controllers and environments with respect to imprecise local clocks, and sampling and actuation times.

The HybridSynchAADL modeling language is a subset of AADL extended with the property set `Hybrid_SynchAADL`. We use a subset of the AADL language without changing the meaning of the AADL constructs or adding a new annex, unlike other AADL extensions [3,34], so that AADL modelers can easily develop and understand HybridSynchAADL models.

```
property set Hybrid_SynchAADL is
  Synchronous: inherit aadlboolean applies to (system);
  isEnvironment: inherit aadlboolean applies to (system);
  ContinuousDynamics: aadlstring applies to (system);
  Max_Clock_Deviation: inherit Time applies to (system);
  Sampling_Time: inherit Time_Range applies to (system);
  Response_Time: inherit Time_Range applies to (system);
end Hybrid_SynchAADL;
```

There are two kinds of components in HybridSynchAADL: *continuous* environments and *discrete* controllers. Environments are specified as system components whose continuous dynamics is specified using continuous functions or ordinary differential equations. Discrete controllers are usual AADL software components in the Synchronous AADL subset [6, 10] of AADL.[3] The top-level system component declares the following properties to state that the model is a synchronous design and to declare the period of the system, respectively.

```
Hybrid_SynchAADL::Synchronous => true;
Period => period;
```

*Example 1.* We use a simple networked thermostat system as a running example. There are two thermostats that control the temperatures of two rooms located in different places. The goal is to maintain similar temperatures in both rooms. For this purpose, the controllers communicate with each other over a network, and turn the heaters on or off, based on the current temperature of the room and the temperature of the other room. Figure 2 shows the architecture of this networked thermostat system. For room $i$, for $i = 1, 2$, the controller `ctrl`$i$ controls its environment `env`$i$ (using "connections" explained below).

## 3.1 Environment Components

*Environment components* specify physical environments of Hybrid PALS models [9]. An environment component involves real-valued variables that continuously change over time according to their dynamics. An environment component can have different *modes* to specify different continuous behaviors. A command from a discrete controller may change the mode of the environment or the value

[3] Just like Synchronous AADL can be extended to *multi-rate* controllers [10], it is possible to extend HybridSynchAADL to the multi-rate case in the same way, but for clarity and ease of exposition we focus on the single-rate case in this paper.

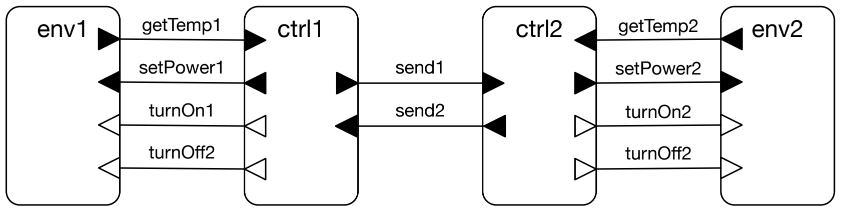


**Fig. 2.** A networked thermostat system.

of a variable. HybridSynchAADL uses existing AADL constructs, extended with the property set `Hybrid_SynchAADL`, to model such environments. Each environment component declares the following component property to state that it specifies a physical environment.

```
Hybrid_SynchAADL::isEnvironment => true;
```

*Continuous Dynamics.* Data subcomponents are used to specify the state of an environment component. An environment component contains data subcomponents of type `Base_Types::Float`.[4] The values of these data subcomponents change continuously over time. The continuous dynamics of these values are specified using either ordinary differential equations (ODEs) or continuous real functions, using the component property:

```
Hybrid_SynchAADL::ContinuousDynamics => "continuous_dynamics";
```

Consider a set of ODEs over $n$ variables $x_1, \ldots, x_n$, say, $\dot{x_i} = e_i(x_1, \ldots, x_n)$, where $e_i$ denotes an expression over the variables $x_1, x_2, \ldots, x_n$, for $1 \leq i \leq n$. In HybridSynchAADL, they are written as a semicolon-separated string of the following form, where `d/dt(`$x$`)` denotes the derivative $\dot{x}$:

$$\texttt{d/dt(}x_1\texttt{) = } e_1(x_1, \ldots, x_n)\texttt{; . . . ; d/dt(}x_n\texttt{) = } e_n(x_1, \ldots, x_n)\texttt{;}$$

If a closed-form solution of ODEs is already known, we can directly specify concrete continuous functions, which are parameterized by a time parameter $t$ and the initial values $x_1(0), \ldots, x_n(0)$ of the variables $x_1, \ldots, x_n$. They are written as a semicolon-separated string of the following form:[5]

$$x_1\texttt{(}t\texttt{) = } e_1(t, x_1(0), \ldots, x_n(0))\texttt{; . . . ; } x_n\texttt{(}t\texttt{) = } e_n(t, x_1(0), \ldots, x_n(0))\texttt{;}$$

Sometimes an environment component may include real-valued parameters or state variables that have the same constant values in each iteration, and can only be changed by a controller command; their dynamics can be specified as `d/dt(x) = 0` or `x(`$t$`) = x(0)`, and can be omitted in HybridSynchAADL.

[4] Since an environment models a physical object, its state involves real numbers. We use `Base_Types::Float` to denote real numbers, instead of defining a new data type.
[5] We may use any name for the time parameters ('`t`' above).

*Mode Transitions.* In Hybrid PALS, an environment component can have different continuous behaviors. In HybridSynchAADL, this can be specified by assigning different values to `Hybrid_SynchAADL::ContinuousDynamics` in different modes, using the following AADL syntax:

```
Hybrid_SynchAADL::ContinuousDynamics =>
      "continuous_dynamics_1" in modes (mode_1),
      . . .
      "continuous_dynamics_n" in modes (mode_n);
```

Modes and mode transitions are declared in the usual AADL way: a command from a controller through an input port can trigger a mode transition, and change the continuous dynamics of the environment accordingly.

*Sampling and Updating Data.* An environment component interacts with discrete controllers by sending its state values, and by receiving actuator commands that may update the values of state variables. This behavior is specified in HybridSynchAADL using *connections between ports and data subcomponents.*

– A connection from a data subcomponent inside the environment to an output data port declares that the value of the data subcomponent is “sampled” by a discrete controller through the output port of the environment component.
– A connection from an environment’s data input port or event data input port to a data subcomponent inside the environment declares that a controller command arrived at the input port updates the value of the data subcomponent of the environment component.

When a discrete controller sends actuator commands, some input ports of the environment component may receive no value (more precisely, some “don’t care” value $\perp$). In this case, the behavior of the environment is unchanged.

*Example.* Figure 4 shows an environment component `RoomEnv` for our networked thermostat system (`Hybrid_SynchAADL::isEnvironment => true`). Figure 3 shows its architecture. It has data output port `temp`, data input port `power`, and event input ports `on_control` and `off_control`. These ports will be connected to a controller component.

The implementation of `RoomEnv` has two data subcomponents `x` and `p` to denote the temperature of the room and the heater’s power, respectively. They represent the state variables of `RoomEnv` with the specified initial values. There are two modes `heaterOn` and `heaterOff` with their respective continuous dynamics, specified by `Hybrid_SynchAADL::ContinuousDynamics`, where `heaterOff` is the initial mode. Because `p` is a constant, `p`’s dynamics `d/dt(p) = 0` is omitted.

The values of the state variable `x` change continuously according to the mode and the continuous dynamics. The value of `x` is sent to the controller through the output port `temp`, declared by the connection `port x -> temp`. When a discrete controller sends a actuation command through input ports `power`, `on_control`, and `off_control`, the mode changes according to the mode transitions, and the value of `p` can be updated by the value of input port `power`, declared by the connection `port x -> temp`.

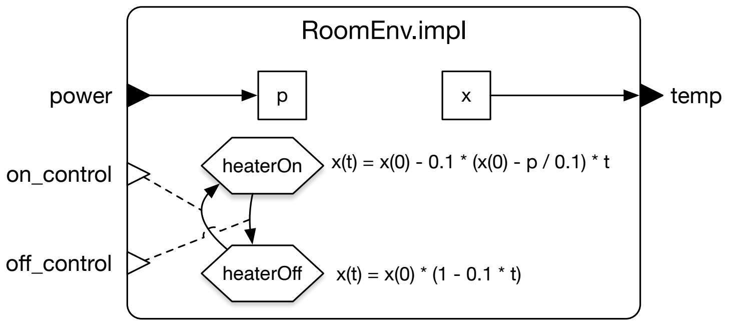

RoomEnv.impl
power
p
x
temp
on_control
off_control
heaterOn
heaterOff
x(t) = x(0) - 0.1 * (x(0) - p / 0.1) * t
x(t) = x(0) * (1 - 0.1 * t)

**Fig. 3.** An environment of the thermostat controller.

```
system RoomEnv
  features
    temp : out data port Base_Types::Float;
    power : in data port Base_Types::Float;
    on_control: in event port;
    off_control : in event port;
  properties
    Hybrid_SynchAADL::isEnvironment => true;
end RoomEnv;

system implementation RoomEnv.impl
  subcomponents
    x: data Base_Types::Float {Data_Model::Initial_Value => ("15");};
    p: data Base_Types::Float {Data_Model::Initial_Value => ("5");};
  connections
    C: port x -> temp;
    R: port power -> p;
  modes
    heaterOff: initial mode;            heaterOn: mode;
    heaterOff -[on_control]-> heaterOn;
    heaterOn -[off_control]-> heaterOff;
  properties
    Hybrid_SynchAADL::ContinuousDynamics =>
      "x(t) = x(0) - 0.1 * (x(0) - p / 0.1) * t;" in modes (heaterOn),
      "x(t) = x(0) * (1 - 0.1 * t);" in modes (heaterOff);
end RoomEnv.impl;
```

**Fig. 4.** A RoomEnv component.

## 3.2 Discrete Components

Like in Synchronous AADL, a (discrete) controller is specified using the behavioral and structural subset of AADL, namely, hierarchical system, process, and thread components; data subcomponents; ports and connections; and thread behaviors defined using the AADL *behavior annex* [25]. The hardware, memory, and scheduling features of AADL is not considered in HybridSynchAADL, because they are not relevant to synchronous *designs*.

*Dispatch.* The execution of an AADL thread is specified by the *dispatch protocol*. A thread with an *event-trigerred* dispatch (such as aperiodic, sporadic, timed, or hybrid dispatch protocols) is dispatched when it receives an event. Since all "controller" components are executed in lock-step in HybridSynchAADL, each thread must have *periodic* dispatch by which the thread is dispatched at the beginning of each period. The periods of all the threads are identical to the period declared in the top-level component. In AADL, this behavior is declared by the thread component property:

```
Dispatch_Protocol => Periodic;
```

*Timing Properties.* In HybridSynchAADL, a controller component interacts with its environment according to its sampling and actuating times. A controller receives (samples) the state of the environment at some sampling time, and sends a controller command to (actuates) the environment at some actuation time. Sampling and actuating times are defined with respect to the imprecise local clock of the controller that may differ from the "ideal clock" by up to the maximal clock skew. These time values are declared by the component properties:[6]

```
Hybrid_SynchAADL::Max_Clock_Deviation => time;
Hybrid_SynchAADL::Sampling_Time => lower bound .. upper bound;
Hybrid_SynchAADL::Response_Time => lower bound .. upper bound;
```

The upper sampling time bound must be strictly smaller than the upper bound of actuation time, and the lower bound of actuation time must be strictly greater than the lower bound of sampling time. Also, the upper bounds of both sampling and actuating times must be strictly smaller than the maximal execution time to meet the (Hybrid) PALS constraints [9].

*Initial Values and Parameters.* In AADL, *data* subcomponents represent data values, including Booleans, integers, and floating-point numbers. The initial values of data subcomponents and output ports are specified using the property:

```
Data_Model::Initial_Value => ("value");
```

[6] For sampling and actuating times, we consider time intervals, instead of exact values, because measuring times may involve some numerical errors.

Sometimes initial values can be *parameters*, instead of concrete values. In HybridSynchAADL, such unknown parameters can be declared using the AADL property below. E.g., you can check whether a certain property holds from initial values satisfying a certain constraint for those parameters (see Section 6).

```
Data_Model::Initial_Value => ("param");
```

*Example.* Consider again our networked thermostat system. Figure 5 shows a thread component `ThermostatThread` that turns the heater on or off depending on the average value `avg` of the current temperatures of the two rooms. The component has event output ports `on_control` and `off_control`, data input ports `curr` and `tin`, and data output ports `set_power` and `tout`. The ports `on_control`, `off_control`, `set_power`, and `curr` are connected to an environment, and `tin` and `tout` are connected to another controller component (see Fig. 6). The implementation of `ThermostatThread` has data subcomponent `avg` whose initial value is declared as a parameter.

When the thread dispatches, the transition from state `init` to `exec` is taken, which updates `avg` using the values of the input ports `curr` and `tin`, and assigns to the output port `tout` the value of `curr`. Since `exec` is not a complete state, the thread continues executing by taking one of the other transitions, which may send an event. For example, if the value of `avg` is smaller than 10, a control command that sets the heater's power to 5 is sent through the port `set_power`, and an event is sent through the port `off_control`. The resulting state `init` is a complete state, and the execution of the current dispatch ends.

## 3.3 Communication

There are three kinds of ports in AADL: *data* ports, *event* ports, and *event data* ports. Event and event data ports can trigger the execution of threads, whereas data ports cannot. In HybridSynchAADL, controller and environment components may have all three kinds, but the connections are constrained for synchronous behaviors: no connection is allowed between environments, or between environment components and the enclosing system components.

*Connections Between Discrete Controllers.* All (non-actuator) outputs of discrete components generated in one iteration are available to the receiving discrete components at the beginning of the *next* iteration. As explained in [6, 10], *delayed connections between data ports* meet this requirement. Therefore, two discrete components can be connected only through data ports with delayed connections, declared by the connection property:

```
Timing => Delayed
```

```
thread ThermostatThread
  features
    on_control: out event port;
    off_control: out event port;
    set_power: out data port Base_Types::Float;
    curr: in data port Base_Types::Float;
    tin: in data port Base_Types::Float;
    tout: out data port Base_Types::Float
            {Data_Model::Initial_Value => ("0");};
  properties
    Dispatch_Protocol => Periodic;
    Hybrid_SynchAADL::Max_Clock_Deviation => 0.3ms;
    Hybrid_SynchAADL::Sampling_Time => 1ms .. 5ms;
    Hybrid_SynchAADL::Response_Time => 7ms .. 9ms;
end ThermostatThread;

thread implementation ThermostatThread.impl
  subcomponents
    avg : data Base_Types::Float
            {Data_Model::Initial_Value => ("param");};
  annex behavior_specification{**
    states
      init : initial complete state;  exec : state;
    transitions
      init -[on dispatch]-> exec { avg := (tin + curr) / 2; tout := curr };
      exec -[avg > 25]-> init { off_control! };
      exec -[avg < 20 and avg >= 10]-> init { set_power := 5; on_control! };
      exec -[avg < 10]-> init { set_power := 10; on_control! };
  **};
end ThermostatThread.impl;
```

**Fig. 5.** A simple thermostat controller.

*Connections Between Controllers and Environments.* In HybridSynchAADL, interactions between a controller and an environment component occur *instantaneously* at the times specified by the sampling and actuating times of the controller.[7] Because an environment does not "actively" send data for sampling, every output port of an environment must be a data output port.

On the other hand, any types of input ports, such as data, event, event data ports, are available for environment components. Specifically, a discrete controller can trigger a mode transition of an environment through event ports. Therefore, no extra requirement is needed for connections, besides the usual constraints for port to port connections in AADL.

[7] More precisely, processing times and delays between environments and controllers are modeled using sampling and actuating times.

```
system implementation TwoThermostats.impl
  subcomponents
    ctrl1: system Thermostat.impl;        ctrl2: system Thermostat.impl;
    env1: system RoomEnv.impl;            env2: system RoomEnv.impl;
  connections
    turnOn1:   port ctrl1.on_control  -> env1.on_control;
    turnOff1:  port ctrl1.off_control -> env1.off_control;
    setPower1: port ctrl1.set_power   -> env1.power;
    getTemp1:  port env1.temp         -> ctrl1.curr;
    send1:     port ctrl1.tout         > ctrl2.tin;
    turnOn2:   port ctrl2.on_control  -> env2.on_control;
    turnOff2:  port ctrl2.off_control -> env2.off_control;
    setPower2: port ctrl2.set_power   -> env2.power;
    getTemp2:  port env2.temp         -> ctrl2.curr;
    send2:     port ctrl2.tout        -> ctrl1.tin;
  properties
    Hybrid_SynchAADL::Synchronous => true;
    Period => 10 ms;
    Timing => Delayed applies to send1, send2;
end TwoThermostats.impl;
```

**Fig. 6.** A top level component with two thermostat controllers.

*Example.* Figure 6 shows an implementation of a top-level system component `TwoThermostats` of our networked thermostat system. This component has no ports and contains two thermostats and their environments. The controller system component `Thermostat.impl` is implemented using the thread component `ThermostatThread.impl` in Fig. 5, and the environment component `RoomEnv.impl` is given in Fig. 4. Each discrete controller `ctrl`$i$, for $i = 1, 2$, is connected to its environment component `env`$i$ using four connections. The controllers `ctrl1` and `ctrl2` are connected with each other using delayed data connections.

## 4 Formal Semantics of HybridSynchAADL

This section presents the semantics of HybridSynchAADL in *rewriting modulo SMT* [13,44]. Because the HybridSynchAADL language extends Synchronous AADL, the semantics of discrete controllers extends that of Synchronous AADL [7,11], while the semantics of continuous environments, and nontrivial interactions between controllers and environments, is new. Specifically, an SMT encoding for synchronous Hybrid PALS models in [8] is not directly applicable, since HybridSynchAADL models are hierarchical. The full semantics of HybridSynchAADL is available in `https://tinyurl.com/r8pqwtr`.

### 4.1 Overview

The semantics of HybridSynchAADL is defined in an object-oriented style. Each component in a HybridSynchAADL model is represented as an object instance of the corresponding class, and rewrite rules involving such objects specify the behavior of the component. Since AADL can have a hierarchical structure, our semantics is based on hierarchical objects, where some attribute value or an object may contain other objects.

We use *constrained object* terms of the form $\phi(x_1, \dots, x_n) \parallel \mathit{obj}(x_1, \dots, x_n)$ to symbolically represent an infinite number of objects, where an SMT constraint $\phi(x_1, \dots, x_n)$ and an object "pattern" $\mathit{obj}(x_1, \dots, x_n)$ over SMT variables $x_1, \dots, x_n$ represent (possibly infinite) sets of objects. Such object patterns may be hierarchical; for example, $\mathit{obj}(x_1, \dots, x_n)$ may include other object patterns over $x_1, \dots, x_n$ in its attributes. Similarly, *constrained configurations* have the form $\phi \parallel \mathit{conf}$ with $\mathit{conf}$ a multiset of object patterns.

Our semantics defines various *semantic operations* to formally specify the behavior of components, behavior transitions, environments, communications, etc. These semantic operations are defined on constrained terms. A semantic operation $f$ specifies a *symbolic* rewrite relation

$$f(\phi \parallel t) \rightsquigarrow \phi' \parallel t'.$$

Notice that $f$ can be *nondeterministic*; e.g., there could be many constrained terms $\phi'_1 \parallel t'_1, \dots, \phi'_n \parallel t'_n$ with $f(\phi \parallel t) \rightsquigarrow \phi'_i \parallel t'_i$ for $1 \leq i \leq n$. In this paper, we often call $f$ a *semantic function* if $f$ is a deterministic semantic operation.

In our semantics, these semantic operations are declared using the function `check-sat` that invokes the underlying SMT solver to check the satisfiability of an SMT constraint. For example, many semantic operations in our semantics are declared using conditional rewrite rules of the form

```
crl f(PHI || u) => (PHI and φ) || v
 if condition /\ check-sat(PHI and φ) .
```

where `PHI` is a variable that denotes an SMT constraint, $\varphi$ denotes a newly generated constraint, and `check-sat` checks the satisfiability of the accumulated constraints using the SMT solver.

"Continuous" semantic operations for environment components can easily be specified using conditional rewrite rules of the above form, provided that the continuous dynamics can be specified as SMT constraints (e.g., polynomials).

### 4.2 SMT Expressions

In our semantics, *SMT values* are terms of sort `Value`. There are three kinds of SMT values. Boolean values are either $[\mathit{true}]$ or $[\mathit{false}]$, and real values are $[r]$ for rational constants $r \in \mathbb{Q}$ (a value $v$ is written as the term `[`$v$`]` to avoid parsing issues in Maude). A unit value $*$ denotes the presence of an event.

```
sorts Value BoolValue RealValue UnitValue .
subsorts BoolValue RealValue UnitValue < Value  .
op [_] : Bool -> BoolValue [ctor] .
op [_] : Rat -> RealValue [ctor] .
op * : -> UnitValue [ctor] .
```

It is worth noting that SMT values are different from *syntactic values* that are defined by the syntax of AADL. Each syntactic value corresponds to a SMT value, but different syntactic values may correspond to the same SMT value, when they have the same meaning (e.g., `1` and `1.0`). Also, some SMT value may have no syntactic counterpart (e.g., the unit value $*$).

*SMT variables* are declared as terms of sort `SMTVar`, and there are two kinds of SMT variables: Boolean variables of the form $b(id)$, and real variables of the form $r(id)$, where $id$ is a natural number.

```
sorts SMTVar SMTBoolVar SMTRealVar .
subsorts SMTBoolVar SMTRealVar < SMTVar .
op b : Nat -> SMTBoolVar [ctor] .
op r : Nat -> SMTRealVar [ctor] .
```

A fresh variable can be obtained by maintaining a pair of counters `<` $i$`,` $j$ `>`, where $i$ and $j$ are next indices for real and Boolean variables, respectively. The function `gen` generates a fresh variable of a given type:

```
eq gen(< I, J >, Boolean) = {b(J), < I, J + 1 >} .
eq gen(< I, J >, Real)    = {r(I), < I + 1, J >} .
eq gen(< I, J >, Unit)    = {   *, < I, J >} .
```

*Symbolic expressions* are terms of sort `Exp`, a supersort of both `Value` and `SMTVar`, and are constructed by symbolic values, symbolic variables, and the usual logical, arithmetic, comparison, and conditional operators (which are available in SMT). For example, the symbolic expression

```
(([1] + r(1) > [2] * r(2)) or b(1)) === (b(2) and [true])
```

of sort `BoolExp` represents the pattern $(1 + x_1 > 2 * x_2 \ or \ b_1) = (b_2 \ and \ true)$.[8]

```
sort Exp .
subsorts Value SMTVar < Exp .

sort BoolExp .
subsorts BoolValue SMTBoolVar < BoolExp < Exp .
op not_ : BoolExp -> BoolExp [prec 53] .
op _and_ : BoolExp BoolExp -> BoolExp [assoc comm prec 55] .
op _or_ : BoolExp BoolExp -> BoolExp [assoc comm prec 59] .
. . .
```

[8] [1] and [2] are symbolic values of sort `RealValue`, $[true]$ is a symbolic value of sort `BoolValue`, $r(1)$ and $r(2)$ are symbolic variables of sort `RealVar`, and $b(1)$ and $b(2)$ are symbolic variables of sort `BoolVar`.

```
sort RealExp.
subsorts RealValue SMTRealVar < RealExp < Exp .
op -_ : RealExp -> RealExp .
op _+_ : RealExp RealExp -> RealExp [assoc comm prec 33] .
op _*_ : RealExp RealExp -> RealExp [assoc comm prec 31] .
. . .
sort UnitExp .
subsorts UnitValue < UnitExp < Exp .
op * : -> UnitValue [ctor] .
op _===_ : UnitExp UnitExp -> BoolExp [gather (e E) prec 51] .
. . .
```

## 4.3 Representing HybridSynchAADL Models

*Components.* A component instance in HybridSynchAADL is represented as an instance of a subclass of the following base class `Component`.

```
class Component | features : Configuration,
                  subcomponents : Configuration,
                  connections : Set{Connection},
                  properties : PropertyAssociation .
```

The attribute `features` denotes a multiset of `Port` objects, representing the ports of a component; `subcomponents` denotes a multiset of `Component` objects, representing the hierarchical structure of components; `connections` denotes its connections; and `properties` denotes its properties.

System, process, and thread group components in AADL are represented as object instances of a subclass of the following class `Ensemble`:

```
class Ensemble .
subclass Ensemble < Component .

class System .   class Process .   class ThreadGroup .
subclass System Process ThreadGroup < Ensemble .
```

*Thread Components.* Thread components are represented as object instances of the `Thread` class that contains extra attributes for thread behaviors:

```
class Thread | variables : Map{VarId,DataType},
               transitions : Set{Transition}
               currState : Location,
               completeStates : Set{Location},
               varGen : VarGen .
subclass Env < Component .
```

The attribute `variables` denotes the local (temporary) variables and their types; `currState` denotes the current location of the thread; `completeStates` denotes the complete states; `transitions` denotes its transitions; and `varGen` denotes counters `<` $i$`,` $j$ `>` to generate a fresh symbolic variable.

A transition system of a thread is represented as a (semi-colon-separated) set of transitions of the form:

$$s \texttt{ -[}guard\texttt{]-> } s' \ \texttt{\{}actions\texttt{\}}$$

where $s$ is a source state, $s'$ is a destination state, *guard* is a Boolean condition, and `{`*actions*`}` is a behavior action block.

*Environment Components.* Environment components are represented as object instances of the `Env` class. The attribute `currMode` denotes the current mode, `jumps` denotes the mode transitions, `flows` denotes the continuous dynamics, `sampling` and `response` denote respectively the sets of sampling and actuating times, and `varGen` denotes a fresh variable generator.

```
class Env | currMode : Location,
            jumps : Set{EnvJump},
            flows : Set{EnvFlow},
            sampling : Set{InterTiming},
            response : Set{InterTiming},
            varGen : VarGen .
subclass Env < Component .
```

A mode transition system is represented as a (semi-colon-separated) set of mode transitions $m$ `-[`*triggers*`]->` $m'$, where *triggers* is a set of port names.

A continuous dynamics in `flows` is represented as a (semi-colon-separated) set of continuous dynamics assignments of the form

$$m \ \texttt{[}continuous\_dynacmis\texttt{]}$$

where $m$ is a mode and *continuous_dynacmis* is either ODEs or continuous functions as explained in Section 3.1.

A set `Set{InterTiming}` for `sampling` and `actuation` is represented as a (comma-separated) set of interval assignments of the form $o$ `: (`$l$`,` $u$`)`, where $o$ is an object identifier of the corresponding controller, $l$ is a lower time bound, and $u$ is an upper time bound.

*Data Components.* Data components specify state variables of threads and environments, where the attribute `value` denotes the current value:

```
class Data | value : DataContent .
subclass Data < Component .
```

In HybridSynchAADL, a data component has either no value (more precisely, some "don't care" value $\bot$) or a (Boolean or real) value. We *symbolically represent* the data content of as a pair $e$ `#` $b$ of an SMT expression $e$ and a Boolean condition $b$. The data component has no value (i.e., $\bot$) if $b$ is *false*, and a value represented by the expression $e$ if $b$ is *true*.

```
sort DataContent .
op _#_ : Exp BoolExp -> DataContent [ctor] .
```

*Ports.* A port is represented as an object instance of a subclass of the class `Port`, where the attribute `content` denotes its data content, and `properties` denotes its properties, such as `DataModel::InitialValue`. The subclasses `InPort` and `OutPort` denote input and output ports, respectively.

```
class Port | content : DataContent,
             properties : PropertyAssociation .
class InPort .
class OutPort .
subclass InPort OutPort < Port .
```

We distinguish between the ports of discrete components and the ports of environment components, because their behaviors are significantly different. As mentioned in Section 3.3, the communication between discrete components is *delayed*, i.e., outputs generated in one iteration are available at the destination discrete components in the next iteration. On the other hand, the communication between discrete and environment components is *immediate*.

Ports of discrete components are represented as instances of the `DataPort` class. An input data port contains the extra attribute `cache` to keep the previously received value; if an input port $p$ has received $\bot$ (i.e., $e$ `#` *false*) in the latest dispatch, the thread can use the value in the `cache`, while the behavior annex expression $p$`'fresh` becomes *false* [6, 10].

```
class DataPort .
subclass DataPort < Port .

class DataInPort | cache : DataContent .
class DataOutPort .
subclass DataInPort  < InPort  DataPort .
subclass DataOutPort < OutPort DataPort .
```

Ports of environment components are represented as instances of the `EnvPort` class, containing two extra attributes `target` and `envCache`. Because the port communication depends on the sampling and actuating times of the connected controller, an environment port keeps the identifier of the `target` component.[9] To symbolically encode the immediate communication, the attribute `envCache` contains a data content in the previous iteration (see Section 4.7).

```
class EnvPort | target : ComponentRef,
                envCache : DataContent .
subclass EnvPort < Port .
```

[9] This implies that no *fan-out* connection from an environment output port $p$ exists, i.e., $p$ can only be connected to one controller input port.

```
class EnvInPort .     class EnvOutPort .
subclass EnvInPort  < EnvPort InPort .
subclass EnvOutPort < EnvPort OutPort .
```

*Connections.* A connection set is represented as a semi-colon-separated set of connections of the form $p_i$ `-->` $p_o$, where $p_i$ denotes the source port name and $p_o$ denotes the target port name. The name of a port $p$ in a subcomponent $c$ is written as a term $c$ `..` $p$. That is, a connection from an output port $p_1$ in $c_1$ to an input port $p_2$ in $c_2$ is written as the term $c_1$ `..` $p_1$ `-->` $c_2$ `..` $p_2$. A level-up (resp., level-down) connection, connecting a port $p$ in a subcomponent $c$ to the port $p'$ in the "current" component is written as the term $c$ `..` $p$ `-->` $p'$ (resp., $p'$ `-->` $c$ `..` $p$).

```
sort Connection .
op _-->_ : FeatureRef FeatureRef -> Connection [ctor] .

sort FeatureRef .
subsort FeatureId < FeatureRef .
op _.._ : ComponentRef FeatureId -> FeatureRef [ctor] .

sort ComponentRef .
subsort ComponentId < ComponentRef .
op _._ : ComponentRef ComponentRef -> ComponentRef [ctor assoc] .
```

We use different representations for *internal* connections of an environment component between ports and data subcomponents. This makes it easier to distinguish different types of connections when defining semantic operations. A connection from a data subcomponent $d$ to an output port $p$ for sampling data is written as the term $d$ `==>` $p$, and a connection from an input port $p$ to a data subcomponent $d$ for updating data is written as the term $p$ `=>>` $d$.

*Example.* An instance of the `TwoThermostats.impl` component in Fig. 6 is represented by the following object, where property values are enclosed by `{{...}}`.

```
< TwoThermostatsimplInstance : System |
  features : none,
  subcomponents : < ctrl1 : System | ... >
                  < ctrl2 : System | ... >
                  < env1 : Env | ... >
                  < env2 : Env | ... >,
  connections : ctrl1 .. oncontrol --> env1 .. oncontrol ;
                ctrl1 .. offcontrol --> env1 .. offcontrol ;
                ctrl1 .. setpower --> env1 .. power ;
                env1 .. temp --> ctrl1 .. curr ;
                ctrl1 .. tou --> ctrl2 .. tin ;
                ctrl2 .. oncontrol --> env2 .. oncontrol ;
                ctrl2 .. offcontrol --> env2 .. offcontrol ;
                ctrl2 .. setpower --> env2 .. power ;
                env2 .. temp --> ctrl2 .. curr ;
                ctrl2 .. tou --> ctrl1 .. tin,
  properties : TimingProperties::Period => {{10}} ;
               HybridSynchAADL::Synchronous => {{true}} >
```

An instance of `ThermostatThread.impl` in Fig. 5 can be represented by the following object, where the initial value of `avg` is a symbolic variable $r(0)$. For parsing purposes, syntactic values are enclosed by `[[...]]`, component identifiers are enclosed by `c{...}`, and port identifiers are enclosed by `f{...}` in transitions.

```
< ctrlThread : Thread |
    features :
        < oncontrol : DataOutPort | content : * # [false], property : none >
        < offcontrol : DataOutPort | content : * # [false], property : none >
        < setpower : DataOutPort | content : [0] # [false], property : none >
        < curr : DataInPort | content : [0] # [false], cache : [0] # [false],
                              property : none >
        < tin  : DataInPort | content : [0] # [false], cache : [0] # [false],
                              property : none >
        < tou : DataOutPort | content : [0] # [true], property : none >,
    subcomponents :
      < avg : Data | value : r(0) # [true], features : none,
                     subcomponents : none, connections : empty,
                     property : none >,
    connections : empty,
    properties :
        TimingProperties::Period => {{10}} ;
        HybridSynchAADL::SamplingTime => {{1.0 .. 5.0}} ;
        HybridSynchAADL::ResponseTime => {{7.0 .. 9.0}} ;
        HybridSynchAADL::Synchronous => {{true}},
    variables : empty,
    transitions :
        init -[on dispatch]-> exec {
            (c{avg} := ((f[tin] + f[curr]) / [[2]])) ;
            f{tou} := f[curr] } ;
        exec -[c[avg] > [[25]]]-> init {
            offcontrol !} ;
        exec -[(c[avg] < [[20]]) and (c[avg] > [[10]])]-> init {
            (f{setpower} := [[5]]) ;
            oncontrol !} ;
        exec -[c[avg] <= [[10]]]-> init {
            (f{setpower} := [[10]]) ;
            oncontrol !} ;
        exec -[otherwise]-> init {skip},
    currState : init,
    completeStates : init,
    varGen : < 1, 0 >
>
```

An instance of `RoomEnv.impl` in Fig. 4 can be represented by the following object. The contents of environment ports include symbolic variables, such as $r(1)$, $b(0)$, and $b(3)$, to symbolically encode the immediate communication (see Section 4.7). For parsing purposes, syntactic values are enclosed by `[[...]]` and component identifiers are enclosed by `c{...}` in `flows`.

```
< env1 : Env |
    features : < temp : EnvOutPort | content : r(1) # b(3),
                                     envCache : r(1) # b(3),
                                     target : ctrl1, property : none >
               < offcontrol : EnvInPort | content : * # [false],
                                          envCache : * # b(0),
                                          target : ctrl1, property : none >
               < oncontrol : EnvInPort | content : * # [false],
                                         envCache : * # b(1),
                                         target : ctrl1, property : none >
               < power : EnvInPort | content : [0] # [false],
                                     envCache : r(0) # b(2),
                                     target : ctrl1, property : none >,
    subcomponents : < p : Data | value : [5] # [true], ... >
                    < x : Data | value : [15] # [true], ... >,
    connections : x ==> temp ; power =>> p,
    properties : Hybrid_SynchAADL::isEnvironment => {{true}} ;
                 TimingProperties::Period => {{10}} ;
                 HybridSynchAADL::Synchronous => {{true}},
    currMode : toff,
    jumps : ton -[offcontrol]-> toff ; toff -[oncontrol]-> ton,
    flows : ton [x(t)= c[x] - ([[0.1]] * (c[x] - c[p] / [[0.1]]) * v[t])] ;
            toff [x(t)= c[x] * ([[1.0]] - ([[0.1]] * v[t]))],
    varGen : < 2, 4 >,
    sampling : ctrl1 : (1,5),
    response : ctrl1 : (7,9) >
```

## 4.4 Symbolic Synchronous Steps

The semantics of a single AADL component is specified using the *partial operation* `executeStep` that executes one synchronous iteration of the component, by means of equations and rewrite rules. Unlike in the formal semantics of Synchronous AADL in which `executeStep` is defined for a (concrete) object, `executeStep` is applied to a constrained object that symbolically represents a (possibly infinite) set of object instances.

```
op executeStep : ConstObject ~> ConstObject .

sort ConstObject .
subsort Object < ConstObject .
op _||_ : BoolExp Object -> ConstObject [ctor] .
```

In our semantics, all semantic operations, including `executeStep`, are partial. Since a term containing partial operations does not have a sort, this is used to ensure that equations and rules for semantic operations are only applied to an object of sort `Object` in which all subcomponents have already finished their semantic operations.

A (symbolic) synchronous step of the entire system, given by a top-level *closed* system component with no ports, is formalized by the following rule, where `SYSTEM` is a variable of sort `Object`:

```
rl [step]: {PHI || < C : System | features : none >}
        => {PHI and PHI' || SYSTEM}
if executeStep(PHI || < C : System | >) => PHI' || SYSTEM .
```

In the condition of the rule, any term of sort `Object` that includes no partial operations, where `executeStep` has been completely evaluated obtained by rewriting `executeStep(PHI || < C : System | >)` in zero or more steps can be nondeterministically assigned to the variable `SYSTEM` of sort `Object`.

## 4.5 Ensemble Behavior

The following rewrite rule defines the behavior of ensemble components (such as systems and processes), provided that the behavior of all the subcomponents is defined using `executeStep`. (We explain how the semantics of threads and environment components is defined by `executeStep` below.) This rule specifies the synchronous composition of the subcomponents of an ensemble.

```
crl executeStep(PHI || < C : Ensemble | >)  =>  PHI' || transferResults(OBJ')
 if OBJ := transferInputs(< C : Ensemble | >)
 /\ propagateExec(PHI, OBJ) => PHI' || OBJ'
 /\ check-sat(PHI and PHI') .
```

First, each input port of the subcomponents receives a value from its source by `transferInputs`. Next, `exectueStep` is applied to each subcomponent, along with the constraint `PHI`, by `propagateExec`. Then, any term of sort `Object` obtained by rewriting `propagateExec(PHI, OBJ)`, where `executeStep` has been completely evaluated in each subcomponent, is nondeterministically assigned to `OBJ'` of sort `Object`, together with the new constraint `PHI'`. Finally, the new outputs of the subcomponents are transferred by `transferResults`.

*Propagating Executions.* Given an ensemble `C` and a Boolean constraint `PHI`, the function `propagateExec` simply applies the operation `executeStep` to each subcomponent *Obj* constrained by `PHI`. Each term `executeStep(PHI ||` *Obj*`)` can then be individually executed using rewrite rules and equations.

```
 eq propagateExec(PHI, < C : Ensemble | subcomponents : COMPS >)
  = < C : Ensemble | subcomponents : propExecAux(PHI, COMPS, none) > .

 eq propExecAux(PHI, < C : Component | > COMPS, COMPS')
```

```
  = propExecAux(PHI, COMPS,
                executeStep(PHI || < C : Component | >) COMPS') .

 eq propExecAux(PHI, none, COMPS') = COMPS'  .
```

*Transferring Data.* Consider an ensemble component `C`. The semantic function `transferInputs` moves data in the input ports of `C` or the feedback output ports of its subcomponents into their connected input ports. The semantic function `transferResults` transfers data in the output ports of the subcomponents to their connected output ports of `C`; if such an output port is also connected to another subcomponent, it keeps the data for the feedback output in the next step. These functions are declared in the same way as those for Synchronous AADL [10], except that data contents are pairs of symbolic values.

### 4.6 Thread Behavior

The following rule defines the behavior of threads. The function `readFeature` returns a map from each input port to its current value, and `readData` returns a map from each data subcomponent to its value. The operation `execTrans` *non-deterministically* assigns any possible computation result of the behavior transition system to the pattern `L' | FMAP' | DATA' | PHI' | GEN'`. The function `writeFeature` updates the content of each output port, and `writeData` updates the value of each data subcomponent. The function `check-sat` invokes an SMT solver to check whether the generated constraint is satisfiable. The constants `#loopbound#` and `#transbound#` denote a loop unrolling bound and a transition bound, respectively, for symbolic analysis.[10]

```
crl executeStep(
     PHI  || < C : Thread | features : PORTS,   subcomponents : COMPS
                            properties : PROPS, currState : L,
                            transitions : TRS,  completeStates : LS,
                            variables : VIS,    varGen : GEN >)
 =>
     PHI' || < C : Thread | features : writeFeature(FMAP',PORTS'),
                            subcomponents : writeData(DATA',COMPS),
                            currState : L', varGen : GEN' >
 if {PORTS',FMAP} := readFeature(PORTS)
 /\ DATA := readData(COMPS)
 /\ execTrans(feature(FMAP) data(DATA) prop(PROPS) const([true])
              location(L) complete(LS) trans(TRS)  local(defaultVal(VIS))
              lbound(#loopbound#)  tbound(#transbound#)   vargen(GEN))
    => L' | FMAP' | DATA' | PHI' | GEN'
 /\ check-sat(PHI and PHI') .
```

[10] `loopBound` limits the number of loop unrolling when symbolically executing behavior actions, and `transBound` limits the number of visiting the same behavior locations in one synchronous step when symbolically executing behavior transitions.

*Behavior Configurations.* We represent a group of "named" function arguments $id_1 : arg_1, id_2 : arg_2, \ldots, id_n : arg_n$ as a a multiset of *behavior configuration items* of the form $id_1(arg_1)\ id_2(arg_2) \ldots id_n(arg_n)$. For example, execTrans takes a number of behavior configuration items, including port values feature, data values data, constraints const, and so on. The auxiliary function addConst adds a given constraint PHI' to a behavior configuration.

```
sort BehaviorConf .
subsort BehaviorConfItem < BehaviorConf .
op none : -> BehaviorConf [ctor] .
op __ : BehaviorConf BehaviorConf -> BehaviorConf [ctor comm assoc id: none] .

sort BehaviorConfItem .
op const : BoolExp -> BehaviorConfItem [ctor] .
op feature : FeatureMap -> BehaviorConfItem [ctor] .
op data : DataValuation -> BehaviorConfItem [ctor] .
op prop : PropertyAssociation -> BehaviorConfItem [ctor] .
. . .

eq addConst(const(PHI) REST, PHI') = const(PHI and PHI') REST .
```

*Feature Operations.* Given a set of ports (for discrete components), the semantic function readFeature builds a map from port identifiers to their current values, removes the value from each input port, and returns a pair of the result ports and the map. This function is defined in a tail-recursive style by using an auxiliary function with extra arguments to carry intermediate results:

```
eq readFeature(PORTS) = readFeature(PORTS, none, empty) .
eq readFeature(none, PORTS, FMAP) = {PORTS, FMAP} .
```

A feature map built by readFeature has different feature map contents for input and output ports, because a behavior annex expression *p*'fresh needs to know whether the value of input port *p* is "fresh". A feature map content of an input port is given by a pair D : F of data content D and freshness flag F (where D is also a pair E # B in which B indicates the presence of the value).

```
sort FeatureMapContent .
subsort DataContent < FeatureMapContent .
op _:_ : DataContent BoolExp -> FeatureMapContent [ctor] .
```

Consider a port P with a content E # B and a cached content E' # B'. If the content is present (i.e., B is *true*), P corresponds to the pair (E # B) : true in the resulting map FMAP; otherwise, P corresponds to the pair (E' # B') : false using the cached value. The resulting content can be compactly represented as ((B ? E : E') # (B or B')) using the conditional operator. Then, the cache attribute is updated, and the content is set absent.

```
eq readFeature(< P : DataInPort | content : E # B, cache : E' # B' > PORTS,
               PORTS', FMAP)
```

```
 = readFeature(PORTS,
      < P : DataInPort | content : E # [false],
                         cache : (B ? E : E') # (B or B') > PORTS',
      insert(P, ((B ? E : E') # (B or B')) : B, FMAP)) .
```

Finally, each output port is related to `E # [false]` in the resulting map `FMAP`, indicating $\perp$ with the second item `[false]`, because behavior transitions cannot read a value from output ports.

```
eq readFeature(< P : DataOutPort | content : E # B > PORTS, PORTS', FMAP)
 = readFeature(PORTS, < P : DataOutPort | > PORTS',
              insert(P, E # [false], FMAP)) .
```

The semantic function `writeFeature` replaces the content of each output port `P` by the corresponding content `D'` in the map `FMAP`.

```
eq writeFeature(FMAP, PORTS) = writeFeature(FMAP, PORTS, none) .
eq writeFeature(((P |-> D'), FMAP),
               < P : DataOutPort | content : D > PORTS, PORTS')
 = writeFeature(FMAP, PORTS, < P : DataOutPort | content : D' > PORTS') .
eq writeFeature(FMAP, PORTS, PORTS') = PORTS PORTS' [owise] .
```

*Data Operations.* Given data components, the semantic function `readData` builds a map from each identifier to its value, and `writeData` updates the values of the data subcomponents using a given map, defined as follows:

```
eq readData(COMPS) = readData(COMPS, empty) .
eq readData(< C : Data | value : D > COMPS, DATA)
 = readData(COMPS, insert(C, D, DATA)) .
eq readData(none, DATA) = DATA .

eq writeData(DATA, COMPS) = writeData(DATA, COMPS, none) .
eq writeData((C |=> D', DATA), < C : Data | value : D > COMPS, COMPS')
 = writeData(DATA, COMPS, COMPS' < C : Data | value : D' >) .
eq writeData(DATA, COMPS, COMPS') = COMPS COMPS' [owise] .
```

*Executing Transitions.* The behavior of the semantic operation `execTrans` is defined with respect to behavior configurations as follows.

```
crl [trans]:
    execTrans(location(SL) trans(TRS) tbound(s(N)) local(VAL) REST)
 =>
    execTStep(VAL, location(L') trans(TRS) tbound(N)
                   addConst(execAction(ACT, local(VAL) REST), B and B'))
   if (L -[GUARD]-> L' ACT) ; TRS' := TRS
   /\ B  := locConst(SL, L) /\ check-sat(B)
   /\ B' := guardConst(GUARD, outTrs(L, TRS'), local(VAL) REST) .
```

A transition `L -[GUARD]-> L' ACT` is nondeterministically chosen from the set `TRS`. The constraint `B` states that the source state `L` is the same as the current state `SL`, and `B'` indicates that the guard condition `GUARD` evaluates to *true*. The operation `execAction` symbolically executes the actions `ACT` of the chosen transition and returns a new behavior configuration.

If the next state `L'` is a complete state, the operation ends with a result, provided that the constraint `PHI` is satisfiable.[11] Otherwise, `execTrans` is applied again with the new configuration. The number of such iterations is limited by the bound `tbound` (in the rule `trans`) to avoid infinite symbolic computation.

```
eq execTStep(VAL, location(L') complete(LS) local(VAL') REST)
 = if L' in LS then transResult(L', REST)
   else execTrans(location(L') complete(LS) local(VAL) REST) fi .

ceq transResult(L, feature(FMAP) data(DATA) const(PHI) vargen(GEN) REST)
  = L | FMAP | DATA | PHI | GEN  if check-sat(PHI) .
```

The auxiliary functions are defined as follows. The function `locConst` returns the constraint for two behavior states being equal, assuming that states are encoded as terms `loc(`$r$`)` with a rational constant $r$. The functions `guardConst` and `allGuardsFalse` return Boolean constraints obtained by guard conditions. The function `outTrs` returns the set of transitions from a given state.

```
eq locConst(loc(R), loc(R')) = R === R' .

eq guardConst(on dispatch, TRS, REST) = [true] .
ceq guardConst(GE, TRS, REST) = E and B if E # B := eval(GE, REST) .
ceq guardConst(otherwise, TRS, REST)
  = allGuardsFalse(TRS, REST) if noOwise(TRS) .

eq allGuardsFalse((L -[GUARD]-> L' ACT) ; TRS, REST)
 = not(guardConst(GUARD, empty, REST)) and allGuardsFalse(TRS, REST) .
eq allGuardsFalse(empty, REST) = [true] .

eq noOwise((L -[otherwise]-> L' ACT) ; TRS) = false .
eq noOwise(TRS) = true [owise] .

eq outTrs(L, (L -[GE]-> L' ACT) ; TRS) = (L -[GE]-> L' ACT) ; outTrs(L,TRS) .
eq outTrs(L, TRS) = empty [owise] .
```

*Evaluating Expressions.* The semantic function `eval` evaluates (syntactic) AADL behavior expressions to (semantic) data content, given a behavior configuration that contains symbolic expressions and constraints. By construction, when `eval` evaluates *exp* to a data content $e$ `#` $b$, the second item $b$ indicates that all the identifiers in *exp* are well defined in the given behavior configuration.

[11] If `PHI` is unsatisfiable (i.e., when `check-sat(PHI)` returns false), the corresponding execution is not realizable (e.g., due to some runtime errors like division by 0). In this case, the execution path ends with a deadlock term with no sort.

The following equations define the case of syntactic values (which are enclosed by `[[...]]` for parsing purposes), where `BCF` denotes behavior configurations. We only consider Boolean values, integers, and floating point numbers in HybridSynchAADL.

```
eq eval([[B:Bool]],  BCF) = [B:Bool]       # [true] .
eq eval([[I:Int]],   BCF) = [I:Int]        # [true] .
eq eval([[F:Float]], BCF) = [rat(F:Float)] # [true] .
```

The following equations define the cases for identifiers, namely, local variable identifier `VI`, port identifier `PI`, data component identifier `C`, property identifier `PR`, and a fresh expression for port `PI`.

```
 eq eval(v[VI],     local(VAL) REST)    = VAL[VI] .
 eq eval(f[PI],     feature(FMAP) REST) = getData(FMAP[PI]) .
 eq eval(c[C],      data(DATA) REST)    = DATA[C] .
 eq eval(p[PR],     prop(PROPS) REST)   = eval(value(PROPS[PR]), REST) .
ceq eval(fresh(PI), feature(FMAP) REST) = B # B1  if E # B1 : B := FMAP[PI] .
```

The cases for the other expressions are defined by propagating `eval` to the subexpressions. For example, the case of addition is defined as follows. The second equation defines the addition of two data contents.

```
eq eval(AE1 + AE2, REST) = eval(AE1, REST) + eval(AE2, REST) .

eq (E1 # B1) + (E2 # B2) = (E1 + E2) # (B1 and B2) .
```

*Executing Actions.* The semantic operation `execAction` computes a behavior action based a given behavior configuration, and returns the resulting behavior configuration. These behavior configurations contain symbolic expressions and constraints, and represent (possible infinite) sets of concrete configurations.

For example, the semantics of an assignment action *id* `:=` *exp*, assigning the evaluated value of *exp* to the identifier *id*, is defined as follows.

```
ceq execAction(v{VI} := AE, REST)
  = local(insert(VI, E # [true], VAL)) addConst(REST, B)
 if E # B := eval(AE,REST) .

ceq execAction(f{PI} := AE, REST)
  = feature(insert(PI, E # [true], FMAP)) addConst(REST, B)
 if E # B := eval(AE,REST) .

ceq execAction(c{C} := AE, REST)
  = data(insert(C, E # [true], DATA)) addConst(REST, B)
 if E # B := eval(AE,REST) .
```

For a conditional statement, the branch condition can evaluate to either *true* or *false*, according to a given (concrete) behavior configuration. Therefore, `execAction` produces both cases with different constraints. For example:

```
crl execAction(if (AE) AS end if, REST)
 => execAction(AS, addConst(REST, E and B))
 if E # B := eval(AE, REST) .

crl execAction(if (AE) AS end if, REST)
 => addConst(REST, E and B)
 if E # B := eval(not(AE), REST) .
```

For a loop statement, `execAction` produces both *true* and *false* cases for the branch condition, where the number of loop iterations is limited by the bound `lbound` to avoid infinite symbolic computation. For example:

```
crl execAction(while (AE) {AS}, lbound(s(N)) REST)
 => execAction({AS ; while (AE) {AS}}, lbound(N) addConst(REST, E and B))
 if E # B := eval(AE, REST) .

crl execAction(while (AE) {AS}, REST) => addConst(REST, E and B)
 if E # B := eval(AE, REST) .
```

Finally, for a sequence of actions $\{Action_1\ ;\ \cdots\ ;\ Action_n\}$, each action in the sequence is executed based on the execution results of the previous actions:

```
eq execAction({A ; ASQ}, REST) = execAction({ASQ}, execAction(A, REST)) .
eq execAction({A}, REST) = execAction(A, REST) .
```

## 4.7 Environment Behavior

In HybridSynchAADL, an environment component interacts with each of its controllers *in a single iteration*. For example, consider an environment $E$ that is connected to two controllers $C_1$ and $C_2$. Figure 7 shows a timeline of their interactions. Initially, the state variables of $E$ have values $\vec{v}_0$ and change over time according to $E$'s continuous dynamics. For $i = 1, 2$, environment $E$ sends the state values $\vec{v}_n$ at time $t_n^s$ to controller $C_n$, and receives $C_n$'s command $\alpha_n$ at time $t_n^a$ (and may also change its continuous dynamics by $\alpha_n$), according to the sampling and actuating times of $C_n$.

The semantics of environment components cannot be directly specified as synchronous composition. Indeed, the environment behavior is *asynchronous*, since the order of "interaction events" in a single iteration (e.g., *sampling*$(C_1)$, *sampling*$(C_2)$, *response*$(C_1)$, and *response*$(C_2)$ in Fig. 7) can lead to different behaviors. The synchronous semantics requires that the interactions between components must be *delayed*, but the interactions between environment and controller components are *immediate*. Hence, any *concrete semantics* of HybridSynchAADL is not likely definable as synchronous composition.

Previously, there are two approaches to deal with asynchronous interactions. A typical way is to explicitly enumerate all possible interleavings of components [40], but it can lead to state-space explosion. In Hybrid PALS [8], a controller and an environment are combined into a single *environment-restricted*

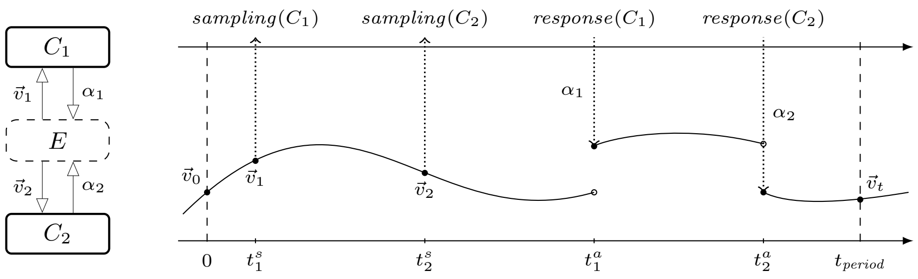


**Fig. 7.** Interactions between an environment $E$ and two controllers $C_1$ and $C_2$

machine, where a controller is a "flat" state machine. However, this technique is not applicable to HybridSynchAADL, because a controller may include arbitrarily complex (hierarchical) subcomponents. Defining environment restrictions for generic AADL components is thus very difficult.

In this paper we present an alternative approach to symbolically encode asynchronous interactions in a *modular* way. We encode the values of input and output ports at different sampling and actuating times into symbolic variables, and perform `executeStep` of each component *independently*. The correspondence between the input and output ports are then symbolically declared using equality constraints. This relies on the fact that an environment interacts *only once* with each of its controllers in a single iteration.

The semantics of environment components can be *symbolically represented as logical constraints* to specify the environment behavior in one-step iteration. Using this approach, the environment semantics can be defined as an operation that builds such symbolic constraints:

$$(x_1, \ldots, x_n) \quad \mapsto \quad \phi(x_1, x_2, \ldots, x_n)$$

where $x_1, \ldots, x_n$ are symbolic variables to completely represent all the necessary information for the environment and its interactions. Observe that `executeStep` for threads can also be interpreted in this way for symbolic inputs.

Therefore, the behavior of environment components is also specified in the operation `executeStep` in our symbolic semantics. The operation `executeStep` builds *constrained objects* with logical constraints to encode the environment behavior. All information required for interaction with discrete controllers—including the values of input and output ports at different sampling and actuating times—is encoded as a set of symbolic variables. The immediate communication between environment and controller components is also encoded as symbolic constraints. As a result, the semantics of ensemble components with environment components is specified in the same way as in Section 4.5.

*Executing Symbolic Steps.* The following rule defines the behavior of environment components. The function `readEnvFeature` returns a map from each input port to its symbolic content, and `writeEnvFeature` updates the content of each output

| # Iteration | | 1 | 2 | 3 | 4 | ... |
|---|---|---|---|---|---|---|
| Controller | | $d_1$ | $d_2$ | $d_3$ | $d_4$ | ... |
| Input port $p$ | content | · | $d_1$ | $d_2$ | $d_3$ | ... |
| | envCache | · | $x_1$ | $x_2$ | $x_3$ | ... |
| readEnvFeature | FMAP | $x_1$ | $x_2$ | $x_3$ | $x_4$ | ... |
| | constraint | *true* | $x_1 = d_1$ | $x_2 = d_2$ | $x_3 = d_3$ | ... |

**Fig. 8.** The behavior of `readEnvFeature`.

port. These functions also return extra constraints to encode the environment communication. The operation `execEnv` builds logical constraints to encode the behavior of the environment in one-step iteration, and each of them is assigned to the pattern `L' | FMAP' | DATA' | PHI' | GEN2`. The function `check-sat` then invokes an SMT solver to check whether the generated constraint is satisfiable.

```
crl executeStep(
     PHI  || < C : Env | features : PORTS,    subcomponents : COMPS,
                         connections : CONXS, properties : PROPS,
                         currMode : L,        jumps : JUMPS,
                         flows : FLOWS,       sampling : STS,
                         response : RTS,      varGen : GEN >)
 =>
     (IPHI and PHI' and OPHI) ||
     < C : Env | features : PORTS',
                 subcomponents : writeData(DATA',COMPS),
                 currState : L',
                 varGen : GEN' >
 if {PORTS1,FMAP,IPHI,GEN1} := readEnvFeature(PORTS, GEN)
 /\ DATA := readData(COMPS)
 /\ execEnv(feature(FMAP) data(DATA)   prop(PROPS)  vargen(GEN1) mode(L)
            time([0])     jumps(JUMPS) flows(FLOWS) sampling(STS)
            response(RTS) envcon(CONXS,PORTS)       const([true]))
    => L' | FMAP' | DATA' | PHI' | GEN2
 /\ {PORTS',OPHI,GEN'} := writeEnvFeature(FMAP', PORTS1, GEN2)
 /\ check-sat(PHI and IPHI and PHI' and OPHI) .
```

*Environment Feature Operations.* Given a set of environment ports, the function `readEnvFeature` builds a map from each port identifier to a symbolic variable denoting the value *sent from the controller in the same iteration.* As described in Fig. 8, we use an extra attribute `envCache` that contains the symbolic variable for the previous round. Suppose that a controller sends a data content $d_i$ to an input port $p$ in the $i$-th iteration; the `content` of $p$ is then $d_{i-1}$ at the beginning of the $i$-th iteration. The function `readEnvFeature` relates the port identifier $p$ to a fresh variable $x_i$, and generates the constraint $x_{i-1} = d_{i-1}$.

This function is defined by the following equations using an auxiliary function with extra arguments to carry intermediate results. In the third equation, each

| # Iteration | | 0 | 1 | 2 | 3 | ... |
|---|---|---|---|---|---|---|
| Environment | `FMAP` | | $d_1$ | $d_2$ | $d_3$ | ... |
| | `envCache` | | $x_0$ | $x_1$ | $x_2$ | ... |
| `writeEnvFeature` | content of `PI` | $x_0$ | $x_1$ | $x_2$ | $x_3$ | ... |
| | constraint | | $x_0 = d_1$ | $x_1 = d_2$ | $x_2 = d_3$ | ... |
| Controller | | | $x_0$ | $x_1$ | $x_2$ | ... |

**Fig. 9.** The behavior of `writeEnvFeature`.

input port `PI` is related to a symbolic content `V # BV` with fresh variables `V` and `BV`, the current `content E # B` and `envCach E' # B'` are declared to be identical as a constraint, the `envCach` attribute is updated, and the `content` attribute is set absent. In the last equation, each output port is related to `E # [false]`, indicating $\perp$ with the second item `[false]`.

```
eq readEnvFeature(PORTS, GEN)
 = readEnvFeature(PORTS, none, empty, [true], GEN) .

eq readEnvFeature(none, PORTS, FMAP, PHI, GEN)
 = {PORTS, FMAP, PHI, GEN} [owise] .

ceq readEnvFeature(< PI : EnvInPort | content  : E  # B,
                                      envCache : E' # B' > PORTS, PORTS',
                   FMAP, PHI, GEN)
  = readEnvFeature(PORTS, PORTS' < PI : EnvInPort | content  : E # [false],
                                                    envCache : V # BV >,
                   insert(PI, V # BV : [true], FMAP),
                   PHI and E === E' and B === B', GEN2)
 if {V, GEN1} := gen(GEN, type(E)) /\ {BV,GEN2} := gen(GEN1,Boolean) .

 eq readEnvFeature(< PI : EnvOutPort | content : E # B > PORTS, PORTS',
                   FMAP, PHI, GEN)
 = readEnvFeature(PORTS, PORTS' < PI : EnvOutPort | >,
                   insert(PI, E # [false], FMAP), PHI, GEN) .
```

The function `writeFeature` replaces the content of each output port `P` by a symbolic variable and declares that *the data content sent in the previous round is identical to the current content* in the map `FMAP`. Thus, the corresponding input port of the controller receives the current content in the same iteration. Similarly, we use `envCache`, containing the symbolic variable sent in the previous round, to implement this behavior, as described in Fig. 9, where $x_0$ denotes the initial content. The function `writeFeature` is defined as follows.

```
eq writeEnvFeature(FMAP, PORTS, GEN)
 = writeEnvFeature(PORTS, none, FMAP, [true], GEN) .

ceq writeEnvFeature(< PI : EnvOutPort | content  : D,
                                        envCache : E # B > PORTS, PORTS',
                   FMAP, PHI, GEN)
  = writeEnvFeature(PORTS, PORTS' < PI : EnvOutPort | content  : V # BV,
                                                      envCache : V # BV >,
                   FMAP, PHI and E === E' and B === B', GEN2)
  if E' # B' := FMAP[PI]
  /\ {V, GEN1} := gen(GEN, type(E)) /\ {BV,GEN2} := gen(GEN1,Boolean) .

eq writeEnvFeature(PORTS, PORTS', FMAP, PHI, GEN)
 = {PORTS PORTS', PHI, GEN} [owise] .
```

Observe that the constraints for environment inputs in one iteration are built by `readEnvFeature` in the next iteration. Therefore, `executeStep` on ensemble components is slightly modified to check such constraints as follows:

```
crl executeStep(PHI || < C : Ensemble | >)  =>  PHI' || transferResults(OBJ')
 if OBJ := transferInputs(< C : Ensemble | >)
 /\ propagateExec(PHI, OBJ) => PHI' || OBJ'
 /\ check-sat(PHI and PHI' and finalConst(OBJ')) .

eq finalConst(< C : Env | features : PORTS, varGen : GEN > COMPS)
 = getConst(readEnvFeature(PORTS,GEN)) and finalConst(COMPS) .
eq finalConst(< C : Ensemble | > COMPS)
 = finalAux(transferInputs(< C : Ensemble | >)) and finalConst(COMPS) .
eq finalAux(< C : Component | subcomponents : COMPS >) = finalConst(COMPS) .
eq finalConst(COMPS) = [true] [owise].
```

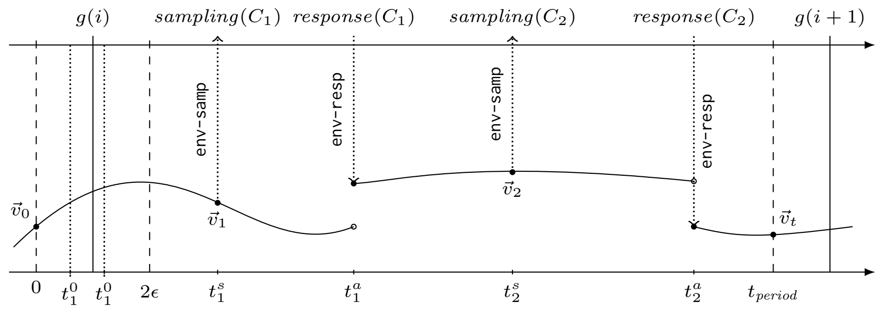


**Fig. 10.** The behavior of an environment interacting with two controllers.

*Executing Environments.* Figure 10 depicts the behavior of an environment that interacts with two controllers $C_1$ and $C_2$. Let $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ denote the *global time* $g(i)$ at the beginning of the $i$-th period, where $g(i+1) - g(i) = t_{period}$. The environment time frame is "shifted" to the left from the global time frame $[g(i), g(i+1)]$ by a maximal clock skew $\epsilon > 0$. Because each controller $C_n$,

$n = 1, 2$, runs according to its local clock, the period of $C_n$ begins at any time $0 < t_n^0 < 2\epsilon$, and its sampling and actuation happen according to the sampling and actuating times of $C_n$ with respect to $t_n^0$. That is, in Fig. 10, $t_n^s - t_n^0$ and $t_n^a - t_n^0$ denote the sampling and actuating times declared by $C_n$.

The semantics of environment components are specified using three rewrite rules; `env-cont` for continuous state changes, `env-samp` for sampling operations, and `env-resp` for actuation operations. These rules are defined using two semantic operations; `execEnv` for the continuous behavior, and `envStep` for the discrete behavior. Basically, our semantics alternatively applies the operations `execEnv` and `envStep` to build the symbolic constraints for the environment behavior of one iteration, given a symbolic behavior configuration.

The following rule `env-cont` specifies the behavior of an environment. It performs a "continuous transition" from a state at time `T` to a state at time `T'`, where `T'` is given as a fresh symbolic variable, according to its continuous dynamics. The constraint `B` states that the mode `L` for the continuous dynamics `FLOW` is the same as the current mode `SL`. The function `execFlow` builds symbolic values and constraints to encode the new environment states given by evolving the environment by time `T' - T`. Finally the function `updateEnvData` updates the data subcomponents to have the symbolic values for time `T'`.

```
crl [env-cont]:
    execEnv(time(T)  mode(SL) vargen(GEN)  REST)
 => envStep(time(T') mode(L)  vargen(GEN') addConst(REST', T <= T' and B))
 if flows((L FLOW) ; FLOWS) ECF'' := ECF
 /\ B := locConst(SL,L) /\ check-sat(B)
 /\ {T',GEN'} := gen(GEN,Real)
 /\ DATA  := execFlow(FLOW, T' - T, REST)
 /\ REST' := updateEnvData(DATA, REST) .
```

The function `execFlow` computes the values of continuous dynamics for a given input `T`. As mentioned, continuous dynamics in HybridSynchAADL are specified using either ODEs or a set of continuous real functions. Currently, we only consider the case of continuous real functions;[12] a function of the form `C(VI) = AE` over an input argument `VI` is evaluated using the function `eval`, while `T` is assigned to the local variable identifier `VI` as follows:

```
eq execFlow([FUNCS], T, REST) = execFuncFlow(FUNCS, T, empty, REST) .

ceq execFuncFlow((C(VI) = AE) ; FUNCS, T, DATA, REST)
  = execFuncFlow(FUNCS, T, insert(C, D, DATA), REST)
 if D := eval(AE, local(VI |-> T # [true]) REST) .

eq execFuncFlow(empty, T, DATA, REST) = DATA .
```

[12] If ODEs have closed-form solutions, we can define them as continuous functions. It is possible to directly generate SMT constraints with ODEs, and solve them using a specialized SMT solver, such as `dReal` [28]. Because `dReal` has not been integrated with Maude, the current version of the semantics only supports continuous functions.

The function `updateEnvData` simply updates the behavior configuration item `data` with the result `DATA` of applying `execFlow`.

```
eq updateEnvData((C |=> E # B, DATA), data(DATA') REST)
 = updateEnvData(DATA, data(insert(C, E # [true], DATA')) addConst(REST,B)) .
eq updateEnvData(empty, REST) = REST .
```

After each continuous transition, the operation `envStep` is applied to perform discrete operations, such as sampling and responding. If no more such discrete operation remains, the current iteration of the environment ends with a result (using the same function `transResult` for threads), with an assertion to state that the end time `T'` is the same as the period `TimingProperties::Period`.

```
ceq envStep(time(T) mode(L) sampling(empty) response(empty) prop(PROPS) REST)
  = transResult(L, prop(PROPS) addConst(REST, B and (T === PER)))
 if PER # B := eval(p[TimingProperties::Period], prop(PROPS)) .
```

*Environment Sampling.* The following rule `env-samp` specifies the behavior of sampling operations. A controller `C` and its sampling time bound $(lt, ut)$ are first nondeterministically chosen in the left-hand side of the rule. The function `timeConst` gives the constraint for the sampling time `T` with respect to the clock skew and the sampling time bound. The function `updateEnvFeature` updates the output ports connected to `C` with the corresponding state values.

```
crl [env-samp]:
    envStep(time(T) sampling((C :(LT,UT), SIT)) REST)
 => execEnv(time(T) sampling(SIT) addConst(REST',B))
 if B := timeConst(T, LT, UT, REST)
 /\ REST' := updateEnvFeature(C, REST) .
```

As explained, because the period of controller `C` happens at any time between 0 and $2\epsilon$, the sampling happens between $lt$ and $ut + 2\epsilon$ as follows.

```
ceq timeConst(T, LT, UT, prop(PROPS) REST)
  = ([LT] <= T) and (T <= [UT] + [2] * SK) and B
 if SK # B := eval(p[HybridSynchAADL::MaxClockDeviation], prop(PROPS)) .
```

The function `updateEnvFeature` updates the content of the output port `PI` using the content of state varaible `CI`, if there is an internal connection `CI ==> PI` from `CI` to `PI`, provided that `PI` is connected to the controller `C`.

```
eq updateEnvFeature(C, envcon(CONXS,PORTS) REST)
 = updateEnvFeature(CONXS, C, envcon(CONXS,PORTS) REST) .

ceq updateEnvFeature(CI ==> PI ; CONXS, C, data(DATA) feature(FMAP) REST)
  = updateEnvFeature(CONXS, C, data(DATA) feature(FMAP') REST)
  if validTarget(PI, C, REST) /\ FMAP' := insert(PI, DATA[CI], FMAP) .

eq updateEnvFeature(CONXS, C, REST) = REST [owise] .
eq validTarget(PI, C, envcon(CONXS,< PI : EnvPort | target : C > PORTS) REST)
 = true .
eq validTarget(PI, C, REST) = false [owise] .
```

*Environment Actuation.* The rules with label `env-resp` specify the behavior of actuation operations. There are two cases: either a mode change is triggered or not. In the first rule, there is a mode transition from the current mode `L`, where one of its triggers, `PI`, which is connected to the controller `C` (`validTarget`), has received a content (`isPortPresent`). In the second rule, all input ports in the mode transitions from `L` that are connected with `C` has received no content (`allPortsAbsent`). For both cases, the constraint `B` is considered for the actuation time `T` with respect to the actuation time bound $(lt, gt)$ and the clock skew, and all data subcomponents "connected" to `C` are updated (`updateRespData`).

```
crl [env-resp]:
   envStep(time(T) mode(L)  response((C :(LT,UT), SIT)) REST)
=> execEnv(time(T) mode(L') response(SIT) addConst(REST', B and B'))
 if jumps(L -[PI,PRS]-> L' ; JUMPS) REST'' := REST
 /\ validTarget(PI, C, REST)
 /\ B' := isPortPresent(PI, REST)
 /\ B  := timeConst(T, LT, UT, REST)
 /\ REST' := updateRespData(C, REST) .

crl [env-resp]:
   envStep(time(T) mode(L) response((C :(LT,UT), SIT)) REST)
=> envStep(time(T) mode(L) response(SIT) addConst(REST', B and B'))
 if B' := allPortsAbsent(L, C, REST)
 /\ B  := timeConst(T, LT, UT, REST)
 /\ REST' := updateRespData(C, REST) .
```

The function `isPortPresent` gives a constraint for the content of a given input port `PI` having a value with the second item `[true]` (note that `B'` is always `[true]` by construction). The function `allPortsAbsent` returns a constraint stating that all trigger input ports of each mode transition from mode `L` are not present if they are connected to controller `C`.

```
eq isPortPresent(PI, feature((PI |-> (E # B : B'), FMAP)) REST) = B and B' .

eq allPortsAbsent(L, C, jumps(JUMPS) REST)
 = allPortsAbsent(L, JUMPS, C, REST, [true]) .

eq allPortsAbsent(L, (L -[PRS]-> L') ; JUMPS, C, REST, PHI)
 = allPortsAbsent(L, JUMPS, C ,REST, PHI and allPortsAbsent(PRS,C,REST)) .
eq allPortsAbsent(L, JUMPS, C, REST, PHI) = PHI [owise] .

ceq allPortsAbsent((PI, PRS), C, REST)
  = not isPortPresent(PI,REST) and allPortsAbsent(PRS,C,REST)
  if validTarget(PI, C, REST) .
eq allPortsAbsent(PRS, C, REST) = [true] [owise] .
```

The function updateRespData updates the content of the state variable CI by the content of the input port PI, provided that there is an internal connection PI =>> CI, PI is connected to the controller C, and PI has received a value. If PI is not present (i.e., the second item B is *false*), CI is not updated (i.e., the previous value is used). This is encoded using the conditional operator _?_:_.

```
eq updateRespData(C, envcon(CONXS,PORTS) feature(FMAP) REST)
 = updateRespData(CONXS, C, FMAP, envcon(CONXS,PORTS) feature(FMAP) REST) .

ceq updateRespData((PI =>> CI) ; CONXS, C, FMAP, data(DATA) REST)
  = updateRespData(CONXS, C, FMAP, data(DATA') REST')
 if validTarget(PI, C, REST)
 /\ E  # B : B'' := FMAP[PI]
 /\ E' # B' := DATA[CI]
 /\ DATA' := insert(CI, (B ? E : E') # (B or B'), DATA)
 /\ REAT' := addConst(REST, B'' and (B or B')) .

eq updateRespData(CONXS, C, FMAP, REST) = REST [owise] .
```

## 5 Merging Symbolic States

This section presents a state-space reduction method for our symbolic semantics of HybridSynchAADL. Recall that the behavior of threads and environments is specified by using two operations execTrans and execEnv. Even for one component, executeStep can produce many different execution results. In particular, for an environment interacting with $n$ controllers, the rules in Section 4.7 can generate $O((2n)!/2^n)$ different symbolic execution results, according to nondeterministic orders of sampling and actuating events. The modular encoding can symbolically eliminate the interleavings of components, but cannot eliminate the nondeterminism in a component.

To *symbolically* reduce the number of different execution results, we merge two terms that are syntactically identical except for SMT subterms into one constrained term. Let $t(u_1, \ldots, u_n)$ be a term with SMT subterms $u_1, \ldots, u_n$, and $x_1, \ldots, x_n$ be fresh SMT variables that do not appear in $t$. By definition, an *abstraction of built-ins* for $t$, denoted by $abs(t)$, is a constrained term

$$(x_1 = u_1 \wedge \cdots \wedge x_n = u_n) \parallel t(x_1, \ldots, x_n),$$

and it is semantically equivalent to $t$ (i.e., $[\![abs(t)]\!] = [\![true \parallel t]\!]$) [45].

**Definition 1.** *Two abstractions of built-ins $\phi_1 \parallel t_1$ and $\phi_2 \parallel t_2$ are* mergeable *iff there is a renaming substitution $\rho$ with $t_1 = \rho t_2$ (i.e., $t_1$ and $t_2$ are equivalent up to renaming). In this case, the* merged term *is the constrained term*

$$(\phi_1 \vee \rho\phi_2) \parallel t_1.$$

For example, $y = 2 + x \parallel f(y)$ and $z = 3 \parallel f(z)$ can be merged into the constrained term $(y = 2 + x \vee y = 3) \parallel f(y)$. We can easily show the following proposition that ensures the soundness and completeness of our method.

**Algorithm 1:** Semantic operation $f$ with state merging

```
    Input: A constrained object $\phi \parallel t$
    Output: A set of constrained objects
1   $post \longleftarrow \{(\phi' \parallel t') \mid f(\phi \parallel t) \rightsquigarrow \phi' \parallel t'\}$;
2   $\widehat{post} \longleftarrow \{(\phi' \wedge \psi \parallel t'') \mid (\psi \parallel t'') = abs(t'),\ \phi' \parallel t' \in post\}$;
3   while $\exists(\varphi_1 \parallel t_1), (\varphi_2 \parallel t_2) \in \widehat{post}$ with mergeable $t_1$ and $t_2$ do
4       $\varphi \parallel u$ is a merged term of $\varphi_1 \parallel t_1$ and $\varphi_2 \parallel t_2$;
5       $\widehat{post} \longleftarrow (\widehat{post} \cup \{\varphi \parallel u\}) \setminus \{\varphi_1 \parallel t_1,\ \varphi_2 \parallel t_2\}$;
6   return $\widehat{post}$;
```

**Proposition 1.** $[\![(\phi_1 \vee \rho\phi_2) \parallel t_1]\!] = [\![\phi_1 \parallel t_1]\!] \cup [\![\phi_2 \parallel t_2]\!]$

*Proof.* By definition (Section 2.4), $u \in [\![(\phi_1 \vee \rho\phi_2) \parallel t_1]\!]$ iff there is a substitution $\theta$ such that $u = \theta t_1$ and $\mathcal{T} \models \theta(\phi_1 \vee \rho\phi_2)$. Because $t_1 = \rho t_2$, $u = \theta\rho t_2$. Also, $\mathcal{T} \models \theta\phi_1$ or $\mathcal{T} \models \theta\rho\phi_2$. Thus, one of the following cases must hold: (i) $u = \theta t_1$ and $\mathcal{T} \models \theta\phi_1$, or (ii) $u = \theta\rho t_2$ and $\mathcal{T} \models \theta\rho\phi_2$. By definition, $u \in [\![\phi_1 \parallel t_1]\!]$ or $u \in [\![\rho(\phi_2 \parallel t_2)]\!]$. Because $\rho$ is a renaming substitution, $u \in [\![\rho(\phi_2 \parallel t_2)]\!]$ iff $u \in [\![\phi_2 \parallel t_2]\!]$. Consequently, $[\![(\phi_1 \vee \rho\phi_2) \parallel t_1]\!] = [\![\phi_1 \parallel t_1]\!] \cup [\![\phi_2 \parallel t_2]\!]$.

Algorithm 1 shows the new "merging" operation. It collects all the execution results by `executeStep` and merges all mergeable results. For our HybridSynchAADL semantics, by construction, Algorithm 1 always generates a single "merged" result. Hence, the `step` rule for the entire system will yield a single symbolic state for one synchronous step. Our method is inspired by state merging methods for symbolic execution [32], but has been generalized to deal with arbitrary constrained objects in HybridSynchAADL.

In order to obtain abstractions of built-ins for two terms $t_1$ and $t_2$, we define a function `symAbs(`$t_1$`, `$t_2$`)` that returns a triple $(u, \phi_1, \phi_2)$, where $\phi_1 \parallel u$ and $\phi_2 \parallel u$ are abstractions of $t_1$ and $t_2$, respectively, with the same set of fresh SMT variables. (We do not need to perform extra renaming by using the same fresh variables.) For example, `symAbs(`$e_1, e_2$`)` for two SMT expressions $e_1$ and $e_2$ is a triple $(x, x = e_1, x = e_2)$ with a fresh variable $x$, specified using the following equation, where an extra arguments `GEN` is used to generate fresh variables.

```
ceq symAbs(E1, E2, GEN)
  = {X, X === E1, X === E2, GEN'}
 if {X,GEN'} := gen(GEN,type(E1)) /\ type(E1) == type(E2) .
```

We define `symAbs` for each "pattern" of terms, such as locations, data contents, and data valuations, that can appear in the execution results of `execTrans` and `execEnv`. To illustrate, consider locations of the form `loc(`$r$`)` and data contents of the form $e$ `#` $b$ in our semantics. Using `symAbs` for SMT expressions described above, we can easily define `symAbs` for these cases as follows.

```
ceq symAbs(loc(R), loc(R'), GEN) = {loc(MR), CS, CS', GEN'}
 if {MR, CS, CS', GEN'} := symAbs(R, R', GEN) .
```

```
ceq symAbs(E # B, E' # B', GEN)
  = {ME # MB, CS1 and CS2, CS1' and CS2', GEN2}
 if {ME, CS1, CS1', GEN1} := symAbs(E, E', GEN)
 /\ {MB, CS2, CS2', GEN2} := symAbs(B, B', GEN1) .
```

The new operations `execTransMerge` and `execEnvMerge` find all the execution results obtained from `execTrans` and `execEnv`, respectively, and merge them into a single term using Algorithm 1. The function `collectResults` uses Maude's reflective features to compute a `;;`-separated list of execution results, each of which has the form `L | FMAP | DATA | PHI | GEN`, as explained in Sections 4.6 and 4.7. The function `symMerge` syntactically merges those terms into a single term, by means of the abstraction function `symAbs` for each pattern.

```
eq execTransMerge(BCF) = symMerge(collectResults('execTrans[upTerm(BCF)])) .
eq execEnvMerge(ECF)   = symMerge(collectResults('execEnv[upTerm(ECF)])) .

ceq symMerge(BTRS) = symMerge(BTRS, GEN)  if GEN := maxGen(BTRS) .
ceq symMerge((L  | FMAP  | DATA  | CS  | GEN)  ;;
             (L' | FMAP' | DATA' | CS' | GEN') ;; BTRS, GEN0)
  = symMerge((ML | MFMAP | MDATA | MCS | GEN3) ;; BTRS, GEN3)
 if {ML,   CS1,CS1',GEN1} := symAbs(L,    L',    GEN0)
 /\ {MFMAP,CS2,CS2',GEN2} := symAbs(FMAP, FMAP', GEN1)
 /\ {MDATA,CS3,CS3',GEN3} := symAbs(DATA, DATA', GEN2)
 /\ MCS := (CS and CS1 and CS2 and CS3) or (CS' and CS1' and CS2' and CS3') .
eq symMerge(BTR, GEN0) = BTR .
```

## 6 The HybridSynchAADL Analyzer

This section explains how the HybridSynchAADL Analyzer can be used to formally analyze HybridSynchAADL models under the virtually synchronous Hybrid PALS semantics. The tool is a plugin of OSATE, the standard tool environment for AADL, and supports the modeling and formal analysis of HybridSynchAADL models within OSATE. The HybridSynchAADL Analyzer:

1. provides an intuitive language to easily specify properties of HybridSynchAADL models,
2. automatically synthesizes a rewriting-modulo-SMT model from a given HybridSynchAADL model, and
3. performs symbolic reachability analysis based on the formal semantics of HybridSynchAADL using Maude and an SMT solver.[13]

[13] We use Yices2 as the underlying SMT solver. Yices2 does support polynomial constraints but does not support general classes of ODEs. Thus, the current version of our tool supports polynomial continuous dynamics only.

## 6.1 Property Specification Language

HybridSynchAADL Analyzer's *property specification language* allows the user to easily specify invariant and reachability properties of HybridSynchAADL models in an intuitive way, without having to understand Maude or the formal representations of the models. Such properties are given by propositional logic formulas whose atomic propositions are AADL Boolean expressions. Since HybridSynchAADL models are typically infinite-state systems, we only consider properties over behaviors up to a given time bound.

*Syntax.* Atomic propositions are given by AADL Boolean expressions in the AADL behavior annex syntax that involves data components, property values, and numeric constants. Each identifier in these conditions is fully qualified with its component path in the AADL syntax.[14] A "named" proposition can be declared using AADL Boolean expressions with an identifier as follows:

**proposition** [*id*]: *AADL Boolean Expression*

Such user-defined propositions can appear in propositional logic formulas, with the prefix ? for parsing purposes, for invariant and reachability properties.

An invariant property is composed of an identifier *name*, an initial condition $\varphi_{init}$, an invariant condition $\varphi_{inv}$, and a time bound $\tau_{bound}$, where $\varphi_{init}$ and $\varphi_{inv}$ are in propositional logic. Intuitively, the invariant property holds if for every initial state satisfying the initial condition $\varphi_{init}$, all reachable states within the time bound $\tau_{bound}$ satisfy the invariant condition $\varphi_{inv}$.

**invariant** [*name*]: $\varphi_{init}$ ==> $\varphi_{inv}$ **in time** $\tau_{bound}$

A reachability property is also composed of an identifier *name*, an initial condition $\varphi_{init}$, a goal condition $\varphi_{goal}$, and a time bound $\tau_{bound}$. A reachability property holds if there exists some state that satisfies the goal condition $\varphi_{inv}$ and that is reachable from an initial state satisfying the initial condition $\varphi_{init}$ within the time bound $\tau_{bound}$. It is worth noting that a reachability property can be expressed as an invariant property by *negating the goal condition.*

**reachability** [*name*]: $\varphi_{init}$ ==> $\varphi_{goal}$ **in time** $\tau_{bound}$

We can simplify component paths that appear repeatedly in conditions using component scopes. A *scoped expression* of the form

$$path \mid exp$$

denotes that the component path of each identifier in the expression *exp* begins with *path*. For example, $c_1$ . $c_2$ | (($x_1$ > $x_2$) and ($b_1$ = $b_2$)) is equivalent to ($c_1$ . $c_2$ . $x_1$ > $c_1$ . $c_2$ . $x_2$) and ($c_1$ . $c_2$ . $b_1$ = $c_1$ . $c_2$ . $b_2$). These scopes can be nested so that one scope may include another scope. For example, $c_1$ | (($c_2$ | ($x$ > $c_3$ . $y$)) = ($c_4$ | ($c_5$ | $b$))) is equivalent to the expression ($c_1$ . $c_2$ . $x$ > $c_1$ . $c_2$ . $c_3$ . $y$) = $c_1$ . $c_4$ . $c_5$ . $b$.

[14] A component path is given by a period-separated path of component identifiers in AADL; for example, $c_1.c_2.b$.

*Semantics.* The semantics of the property specification language is defined by means of equations in Maude. For a Boolean expression `COND` and a top-level component `OBJ`, its value, written $[\![$`COND`$]\!]$ `OBJ`, is *true* if the *normalized* expression `normal(COND)` without component scopes evaluates to a data content $b$ # $b'$, and both $b$ and $b'$ are true. (Recall that the second item $b'$ indicates whether all the identifiers in `COND` have some values in `OBJ`.)

```
ceq [[ COND ]] OBJ = B and B'
 if B # B' := evalPS(normal(COND), OBJ) .
```

By definition, scoped expressions are equivalent to *normalized* expressions without component scopes, where each identifier is fully qualified with the full component path. This transformation is specified using the following equations, where `E` and `E'` denote expressions, `PATH` and `PATH'` denote component paths, `VAR` denotes a variable identifier, and `nil` denotes the empty path.

```
eq normal(E)                = normal(nil, E) .
eq normal(PATH, PATH' | E)  = normal(PATH . PATH', E) .
eq normal(PATH, VAR)        = PATH . VAR .
eq normal(PATH, VALUE)      = VALUE .

eq normal(PATH, E and E') = normal(PATH, E) and normal(PATH, E') .
eq normal(PATH, E or  E') = normal(PATH, E) or  normal(PATH, E') .
eq normal(PATH, E +   E') = normal(PATH, E) +   normal(PATH, E') .
...
eq normal(PATH,   not(E)) = not(normal(PATH, E)) .
```

The function `evalPS(E, COMPS)` evaluates a normalized expression `E` to its data content $e$ # $b$, given a set of components `COMPS`. The following equations specify `evalPS`, where `BehComponent` is a superclass of both classes `Thread` and `Env`. Notice that `evalPS` uses `eval` as a subroutine, which is defined in Section 4.6 to evaluate expressions with respect to behavior configurations.

```
eq evalPS(C . PATH . VAR, < C : Ensemble | subcomponents : COMPS > REST)
 = evalPS(PATH . VAR, COMPS) .

eq evalPS(C . VAR, < C : BehComponent | subcomponents : COMPS,
                                        properties : PROPS > REST)
 = eval(VAR, data(readData(COMPS)) prop(PROPS)) .

eq evalPS(VALUE, COMPS) = eval(VALUE, none) .

eq evalPS(E and E', COMPS) = evalPS(E, COMPS) and evalPS(E', COMPS) .
eq evalPS(E or  E', COMPS) = evalPS(E, COMPS) or  evalPS(E', COMPS) .
eq evalPS(E +   E', COMPS) = evalPS(E, COMPS) +   evalPS(E', COMPS) .
...
eq evalPS(not(E),   COMPS) = not(evalPS(E, COMPS)) .
```

Reachability and invariant properties in our property specification language correspond to Maude's search command. A reachability property of the form

$\varphi_{init}$ `==>` $\varphi_{goal}$ `in time` $\tau_{bound}$ corresponds to the following search command to find its witness (i.e., the property holds if the search command finds a solution), where $N_{bound}$ is the quotient of $\tau_{bound}$ by the period of the model, and `initState` denotes the term representation of the entire model:

```
search [$N_{bound}$] {([[$\varphi_{init}$]] initState) || initState}
           =>* {COND || OBJ}
      such that check-sat(COND and finalConst(OBJ) and ([[$\varphi_{goal}$]] OBJ)) .
```

Similarly, an invariant property of the form $\varphi_{init}$ `==>` $\varphi_{inv}$ `in time` $\tau_{bound}$ is specified as the following search command to find its counterexample (i.e., the property holds if the search command *cannot* find a solution):

```
search [$N_{bound}$] {([[$\varphi_{init}$]] initState) || initState}
           =>* {COND || OBJ}
      such that check-sat(COND and finalConst(OBJ) and not([[$\varphi_{inv}$]] OBJ)) .
```

*Example.* Consider the thermostat system in Section 3 that consists of two thermostat controllers `ctrl1` and `ctrl2` and their environments `env1` and `env2`, respectively. The following declares two propositions `inRan1` and `inRan2` using the property specification language. For example, `inRan1` holds if the value of `env1`'s data subcomponent `x` is between 10 and 25.

```
proposition [inRan1]: env1 | (x > 10 and x <= 25)
proposition [inRan2]: env2 | (x > 5  and x <= 10)
```

The following declares the invariant property `inv`. The initial condition states that the value of `env1`'s data subcomponent `x` satisfies $|x-15| < 3$ and the value of `env2`'s data subcomponent `x` satisfies $|x - 7| < 1$. This property holds if for each initial state satisfying the initial condition, any reachable state within the time bound 30 satisfies the conditions `inRan1`, `inRan2`, and `env1.x > env2.x`.

```
invariant [inv]: abs(env1.x - 15) < 3 and abs(env2.x - 7) < 1
     ==> ?inRan1 and ?inRan2 and (env1.x > env2.x) in time 30
```

This requirement `inv` corresponds to the following search command in Maude. The constant `initState` is replaced by the term representation of the entire model, and `inRan1` and `inRan2` are replaced by the related Boolean expressions.

```
search [3] {([[abs(env1 . x - 15) < 3 and abs(env2 . x - 7) < 1}]] initState)
           || initState}
       =>* {COND || OBJ} such that
 check-sat(COND and not([[inRan1 and inRan2 and env1 . x > env2 . x]] OBJ)) .
```

## 6.2 Tool Architecture and Interface

Figure 11 depicts the architecture of the HybridSynchAADL Analyzer. The tool first statically checks whether a given model is a valid HybridSynchAADL

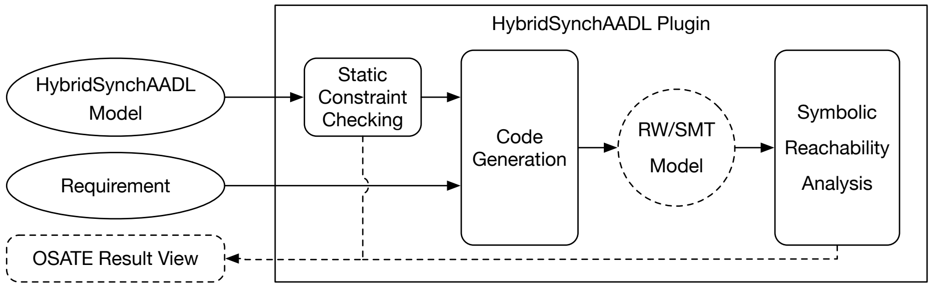


**Fig. 11.** The architecture of the HybridSynchAADL Analyzer.

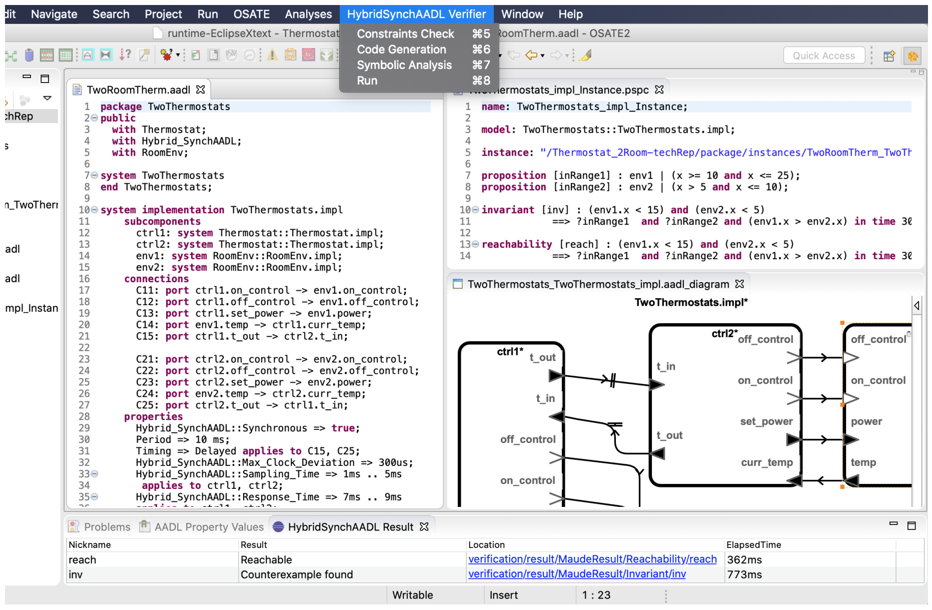
model. It then uses OSATE's code generation capability to synthesize the corresponding rewriting-modulo-SMT model automatically. Finally, our tool can perform the symbolic analysis, using rewriting modulo SMT, to check whether the model satisfies given invariant and reachability requirements. The tool internally invokes Maude and an SMT solver to symbolically execute the model in rewriting modulo SMT. The Result view in OSATE displays the results of constraint checking and symbolic analysis for the user in a readable format.

By syntactically validating a HybridSynchAADL model, we ensure that the corresponding model in rewriting modulo SMT is symbolically executable. The tool checks whether a given model satisfies all the syntactic constraints of HybridSynchAADL in Section 3. For example, environment components (with `Hybrid_SynchAADL::isEnvironment`) can only contain data subcomponents of type `Base_Types::Float`, and must declare the continuous dynamics using the property `Hybrid_SynchAADL::ContinuousDynamics`. The tool also checks other "trivial" constraints that are assumed in the semantics of HybridSynchAADL; e.g., all input ports are connected to some output ports, the model contains no AADL features that are not supported by the semantics, and so on.

Figure 12 illustrates the interface of the HybridSynchAADL Analyzer tool that is fully integrated with OSATE. The left editor shows the AADL code of `TwoThermostats` in Section 3, the bottom right editor shows its graphical representation, and the top right editor shows two requirements. All available functionalities of OSATE are applicable. The HybridSynchAADL menu contains three items for constraint checking, rewriting-modulo-SMT code generation, and symbolic formal analysis, and `Run` that executes all three commands. The `Symbolic Analysis` item has been already clicked, and the `Result` view at the bottom summaries the analysis results.

## 7 Case Studies

This section presents case studies on designing virtually synchronous CPSs for controlling distributed drones using HybridSynchAADL. In these models, the controllers of multiple drones collaborate to achieve common maneuver goals, such as rendezvous and formation control depicted in Fig. 13. These controllers



```
package TwoThermostats
public
  with Thermostat;
  with Hybrid_SynchAADL;
  with RoomEnv;

system TwoThermostats
end TwoThermostats;

system implementation TwoThermostats.impl
    subcomponents
      ctrl1: system Thermostat::Thermostat.impl;
      ctrl2: system Thermostat::Thermostat.impl;
      env1: system RoomEnv::RoomEnv.impl;
      env2: system RoomEnv::RoomEnv.impl;
    connections
      C11: port ctrl1.on_control -> env1.on_control;
      C12: port ctrl1.off_control -> env1.off_control;
      C13: port ctrl1.set_power -> env1.power;
      C14: port env1.temp -> ctrl1.curr_temp;
      C15: port ctrl1.t_out -> ctrl2.t_in;

      C21: port ctrl2.on_control -> env2.on_control;
      C22: port ctrl2.off_control -> env2.off_control;
      C23: port ctrl2.set_power -> env2.power;
      C24: port env2.temp -> ctrl2.curr_temp;
      C25: port ctrl2.t_out -> ctrl1.t_in;
    properties
      Hybrid_SynchAADL::Synchronous => true;
      Period => 10 ms;
      Timing => Delayed applies to C15, C25;
      Hybrid_SynchAADL::Max_Clock_Deviation => 300us;
      Hybrid_SynchAADL::Sampling_Time => 1ms .. 5ms
       applies to ctrl1, ctrl2;
      Hybrid_SynchAADL::Response_Time => 7ms .. 9ms
```

```
name: TwoThermostats_impl_Instance;

model: TwoThermostats::TwoThermostats.impl;

instance: "/Thermostat_2Room-techRep/package/instances/TwoRoomTherm_TwoTh

proposition [inRange1] : env1 | (x >= 10 and x <= 25);
proposition [inRange2] : env2 | (x > 5 and x <= 10);

invariant [inv] : (env1.x < 15) and (env2.x < 5)
            ==> ?inRange1  and ?inRange2 and (env1.x > env2.x) in time 3

reachability [reach] : (env1.x < 15) and (env2.x < 5)
            ==> ?inRange1  and ?inRange2 and (env1.x > env2.x) in time 3
```

| Nickname | Result | Location | ElapsedTime |
|---|---|---|---|
| reach | Reachable | verification/result/MaudeResult/Reachability/reach | 362ms |
| inv | Counterexample found | verification/result/MaudeResult/Invariant/inv | 773ms |

**Fig. 12.** The interface of the HybridSynchAADL Analyzer tool in OSATE.

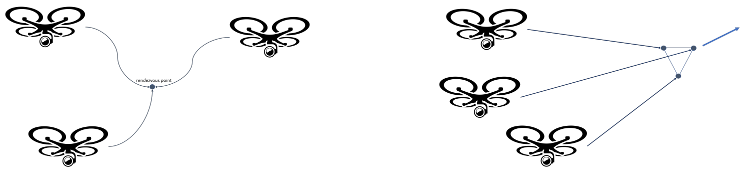


**Fig. 13.** Rendezvous and formation control of distributed drones

are physically distributed, because each controller is included in the hardware of each drone. The models take into account continuous dynamics, asynchronous communication, network delays, clock skews, etc. Furthermore, the interactions between the controllers and the drones are affected by imprecise local clocks.

## 7.1 Distributed Consensus Algorithms

We use distributed consensus algorithms in [43] to synchronize the movements of distributed drones. Each drone has an *information state* that represents the drone's local view of the coordination task. Examples of information states include the rendezvous position, the center of a formation, etc. There is no centralized controller with information about the entire system. Each drone repeatedly exchanges the information state with neighboring drones, and eventually, the information states of all drones converge to a common value.

We summarize distributed consensus algorithms for rendezvous and formation control in [43] below. Suppose that there are $N$ drones in moving two-dimensional space. Let two-dimensional vectors $\vec{x}_i$, $\vec{v}_i$, and $\vec{a}_i$, for $1 \leq i \leq N$, denote, respectively, the position, velocity and, acceleration of the $i$-th drone. The continuous dynamics of the $i$-th drone is then specified by the differential equations $\dot{\vec{x}}_i = \vec{v}_i$ and $\dot{\vec{v}}_i = \vec{a}_i$. Let $A$ denote the adjacency matrix representing the underlying communication network. In particular, if the $(i, j)$ entry $A_{ij}$ of $A$ is 0, the $i$-th drone cannot receive information from the $j$-th drone. The controller of a drone gives the value of acceleration as a control input.

The goal of the rendezvous problem [43] is for all drones to arrive near a common location simultaneously. In a distributed consensus algorithm, the acceleration $\vec{a}_i$ of the $i$-th drone is given by the following equation:

$$\vec{a}_i = -\sum_{j=1}^{N} A_{ij}\big((\vec{x}_i - \vec{x}_j) + \gamma(\vec{v}_i - \vec{v}_j)\big),$$

where $\gamma > 0$ denotes the coupling strength between $\vec{v}_i$. The information state $\vec{x}_i$ of the $i$-th drone is directed toward the information states of its neighbors and eventually converges to a consensus value. It is worth noting that the exact location and time of the rendezvous are not given.

In formation control problems [43], one drone is designated as a leader and the other drones follow the leader in a given formation. The information state is the position of the leader that is *continuously changing*. Suppose that the $N$-th drone is the leader and the others are the followers. The acceleration $\vec{a}_i$ of the $i$-th drone is given by the following equation:

$$\vec{a}_i = \vec{a}_N - \alpha\big((\vec{e}_i - \vec{x}_N) + \gamma(\vec{v}_i - \vec{v}_N)\big) - \sum_{j=1}^{N-1} A_{ij}\big((\vec{e}_i - \vec{e}_j) + \gamma(\vec{v}_i - \vec{v}_j)\big),$$

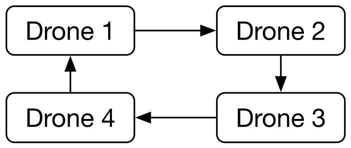
where $\vec{e}_i = \vec{x}_i - \vec{o}_i$ with $\vec{o}_i$ an *offset vector* for the formation, and $\alpha$ and $\gamma$ are positive constants. The position $\vec{x}_i$ of the $i$-th drone eventually converges to $\vec{x}_N - \vec{o}_i$, while the position $\vec{x}_N$ of the leader changes with velocity $\vec{v}_N$.

For both cases, a simplified model for drones with *single-integrator* dynamics is also considered, assuming acceleration is negligible. Acceleration $\vec{a}_i$ is always 0, and the controller of a drone directly gives the value of velocity as a control input. For single-integrator dynamics, the following equations provide velocity $\vec{v}_i$ for rendezvous and formation control, respectively [43]:

$$\vec{v}_i = -\sum_{j=1}^{N} A_{ij}(\vec{x}_i - \vec{x}_j),$$

$$\vec{v}_i = \vec{v}_N - \alpha(\vec{e}_i - \vec{x}_N) - \sum_{j=1}^{N-1} A_{ij}(\vec{e}_i - \vec{e}_j).$$

This model provides a reasonable approximation when the velocity is low and is often much easier to analyze using SMT solving.



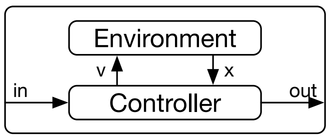


**Fig. 14.** The architecture of four drones (left), and a drone component (right).

## 7.2 The HybridSynchAADL Models

This section presents a HybridSynchAADL model that specifies rendezvous for four distributed drones. We show models for single-integrator dynamics. A controller for double-integrator dynamics with acceleration needs to exchange velocity as well as positions with other controllers, and thus the HybridSynchAADL model requires much more text to specify connections. Nevertheless, we have developed a variety of HybridSynchAADL models for both rendezvous and formation control of different numbers of drones with respect to single-integrator and double-integrator dynamics. All these models are available at `https://tinyurl.com/r8pqwtr`.

Figure 14 illustrates the structure of our model for rendezvous. There are four drone components. Each drone is connected with two other drones to exchange positions. For example, Drone 1 sends its position to Drone 2, and receives the position of Drone 4. A drone component consists of an environment and its controller. An environment component specifies the physical model of the drone, including position and velocity. A controller component interacts with the environment according to the sampling and actuating times. All controllers in the model have the same period.

In each round, a controller determines a new velocity to synchronize its movement with the other drones. The controller obtains the position $\vec{x}$ from its environment according to the sampling time. The position of the connected drone is sent in the previous round, and is already available to the controller at the beginning of the round. The controller sends the current position $\vec{x}$ through its output port. In the meantime, the environment changes its position according to the velocity indicated by its controller, where the new velocity $\vec{v}$ from the controller become effective according to the actuation time.

*Top-Level Component.* Figure 15 shows the top-level system component for our model. `FourDronesSystem` includes four subcomponents for drones. Each drone sends the position $(X, Y)$ through two output ports `oX` and `oY`, and receives the position of the other drone through two input ports `iX` and `iY`. The component is declared to be synchronous with a 100 ms period. The connections between drone components are delayed and the output ports have some initial values. The maximal clock skew is given by `Hybrid_SynchAADL::Max_Clock_Deviation`.

*Drone Component.* A drone component in Figure 16 have two input ports `iX` and `iY` and two output ports `oX` and `oY` for communicating with other drones.

```
system FourDronesSystem
end FourDronesSystem;

system implementation FourDronesSystem.rend
  subcomponents
    dr1: system Drone::Drone.rend;    dr2: system Drone::Drone.rend;
    dr3: system Drone::Drone.rend;    dr4: system Drone::Drone.rend;
  connections
    C1:  port dr1.oX -> dr2.iX;    C2: port dr1.oY -> dr2.iY;
    C3:  port dr2.oX -> dr3.iX;    C4: port dr2.oY -> dr3.iY;
    C5:  port dr3.oX -> dr4.iX;    C6: port dr3.oY -> dr4.iY;
    C7:  port dr4.oX -> dr1.iX;    C8: port dr4.oY -> dr1.iY;
  properties
    Hybrid_SynchAADL::Synchronous => true;
    Period => 100ms;
    Hybrid_SynchAADL::Max_Clock_Deviation => 10ms;
    Timing => Delayed applies to C1, C2, C3, C4, C5, C6, C7, C8;
    Data_Model::Initial_Value => ("0.0") applies to
        dr1.oX, dr2.oX, dr3.oX, dr4.oX,
        dr1.oY, dr2.oY, dr3.oY, dr4.oY;
end FourDronesSystem.rend;
```

**Fig. 15.** The top-level system component `FourDronesSystem`.

The implementation contains a controller component `ctrl` and an environment component `env`. The controller `ctrl` communicates with the outside components using `Drone`'s ports `iX` and `iY`, Also, `ctrl` receives the current position from `env` using the ports `currX` and `currY`, and sends a new velocity to `env` using the ports `velX` and `velY`, according to the sampling and actuating times with respect to local clocks, declared as the properties of the implementation `Drone.rend`.

*Environment.* An environment component in Figure 17 has two input ports `velX` and `velY` and two output ports `currX` and `currY`, and is declared with the property `Hybrid_SynchAADL::isEnvironment`. It has four subcomponents to represent the position $(x, y)$ and velocity $(v_x, v_y)$ of the drone. The values of $x$ and $y$ are sent to the controller using the output ports `currX` and `currY`, and the values of $vx$ and $vy$ are updated by the controller using the input ports `velX` and `velY`. The dynamics of $(x, y)$ is given as continuous functions $x(t) = v_x t + x(0)$ and $y(t) = v_y t + y(0)$ over time $t$ in `Hybrid_SynchAADL::ContinuousDynamics`, which are actually equivalent to the differential equations $\dot{x} = v_x$ and $\dot{y} = v_y$.

*Controller.* Figure 18 shows a controller system component. As explained above, there are four ports `iX`, `iY`, `oX`, and `oY` for communicating with other controllers, and four ports `currX`, `currY`, `velX`, and `velY` for interacting with the environment. The system implementation `DroneControl.rend` includes the process component `ctrlProc`. As shown in Figure 19, `ctrlProc` again includes the thread component

```
system Drone
  features
    iX: in data port Base_Types::Float;
    iY: in data port Base_Types::Float;
    oX: out data port Base_Types::Float;
    oY: out data port Base_Types::Float;
end Drone;

system implementation Drone.rend
  subcomponents
    ctrl: system DroneControl::DroneControl.rend;
    env:  system Environment::Environment.impl;
  connections
    C1: port ctrl.oX -> oX;             C2: port ctrl.oY -> oY;
    C3: port iX -> ctrl.iX;             C4: port iY -> ctrl.iY;
    C5: port env.currX -> ctrl.currX;    C6: port env.currY -> ctrl.currY;
    C7: port ctrl.velX -> env.velX;      C8: port ctrl.velY -> env.velY;
  properties
    Hybrid_SynchAADL::Sampling_Time => 2ms .. 4ms;
    Hybrid_SynchAADL::Response_Time => 6ms .. 9ms;
end Drone.rend;
```

**Fig. 16.** A drone component in HybridSynchAADL.

cThread in its implementation DroneControlProc.rend. The input and output ports of a wrapper component (e.g., ctrlProc) are connected to the ports of the enclosed subcomponent (e.g., cThread).

Figure 20 shows a thread component for a controller. It has the periodic dispatch protocol, and the period is inherited from the top-level component. When the thread dispatches, the transition from init to exec is taken. There are two possibilities from state exec. When the distance between the current position and the connected drone is too close, the new velocity is set to $(0, 0)$ so that the drone stops to avoid a collision. Otherwise, the new velocity is set toward the connected drone according to a *discretized* version of the distributed consensus algorithm above where $v_x$ and $v_y$ are chosen from $\{-2, -1, 0, 1, 2\}$. Finally, the current position is assigned to the output ports.

## 7.3 Formal Analysis Using the HybridSynchAADL Analyzer

We consider two properties for our model: (i) there is no collision between any drones; and (ii) all drones could eventually gather together. Our goal is to formally analyze these properties up to bound 300 ms from given initial states. These properties are specified in our property specification language using the atomic propositions collision, gather, and initial as follows:

```
invariant [noCollision]: ?initial ==> not ?collision in time 300;
```

```
system Environment
  features
    currX: out data port Base_Types::Float;
    currY: out data port Base_Types::Float;
    velX: in data port Base_Types::Float;
    velY: in data port Base_Types::Float;
  properties
    Hybrid_SynchAADL::isEnvironment => true;
end Environment;

system implementation Environment.impl
  subcomponents
    x:  data Base_Types::Float;     y: data Base_Types::Float;
    vx: data Base_Types::Float;    vy: data Base_Types::Float;
  connections
    C1: port x -> currX;           C2: port y -> currY;
    C3: port velX -> vx;           C4: port velY -> vy;
  properties
    Hybrid_SynchAADL::ContinuousDynamics =>
      "x(t) = vx * t + x(0); y(t) = vy * t + y(0);";
    Data_Model::Initial_Value => ("param") applies to
      x, y, vx, vy;
end Environment.impl;
```

**Fig. 17.** An environment component in HybridSynchAADL.

```
reachability [rendezvous]: ?initial ==> ?gather in time 300;
```

The property `noCollision` holds if all states reachable from any `initial` states within 300 ms satisfy `not ?collision`, and the property `rendezvous` holds if some state reachable from some `initial` state within 300 ms satisfies `gather`.

To specify these properties we define atomic propositions `collision`, `gather`, and `initial` in the property specification languages as follows. The initial values of `x`, `y`, `vx`, and `vy` are declared to be parametric in Fig. 17. Notice that there are infinitely many initial states satisfying the proposition `initial`.

```
proposition [collision]:
  (abs(dr1.env.x - dr2.env.x) < 0.2 and abs(dr1.env.y - dr2.env.y) < 0.2) or
  (abs(dr1.env.x - dr3.env.x) < 0.2 and abs(dr1.env.y - dr3.env.y) < 0.2) or
  (abs(dr1.env.x - dr4.env.x) < 0.2 and abs(dr1.env.y - dr4.env.y) < 0.2) or
  (abs(dr2.env.x - dr3.env.x) < 0.2 and abs(dr2.env.y - dr3.env.y) < 0.2) or
  (abs(dr2.env.x - dr4.env.x) < 0.2 and abs(dr2.env.y - dr4.env.y) < 0.2) or
  (abs(dr3.env.x - dr4.env.x) < 0.2 and abs(dr3.env.y - dr4.env.y) < 0.2);

proposition [gather]:
    abs(dr1.env.x - dr2.env.x) < 1 and abs(dr1.env.y - dr2.env.y) < 1 and
    abs(dr1.env.x - dr3.env.x) < 1 and abs(dr1.env.y - dr3.env.y) < 1 and
    abs(dr1.env.x - dr4.env.x) < 1 and abs(dr1.env.y - dr4.env.y) < 1 and
```

```
system DroneControl
  features
    iX: in data port Base_Types::Float;
    iY: in data port Base_Types::Float;
    oX: out data port Base_Types::Float;
    oY: out data port Base_Types::Float;
    currX: in data port Base_Types::Float;
    currY: in data port Base_Types::Float;
    velX: out data port Base_Types::Float;
    velY: out data port Base_Types::Float;
end DroneControl;

system implementation DroneControl.rend
  subcomponents
    ctrlProc: process DroneControlProc.rend;
  connections
    C1: port ctrlProc.oX -> oX;         C2: port ctrlProc.oY -> oY;
    C3: port iX -> ctrlProc.iX;         C4: port iY -> ctrlProc.iY;
    C5: port currX -> ctrlProc.currX;   C6: port currY -> ctrlProc.currY;
    C7: port ctrlProc.velX -> velX;     C8: port ctrlProc.velY -> velY;
end DroneControl.rend;
```

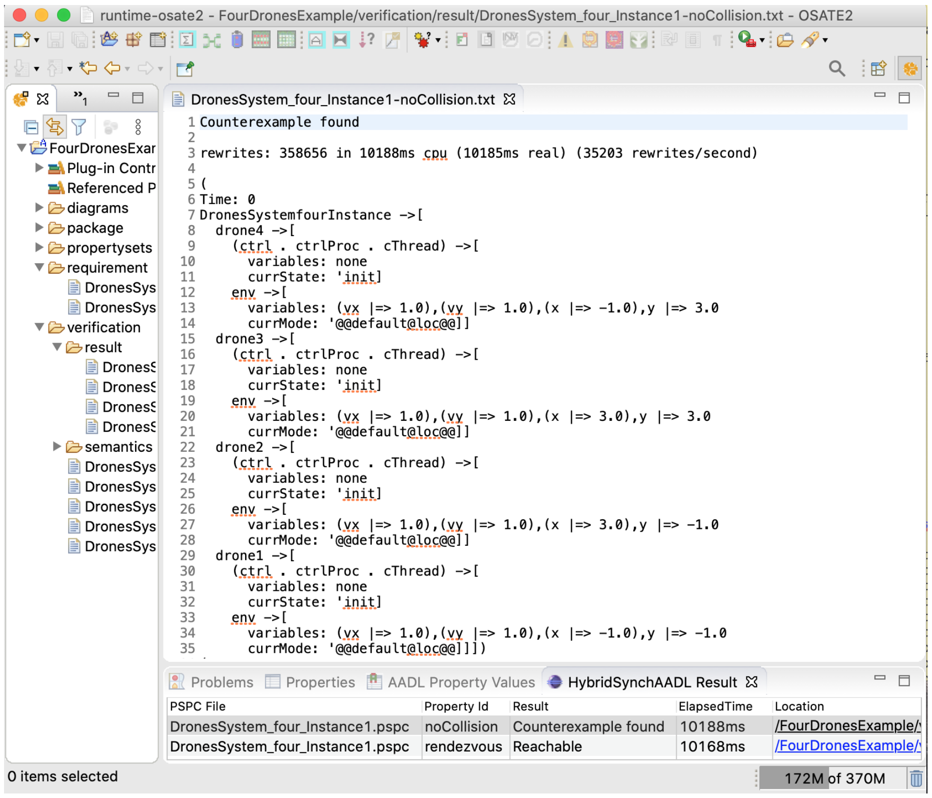
**Fig. 18.** A controller system component.

```
process DroneControlProc
  features
    iX: in data port Base_Types::Float;
    iY: in data port Base_Types::Float;
    oX: out data port Base_Types::Float;
    oY: out data port Base_Types::Float;
    currX: in data port Base_Types::Float;
    currY: in data port Base_Types::Float;
    velX: out data port Base_Types::Float;
    velY: out data port Base_Types::Float;
end DroneControlProc;

process implementation DroneControlProc.rend
  subcomponents
    cThread: process DroneControlThread.rend;
  connections
    C1: port cThread.oX -> oX;         C2: port cThread.oY -> oY;
    C3: port iX -> cThread.iX;         C4: port iY -> cThread.iY;
    C5: port currX -> cThread.currX;   C6: port currY -> cThread.currY;
    C7: port cThread.velX -> velX;     C8: port cThread.velY -> velY;
end DroneControlProc.rend;
```

**Fig. 19.** A controller process component

```
thread DroneControlThread
  features
    iX: in data port Base_Types::Float;
    iY: in data port Base_Types::Float;
    oX: out data port Base_Types::Float;
    oY: out data port Base_Types::Float;
    currX: in data port Base_Types::Float;
    currY: in data port Base_Types::Float;
    velX: out data port Base_Types::Float;
    velY: out data port Base_Types::Float;
  properties
    Dispatch_Protocol => Periodic;
end DroneControlThread;

thread implementation DroneControlThread.rend
  annex behavior_specification {**
    variables
      nx, ny : Base_Types::Float;
    states
      init: initial complete state;
      exec, output: state;
    transitions
      init -[on dispatch]-> exec;
      exec -[abs(currX - iX) < 0.5 and abs(currY - iY) < 0.5]-> output {
        velX := 0;
        velY := 0
      };
      exec -[otherwise]-> output {
        nx := -#DroneSpec::A * (currX - inX);
        ny := -#DroneSpec::A * (currY - inY);
        if (nx > 1.5)      velX := 2   elsif (nx > 0.5)   velX := 1
        elsif (nx >= -0.5) velX := 0   elsif (nx >= -1.5) velX := -1
        else               velX := -2  end if;
        if (ny > 1.5)      velY := 2   elsif (ny > 0.5)   velY := 1
        elsif (ny >= -0.5) velY := 0   elsif (ny >= -1.5) velY := -1
        else               velY := -2  end if
      };
      output -[ ]-> init {
        oX := currX;
        oY := currY;
      };
  **};
end DroneControlThread.rend;
```

**Fig. 20.** A controller thread in HybridSynchAADL



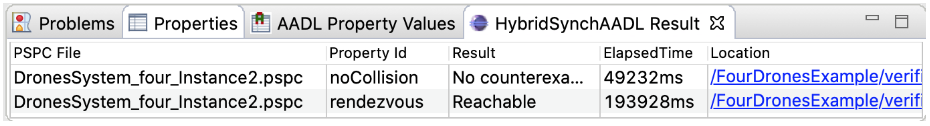
```
Counterexample found

rewrites: 358656 in 10188ms cpu (10185ms real) (35203 rewrites/second)

(
Time: 0
DronesSystemfourInstance ->[
  drone4 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: none
      currState: 'init]
    env ->[
      variables: (vx |=> 1.0),(vy |=> 1.0),(x |=> -1.0),y |=> 3.0
      currMode: '@@default@loc@@]]
  drone3 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: none
      currState: 'init]
    env ->[
      variables: (vx |=> 1.0),(vy |=> 1.0),(x |=> 3.0),y |=> 3.0
      currMode: '@@default@loc@@]]
  drone2 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: none
      currState: 'init]
    env ->[
      variables: (vx |=> 1.0),(vy |=> 1.0),(x |=> 3.0),y |=> -1.0
      currMode: '@@default@loc@@]]
  drone1 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: none
      currState: 'init]
    env ->[
      variables: (vx |=> 1.0),(vy |=> 1.0),(x |=> -1.0),y |=> -1.0
      currMode: '@@default@loc@@]]])
```

| PSPC File | Property Id | Result | ElapsedTime | Location |
|---|---|---|---|---|
| DronesSystem_four_Instance1.pspc | noCollision | Counterexample found | 10188ms | /FourDronesExample/ |
| DronesSystem_four_Instance1.pspc | rendezvous | Reachable | 10168ms | /FourDronesExample/ |

**Fig. 21.** The formal analysis results for the properties `noCollision` and `rendezvous`, where a counterexample of `noCollision` is shown in the editor.

```
    abs(dr2.env.x - dr3.env.x) < 1 and abs(dr2.env.y - dr3.env.y) < 1 and
    abs(dr2.env.x - dr4.env.x) < 1 and abs(dr2.env.y - dr4.env.y) < 1 and
    abs(dr3.env.x - dr4.env.x) < 1 and abs(dr3.env.y - dr4.env.y) < 1;

proposition [initial]:
    drone1.env.x < 0   and drone1.env.y < 0 and
    drone2.env.x > 1.5 and drone2.env.y < 0 and
    drone3.env.x > 1.5 and drone3.env.y > 1.5 and
    drone4.env.x < 0   and drone4.env.y > 1.5;
```

Figure 21 shows the results of symbolic reachability analysis for the properties `noCollision` and `rendezvous`. We observe that there exist a counterexample for the invariant property `noCollision` and a witness for the reachability property `rendezvous` exist. Since `initial` contains no constraint on the velocities, the drones can have arbitrary speed in the initial state.



| PSPC File | Property Id | Result | ElapsedTime | Location |
|---|---|---|---|---|
| DronesSystem_four_Instance2.pspc | noCollision | No counterexa... | 49232ms | /FourDronesExample/verif |
| DronesSystem_four_Instance2.pspc | rendezvous | Reachable | 193928ms | /FourDronesExample/verif |

**Fig. 22.** The results for the modified properties `noCollision` and `rendezvous`.

Consider the modified versions of `noCollision` and `rendezvous` with the extra constraint `velconst` in the initial condition as follows. As shown in Figure 22, there exists no counterexample of `noCollision` in this case.

```
invariant [noCollision]:
   ?initial and ?velconst ==> not ?collision in time 300;

reachability [rendezvous]:
   ?initial and ?velconst ==> ?gather in time 300;

proposition [velconst]:
   abs(drone1.env.vx) <= 2 and abs(drone1.env.vy) <= 2 and
   abs(drone2.env.vx) <= 2 and abs(drone2.env.vy) <= 2 and
   abs(drone3.env.vx) <= 2 and abs(drone3.env.vy) <= 2 and
   abs(drone4.env.vx) <= 2 and abs(drone4.env.vy) <= 2;
```

## 8 Evaluation

This section evaluates the HybridSynchAADL Analyzer tool by answering the following research questions: (1) how effective is our symbolic reachability analysis method for analyzing invariant properties, compared to randomized simulations? (2) how effective is our state merging technique for symbolic analysis? (3) how effective is the Hybrid PALS methodology for reducing the complexity of analyzing virtually synchronous CPSs?

To answer these questions, we have analyzed HybridSynchAADL models of networked thermostat systems and rendezvous and formation control of distributed drones. We consider many variants of these models in the experiments, such as different numbers of components, different sampling and actuating times, single-integrator and double-integrator dynamics, etc.

We have run all experiments on a 16-core 32-thread Intel Xeon 2.8GHz with 256 GB memory. We use a specialized implementation of Maude connected with Yices 2.6, where MCSAT is enabled for nonlinear arithmetic. We have repeated each experiment 10 times (with different random seeds) and report the average results and the standard deviation. The models and the experimental results are available in `https://tinyurl.com/r8pqwtr`.

| Model | Sampling | Actuation | $\epsilon$ | Invariant property |
|---|---|---|---|---|
| Thermostat | $20 \sim 30$ | $60 \sim 70$ | 5 | temperatures are between 20 and 50 |
| Rendezvous | $30 \sim 50$ | $60 \sim 80$ | 10 | distance between drones greater than 0.5 |
| Formation | $30 \sim 50$ | $60 \sim 80$ | 10 | distance between drones greater than 0.3 |

**Table 1.** Timing parameters and properties.

### 8.1 Analyzing Invariant Properties

We compare symbolic analysis with randomized simulations, where concrete sampling and actuating times are randomly chosen. We measure the time taken to find counterexamples for invariant properties up to a bound, given a set of "faulty" models obtained by manually injecting bugs and a concrete initial state without parameters. We perform randomized simulations *repeatedly* until a counterexample is found for the invariant properties. At the same time, we perform symbolic reachability analysis (with state merging) for the same invariant properties to find counterexamples.

Table 1 summarizes the time parameters and invariant properties used in this experiment, where $\epsilon$ denotes the maximal clock skew, and all models have period 100 (milliseconds). To build faulty HybridSynchAADL models, we manipulate the sampling and actuating times so that the controllers respond too late to the environments. We consider different time bounds $\tau_b$ for the invariant properties to observe different analysis results due to different time bounds. We set a timeout of 20 minutes for this experiment.

Table 2 shows the experimental results. The number $N$ denotes the number of thermostat or drone components in the model. For randomized simulations (`random`), "timeout" (`-`) means that a counterexample of the bounded property is not found (`NF`) until timeout (20 minutes) by repeated randomized simulations.[15] For symbolic analysis (`symbolic`), once a counterexample is found (`FO`) for one bound $T$, the results for larger bounds $T' > T$ are omitted in the table and grayed out, because they are exactly the same as for $T$, since symbolic reachability analysis uses a breadth-first strategy.

As expected, we observe that randomized simulations are often effective for finding "obvious" bugs. E.g., consider the results for "Rendezvous" with single-integrator dynamics for $N = 2$ in Table 2. A counterexample was found by randomized simulations in less than 1 second when $\tau_b = 500$, but it took more than 1 minute when $\tau_b = 400$, and no counterexample could be found when $\tau_b \leq 300$. Because the injected faults are caused by excessive sampling and actuating times, invariant violations are easier to find with a larger bound. Also, randomized simulations cannot show the absence of counterexamples.

[15] Since different random seeds were used in the repeated experiments, randomized simulations often gave very different results, whereas symbolic analysis always gave similar results. In Table 2, `FO`($k$) is written if a counterexample found $k$ times in the repeated experiments; the time for timeout cases is considered to be 20 minutes.

| Model | N | Method | Bound = 100 | | | Bound = 200 | | | Bound = 300 | | | Bound = 400 | | | Bound = 500 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Result | Time (s) | | Result | Time (s) | | Result | Time (s) | | Result | Time (s) | | Result | Time (s) | |
| | | | | AVG | STD | | AVG | STD | | AVG | STD | | AVG | STD | | AVG | STD |
| Thermostat | 2 | random | TO | - | - | TO | - | - | TO | - | - | TO | - | - | TO | - | - |
| | | symbolic | NF | 0.20 | 0.00 | NF | 0.46 | 0.00 | NF | 0.97 | 0.02 | NF | 2.02 | 0.02 | FO | 4.31 | 0.09 |
| | 3 | random | TO | - | - | TO | - | - | TO | - | - | TO | - | - | FO(10) | 25.67 | 24.13 |
| | | symbolic | NF | 0.25 | 0.01 | NF | 0.64 | 0.01 | NF | 1.51 | 0.03 | NF | 3.55 | 0.09 | FO | 8.40 | 0.15 |
| | 4 | random | TO | - | - | TO | - | - | TO | - | - | TO | - | - | FO(10) | 18.87 | 13.67 |
| | | symbolic | NF | 0.30 | 0.01 | NF | 0.89 | 0.01 | NF | 2.16 | 0.05 | NF | 6.05 | 0.14 | FO | 13.96 | 0.47 |
| Drone rendezvous (single) | 2 | random | TO | - | - | TO | - | - | TO | - | - | FO(10) | 78.80 | 80.33 | FO(10) | 0.39 | 0.40 |
| | | symbolic | NF | 0.56 | 0.01 | NF | 1.57 | 0.03 | FO | 3.36 | 0.05 | | | | | | |
| | 3 | random | TO | - | - | FO(10) | 127.12 | 104.80 | FO(10) | 318.68 | 353.53 | FO(10) | 1.86 | 1.70 | FO(10) | 0.10 | 0.05 |
| | | symbolic | NF | 0.80 | 0.02 | FO | 2.40 | 0.03 | | | | | | | | | |
| | 4 | random | TO | - | - | TO | - | - | TO | - | - | TO | - | - | TO | - | - |
| | | symbolic | NF | 1.07 | 0.01 | NF | 3.64 | 0.04 | NF | 10.38 | 0.12 | FO | 28.38 | 0.39 | | | |
| Drone formation (single) | 2 | random | TO | - | - | TO | - | - | TO | - | - | TO | - | - | FO(10) | 262.09 | 222.48 |
| | | symbolic | NF | 0.77 | 0.02 | FO | 2.24 | 0.05 | | | | | | | | | |
| | 3 | random | TO | - | - | TO | - | - | TO | - | - | TO | - | - | FO(9) | 778.33 | 651.81 |
| | | symbolic | NF | 1.05 | 0.00 | FO | 3.70 | 0.04 | | | | | | | | | |
| | 4 | random | TO | - | - | TO | - | - | TO | - | - | TO | - | - | FO(4) | 992.16 | 318.81 |
| | | symbolic | NF | 1.40 | 0.02 | FO | 5.06 | 0.10 | | | | | | | | | |
| Drone rendezvous (double) | 2 | random | TO | - | - | TO | - | - | FO(10) | 8.24 | 10.89 | FO(10) | 3.09 | 2.18 | FO(10) | 4.40 | 5.54 |
| | | symbolic | NF | 0.79 | 0.02 | FO | 2.37 | 0.05 | | | | | | | | | |
| | 3 | random | TO | - | - | TO | - | - | FO(10) | 157.15 | 138.30 | FO(10) | 182.69 | 192.10 | FO(10) | 165.28 | 185.01 |
| | | symbolic | NF | 1.14 | 0.02 | FO | 14.68 | 0.30 | | | | | | | | | |
| | 4 | random | TO | - | - | TO | - | - | FO(3) | 1052.85 | 246.59 | FO(5) | 871.03 | 457.65 | FO(1) | 1127.25 | 230.07 |
| | | symbolic | NF | 1.50 | 0.02 | FO | 6.77 | 0.09 | | | | | | | | | |
| Drone formation (double) | 2 | random | TO | - | - | TO | - | - | FO(1) | 1087.96 | 354.31 | FO(10) | 5.32 | 4.43 | FO(10) | 2.66 | 1.92 |
| | | symbolic | NF | 1.12 | 0.02 | FO | 3.67 | 0.07 | | | | | | | | | |
| | 3 | random | TO | - | - | TO | - | - | TO | - | - | FO(10) | 80.57 | 61.01 | FO(10) | 60.70 | 60.69 |
| | | symbolic | NF | 1.56 | 0.01 | TO | - | - | TO | - | - | TO | - | - | TO | - | - |
| | 4 | random | TO | - | - | TO | - | - | FO(5) | 793.13 | 499.14 | FO(10) | 42.36 | 32.69 | FO(10) | 25.75 | 24.14 |
| | | symbolic | NF | 2.07 | 0.04 | FO | 9.74 | 0.18 | | | | | | | | | |

**Table 2.** Comparing randomized simulation and symbolic reachability analysis.

In contrast, we observe that symbolic reachability analysis can be effective to: (i) find subtle counterexamples; and (ii) verify that bounded invariants hold. Consider, e.g., "Drone Rendezvous" with single-integrator dynamics for $N = 4$. Randomized simulations do not find counterexamples of the invariant properties in 20 minutes, whereas symbolic analysis finds one for $\tau_b = 400$ in less than 30 seconds, and verifies in less than 11 seconds that no counterexample exists for bound 300 (that is, the invariant property holds up to bound 300).

For double-integrator dynamics (where control input is given by acceleration instead of velocity), symbolic analysis sometimes cannot find a counterexample before timeout. This is caused by *high-order* nonlinear constraints generated by double-integrator dynamics. Consider double-integrator "Formation" for $N = 3$. Symbolic analysis timed out for bounds greater than 100, although it can verify that no counterexample exists for bound 100. Both single-integrator and double-integrator models involve nonlinear constraints, but double-integrator models give higher-order constraints that Yices2 cannot effectively deal with.

## 8.2 Effect of Merging Symbolic States

We evaluate the effect of state merging for symbolic reachability analysis. We have performed symbolic reachability analysis, with state merging and without state merging, for generating all reachable symbolic states up to given bounds.[16] In both cases, we measure the execution time, the size of accumulated SMT formulas, the number of calls to the SMT solver (Yices2), and the number of reachable symbolic states up to the bound. We set a timeout of 180 minutes.

Table 3 summarizes the experimental results, where a timeout is denoted by '-', and $|Const|$ denotes the accumulated size of SMT constraints in thousands. We consider again the five HybridSynchAADL models with different numbers of components. In this experiment, we use parameterized values for initial states of benchmarks models, instead of concrete initial states. More precisely, data subcomponents for temperatures and drone positions in the models declare the property `Data_Model::Initial_Value => ("param")`.

The experimental results show that, despite the increased burden on the SMT solver, the state-space reduction by state merging almost always significantly improves the performance of symbolic analysis. Symbolic analysis with state merging always generates *one symbolic state* for each step, whereas analysis without merging may generate a huge number of symbolic states. E.g., for "Drone rendezvous" with $N = 3$ and bound 100, symbolic analysis without merging generates $140,609$ symbolic states in one synchronous step.

The reason is that symbolic analysis with state merging involves a much smaller number of SMT calls and a much smaller size of accumulated SMT constraints than those without state merging. For instance, consider "Drone rendezvous" with single-integrator dynamics for $N = 3$ and bound 100. With state merging, the size of the accumulated SMT constraints is about $58,800$ and the number of SMT calls is 441. Without merging, the size of the constraints is about 245 million and the number of SMT calls is more than 1 million.

## 8.3 Complexity Reduction by Hybrid PALS

To gauge the complexity reduction obtained by Hybrid PALS, we have developed a *concrete* asynchronous distributed semantics for HybridSynchAADL models This semantics is adapted from the formal semantics for a subset of AADL in [40]. We measured the execution times for generating all reachable *concrete* states up to bounds in distributed asynchronous models. We set a timeout of 360 minutes.

For the experiment, we consider *highly simplified* distributed models with the following assumptions: (i) all clocks are perfectly synchronized; (ii) there is no network delay; (iii) controllers (nondeterministically) choose one of the predefined values for sampling and actuating times; and (iv) controllers take zero time to perform transitions. We use floating-point arithmetic to compute the continuous dynamics of the environments given by polynomials.

[16] We use *false* for the reachability goal condition, and run the Maude search command to find a symbolic state satisfying the goal. Since there exists no state satisfying *false*, the search command enumerates all reachable state up to a bound.

| Model | N | Bound | Symbolic with merging | | | | | Symbolic without merging | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Time (s) | | \|Const\| (k) | # SMT call | # Sym. State | Time (s) | | \|Const\| (k) | # SMT call | # Sym. State |
| | | | AVG | STD | | | | AVG | STD | | | |
| Thermostat | 2 | 100 | 0.2 | 0.0 | 4.5 | 81 | 2 | 0.9 | 0.0 | 85.5 | 1,478 | 10 |
| | | 200 | 0.4 | 0.0 | 18.5 | 205 | 3 | 25.1 | 0.3 | 2,829.4 | 24,808 | 66 |
| | | 300 | 0.9 | 0.0 | 44.1 | 329 | 4 | 335.5 | 2.6 | 32,401.9 | 182,423 | 223 |
| | | 400 | 1.9 | 0.0 | 81.4 | 453 | 5 | 1,908.7 | 39.4 | 164,158.8 | 683,809 | 591 |
| | 3 | 100 | 0.2 | 0.0 | 6.7 | 120 | 2 | 13.8 | 0.3 | 1,235.0 | 17,840 | 28 |
| | | 200 | 0.6 | 0.0 | 29.9 | 305 | 3 | 1,560.2 | 6.4 | 175,734.9 | 1,136,343 | 465 |
| | | 300 | 1.5 | 0.0 | 75.8 | 490 | 4 | - | - | - | - | - |
| | | 400 | 4.8 | 0.1 | 144.4 | 675 | 5 | - | - | - | - | - |
| | 4 | 100 | 0.3 | 0.0 | 9.0 | 159 | 2 | 208.7 | 3.4 | 17,364.3 | 214,238 | 82 |
| | | 200 | 0.9 | 0.0 | 42.9 | 405 | 3 | - | - | - | - | - |
| | | 300 | 2.7 | 0.1 | 114.1 | 651 | 4 | - | - | - | - | - |
| | | 400 | 9.1 | 0.2 | 222.9 | 897 | 5 | - | - | - | - | - |
| Drone Rendezvous (Single) | 2 | 100 | 0.6 | 0.0 | 39.2 | 295 | 2 | 44.1 | 1.0 | 3,708.6 | 26,503 | 2,705 |
| | | 200 | 1.5 | 0.0 | 137.1 | 589 | 3 | - | - | - | - | - |
| | | 300 | 2.9 | 0.0 | 306.0 | 883 | 4 | - | - | - | - | - |
| | | 400 | 4.9 | 0.1 | 544.1 | 1,177 | 5 | - | - | - | - | - |
| | 3 | 100 | 0.8 | 0.0 | 58.8 | 441 | 2 | 3,173.6 | 51.3 | 245,569.1 | 1,378,503 | 140,609 |
| | | 200 | 2.4 | 0.0 | 232.7 | 881 | 3 | - | - | - | - | - |
| | | 300 | 5.1 | 0.1 | 554.4 | 1,321 | 4 | - | - | - | - | - |
| | | 400 | 8.9 | 0.0 | 1018.9 | 1,761 | 5 | - | - | - | - | - |
| | 4 | 100 | 1.1 | 0.0 | 78.4 | 587 | 2 | - | - | - | - | - |
| | | 200 | 3.4 | 0.1 | 346.2 | 1,173 | 3 | - | - | - | - | - |
| | | 300 | 7.7 | 0.1 | 866.5 | 1,759 | 4 | - | - | - | - | - |
| | | 400 | 14.4 | 0.1 | 1628.9 | 2,345 | 5 | - | - | - | - | - |
| Drone Formation (Single) | 2 | 100 | 0.8 | 0.0 | 53.3 | 331 | 2 | 115.9 | 1.6 | 9,063.2 | 53,043 | 5,409 |
| | | 200 | 2.1 | 0.0 | 191.9 | 661 | 3 | - | - | - | - | - |
| | | 300 | 4.7 | 0.1 | 451.3 | 991 | 4 | - | - | - | - | - |
| | | 400 | 28.3 | 0.5 | 836.2 | 1,321 | 5 | - | - | - | - | - |
| | 3 | 100 | 1.1 | 0.0 | 79.3 | 477 | 2 | 8,563.6 | 33.3 | 576,700.2 | 2,757,043 | 281,217 |
| | | 200 | 3.4 | 0.1 | 313.9 | 953 | 3 | - | - | - | - | - |
| | | 300 | 7.9 | 0.0 | 778.0 | 1,429 | 4 | - | - | - | - | - |
| | | 400 | - | - | - | - | - | - | - | - | - | - |
| | 4 | 100 | 1.4 | 0.0 | 105.3 | 623 | 2 | - | - | - | - | - |
| | | 200 | 4.8 | 0.1 | 455.5 | 1,245 | 3 | - | - | - | - | - |
| | | 300 | 13.4 | 0.2 | 1179.5 | 1,867 | 4 | - | - | - | - | - |
| | | 400 | - | - | - | - | - | - | - | - | - | - |
| Drone Rendezvous (Double) | 2 | 100 | 0.8 | 0.0 | 51.2 | 295 | 2 | 66.6 | 0.4 | 5,035.8 | 26,503 | 2,705 |
| | | 200 | 2.3 | 0.0 | 182.8 | 589 | 3 | - | - | - | - | - |
| | | 300 | 4.6 | 0.1 | 412.9 | 883 | 4 | - | - | - | - | - |
| | | 400 | 7.8 | 0.1 | 739.3 | 1,177 | 5 | - | - | - | - | - |
| | 3 | 100 | 1.1 | 0.0 | 76.8 | 441 | 2 | 5,007.1 | 102.6 | 329,009.0 | 1,378,503 | 140,609 |
| | | 200 | 3.6 | 0.0 | 310.2 | 881 | 3 | - | - | - | - | - |
| | | 300 | 8.1 | 0.1 | 749.0 | 1,321 | 4 | - | - | - | - | - |
| | | 400 | 14.8 | 0.2 | 1386.2 | 1,761 | 5 | - | - | - | - | - |
| | 4 | 100 | 1.5 | 0.0 | 102.4 | 587 | 2 | - | - | - | - | - |
| | | 200 | 5.3 | 0.0 | 461.6 | 1,173 | 3 | - | - | - | - | - |
| | | 300 | 12.4 | 0.0 | 1171.7 | 1,759 | 4 | - | - | - | - | - |
| | | 400 | 24.2 | 0.1 | 2218.1 | 2,345 | 5 | - | - | - | - | - |
| Drone Formation (Double) | 2 | 100 | 1.1 | 0.0 | 71.4 | 331 | 2 | 139.2 | 1.7 | 9,960.4 | 49,299 | 3,537 |
| | | 200 | 3.5 | 0.1 | 261.6 | 661 | 3 | - | - | - | - | - |
| | | 300 | 28.7 | 0.5 | 627.6 | 991 | 4 | - | - | - | - | - |
| | | 400 | - | - | - | - | - | - | - | - | - | - |
| | 3 | 100 | 1.6 | 0.0 | 106.0 | 477 | 2 | 9,140.9 | 138.2 | 503,500.4 | 1,817,299 | 183,873 |
| | | 200 | 5.5 | 0.1 | 426.3 | 953 | 3 | - | - | - | - | - |
| | | 300 | - | - | - | - | - | - | - | - | - | - |
| | | 400 | - | - | - | - | - | - | - | - | - | - |
| | 4 | 100 | 2.0 | 0.0 | 140.7 | 623 | 2 | - | - | - | - | - |
| | | 200 | 7.7 | 0.0 | 616.7 | 1,245 | 3 | - | - | - | - | - |
| | | 300 | - | - | - | - | - | - | - | - | - | - |
| | | 400 | - | - | - | - | - | - | - | - | - | - |

**Table 3.** Symbolic analysis with merging and without merging.

| Model | Bound | N = 2 | | | | | | N = 3 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | \|Time sample\| = 1 | | | \|Time sample\| = 2 | | | \|Time sample\| = 1 | | | \|Time sample\| = 2 | | |
| | | Time (s) | | # State (k) | Time | | # State (k) | Time (s) | | # State (k) | Time | | # State (k) |
| | | AVG | STD | | AVG | STD | | AVG | STD | | AVG | STD | |
| Thermostat | 50 | 5.0 | 0.0 | 9.9 | 51.4 | 0.3 | 93.0 | 4,117.6 | 233.0 | 1,808.9 | - | - | - |
| | 100 | 5.0 | 0.1 | 10.0 | 53.6 | 0.2 | 99.8 | 4,066.7 | 118.8 | 1,812.7 | - | - | - |
| | 150 | 6.0 | 0.0 | 12.5 | 95.3 | 0.2 | 187.0 | 5,852.0 | 214.1 | 2,365.1 | - | - | - |
| | 200 | 6.2 | 0.0 | 13.2 | 116.3 | 0.4 | 250.7 | 6,185.6 | 233.4 | 2,433.4 | - | - | - |
| | 250 | 8.4 | 0.0 | 18.7 | 258.0 | 1.3 | 506.2 | - | - | - | - | - | - |
| | 300 | 8.7 | 0.0 | 20.0 | 298.3 | 1.2 | 596.4 | - | - | - | - | - | - |
| Drone Rendezvous (Single) | 50 | 5.2 | 0.0 | 9.3 | 50.7 | 0.4 | 84.0 | 951.0 | 4.1 | 886.1 | - | - | - |
| | 100 | 5.8 | 0.0 | 10.7 | 53.4 | 0.5 | 90.5 | 989.8 | 4.3 | 939.7 | - | - | - |
| | 150 | 6.1 | 0.0 | 11.5 | 55.8 | 0.2 | 97.8 | 1,034.6 | 7.6 | 988.9 | - | - | - |
| | 200 | 9.3 | 0.1 | 19.2 | 71.2 | 0.4 | 134.4 | 2,554.9 | 92.5 | 2,253.9 | - | - | - |
| | 250 | 10.8 | 0.1 | 23.4 | 85.5 | 0.5 | 171.4 | 3,345.1 | 62.4 | 2,902.1 | - | - | - |
| | 300 | 14.3 | 0.1 | 30.9 | 102.6 | 0.8 | 207.3 | 5,481.4 | 156.8 | 3,923.7 | - | - | - |
| Drone Formation (Single) | 50 | 7,076.2 | 132.3 | 3,540.3 | - | - | - | - | - | - | - | - | - |
| | 100 | 8,070.0 | 499.2 | 3,888.4 | - | - | - | - | - | - | - | - | - |
| | 150 | 8,258.6 | 791.0 | 4,037.5 | - | - | - | - | - | - | - | - | - |
| | 200 | - | - | - | - | - | - | - | - | - | - | - | - |
| | 250 | - | - | - | - | - | - | - | - | - | - | - | - |
| | 300 | - | - | - | - | - | - | - | - | - | - | - | - |
| Drone Rendezvous (Double) | 50 | 123.6 | 0.7 | 148.7 | 1,449.1 | 10.2 | 1,337.2 | - | - | - | - | - | - |
| | 100 | 148.9 | 1.2 | 188.2 | 1,595.1 | 15.0 | 1,500.6 | - | - | - | - | - | - |
| | 150 | 170.5 | 0.9 | 226.2 | 1,918.1 | 100.0 | 1,813.1 | - | - | - | - | - | - |
| | 200 | 839.9 | 7.3 | 1,121.2 | 10,638.3 | 1,209.9 | 5,495.6 | - | - | - | - | - | - |
| | 250 | 1,505.5 | 10.7 | 1,869.6 | - | - | - | - | - | - | - | - | - |
| | 300 | 2,641.6 | 169.8 | 2,764.0 | - | - | - | - | - | - | - | - | - |
| Drone Formation (Double) | 50 | - | - | - | - | - | - | - | - | - | - | - | - |
| | 100 | - | - | - | - | - | - | - | - | - | - | - | - |
| | 150 | - | - | - | - | - | - | - | - | - | - | - | - |
| | 200 | - | - | - | - | - | - | - | - | - | - | - | - |
| | 250 | - | - | - | - | - | - | - | - | - | - | - | - |
| | 300 | - | - | - | - | - | - | - | - | - | - | - | - |

**Table 4.** Analyzing distributed asynchronous models .

Table 4 shows the experimental results. $|Time\ sample|$ is the number of the predefined values for sampling and actuating times. The number of concrete states is written in thousands. We see that the number of reachable states in distributed asynchronous models even for very simple models with unrealistic assumptions. E.g., for "Thermostat" with $N = 3$ and bound 200 the number of reachable states is greater than 2.4 million, and it took more than 1.7 hours to generate these states, whereas symbolic analysis with state merging in Table 3 needed less than 1 second to do the same.

## 9 Related Work

One distinguishing feature of our work is that we perform model checking verification of virtually synchronous CPSs with typical CPS features such as advanced control programs, continuous behaviors, communication delays, execution times, and clock skews. We consider continuous dynamics and imprecise local clocks at the same time, which is not the case in CPS analysis tools such as [20, 27, 31]. We can deal with continuous behaviors *and* complex control programs, whereas most formal frameworks are strong at analyzing either discrete or continuous behaviors. The latter class includes reachability analysis tools for (networks of "finite-location") hybrid automata, such as SpaceEx [27] and HyComp [20]: hybrid automata cannot deal well with the "discrete complexity" of CPSs, such as complex control programs and data structures.

*Almost-Synchronous Systems.* Our work is related to a broader body of work on analyzing "almost-synchronous" systems, including quasi-synchrony [17, 18, 30, 33], GALS [29, 41], approximate synchrony [22], time-triggered architectures [46, 48], virtual synchrony [8, 39], etc. A common theme of these approaches is to simplify the design and verification of distributed real-time systems using various synchronization methods. Our method makes it possible to model and verify almost-synchronous systems *with continuous behaviors*, including of *continuous behaviors perturbed by clock skews*, which are typically not considered in related work. We also provide a convenient language and modeling environment for modeling almost-synchronous CPSs.

*Hybrid Systems in AADL.* The *Hybrid Annex* for AADL [4] allows specifying continuous behaviors in AADL, and its developers provide a theorem proving support for proving properties in Hoare Logic combined with Duration Calculus [2]. Controller behaviors are defined in Hybrid CSP. Only a "synchronous" subset without message delays is considered, and clock skews, etc., are not taken into account. In contrast: we analyze models specified using AADL's expressive Behavior Annex, we provide automatic model checking analysis instead of interactive theorem proving, and we consider (virtually synchronous) CPSs—with clock skews, network delays, etc.

In [15], an *Uncertainty Annex* is added to the Hybrid Annex. Uncertain Hybrid AADL models can be transformed into networks of priced timed automata that can then be subjected to statistical model checking using Uppaal-SMC to evaluate the *performance* of the models. Another hybrid annex is proposed in [42], and an AADL sublanguage, called AADL+, where continuous behaviors can be defined using stochastic differential equations is given in [35]. Both approaches come with some kind of operational semantics and simulation, but with no formal analysis support.

*PALS and AADL. Synchronous AADL* [7, 12] and its multi-rate extension [11] support the modeling and analysis of synchronous PALS models of virtually synchronous distributed real-time systems without continuous behaviors in AADL. The explicit-state model checker Maude is used to analyze these models. In contrast, we analyze continuous behaviors for all possible sampling/actuation times. This required us to leave the explicit-state world and use Maude with SMT solving. In this way, we can cover all possible behaviors, but are currently restricted to reachability analysis.

*Formal Analysis of Hybrid PALS Models.* The paper [8] shows how some Hybrid PALS synchronous models with simple finite-state machine controllers—and their bounded reachability problem—can be encoded as logical formulas and analyzed by the dReal solver for nonlinear theories over the reals. However, it is difficult to model complex (hierarchical) CPSs in SMT. In contrast, our approach provides a simple and intuitive way of modeling synchronous Hybrid PALS models using a well-known modeling standard. In addition, since we use Maude with SMT solving instead of just SMT solving, we can also analyze systems with complex control programs and data types.

## 10 Concluding Remarks

In this paper we have shown that how complex virtually synchronous CPSs—where each component samples and actuates physical environments at times that depend on local imperfect clocks—can be effectively modeled and analyzed using: (i) a combination of rewriting and SMT solving, and (ii) a novel domain-specific modeling language. We have leveraged Hybrid PALS and the integration of Maude and the SMT solver Yices2 to provide model checking of bounded reachability properties for virtually synchronous distributed CPSs with complex control programs, continuous environments, and nondeterministic but bounded clock skews, network delays, and execution times. To make our verification methodology accessible to modelers, we have defined the HybridSynchAADL modeling language for specifying synchronous Hybrid PALS models in the modeling standard AADL, and have integrated the modeling and verification of such models into the OSATE tool environment for AADL. We have presented a state merging method to reduce the number of symbolic states and the size of accumulated SMT formulas. We have demonstrated the effectiveness of the HybridSynchAADL Analyzer tool, state merging, and the Hybrid PALS methodology on advanced applications, including on a set of autonomous drones that communicate to achieve common goals.

Maude is only integrated with the Yices2 and CVC4 SMT solvers. Although we can specify continuous behaviors using ODEs in HybridSynchAADL, we can only analyze systems with polynomial continuous dynamics. We should therefore integrate Maude with ODE solvers, such as dReal [28] and Flow* [19], to analyze systems whose continuous behaviors are given as (nonlinear) ODEs. Finally, we can also extend HybridSynchAADL from single-rate to multi-rate controllers, in a similar way as [5].

## References

1. Abrial, J., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, LNCS, vol. 1165. Springer (1996)
2. Ahmad, E., Dong, Y., Wang, S., Zhan, N., Zou, L.: Adding formal meanings to AADL with Hybrid Annex. In: Proc. FACS'14. LNCS, vol. 8997. Springer (2015)

3. Ahmad, E., Larson, B.R., Barrett, S.C., Zhan, N., Dong, Y.: Hybrid annex: An aadl extension for continuous behavior and cyber-physical interaction modeling. In: ACM SIGAda Ada Letters. vol. 34, pp. 29–38. ACM (2014)
4. Ahmad, E., Larson, B.R., Barrett, S.C., Zhan, N., Dong, Y.: Hybrid annex: an AADL extension for continuous behavior and cyber-physical interaction modeling. In: Proc. ACM SIGAda annual conference on High integrity language technology (HILT'14). ACM (2014)
5. Bae, K., Meseguer, J., Ölveczky, P.C.: Formal patterns for multirate distributed real-time systems. Science of Computer Programming 91, 3–44 (2014)
6. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous aadl and its formal analysis in real-time maude. In: International Conference on Formal Engineering Methods. pp. 651–667. Springer (2011)
7. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous AADL and its formal analysis in Real-Time Maude. In: Proc. ICFEM'11. LNCS, vol. 6991. Springer (2011)
8. Bae, K., Ölveczky, P.C., Kong, S., Gao, S., Clarke, E.M.: SMT-based analysis of virtually synchronous distributed hybrid systems. In: Proc. HSCC'16. ACM (2016)
9. Bae, K., Ölveczky, P.C., Kong, S., Gao, S., Clarke, E.M.: Smt-based analysis of virtually synchronous distributed hybrid systems. In: Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control. pp. 145–154. ACM (2016)
10. Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of multirate synchronous aadl. In: Proc. FM. Lecture Notes in Computer Science, vol. 8442, pp. 94–109. Springer (2014)
11. Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of Multirate Synchronous AADL. In: Proc. FM'14. LNCS, vol. 8442. Springer (2014)
12. Bae, K., Ölveczky, P.C., Meseguer, J., Al-Nayeem, A.: The SynchAADL2Maude tool. In: Proc. FASE'12. LNCS, vol. 7212. Springer (2012)
13. Bae, K., Rocha, C.: Symbolic state space reduction with guarded terms for rewriting modulo SMT. Science of Computer Programming 178, 20–42 (2019)
14. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) 51(3), 1–39 (2018)
15. Bao, Y., Chen, M., Zhu, Q., Wei, T., Mallet, F., Zhou, T.: Quantitative performance evaluation of uncertainty-aware Hybrid AADL designs using statistical model checking. IEEE Transactions on CAD of Integrated Circuits and Systems 36(12), 1989–2002 (2017)
16. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. pp. 171–177. Springer (2011)
17. Baudart, G., Bourke, T., Pouzet, M.: Soundness of the quasi-synchronous abstraction. In: 2016 Formal Methods in Computer-Aided Design (FMCAD). pp. 9–16. IEEE (2016)
18. Caspi, P., Mazuet, C., Paligot, N.R.: About the design of distributed control systems: The quasi-synchronous approach. In: International Conference on Computer Safety, Reliability, and Security. pp. 215–226. Springer (2001)
19. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Proc. CAV. pp. 258–263. Springer (2013)
20. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: HyComp: An SMT-based model checker for hybrid systems. In: Proc. TACAS. Lecture Notes in Computer Science, vol. 9035. Springer (2015)

21. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude – A High-Performance Logical Framework, Lecture Notes in Computer Science, vol. 4350. Springer (2007)
22. Desai, A., Seshia, S.A., Qadeer, S., Broman, D., Eidson, J.C.: Approximate synchrony: An abstraction for distributed almost-synchronous systems. In: Proc. CAV'15. LNCS, vol. 9207. Springer (2015)
23. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV. LNCS, vol. 8559, pp. 737–744. Springer (July 2014)
24. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language. SEI series in software engineering, Addison-Wesley (2012)
25. Franca, R.B., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The aadl behaviour annex–experiments and roadmap. In: 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007). pp. 377–382. IEEE (2007)
26. França, R., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL behaviour annex - experiments and roadmap. In: Proc. ICECCS'07. IEEE (2007)
27. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Proc. CAV'11. LNCS, vol. 6806. Springer (2011)
28. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: Proc. CADE. Lecture Notes in Computer Science, vol. 7898. Springer (2013)
29. Girault, A., Ménier, C.: Automatic production of globally asynchronous locally synchronous systems. In: International Workshop on Embedded Software. pp. 266–281. Springer (2002)
30. Halbwachs, N., Mandel, L.: Simulation and verification of asynchronous systems by means of a synchronous model. In: Sixth International Conference on Application of Concurrency to System Design (ACSD'06). pp. 3–14. IEEE (2006)
31. Kong, S., Gao, S., Chen, W., Clarke, E.M.: dReach: $\delta$-reachability analysis for hybrid systems. In: Proc. TACAS. Lecture Notes in Computer Science, vol. 7898. Springer (2015)
32. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. Acm Sigplan Notices 47(6), 193–204 (2012)
33. Larrieu, R., Shankar, N.: A framework for high-assurance quasi-synchronous systems. In: 2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE). pp. 72–83. IEEE (2014)
34. Liu, J., Li, T., Ding, Z., Qian, Y., Sun, H., He, J.: Aadl+: a simulation-based methodology for cyber-physical systems. Frontiers of Computer Science 13(3), 516–538 (2019)
35. Liu, J., Li, T., Ding, Z., Qian, Y., Sun, H., He, J.: AADL+: a simulation-based methodology for cyber-physical systems. Frontiers Comput. Sci. 13(3), 516–538 (2019)
36. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
37. Meseguer, J.: Twenty years of rewriting logic. The Journal of Logic and Algebraic Programming 81(7), 721–781 (2012)
38. Meseguer, J., Ölveczky, P.C.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. Theoretical Computer Science 451, 1–37 (2012)

39. Miller, S., Cofer, D., Sha, L., Meseguer, J., Al-Nayeem, A.: Implementing logical synchrony in integrated modular avionics. In: Proc. IEEE/AIAA 28th Digital Avionics Systems Conference (DASC'09). IEEE
40. Ölveczky, P.C., Boronat, A., Meseguer, J.: Formal semantics and analysis of behavioral aadl models in real-time maude. In: Formal Techniques for Distributed Systems, pp. 47–62. Springer (2010)
41. Potop-Butucaru, D., Caillaud, B.: Correct-by-construction asynchronous implementation of modular synchronous specifications. Fundamenta Informaticae 78(1), 131–159 (2007)
42. Qian, Y., Liu, J., Chen, X.: Hybrid AADL: a sublanguage extension to AADL. In: Proc. Internetware'13. ACM (2013)
43. Ren, W., Beard, R.W.: Distributed consensus in multi-vehicle cooperative control. Springer (2008)
44. Rocha, C., Meseguer, J., Muñoz, C.: Rewriting modulo SMT and open system analysis. J. Log. Algebr. Methods Program 86(1), 269–297 (2017)
45. Rocha, C., Meseguer, J., Muñoz, C.A.: Rewriting modulo SMT and open system analysis. In: Proc. WRLA. Lecture Notes in Computer Science, vol. 8663, pp. 247–262. Springer (2014)
46. Rushby, J.: Systematic formal verification for fault-tolerant time-triggered algorithms. IEEE Transactions on Software Engineering 25(5), 651–660 (1999)
47. Sha, L., Al-Nayeem, A., Sun, M., Meseguer, J., Ölveczky, P.C.: PALS: Physically asynchronous logically synchronous systems. Tech. rep., Department of Computer Science, University of Illinois at Urbana-Champaign (2009), `http://hdl.handle.net/2142/11897`
48. Tripakis, S., Pinello, C., Benveniste, A., Sangiovanni-Vincent, A., Caspi, P., Di Natale, M.: Implementing synchronous models on loosely time triggered architectures. IEEE Transactions on Computers 57(10), 1300–1314 (2008)