HybridSynchAADL: Modeling and Formal Analysis of Virtually Synchronous CPSs in AADL

Jaehun Lee[1], Sharon Kim[1], Kyungmin Bae[1], Peter Csaba Ölveczky[2], and Minseok Kang[1]

[1] Pohang University of Science and Technology
[2] University of Oslo

Abstract. We present the HybridSynchAADL modeling language and formal analysis tool for virtually synchronous cyber-physical systems with complex control programs, continuous behaviors, and bounded clock skews, network delays, and execution times. We leverage the Hybrid PALS equivalence, so that it is sufficient to model and verify the simpler underlying synchronous designs. We define the HybridSynchAADL language as a sublanguage of the avionics modeling standard AADL for modeling such designs in AADL. HybridSynchAADL models are given a formal semantics and analyzed using Maude 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 HybridSynchAADL on a number of applications, including 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 distributed control systems such as the steam-boiler benchmark [2]. 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 reduces the difficulty of modeling and verifying distributed *real-time* systems when the infrastructure provides bounds on network delays, clock skews, and execution times [5, 38, 39]: 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 occurs, as long as it happens in a certain time interval. However, many virtually synchronous CPSs are networks of *hybrid* systems with continuous behaviors, where we cannot abstract from the time when a controller interacts with its continuous environment. *Hybrid PALS* [10] extends PALS to virtually synchronous distributed hybrid systems, taking into account sensing and actuating times that depend on imprecise local clocks. Although synchronous Hybrid PALS models can be encoded as SMT problems [10], it is difficult to encode complex virtually synchronous CPSs—with advanced control programs, data types, hierarchical structure, etc.—in SMT.

This paper therefore defines the HybridSynchAADL language for conveniently modeling virtually synchronous distributed hybrid systems using the avionics modeling standard AADL [27] (Section 3). Providing a formal semantics to such models—with control programs written in AADL's expressive Behavior Annex, having to cover all possible continuous behaviors based on imprecise clocks—is challenging. We use Maude [24] combined with SMT solving [14, 45] to *symbolically* encode all possible continuous behaviors with all possible sensing and actuating times depending on imprecise clocks, and provide in Section 6 a Maude-with-SMT semantics for HybridSynchAADL.

Section 4 presents the HybridSynchAADL tool supporting the modeling and verification of HybridSynchAADL models inside the OSATE tool environment for AADL. Our tool provides an intuitive property specification language for specifying bounded reachability properties of HybridSynchAADL models. HybridSynchAADL invokes Maude combined with the SMT solver Yices2 [26] to provide symbolic reachability analysis and randomized simulation for verifying such bounded reachability properties of HybridSynchAADL models with polynomial continuous dynamics. To make this analysis efficient, our tool implements several optimization techniques, including symbolic state merging, modular symbolic encoding, and multithreaded portfolio analysis.

We use our tool to model and verify a number of CPS applications, including networks of thermostats and of water tanks, as well as distributed drones that communicate to reach the "same" location, or fly in formation, without crashing into each other (Section 5). We evaluate, and demonstrate, the effectiveness of the HybridSynchAADL tool by addressing the following questions (Section 7):

1. How effective is our tool compared to state-of-the-art CPS analysis tools?
2. How effective is our method and tool in finding bugs?
3. How effective is our new state merging technique?
4. How effective is Hybrid PALS in verifying virtually synchronous CPSs with continuous behaviors?

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

1. Hybrid PALS, which reduces the formal analysis of a virtually synchronous CPS to that of its synchronous design—albeit having to consider clock skews and sensing and actuation times; and
2. the integration of Maude with SMT solving. Maude is suitable to analyze complex control programs, whereas SMT solving allows us to symbolically analyze continuous behaviors.

HybridSynchAADL combines these techniques to provide an expressive and user-friendly formal modeling and analysis framework for virtually synchronous CPSs that is optimized to make formal analysis feasible. The HybridSynchAADL tool is available at `https://hybridsynchaadl.github.io`.

The rest of this paper is organized as follows. Section 2 explains some background on Hybrid PALS, AADL, and Maude with SMT. Section 3 presents the HybridSynchAADL language. Section 4 presents the HybridSynchAADL tool. Section 5 presents case studies on virtually synchronous CPSs for controlling distributed drones. Section 6 presents its semantics in Maude with SMT. Section 7 shows the experimental results. Section 8 discusses the related work. Finally, Section 9 presents some concluding remarks.

2 Preliminaries

2.1 PALS and Hybrid PALS

When the infrastructure guarantees bounds on clock skews, network delays, and execution times, the PALS pattern [5,38] reduces the problems of designing and verifying virtually synchronous distributed real-time systems to the much easier problems of designing and verifying their underlying synchronous designs. Formally, given a synchronous system design SD, bounds Γ on clock skews, network delays, and execution times, and a period p of each round, the PALS transformation gives us the asynchronous distributed real-time system $PALS(SD, \Gamma, p)$, which is stuttering bisimilar to SD.

The synchronous design SD is formalized as the synchronous composition of an *ensemble* of state machines with input and output ports [38]. 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. All machines performs their transitions simultaneously, and outputs to other machines become inputs at the *next* iteration. PALS was extended to the multi-rate setting in [7], but, for simplicity of exposition, this paper focuses on the single-rate case.

Hybrid PALS [10] extends PALS to virtually synchronous CPSs with physical environments that exhibit continuous behaviors. The *physical environment* E_M of a machine M has real-valued parameters $\vec{x} = (x_1, \ldots, x_l)$. The continuous behaviors of $\vec{x}$ are modeled by a set of ordinary differential equations (ODEs) 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.

The local clock of a machine M can be seen as a function $c_M : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$, where $c_M(t)$ is the value of the local clock at time t, with $\forall t \in \mathbb{R}_{\geq 0},\ |c_M(t) - t| < \epsilon$ for $\epsilon > 0$ the maximal clock skew [38]. In its ith 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*.

2.2 AADL

The *Architecture Analysis & Design Language* (AADL) [27] 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*.[3] 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 AADL's *Behavior Annex* [28]. 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, transitions are 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 a thread is executed. In particular, a *periodic* thread is activated at fixed time intervals.

[3] Hardware components include: *processor* components that schedule and execute threads, *memory* components, *device* components, and *bus* components that interconnect processors, memory, and devices.

2.3 Maude with SMT

Maude [24] 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. A *rewrite theory* [37] is a triple $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory with Σ a signature (declaring sorts, subsorts, and function symbols) and E a set of equations; and 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. A *rewrite* $t \longrightarrow^* t'$ holds if t' is reachable from t using the rewrite rules in $\mathcal{R}$.

A declaration `class C |` $att_1 : s_1, \ldots, att_n : s_n$ declares a class C with attributes $att_1, \ldots, att_n$ of sorts $s_1, \ldots, s_n$. An *object* o of class C is a term

$$\texttt{<}\ o : C\ \texttt{|}\ att_1 : v_1, \ldots, att_n : v_n\ \texttt{>},$$

of sort `Object`, where v_i is the value of the attribute att_i. A `subclass` inherits the attributes and rewrite rules of its superclasses. A *configuration* is a multiset of objects and messages, and has sort `Configuration`, with multiset union denoted by juxtaposition.

In rewriting modulo SMT [14, 45], (possibly infinite) sets of system states can be *symbolically* represented using *constrained terms*. A constrained term is a pair $\phi \parallel t$ of a constraint $\phi(x_1, \ldots, x_n)$ and a term $t(x_1, \ldots, x_n)$ over SMT variables $x_1, \ldots, x_n$, representing all instances of t such that ϕ holds: i.e., given the underlying SMT theory $\mathcal{T}$, we have:

$$[\![\phi \parallel t]\!] = \{t(a_1, \ldots, a_n) \mid \mathcal{T} \models \phi(a_1, \ldots, a_n)\}.$$

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 provides SMT solving and *symbolic reachability analysis* for constrained terms, using connections to Yices2 [26] and CVC4 [19].

3 The HybridSynchAADL Modeling Language

This section presents the HybridSynchAADL language for modeling virtually synchronous CPSs in AADL. HybridSynchAADL can specify environments with continuous dynamics, synchronous designs of distributed controllers, and nontrivial interactions between controllers and environments with respect to imprecise local clocks and sampling and actuation times.

The HybridSynchAADL language is a subset of AADL extended with the following property set `Hybrid_SynchAADL`. We use a subset of AADL without changing the meaning of AADL constructs or adding new a annex, so that AADL modelers easily can 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 [8, 11] of AADL.[4] 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 1 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).

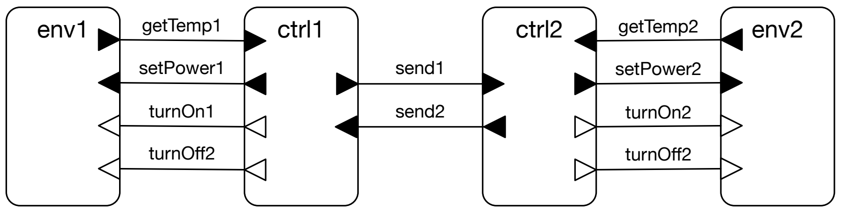


Fig. 1. A networked thermostat system.

[4] Just like Synchronous AADL can be extended to *multi-rate* controllers [11], 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.

3.1 Environment Components

An *environment component* models real-valued state variables that continuously change over time. State variables are specified using data subcomponents of type `Base_Types::Float`. Each environment component declares the property `Hybrid_SynchAADL::isEnvironment => true`.

An environment component can have different *modes* to specify different continuous behaviors (trajectories). A controller command may change the mode of the environment or the value of a variable. The continuous dynamics in each mode is specified using either ODEs or continuous real functions as follows:

```
Hybrid_SynchAADL::ContinuousDynamics =>
     "$dynamics_1$" in modes ($mode_1$), ..., "$dynamics_n$" in modes ($mode_n$);
```

In HybridSynchAADL, a set of ODEs over n variables $x_1, \ldots, x_n$, say, $\frac{\mathrm{d}x_i}{\mathrm{d}t} = e_i(x_1, \ldots, x_n)$ for $i = 1, \ldots, n$, is written as a semicolon-separated string:

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

If a closed-form solution of ODEs is 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 $x_1, \ldots, x_n$:

```
$x_1$($t$) = $e_1(t, x_1(0), \ldots, x_n(0))$; ... ; $x_n$($t$) = $e_n(t, x_1(0), \ldots, x_n(0))$;
```

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.

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 or trigger mode (and hence trajectory) changes. 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 of an environment component declares that the value of the data subcomponent is “sampled” by a controller through the output port of the environment component. A connection from an environment’s input port to a data subcomponent inside the environment declares that a controller command arrived at the input port and updates the value of the data subcomponent. 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 2. Figure 3 gives an environment component `RoomEnv` for our networked thermostat system. Figure 2 shows its architecture. It has data output port `temp`, data input port `power`, and event input ports `on_control` and `off_control`. 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, using continuous functions over time parameter t, where heaterOff is the initial mode. Because p is a constant, p's dynamics d/dt(p) = 0 is omitted. The value 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.

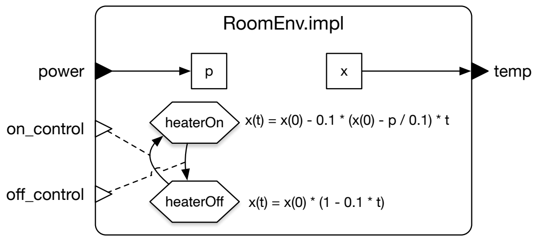

RoomEnv.impl
power
p
x
temp
on_control
off_control
heaterOn
heaterOff
x(t) = x(0) - 0.1 * (x(0) - p / 0.1) * t
x(t) = x(0) * (1 - 0.1 * t)

Fig. 2. 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. 3. A RoomEnv component.

3.2 Controller Components

Discrete controllers are usual AADL software components in the Synchronous AADL subset [9, 12]. A controller component is specified using the behavioral and structural subset of AADL: hierarchical system, process, thread components, data subcomponents; ports and connections; and thread behaviors defined by the Behavior Annex [28]. The hardware and scheduling features of AADL, which are not relevant to synchronous *designs*, are not considered in HybridSynchAADL.

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. A controller receives the state of the environment at some *sampling time*, and sends a controller command to the environment at some *actuation time*. Sampling and actuation take place according to the local clock of the controller, which may differ from the "ideal clock" by up to the maximal clock skew. These time values are declared by the component properties:

```
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 [10].

Initial Values and Parameters. In AADL, *data* subcomponents represent data values, such as 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");
```

Sometimes initial values can be *parameters*, instead of concrete values. E.g., you can check whether a certain property holds from initial values satisfying a certain constraint for those parameters (see Section 4). In HybridSynchAADL, such unknown parameters can be declared using the following AADL property:

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

Example 3. Consider again our networked thermostat system. Figure 4 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. It 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. 5). The implementation 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.

```
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. 4. A simple thermostat controller.

3.3 Communication

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

Connections Between Discrete Controllers. All (non-actuator) output values of controller components generated in an iteration are available to the receiving *controller* components at the beginning of the *next* iteration. As explained in [9,12], *delayed connections between data ports* meet this requirement. Therefore, two controller components can be connected only by data ports with delayed connections, declared by the connection property:

```
Timing => Delayed;
```

Connections Between Controllers and Environments. In HybridSynchAADL, interactions between a controller and an environment occur *instantaneously* at the sampling and actuating times of the controller.[5] Because an environment does not "actively" send data for sampling, every output port of an environment must be a *data* port, whereas its input ports could be of any kind.

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.

Example 4. Figure 5 shows an implementation of a top-level system component `TwoThermostats` of our networked thermostat system, depicted in Figure 1. 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. 4, and the environment component `RoomEnv.impl` is given in Fig. 3. Each discrete controller `ctrl`i, for $i = 1, 2$, is connected to its environment component `env`i using four connections `turnOn`i, `turnOff`i, `setPower`i, and `getTemp`i. The controllers `ctrl1` and `ctrl2` are connected with each other using delayed data connections `send1` and `send2`.

4 The HybridSynchAADL Tool

This section introduces the HybridSynchAADL tool supporting the modeling and formal analysis of HybridSynchAADL models. The tool is an OSATE plugin which: (i) provides an intuitive language to specify properties of models, (ii) synthesizes a rewriting logic model from a HybridSynchAADL model, and (iii) performs various formal analyses using Maude and an SMT solver.

[5] 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. 5. A top level component with two thermostat controllers.

4.1 Property Specification Language

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

Atomic propositions are given by Boolean expressions in the AADL Behavior Annex syntax. Each identifier is fully qualified with its component path in the AADL syntax. A *scoped expression* of the form *path* | *exp* denotes that each component path of each identifier in the expression *exp* begins with *path*. A "named" atomic proposition can be declared 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 φ_{init}, an invariant condition φ_{inv}, and a time bound τ_{bound}, where φ_{init} and φ_{inv} are in propositional logic. Intuitively, the invariant property holds if for every (initial) state satisfying the initial condition φ_{init}, all states reachable within the time bound τ_{bound} satisfy the invariant condition φ_{inv}.

invariant [*name*]: φ_{init} ==> φ_{inv} **in time** τ_{bound}

A *reachability property* (the dual of an invariant) holds if a state satisfying φ_{goal} is reachable from some state satisfying the initial condition φ_{init} within the time bound τ_{bound}. It is worth noting that a reachability property can be expressed as an invariant property by negating the goal condition.

```
reachability [name]: φinit ==> φgoal in time τbound
```

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.

Example 5. 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
```

4.2 Tool Interface

Figure 6 depicts the architecture of the HybridSynchAADL tool. The tool first statically checks whether a given model is a valid model that satisfies the syntactic constraints of HybridSynchAADL in Section 3. It uses OSATE's code generation capability to synthesize the corresponding Maude model from the validated model. Finally, our tool invokes Maude and an SMT solver to check whether the model satisfies given invariant and reachability requirements with respect to the formal semantics of HybridSynchAADL. The `Result` view in OSATE displays the results of the analysis in a readable format.

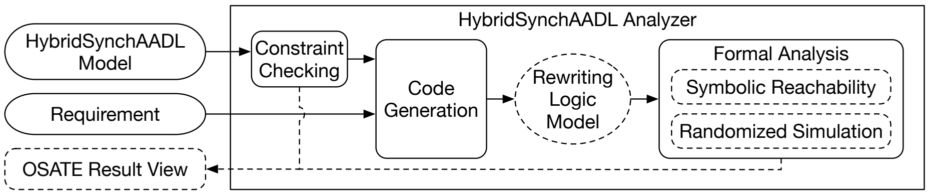
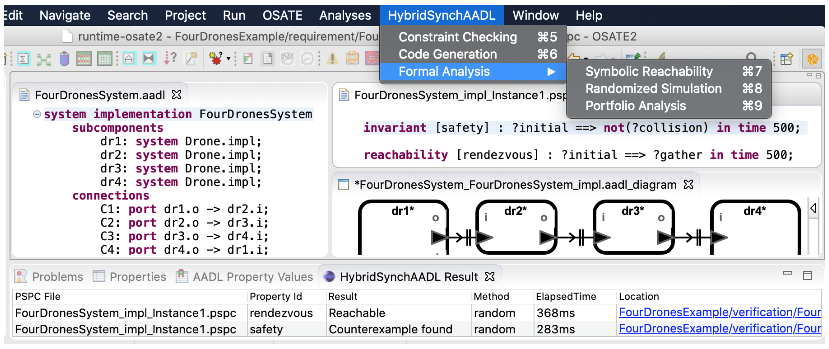


Fig. 6. The architecture of the HybridSynchAADL tool.



```
system implementation FourDronesSystem
  subcomponents
    dr1: system Drone.impl;
    dr2: system Drone.impl;
    dr3: system Drone.impl;
    dr4: system Drone.impl;
  connections
    C1: port dr1.o -> dr2.i;
    C2: port dr2.o -> dr3.i;
    C3: port dr3.o -> dr4.i;
    C4: port dr4.o -> dr1.i;
```

```
invariant [safety] : ?initial ==> not(?collision) in time 500;

reachability [rendezvous] : ?initial ==> ?gather in time 500;
```

PSPC File	Property Id	Result	Method	ElapsedTime	Location
FourDronesSystem_impl_Instance1.pspc	rendezvous	Reachable	random	368ms	FourDronesExample/verification/Four
FourDronesSystem_impl_Instance1.pspc	safety	Counterexample found	random	283ms	FourDronesExample/verification/Four

Fig. 7. Interface of the HybridSynchAADL tool.

By syntactically validating a HybridSynchAADL model, we ensure that the model satisfies all the syntactic constraints of HybridSynchAADL, and thus the corresponding Maude model is executable. 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 `Hybrid_SynchAADL::ContinuousDynamics`. The tool checks other "trivial" constraints that are assumed in the semantics of HybridSynchAADL; e.g., all input ports are connected to some output ports.

HybridSynchAADL provides two formal analysis methods. *Symbolic reachability analysis* can verify that all possible behaviors—imposed by sensing and actuation times based on imprecise clocks—satisfy a given requirement;[6] if not, a counterexample is generated. *Randomized simulation* repeatedly executes the model (using Maude) until a counterexample is found, by randomly choosing concrete sampling and actuating times, nondeterministic transitions, etc.

Our tool also provides *portfolio analysis* that combines symbolic reachability analysis and randomized simulation. HybridSynchAADL runs both methods in parallel using multithreading, and displays the result of the analysis that terminates first. Symbolic reachability analysis can guarantee the absence of a counterexample, whereas randomized simulation is effective for finding "obvious" bugs. Portfolio analysis combines the advantages of both approaches.

[6] Symbolic analysis only supports (nonlinear) polynomial continuous dynamics, since the underlying SMT solver, Yices2, does not support general classes of ODEs.

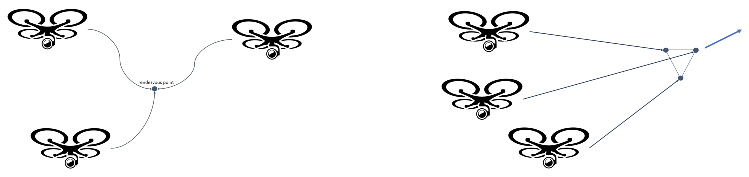


Fig. 8. Rendezvous and formation control of distributed drones

Figure 7 shows the interface of our tool that is fully integrated into OSATE. The left editor shows the code of `FourDronesSystem` in Section 5, the bottom right editor shows its graphical representation, and the top right editor shows two properties in the property specification language. The HybridSynchAADL menu contains three items for constraint checking, code generation, and formal analysis. The `Portfolio Analysis` item has been already clicked, and the `Result` view at the bottom displays the analysis results in a readable format.

5 Case Study: Collaborating Autonomous Drones

This section shows how virtually synchronous CPSs for controlling distributed drones can be modeled and analyzed using HybridSynchAADL. Controllers of multiple drones collaborate to achieve common maneuver goals, such as *rendezvous* or *formation control* depicted in Fig. 8. The controllers are physically distributed, since a controller is included in the hardware of each drone. Our models take into account continuous dynamics, asynchronous communication, network delays, clock skews, execution times, sampling and actuating times, etc.

5.1 Distributed Consensus Algorithms

We use distributed consensus algorithms [44] to synchronize the drone movements. Each drone has an *information state* that represents the drone's local view of the coordination task, such as the rendezvous position, the center of a formation, etc. There is no centralized controller with a "global" view. Each drone periodically exchanges the information state with neighboring drones, and eventually the information states of all drones should converge to a common value.

Consider 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 [44] 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 [44], 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 α and γ 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 [44]:

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

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

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

5.2 The HybridSynchAADL Model

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 HybridSynch-AADL model requires much more text to specify connections. Nevertheless, we

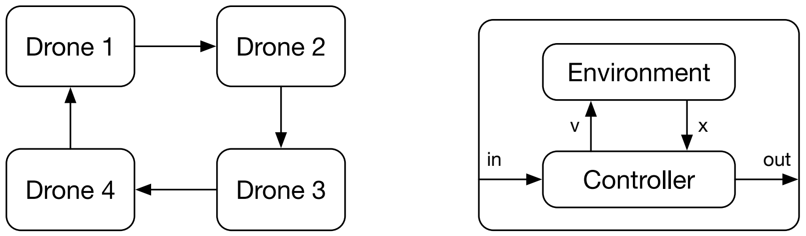


Fig. 9. The architecture of four drones (left), and a drone component (right).

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://hybridsynchaadl.github.io`.

Figure 9 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 $\vec{x}$ and $\vec{y}$ 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 $\vec{x}$ and $\vec{y}$ 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. The top-level component includes four `Drone` components (Fig. 10). Each drone sends its position through its output port `oX` and `oY`, and receives the position of the other drone through its input port `iX` and `iY`. The component is declared to be synchronous with period 100 ms. Also, to meet the constraints of HybridSynchAADL, 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. component in Fig. 11 has an input port `iX` and `iY` and an output port `oX` and `oY`. Its implementation `Drone.impl` contains a controller `ctr` and an environment `env`. The controller `ctr` obtains the current position from `env` via input port `currX` and `currY`, and sends a new velocity to `env` via output port `velX` and `velY`, according to its sampling and actuating times declared in the implementation `FourDronesSystem.impl`.

```
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. 10. The top-level system component `FourDronesSystem`.

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

system implementation Drone.impl
  subcomponents
    ctrl: system DroneControl::DroneControl.impl;
    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.impl;
```

Fig. 11. A drone component in HybridSynchAADL.

```
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. 12. An environment component in HybridSynchAADL.

Environment. Figure 12 shows an `Environment` component that specifies the physical model of the drone. It has two input ports `velX` and `velY` and two output ports `currX` and `currY`. Data subcomponents `x`, `y`, `vx` and `vy` represent the position and velocity of the drone. The value of `x` and `y` are sent to the controller through the output port `currX` and `currY`. When a controller sends an actuation command to ports `velX` and `velY`, the value of `vX` and `vY` are updated by the value of `velX` and `velY`, or the mode changes according to the mode transitions. 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 13 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 14, `ctrlProc` again includes the thread component `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`).

```
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. 13. 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. 14. 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. 15. A controller thread in HybridSynchAADL

Figure 15 shows a thread component for a drone controller. When the thread dispatches, the transition from `init` to `exec` is taken. 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, Finally, the current position is assigned to the output port `oX` and `oY`.

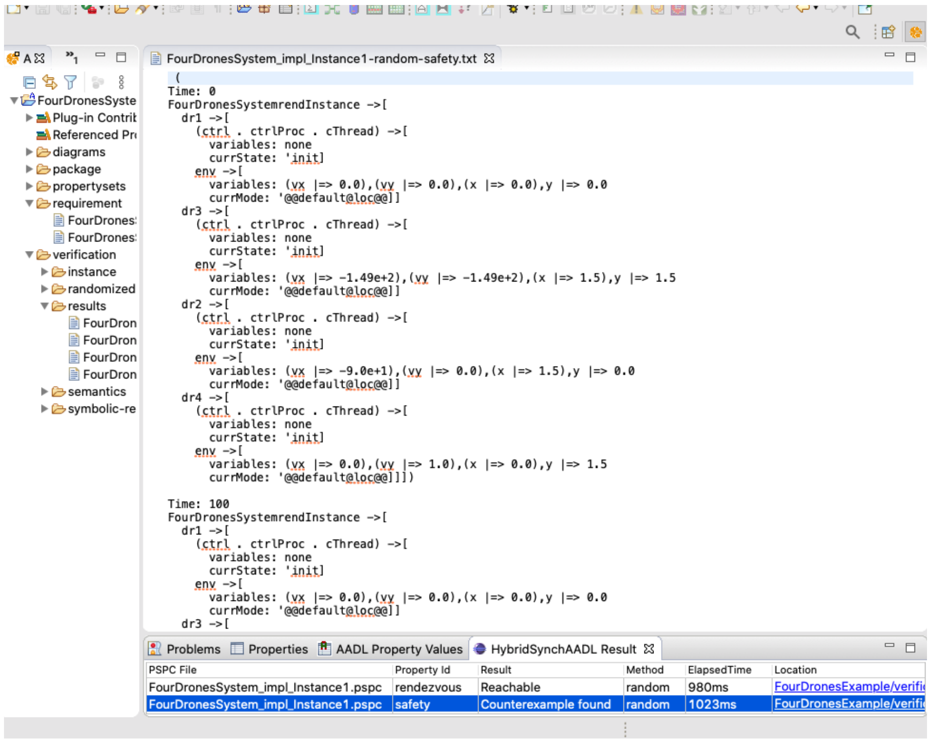
5.3 Formal Analysis Using the HybridSynchAADL Analyzer

We consider two properties of the drone rendezvous model: (i) drones do not collide (`safety`), and (ii) all drones could eventually gather together (`rendezvous`). Because the drone model is a distributed hybrid system, these properties depend on the continuous behavior *perturbed by* sensing and actuating times based on imprecise local clocks. We analyze them up to bound 500 ms using HybridSynchAADL portfolio analysis.

```
invariant [safety]: ?initial ==> not ?collision in time 500;

reachability [rendezvous]: ?initial ==> ?gather in time 500;
```

We define three atomic propositions `collision`, `gather`, and `initial` for four drones `dr1`, `dr2`, `dr3`, and `dr4`. Two drones collide if the distance between them is less than 0.05. All nodes have gathered if the distance between each pair of nodes is less than 1. The initial values of `x`, `y`, `vx`, and `vy` are declared to be parametric in Fig. 12 and constrained by the condition `initial`. There are infinitely many initial states satisfying the proposition `initial`.

```
proposition [collision]:
 (abs(dr1.env.x - dr2.env.x) < 0.05 and abs(dr1.env.y - dr2.env.y) < 0.05) or
 (abs(dr1.env.x - dr3.env.x) < 0.05 and abs(dr1.env.y - dr3.env.y) < 0.05) or
 (abs(dr1.env.x - dr4.env.x) < 0.05 and abs(dr1.env.y - dr4.env.y) < 0.05) or
 (abs(dr2.env.x - dr3.env.x) < 0.05 and abs(dr2.env.y - dr3.env.y) < 0.05) or
 (abs(dr2.env.x - dr4.env.x) < 0.05 and abs(dr2.env.y - dr4.env.y) < 0.05) or
 (abs(dr3.env.x - dr4.env.x) < 0.05 and abs(dr3.env.y - dr4.env.y) < 0.05);

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
 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]:
 abs(dr1.env.x - 0.0) < 0.1 and abs(dr1.env.y - 0.0) < 0.1 and
 abs(dr2.env.x - 1.5) < 0.1 and abs(dr2.env.y - 0.0) < 0.1 and
 abs(dr3.env.x - 1.5) < 0.1 and abs(dr3.env.y - 1.5) < 0.1 and
 abs(dr4.env.x - 0.0) < 0.1 and abs(dr4.env.y - 1.5) < 0.1;
```



```
 (
Time: 0
FourDronesSystemrendInstance ->[
  dr1 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: none
      currState: 'init]
    env ->[
      variables: (vx |=> 0.0),(vy |=> 0.0),(x |=> 0.0),y |=> 0.0
      currMode: '@@default@loc@@]]
  dr3 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: none
      currState: 'init]
    env ->[
      variables: (vx |=> -1.49e+2),(vy |=> -1.49e+2),(x |=> 1.5),y |=> 1.5
      currMode: '@@default@loc@@]]
  dr2 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: none
      currState: 'init]
    env ->[
      variables: (vx |=> -9.0e+1),(vy |=> 0.0),(x |=> 1.5),y |=> 0.0
      currMode: '@@default@loc@@]]
  dr4 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: none
      currState: 'init]
    env ->[
      variables: (vx |=> 0.0),(vy |=> 1.0),(x |=> 0.0),y |=> 1.5
      currMode: '@@default@loc@@]]])

Time: 100
FourDronesSystemrendInstance ->[
  dr1 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: none
      currState: 'init]
    env ->[
      variables: (vx |=> 0.0),(vy |=> 0.0),(x |=> 0.0),y |=> 0.0
      currMode: '@@default@loc@@]]
  dr3 ->[
```

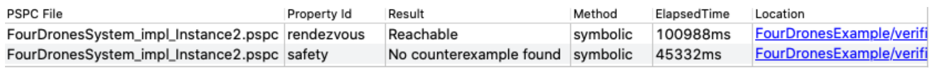
PSPC File	Property Id	Result	Method	ElapsedTime	Location
FourDronesSystem_impl_Instance1.pspc	rendezvous	Reachable	random	980ms	FourDronesExample/verifi
FourDronesSystem_impl_Instance1.pspc	safety	Counterexample found	random	1023ms	FourDronesExample/verifi

Fig. 16. Results for the safety and rendezvous properties.

The result of the analysis is shown in Figure 16. The `HybridSynchAADL Result` view shows that a counterexample is found for `safety` and there is a witness for `rendezvous`. Both results are obtained by randomized simulation. The editor shows a concrete counterexample of `safety`, given by a sequence of states for synchronous steps. For example, the drone *dr3* has the velocity $(-150, -150)$ at time 0 (i.e, in the initial state). Because `initial` has no velocity constraint, the drones can have an unrealistic speed in the initial state.

We therefore modify `safety` and `rendezvous` below by adding such a velocity constraint `velconst` to the initial condition. As shown in Fig. 17, There is now no counterexample to `safety` up to bound 500, where both results are obtained by symbolic reachability analysis.

```
invariant [safety]: ?initial and ?velconst ==> not ?collision in time 500;
reachability [rendezvous]: ?initial and ?velconst ==> ?gather in time 500;

proposition [velconst]:
      abs(dr1.env.vx) <= 0.01 and abs(dr1.env.vy) <= 0.01 and
      abs(dr2.env.vx) <= 0.01 and abs(dr2.env.vy) <= 0.01 and
      abs(dr3.env.vx) <= 0.01 and abs(dr3.env.vy) <= 0.01 and
      abs(dr4.env.vx) <= 0.01 and abs(dr4.env.vy) <= 0.01;
```



PSPC File	Property Id	Result	Method	ElapsedTime	Location
FourDronesSystem_impl_Instance2.pspc	rendezvous	Reachable	symbolic	100988ms	FourDronesExample/verifi
FourDronesSystem_impl_Instance2.pspc	safety	No counterexample found	symbolic	45332ms	FourDronesExample/verifi

Fig. 17. Results for the modified properties.

Although the time bound in the example is small, our verification involves infinitely many (continuous) behaviors, for all possible local clocks, sampling and actuation times, initial states, etc. We therefore precisely verify that the "local" behaviors, *perturbed by* clock skews and sampling/actuation times, are all correct, which is an important problem for virtually synchronous CPSs.

6 Executable Formal Semantics of HybridSynchAADL

This section presents the Maude-with-SMT semantics of HybridSynchAADL that implements our tool's analysis commands. Since the HybridSynchAADL modeling language extends Synchronous AADL, the semantics of discrete controllers extends that of Synchronous AADL [9, 12], while the semantics of continuous environments (and interactions with them) and nontrivial interactions between controllers and environments, is new. Specifically, an SMT encoding for synchronous Hybrid PALS models in [10] is not directly applicable, since HybridSynchAADL models are hierarchical. The full semantics of HybridSynchAADL is available in `https://hybridsynchaadl.github.io`.

6.1 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 $\phi(x_1, \ldots, x_n) \parallel \mathit{obj}(x_1, \ldots, x_n)$ to symbolically represent an infinite number of objects, where an SMT constraint $\phi(x_1, \ldots, x_n)$ and an object "pattern" $\mathit{obj}(x_1, \ldots, x_n)$ over SMT variables $x_1, \ldots, x_n$ represent (possibly infinite) sets of objects. Such object patterns may be hierarchical; for example, $\mathit{obj}(x_1, \ldots, x_n)$ may include other object patterns over $x_1, \ldots, x_n$ in its attributes. Similarly, *constrained configurations* have the form $\phi \parallel \mathit{conf}$ with *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 $\phi'_1 \parallel t'_1, \ldots, \phi'_n \parallel t'_n$ 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, φ 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).

6.2 SMT Expressions

SMT values are terms of sort `Value`. There are three kinds of SMT values. Boolean values are either [*true*] or [*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 $(1 + x_1 > 2 * x_2\ or\ b_1) = (b_2\ and\ true)$.[7]

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

6.3 Representing HybridSynchAADL Models

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

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.

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

[7] [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.

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 `-[`*guard*`]->` s' `{`*actions*`}`, 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 `[`*continuous_dynacmis*`]`where m is a mode and *continuous_dynacmis* is either ODEs or continuous functions as explained in Section 3.

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, 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* [8,11].

```
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.[8] To symbolically encode the immediate communication, the attribute `envCache` contains a data content in the previous iteration (see Section 6.7).

```
class EnvPort | target : ComponentRef,
                envCache : DataContent .
subclass EnvPort < 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. A connection from an output port p_1 in c_1 to an input port p_2 in c_2 is written as 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 6. An instance of the `TwoThermostats.impl` component in Fig. 5 is represented by the following object, where property values are enclosed by `{{...}}`.

[8] 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.

```
< 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}} >
```

Example 7. An instance of ThermostatThread.impl in Fig. 4 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 >
>
```

Example 8. An instance of RoomEnv.impl in Fig. 3 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 6.7). 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) >
```

6.4 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.

6.5 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 [11], except that data contents are pairs of symbolic values.

6.6 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` *nondeterministically* 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.[9]

[9] `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.

```
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') .
```

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.[10] 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)
```

[10] 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.

```
 = 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.

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 Hybrid-SynchAADL.

```
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) .
```

6.7 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 18 shows a timeline of their

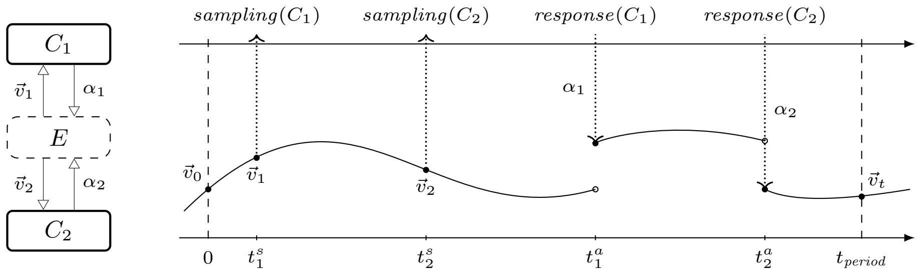


Fig. 18. Interactions between an environment E and two controllers C_1 and C_2

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 α_n at time t_n^a (and may also change its continuous dynamics by α_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. 18) 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 [10], a controller and an environment are combined into a single *environment-restricted* 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 6.5.

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 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. 19, we use an extra attribute `envCache` that contains the symbolic variable

# Iteration		1	2	3	4	...
Controller		d_1	d_2	d_3	d_4	...
Input port p	`content`	$\cdot$	d_1	d_2	d_3	...
	`envCache`	$\cdot$	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. 19. The behavior of `readEnvFeature`.

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 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 $\bot$ 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

# 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. 20. The behavior of `writeEnvFeature`.

round, to implement this behavior, as described in Fig. 20, 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].
```

Executing Environments. Figure 21 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

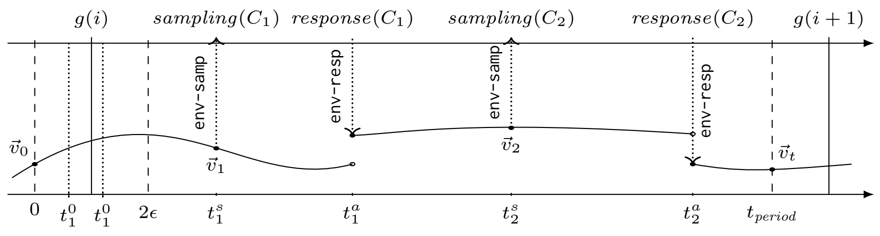


Fig. 21. The behavior of an environment interacting with two controllers.

$[g(i), g(i+1)]$ 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. 21, $t_n^s - t_n^0$ 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;[11] a function of the form

[11] 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

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 .
```

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 2ϵ, 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)) .
```

a specialized SMT solver, such as dReal [30]. Because dReal has not been integrated with Maude, the current version of the semantics only supports continuous functions.

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] .
```

6.8 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 6.7 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 $u_1, \ldots, u_n$,

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 $\widehat{post}$;
```

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., $[\![abs(t)]\!] = [\![true \parallel t]\!]$) [?].

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 ρ 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.

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), $u \in [\![(\phi_1 \vee \rho\phi_2) \parallel t_1]\!]$ iff there is a substitution θ 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 ρ 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 [34], 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, ϕ_1, ϕ_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 6.6 and 6.7. 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 .
```

6.9 Property Specification Language 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 6.6 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 φ_{init} `==>` φ_{goal} `in time` τ_{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 τ_{bound} by the period of the model, and `initState` denotes the term representation of the entire model:

```
search [N_bound] {([[φ_init]] initState) || initState}
           =>* {COND || OBJ}
      such that check-sat(COND and finalConst(OBJ) and ([[φ_goal]] OBJ)) .
```

Similarly, an invariant property of the form φ_{init} `==>` φ_{inv} `in time` τ_{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] {([[φ_init]] initState) || initState}
           =>* {COND || OBJ}
      such that check-sat(COND and finalConst(OBJ) and not([[φ_inv]] OBJ)) .
```

Example 9. Consider the requirement `inv` in Example 5. 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.

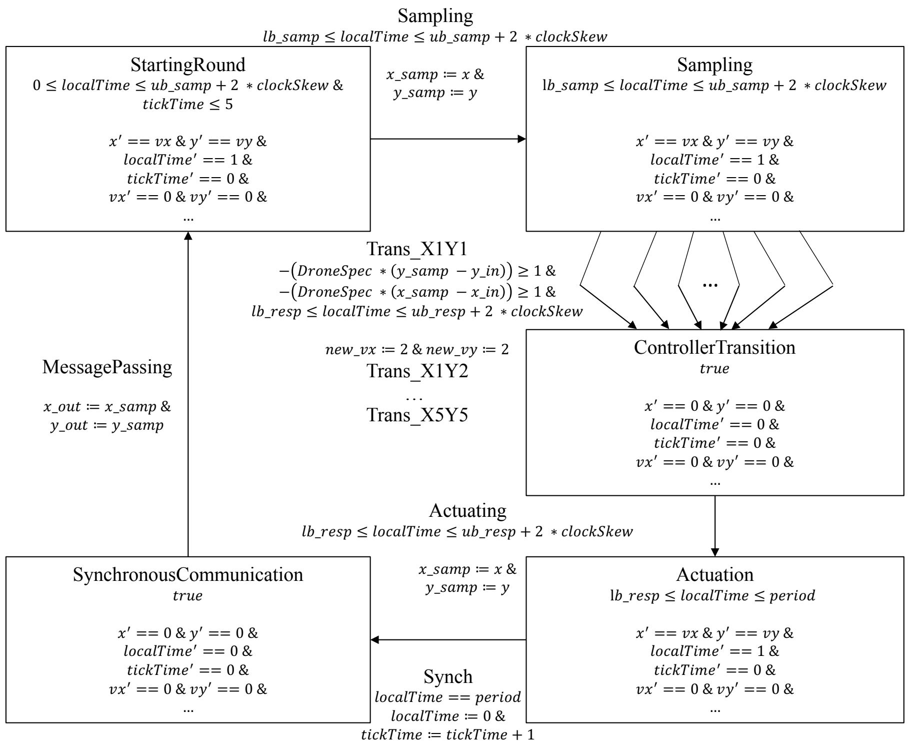
```
search [3] {([[abs(env1 . x - 15) < 3 and abs(env2 . x - 7) < 1}]] initState)
           || initState}
       =>* {COND || OBJ} such that
 check-sat(COND and not([[inRan1 and inRan2 and env1 . x > env2 . x]] OBJ)) .
```

7 Experimental Evaluation

This section evaluates the HybridSynchAADL tool by addressing the following questions: (1) How effective is our tool compared to state-of-the-art CPS analysis tools? (2) How effective is our portfolio analysis method for finding bugs? (3) How effective is our novel state merging technique for symbolic analysis? (4) 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 and water tank systems (adapted from [6, 33, 43]), and rendezvous and formation control of distributed drones. Many variants of these models are considered: 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. The models and the experimental results are available in [1].



Sampling
$lb\_samp \leq localTime \leq ub\_samp + 2 * clockSkew$
StartingRound
$0 \leq localTime \leq ub\_samp + 2 * clockSkew$ &
$tickTime \leq 5$
$x' == vx$ & $y' == vy$ &
$localTime' == 1$ &
$tickTime' == 0$ &
$vx' == 0$ & $vy' == 0$ &
...
$x\_samp := x$ &
$y\_samp := y$
Sampling
$lb\_samp \leq localTime \leq ub\_samp + 2 * clockSkew$
$x' == vx$ & $y' == vy$ &
$localTime' == 1$ &
$tickTime' == 0$ &
$vx' == 0$ & $vy' == 0$ &
...
Trans_X1Y1
$-(DroneSpec * (y\_samp - y\_in)) \geq 1$ &
$-(DroneSpec * (x\_samp - x\_in)) \geq 1$ &
$lb\_resp \leq localTime \leq ub\_resp + 2 * clockSkew$
$new\_vx := 2$ & $new\_vy := 2$
Trans_X1Y2
...
Trans_X5Y5
MessagePassing
$x\_out := x\_samp$ &
$y\_out := y\_samp$
ControllerTransition
$true$
$x' == 0$ & $y' == 0$ &
$localTime' == 0$ &
$tickTime' == 0$ &
$vx' == 0$ & $vy' == 0$ &
...
Actuating
$lb\_resp \leq localTime \leq ub\_resp + 2 * clockSkew$
SynchronousCommunication
$true$
$x' == 0$ & $y' == 0$ &
$localTime' == 0$ &
$tickTime' == 0$ &
$vx' == 0$ & $vy' == 0$ &
...
$x\_samp := x$ &
$y\_samp := y$
Synch
$localTime == period$
$localTime := 0$ &
$tickTime := tickTime + 1$
Actuation
$lb\_resp \leq localTime \leq period$
$x' == vx$ & $y' == vy$ &
$localTime' == 1$ &
$tickTime' == 0$ &
$vx' == 0$ & $vy' == 0$ &
...

Fig. 22. A SpaceEx model

7.1 Comparison with CPS Analysis Tools

We compare HybridSynchAADL's symbolic analysis with three reachability analysis tools for hybrid automata: SpaceEx [29], Hylaa [16], and HyComp [23]. We have developed networks of hybrid automata for distributed thermostats, water tanks, and drone systems, and measure the execution times for analyzing all reachable states of each model up to given bounds using these tools We set a timeout of 20 minutes for this experiment.

In the hybrid automata models, each component is modeled as an automaton with five modes: starting a round, sampling, performing a controller transition, actuation, and synchronous communication. For a fair comparison, the controller behavior of each component is encoded as a single mode transition. In addition, we use flat hybrid automata (obtained using HYST [15]) for Hylaa, which does not support networks of hybrid automata. For example, Figure 22 shows the SpaceEx model for "Drone Rendezvous" with single integrator dynamics.

The experimental results are summarized in Fig. 23 , as execution times (seconds, log scale) over time bounds (ms), with N the number of (thermostat, water tank, or drone) components. The results for double-integrator dynamics

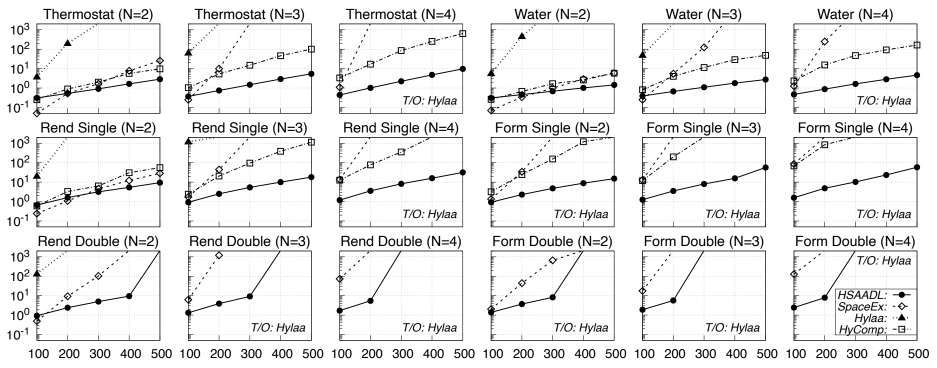


Fig. 23. HybridSynchAADL symbolic analysis vs. SpaceEx, Hylaa, and HyComp.

(where control input is given by acceleration instead of velocity) in the third row do not include HyComp, since it cannot deal with nonlinear polynomial dynamics. We write "T/O" on the graph to indicate that the experiment has timed out for every item. Table 1 also shows the same experimental results in a table, where '-' denotes a timeout.

As shown in Fig. 23, HybridSynchAADL outperforms the other tools in most cases. Consider, e.g., "Formation" with single-integrator dynamics for $N = 4$. HybridSynchAADL needs 59 seconds for bound 500, whereas SpaceEx needs 87 seconds and HyComp needs 68 seconds already for bound 100. Hylaa—currently the best performing tool for affine continuous dynamics [17]—did not perform well. This shows HybridSynchAADL's capability to analyze distributed hybrid systems with a large number (e.g., 5^4) of modes/discrete states.

7.2 Analyzing Invariant Properties

We evaluate the power of HybridSynchAADL for analyzing bounded invariants. We measure the time taken to find counterexamples, using HybridSynch-AADL's three analysis functions, in "faulty" models obtained by modifying the sampling and actuating times. The following table summarizes the time parameters and invariant properties, with ϵ the maximal clock skew. All models have period 100 (milliseconds). The timeout is 20 minutes.

Model	Sampling	Actuation	ϵ	Invariant property
Thermostat	$20 \sim 30$	$60 \sim 70$	5	temperatures are between 20 and 50
Water tank	$30 \sim 40$	$70 \sim 80$	5	water levels are above 30
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

Figure 24 shows the experimental results, with the same x- and y-axis and N as in Fig. 23. Empty shapes indicate that a counterexample is found, and filled circles indicate that symbolic analysis terminates and reports no counterexample. Once a counterexample is found for one bound T by symbolic analysis, the results

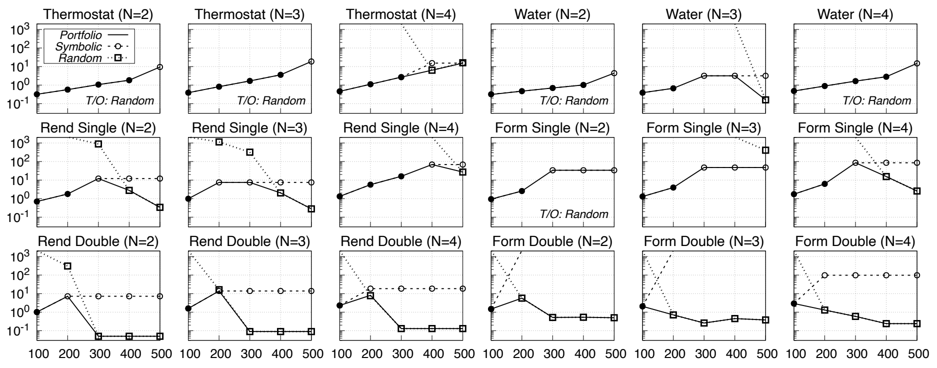


Fig. 24. HybridSynchAADL portfolio analysis.

for larger bounds $T' > T$ are exactly the same, since symbolic analysis uses a breadth-first strategy. The execution time of portfolio analysis is the minimum execution time of symbolic analysis and randomized simulation.

Table. 2 also shows the experiment results in a table. For randomized simulations (`random`), TO means that a counterexample is not found before a timeout. For symbolic analysis (`symbolic`), once a counterexample is found (FO) for one bound T, the results for larger bounds $T' > T$ are omitted and grayed out.

As expected, symbolic analysis is effective to find subtle counterexamples, and randomized simulation is effective for finding obvious bugs. Since the injected faults are caused by excessive sampling and actuating times, violations are easier to find with a larger bound. Consider, e.g., "Formation" with single-integrator dynamics for $N = 4$. Symbolic analysis found a counterexample for bound 300 in 86 seconds, while randomized simulation timed out. For bound 400, randomized simulation found a counterexample in less than 16 seconds. This demonstrates the usefulness of HybridSynchAADL's portfolio analysis.

For double-integrator dynamics, 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. Both single-integrator and double-integrator models involve nonlinear constraints, but double-integrator models give higher-order constraints that Yices2 cannot effectively deal with.

7.3 Effect of State Merging

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.[12]

[12] 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.

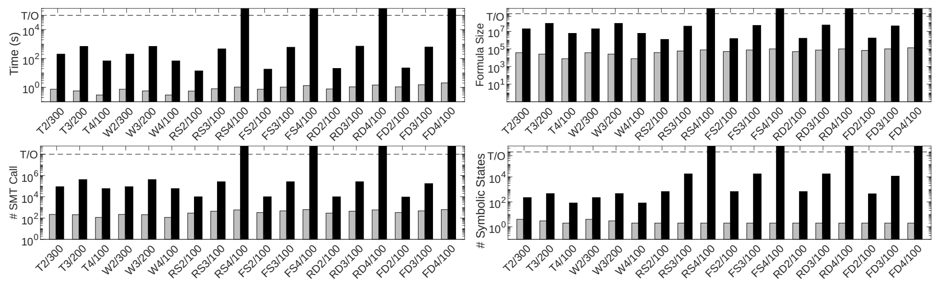


Fig. 25. Symbolic analysis with merging and without merging.

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 $|Const|$ denotes the accumulated size of SMT constraints in thousands. Figure 25 highlights some experimental results. Grey bars denote symbolic analysis with merging, and black bars denote symbolic analysis without merging. A horizontal label has the form *model* $N/bound$, where N denotes the number of components in the model, and `T`, `W`, `RS`, `FS`, `RD`, and `FD` denote thermostats, water tanks, rendezvous single, formation single, rendezvous double, and formation double, respectively.

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 $17,577$ 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 $57,800$ and the number of SMT calls is 438. Without merging, the size of the constraints is about 39 million and the number of SMT calls is more than 260 thousand.

7.4 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

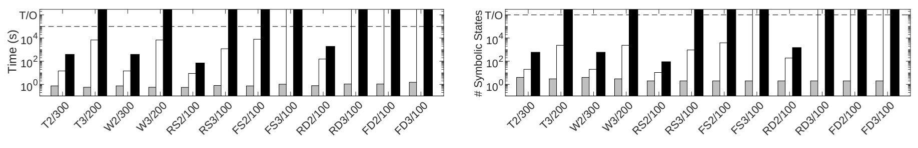


Fig. 26. Comparing Hybrid PALS models and distributed models

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.

Table 4 shows the experimental results, where $|Time\ sample|$ is the number of the predefined values for sampling and actuating times, and the number of concrete states is written in thousands. Figure 26 highlights some experimental results. We can observe that the number of reachable states can be extremely large, even for very simple distributed models with the unrealistic assumptions. For `T3/150` (Thermostat with $N = 3$ and bound 150), the number of reachable states is more than 2.3 million. It took more than 1.8 hours to generate these states, whereas symbolic analysis needed less than 11 second for the same case.

8 Related Work

Our tool can perform model checking of virtually synchronous CPSs with both complex control programs and continuous behaviors (and clock skews, etc.), whereas most formal tools are strong at analyzing either discrete or continuous behaviors. The latter includes reachability analysis tools for hybrid automata [16, 23, 29], which do not deal well with the "discrete complexity" (e.g., complex control programs) of CPSs. HybridSynchAADL can also easily specify and analyze continuous dynamics and imprecise local clocks at the same time.

Almost-Synchronous Systems. Our work is related to a broader body of work on analyzing "almost-synchronous" systems, including quasi-synchrony [20, 21, 32, 35], GALS [31, 41], approximate synchrony [25], time-triggered architectures [46, 47], virtual synchrony [10, 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 [3]. 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 [18], 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 [36]. Both approaches come with some kind of operational semantics and simulation, but with no formal analysis support.

PALS and AADL. Synchronous AADL [9, 13] and its multi-rate extension [12] 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 [10] 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, there is no tool support in [10], and it is difficult to model complex CPSs in SMT. In contrast, this paper provides a tool for modeling 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.

9 Concluding Remarks

We have presented the HybridSynchAADL modeling language and analysis tool for formally modeling and analyzing the *synchronous designs*—and, by the Hybrid PALS equivalence, therefore also of the corresponding *asynchronous distributed system* with bounded clock skews, asynchronous communication, network delays, and execution times—of virtually synchronous networks of hybrid systems with potentially complex control programs in the well-known modeling standard AADL. Our tool provides randomized simulation and symbolic reachability analysis (using Maude combined with SMT), and is fully integrated into the OSATE modeling environment for AADL. We have developed and implemented a number of optimization techniques to improve the performance of the analysis. We demonstrate the efficiency of our tool on a number of distributed hybrid systems, including collaborating drones, and show that in most cases our tool outperforms state-of-the-art hybrid systems reachability analysis tools.

Currently, HybridSynchAADL's symbolic analysis is restricted to systems with (nonlinear) polynomial continuous dynamics, because the underlying SMT solver, Yices2, cannot deal with general classes of ODEs. We should therefore integrate Maude with ODE solvers such as dReal [30] and Flow* [22] 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 [7].

References

1. Supplementary material: HybridSynchAADL technical report, semantics, benchmarks, and the tool, `https://hybridsynchaadl.github.io/tacas21/`
2. Abrial, J., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, LNCS, vol. 1165. Springer (1996)
3. Ahmad, E., Dong, Y., Wang, S., Zhan, N., Zou, L.: Adding formal meanings to AADL with Hybrid Annex. In: Proc. FACS. LNCS, vol. 8997. Springer (2015)
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. Al-Nayeem, A., Sun, M., Qiu, X., Sha, L., Miller, S.P., Cofer, D.D.: A formal architecture pattern for real-time distributed systems. In: Proc. RTSS. IEEE (2009)
6. Bae, K., Gao, S.: Modular SMT-based analysis of nonlinear hybrid systems. In: Proc. FMCAD. pp. 180–187. IEEE (2017)
7. Bae, K., Meseguer, J., Ölveczky, P.C.: Formal patterns for multirate distributed real-time systems. Science of Computer Programming **91**, 3–44 (2014)
8. 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)
9. 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)
10. Bae, K., Ölveczky, P.C., Kong, S., Gao, S., Clarke, E.M.: SMT-based analysis of virtually synchronous distributed hybrid systems. In: Proc. HSCC. ACM (2016)
11. 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)
12. Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of Multirate Synchronous AADL. In: Proc. FM'14. LNCS, vol. 8442. Springer (2014)
13. Bae, K., Ölveczky, P.C., Meseguer, J., Al-Nayeem, A.: The SynchAADL2Maude tool. In: Proc. FASE'12. LNCS, vol. 7212. Springer (2012)
14. Bae, K., Rocha, C.: Symbolic state space reduction with guarded terms for rewriting modulo SMT. Science of Computer Programming **178**, 20–42 (2019)

15. Bak, S., Bogomolov, S., Johnson, T.T.: Hyst: a source transformation and translation tool for hybrid automaton models. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control. pp. 128–133 (2015)
16. Bak, S., Duggirala, P.S.: Hylaa: A tool for computing simulation-equivalent reachability for linear systems. In: Proc. HSCC. pp. 173–178 (2017)
17. Bak, S., Tran, H.D., Johnson, T.T.: Numerical verification of affine systems with up to a billion dimensions. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. pp. 23–32 (2019)
18. 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)
19. 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)
20. Baudart, G., Bourke, T., Pouzet, M.: Soundness of the quasi-synchronous abstraction. In: Proc. FMCAD. pp. 9–16. IEEE (2016)
21. 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. Springer (2001)
22. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Proc. CAV. pp. 258–263. Springer (2013)
23. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: HyComp: An SMT-based model checker for hybrid systems. In: Proc. TACAS. LNCS, vol. 9035. Springer (2015)
24. 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)
25. 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)
26. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV. LNCS, vol. 8559, pp. 737–744. Springer (July 2014)
27. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language. Addison-Wesley (2012)
28. 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)
29. 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. LNCS, vol. 6806. Springer (2011)
30. 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)
31. Girault, A., Ménier, C.: Automatic production of globally asynchronous locally synchronous systems. In: International Workshop on Embedded Software. pp. 266–281. Springer (2002)
32. 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)
33. Henzinger, T.: The theory of hybrid automata. In: Verification of Digital and Hybrid Systems, NATO ASI Series, vol. 170, pp. 265–292. Springer (2000)

34. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. Acm Sigplan Notices **47**(6), 193–204 (2012)
35. 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)
36. 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)
37. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)
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. IEEE (2009)
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. Raisch, J., Klein, E., Meder, C., Itigin, A., O'Young, S.: Approximating automata and discrete control for continuous systems — two examples from process control. In: Hybrid systems V. pp. 279–303. Springer (1999)
44. Ren, W., Beard, R.W.: Distributed consensus in multi-vehicle cooperative control. Springer (2008)
45. Rocha, C., Meseguer, J., Muñoz, C.: Rewriting modulo SMT and open system analysis. Journal of Logical and Algebraic Methods in Programming **86**(1), 269–297 (2017)
46. Rushby, J.: Systematic formal verification for fault-tolerant time-triggered algorithms. IEEE Transactions on Software Engineering **25**(5), 651–660 (1999)
47. 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)

Model	N	Tool	Bound = 100		Bound = 200		Bound = 300		Bound = 400		Bound = 500	
			Time (s)	STD (s)	Time (s)	STD (s)	Time (s)	STD (s)	Time (s)	STD (s)	Time (s)	STD (s)
Thermostat	2	HybridSynchAADL	0.31	0.00	0.53	0.01	0.91	0.01	1.64	0.03	2.86	0.05
		SpaceEx	0.05	0.02	0.53	0.40	1.67	0.15	7.71	0.76	25.62	1.48
		Hylaa	3.78	0.22	198.35	1.41	TO	-	TO	-	TO	-
		Hycomp	0.25	0.01	0.90	0.02	2.02	0.02	5.65	0.05	9.67	0.04
	3	HybridSynchAADL	0.37	0.00	0.75	0.01	1.48	0.03	2.91	0.04	5.41	0.10
		SpaceEx	0.25	0.04	9.92	1.12	TO	-	TO	-	TO	-
		Hylaa	64.14	0.56	TO	-	TO	-	TO	-	TO	-
		Hycomp	1.05	0.02	5.30	0.02	14.97	0.12	45.93	0.21	100.65	0.92
	4	HybridSynchAADL	0.44	0.01	1.02	0.02	2.23	0.03	4.76	0.07	9.68	0.17
		SpaceEx	1.09	0.04	TO	-	TO	-	TO	-	TO	-
		Hylaa	TO	-	TO	-	TO	-	TO	-	TO	-
		Hycomp	3.33	0.02	16.88	0.12	85.05	0.67	250.93	3.04	633.93	5.13
WaterTank	2	HybridSynchAADL	0.31	0.00	0.46	0.00	0.69	0.01	1.02	0.02	1.45	0.02
		SpaceEx	0.07	0.04	0.34	0.19	1.00	0.24	2.90	0.29	5.83	0.43
		Hylaa	5.65	0.23	457.00	2.70	TO	-	TO	-	TO	-
		Hycomp	0.26	0.01	0.67	0.02	1.68	0.03	2.60	0.03	5.84	0.03
	3	HybridSynchAADL	0.39	0.01	0.67	0.02	1.10	0.02	1.79	0.03	2.74	0.04
		SpaceEx	0.25	0.06	5.49	0.34	124.12	4.28	TO	-	TO	-
		Hylaa	48.98	0.32	TO	-	TO	-	TO	-	TO	-
		Hycomp	0.82	0.01	3.98	0.02	11.45	0.07	28.88	0.24	48.09	0.32
	4	HybridSynchAADL	0.47	0.01	0.88	0.02	1.63	0.03	2.82	0.05	4.61	0.07
		SpaceEx	1.25	0.24	247.36	10.49	TO	-	TO	-	TO	-
		Hylaa	TO	-	TO	-	TO	-	TO	-	TO	-
		Hycomp	2.34	0.02	15.47	0.13	47.08	0.52	92.12	1.24	163.45	2.52
Drone Rendezvous (Single)	2	HybridSynchAADL	0.66	0.01	1.61	0.03	3.17	0.07	5.40	0.09	9.40	0.23
		SpaceEx	0.24	0.11	1.08	0.12	4.52	0.20	12.14	0.83	28.37	1.59
		Hylaa	20.43	0.63	TO	-	TO	-	TO	-	TO	-
		Hycomp	0.60	0.05	3.31	0.03	6.36	1.17	30.50	3.79	56.01	14.23
	3	HybridSynchAADL	0.92	0.02	2.51	0.06	5.37	0.09	10.01	0.17	18.13	0.42
		SpaceEx	1.72	0.17	44.27	1.54	TO	-	TO	-	TO	-
		Hylaa	1195.56	7.48	TO	-	TO	-	TO	-	TO	-
		Hycomp	2.39	0.63	20.49	4.50	95.54	9.66	385.57	97.52	1140.35	103.85
	4	HybridSynchAADL	1.19	0.03	3.56	0.08	8.15	0.14	15.96	0.26	31.61	0.49
		SpaceEx	13.70	0.83	TO	-	TO	-	TO	-	TO	-
		Hylaa	TO	-	TO	-	TO	-	TO	-	TO	-
		Hycomp	12.68	2.03	77.08	13.94	356.51	45.57	TO	-	TO	-
Drone Formation (Single)	2	HybridSynchAADL	0.93	0.02	2.28	0.02	4.78	0.10	8.85	0.21	15.16	0.25
		SpaceEx	1.35	0.10	34.03	0.95	TO	-	TO	-	TO	-
		Hylaa	TO	-	TO	-	TO	-	TO	-	TO	-
		Hycomp	3.20	0.25	24.61	4.42	155.42	9.91	1198.55	4.68	TO	-
	3	HybridSynchAADL	1.25	0.02	3.51	0.06	8.05	0.19	15.75	0.28	56.92	0.59
		SpaceEx	13.03	1.11	TO	-	TO	-	TO	-	TO	-
		Hylaa	TO	-	TO	-	TO	-	TO	-	TO	-
		Hycomp	11.89	0.45	199.39	15.98	TO	-	TO	-	TO	-
	4	HybridSynchAADL	1.58	0.01	4.83	0.03	10.26	0.08	23.37	0.27	58.38	0.67
		SpaceEx	86.74	1.53	TO	-	TO	-	TO	-	TO	-
		Hylaa	TO	-	TO	-	TO	-	TO	-	TO	-
		Hycomp	68.13	11.70	843.88	191.83	TO	-	TO	-	TO	-
Drone Rendezvous (Double)	2	HybridSynchAADL	0.93	0.02	2.41	0.05	4.97	0.08	9.45	0.20	TO	-
		SpaceEx	0.49	0.09	9.26	0.57	103.36	2.72	TO	-	TO	-
		Hylaa	132.87	1.27	TO	-	TO	-	TO	-	TO	-
	3	HybridSynchAADL	1.32	0.03	3.89	0.09	9.17	0.19	TO	-	TO	-
		SpaceEx	6.19	0.26	1198.67	3.46	TO	-	TO	-	TO	-
		Hylaa	TO	-	TO	-	TO	-	TO	-	TO	-
	4	HybridSynchAADL	1.71	0.03	5.43	0.14	TO	-	TO	-	TO	-
		SpaceEx	73.35	1.82	TO	-	TO	-	TO	-	TO	-
		Hylaa	TO	-	TO	-	TO	-	TO	-	TO	-
Drone Formation (Double)	2	HybridSynchAADL	1.38	0.02	3.71	0.07	8.24	0.17	TO	-	TO	-
		SpaceEx	2.03	0.21	44.00	2.29	658.35	22.00	TO	-	TO	-
		Hylaa	TO	-	TO	-	TO	-	TO	-	TO	-
	3	HybridSynchAADL	1.89	0.04	5.64	0.10	TO	-	TO	-	TO	-
		SpaceEx	17.68	0.91	TO	-	TO	-	TO	-	TO	-
		Hylaa	TO	-	TO	-	TO	-	TO	-	TO	-
	4	HybridSynchAADL	2.44	0.04	7.86	0.15	TO	-	TO	-	TO	-
		SpaceEx	128.39	1.55	TO	-	TO	-	TO	-	TO	-
		Hylaa	TO	-	TO	-	TO	-	TO	-	TO	-

Table 1. HybridSynchAADL symbolic analysis vs SpaceEx, Hylaa, HyComp

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	STDEV		AVG	STDEV		AVG	STDEV		AVG	STDEV		AVG	STDEV
Thermostat	2	random	TO	-	-	TO	-	-	TO	-	-	TO	-	-	TO	-	-
		symbolic	NF	0.32	0.01	NF	0.57	0.01	NF	1.07	0.02	NF	1.84	0.03	FO	9.45	0.08
		portfolio	NF	0.32	0.01	NF	0.57	0.01	NF	1.07	0.02	NF	1.84	0.03	FO	9.45	0.08
	3	random	TO	-	-	TO	-	-	TO	-	-	TO	-	-	TO	-	-
		symbolic	NF	0.39	0.01	NF	0.82	0.02	NF	1.68	0.03	NF	3.57	0.04	FO	18.83	0.16
		portfolio	NF	0.39	0.01	NF	0.82	0.02	NF	1.68	0.03	NF	3.57	0.04	FO	18.83	0.16
	4	random	TO	-	-	TO	-	-	TO	-	-	FO	6.36	9.84	FO	16.10	9.13
		symbolic	NF	0.46	0.01	NF	1.11	0.02	NF	2.69	0.03	FO	15.29	0.17			
		portfolio	NF	0.46	0.01	NF	1.11	0.02	NF	2.69	0.03	FO	6.36	9.84	FO	15.29	0.17
WaterTank	2	random	TO	-	-	TO	-	-	TO	-	-	TO	-	-	TO	-	-
		symbolic	NF	0.32	0.01	NF	0.47	0.01	NF	0.70	0.02	NF	1.02	0.01	FO	4.41	0.06
		portfolio	NF	0.32	0.01	NF	0.47	0.01	NF	0.70	0.02	NF	1.02	0.01	FO	4.41	0.06
	3	random	TO	-	-	TO	-	-	TO	-	-	TO	-	-	FO	0.16	0.15
		symbolic	NF	0.39	0.01	NF	0.67	0.01	FO	3.17	0.03						
		portfolio	NF	0.39	0.01	NF	0.67	0.01	FO	3.17	0.03	FO	3.17	0.03	FO	0.16	0.15
	4	random	TO	-	-	TO	-	-	TO	-	-	TO	-	-	TO	-	-
		symbolic	NF	0.48	0.01	NF	0.89	0.01	NF	1.64	0.03	NF	2.82	0.04	FO	14.84	0.10
		portfolio	NF	0.48	0.01	NF	0.89	0.01	NF	1.64	0.03	NF	2.82	0.04	FO	14.84	0.10
Drone rendezvous (single)	2	random	TO	-	-	TO	-	-	FO	866.58	435.33	FO	2.78	1.96	FO	0.34	0.24
		symbolic	NF	0.70	0.01	NF	1.78	0.04	FO	12.06	0.26						
		portfolio	NF	0.70	0.01	NF	1.78	0.04	FO	12.06	0.26	FO	2.78	1.96	FO	0.34	0.24
	3	random	TO	-	-	FO	1140.66	187.66	FO	320.08	350.66	FO	2.03	1.49	FO	0.28	0.14
		symbolic	NF	0.99	0.02	FO	7.51	0.24									
		portfolio	NF	0.99	0.02	FO	7.51	0.24	FO	7.51	0.24	FO	2.03	1.49	FO	0.28	0.14
	4	random	TO	-	-	TO	-	-	TO	-	-	TO	-	-	FO	27.20	26.16
		symbolic	NF	1.32	0.05	NF	5.68	0.16	NF	16.99	0.44	FO	68.72	1.45			
		portfolio	NF	1.32	0.05	NF	5.68	0.16	NF	16.99	0.44	FO	68.72	1.45	FO	27.20	26.16
Drone formation (single)	2	random	TO	-	-	TO	-	-	TO	-	-	TO	-	-	TO	-	-
		symbolic	NF	0.94	0.03	NF	2.56	0.04	FO	33.79	0.86						
		portfolio	NF	0.94	0.03	NF	2.56	0.04	FO	33.79	0.86	FO	33.79	0.86	FO	33.79	0.86
	3	random	TO	-	-	TO	-	-	TO	-	-	TO	-	-	FO	410.51	456.68
		symbolic	NF	1.30	0.03	NF	3.97	0.08	FO	47.61	1.23						
		portfolio	NF	1.30	0.03	NF	3.97	0.08	FO	47.61	1.23	FO	47.61	1.23	FO	47.61	1.23
	4	random	TO	-	-	TO	-	-	TO	-	-	FO	15.49	11.62	FO	2.58	3.13
		symbolic	NF	1.74	0.03	NF	6.23	0.14	FO	85.86	2.05						
		portfolio	NF	1.74	0.03	NF	6.23	0.14	FO	85.86	2.05	FO	15.49	11.62	FO	2.58	3.13
Drone rendezvous (double)	2	random	TO	-	-	FO	308.03	1.61	FO	0.05	0.002	FO	0.05	0.002	FO	0.05	0.001
		symbolic	NF	1.01	0.02	FO	7.19	0.07									
		portfolio	NF	1.01	0.02	FO	7.19	0.07	FO	0.05	0.002	FO	0.05	0.002	FO	0.05	0.001
	3	random	TO	-	-	FO	16.26	0.06	FO	0.09	0.001	FO	0.09	0.001	FO	0.09	0.002
		symbolic	NF	1.58	0.00	FO	13.89	0.11									
		portfolio	NF	1.58	0.00	FO	13.89	0.11	FO	0.09	0.001	FO	0.09	0.001	FO	0.09	0.002
	4	random	TO	-	-	FO	7.70	0.04	FO	0.13	0.002	FO	0.13	0.002	FO	0.13	0.002
		symbolic	NF	2.30	0.03	FO	18.56	0.16									
		portfolio	NF	2.30	0.03	FO	7.70	0.04	FO	0.13	0.002	FO	0.13	0.002	FO	0.13	0.002
Drone formation (double)	2	random	TO	-	-	FO	5.71	0.03	FO	0.52	0.004	FO	0.53	0.003	FO	0.50	0.003
		symbolic	NF	1.47	0.03	TO	-	-	TO	-	-	TO	-	-	TO	-	-
		portfolio	NF	1.47	0.03	FO	5.71	0.03	FO	0.52	0.004	FO	0.53	0.003	FO	0.50	0.003
	3	random	TO	-	-	FO	0.71	0.003	FO	0.26	0.002	FO	0.45	0.002	FO	0.38	0.002
		symbolic	NF	2.06	0.04	TO	-	-	TO	-	-	TO	-	-	TO	-	-
		portfolio	NF	2.06	0.04	FO	0.71	0.003	FO	0.26	0.002	FO	0.45	0.002	FO	0.38	0.002
	4	random	TO	-	-	FO	1.30	0.01	FO	0.59	0.004	FO	0.24	0.003	FO	0.24	0.002
		symbolic	NF	2.92	0.03	FO	97.66	1.57	TO	-	-	TO	-	-	TO	-	-
		portfolio	NF	2.92	0.03	FO	1.30	0.01	FO	0.59	0.004	FO	0.24	0.003	FO	0.24	0.002

Table 2. HybridSynchAADL portfolio analysis.

Model	N	Bound	Symbolic with merging					Symbolic without merging				
			Time		|Const| (k)	# SMT call	# Sym. State	Time		|Const| (k)	# SMT call	# Sym. State
			Average (s)	STDEV(σ)				Average (s)	STDEV(σ)			
Thermostat	2	100	0.2	0.0	3.8	59	2	0.5	0.0	51.2	731	10
		200	0.4	0.0	15.7	139	3	15.4	0.4	1,695.2	12,109	66
		300	0.7	0.0	37.5	219	4	203.0	4.2	19,445.1	88,716	223
		400	1.4	0.0	69.2	299	5	1,150.7	31.2	98,587.6	331,723	591
	3	100	0.2	0.0	5.8	87	2	6.2	0.2	559.7	6,581	28
		200	0.6	0.0	25.8	206	3	695.9	16.1	79,627.5	416,184	465
		300	1.2	0.0	65.1	325	4	-	-	-	-	-
		400	2.7	0.0	123.8	444	5	-	-	-	-	-
	4	100	0.3	0.0	7.8	115	2	68.9	1.0	5,921.2	59,123	82
		200	0.8	0.0	37.4	273	3	-	-	-	-	-
		300	1.9	0.0	98.8	431	4	-	-	-	-	-
		400	5.0	0.1	192.3	589	5	-	-	-	-	-
WaterTank	2	100	0.2	0.0	4.1	57	2	0.3	0.0	27.8	339	5
		200	0.3	0.0	15.7	135	3	2.0	0.0	237.2	1,687	17
		300	0.5	0.0	36.2	213	4	9.3	0.0	1,178.7	5,727	49
		400	0.9	0.0	65.5	291	5	35.4	0.0	4,578.3	16,495	129
	3	100	0.2	0.0	6.2	84	2	2.3	0.0	205.0	1,987	9
		200	0.5	0.0	26.2	200	3	30.9	0.0	3,646.8	17,859	57
		300	0.9	0.0	64.1	316	4	282.7	0.0	35,779.3	113,059	313
		400	1.6	0.0	120.3	432	5	2,285.7	0.0	272,514.8	620,707	1,593
	4	100	0.3	0.0	8.4	111	2	16.9	0.0	1,445.2	11,763	17
		200	0.7	0.0	38.2	265	3	477.0	0.0	53,509.6	199,891	209
		300	1.4	0.0	98.0	419	4	9,152.3	0.0	1,046,864.4	2,457,171	2,257
		400	2.5	0.0	188.7	573	5	-	-	-	-	-
Drone Rendezvous (Single)	2	100	0.6	0.0	38.5	293	2	14.0	0.2	1,205.7	9,993	677
		200	1.5	0.0	134.7	585	3	5,942.3	279.8	464,995.2	2,185,825	21,003
		300	2.9	0.1	300.8	877	4	-	-	-	-	-
		400	4.8	0.1	535.8	1,169	5	-	-	-	-	-
	3	100	0.8	0.0	57.8	438	2	470.7	13.0	38,871.4	260,113	17,577
		200	2.4	0.1	228.4	875	3	-	-	-	-	-
		300	4.9	0.1	544.8	1,312	4	-	-	-	-	-
		400	8.8	0.2	1003.0	1,749	5	-	-	-	-	-
	4	100	1.0	0.0	77.0	583	2	-	-	-	-	-
		200	3.4	0.1	339.8	1,165	3	-	-	-	-	-
		300	7.5	0.2	851.3	1,747	4	-	-	-	-	-
		400	14.1	0.3	1603.0	2,329	5	-	-	-	-	-
Drone Formation (Single)	2	100	0.7	0.0	50.7	328	2	18.3	0.5	1,514.6	10,028	677
		200	2.1	0.0	184.6	655	3	-	-	-	-	-
		300	4.4	0.1	437.2	982	4	-	-	-	-	-
		400	8.1	0.2	814.2	1,309	5	-	-	-	-	-
	3	100	1.0	0.0	75.6	473	2	608.6	17.8	46,888.7	260,148	17,577
		200	3.2	0.1	302.9	945	3	-	-	-	-	-
		300	7.2	0.2	756.0	1,417	4	-	-	-	-	-
		400	16.8	0.5	1446.4	1,889	5	-	-	-	-	-
	4	100	1.3	0.0	100.5	618	2	-	-	-	-	-
		200	4.5	0.1	440.4	1,235	3	-	-	-	-	-
		300	10.9	0.2	1148.3	1,852	4	-	-	-	-	-
		400	-	-	-	-	-	-	-	-	-	-
Drone Rendezvous (Double)	2	100	0.8	0.0	49.7	293	2	20.7	0.4	1,641.1	9,993	677
		200	2.2	0.0	177.8	585	3	-	-	-	-	-
		300	4.6	0.1	403.3	877	4	-	-	-	-	-
		400	7.9	0.2	727.6	1,169	5	-	-	-	-	-
	3	100	1.1	0.0	74.5	438	2	722.4	12.3	51,996.4	260,113	17,577
		200	3.5	0.0	301.6	875	3	-	-	-	-	-
		300	7.7	0.1	730.7	1,312	4	-	-	-	-	-
		400	14.1	0.2	1362.0	1,749	5	-	-	-	-	-
	4	100	1.4	0.0	99.4	583	2	-	-	-	-	-
		200	5.1	0.1	448.7	1,165	3	-	-	-	-	-
		300	12.0	0.2	1142.1	1,747	4	-	-	-	-	-
		400	23.4	0.5	2176.8	2,329	5	-	-	-	-	-
Drone Formation (Double)	2	100	1.1	0.0	67.6	328	2	22.2	0.5	1,730.7	9,560	443
		200	3.4	0.1	250.1	655	3	-	-	-	-	-
		300	7.5	0.1	604.8	982	4	-	-	-	-	-
		400	-	-	-	-	-	-	-	-	-	-
	3	100	1.5	0.0	100.7	473	2	627.4	11.3	41,214.9	173,100	11,493
		200	5.2	0.1	408.9	945	3	-	-	-	-	-
		300	12.4	0.2	1044.1	1,417	4	-	-	-	-	-
		400	-	-	-	-	-	-	-	-	-	-
	4	100	2.0	0.1	133.9	618	2	-	-	-	-	-
		200	7.4	0.1	592.6	1,235	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		# States (k)	Time		# States (k)	Time		# States (k)	Time		# States (k)
		Average (s)	STDEV		Average (s)	STDEV		Average (s)	STDEV		Average (s)	STDEV	
Thermostat	50	8.6	4.7	9.9	69.4	37.2	93.0	5,027.4	2,533.6	1,808.9	-	-	-
	100	8.5	4.7	10.0	74.2	42.5	99.8	4,232.3	249.2	1,812.7	-	-	-
	150	10.3	4.8	12.5	127.7	74.5	187.0	6,820.4	1,830.1	2,365.1	-	-	-
	200	9.5	4.1	13.2	149.1	76.4	250.7	-	-	-	-	-	-
	250	13.6	5.8	18.7	332.9	176.5	506.2	-	-	-	-	-	-
	300	14.5	8.2	20.0	391.6	233.9	596.4	-	-	-	-	-	-
WaterTank	50	7.8	2.4	9.8	79.7	35.3	91.1	5,036.6	1,221.9	1,793.2	-	-	-
	100	8.2	1.7	9.8	60.8	4.0	96.5	5,106.2	1,327.5	1,794.1	-	-	-
	150	9.3	1.5	10.9	84.7	8.7	139.6	5,173.0	925.1	1,876.0	-	-	-
	200	9.0	1.4	11.0	103.1	45.7	158.6	5,364.1	1,258.0	1,878.1	-	-	-
	250	9.7	1.2	12.0	183.4	120.3	235.3	5,162.0	100.7	1,952.5	-	-	-
	300	10.2	3.8	12.1	163.1	80.2	254.2	5,462.6	933.5	1,960.9	-	-	-
Drone Rendezvous (Single)	50	8.8	4.5	9.3	67.7	38.0	84.0	1,141.2	648.5	886.1	-	-	-
	100	8.8	5.5	10.7	70.9	43.3	90.5	1,195.2	667.0	939.7	-	-	-
	150	10.1	4.5	11.5	75.9	38.6	97.8	1,229.2	664.5	988.9	-	-	-
	200	15.4	8.5	19.2	92.4	47.5	134.4	3,015.8	1,424.6	2,253.9	-	-	-
	250	15.6	9.2	23.4	109.7	56.5	171.4	3,884.3	1,505.3	2,902.1	-	-	-
	300	20.7	10.2	30.9	131.4	69.5	207.3	5,935.4	230.1	3,923.7	-	-	-
Drone Formation (Single)	50	7,163.6	207.5	3,540.3	-	-	-	-	-	-	-	-	-
	100	7,854.2	169.3	3,888.4	-	-	-	-	-	-	-	-	-
	150	8,196.7	265.8	4,037.5	-	-	-	-	-	-	-	-	-
	200	-	-	-	-	-	-	-	-	-	-	-	-
	250	-	-	-	-	-	-	-	-	-	-	-	-
	300	-	-	-	-	-	-	-	-	-	-	-	-
Drone Rendezvous (Double)	50	158.8	96.5	148.7	1,700.5	892.2	1,337.2	-	-	-	-	-	-
	100	154.3	4.6	188.2	1,927.1	1,063.6	1,500.6	-	-	-	-	-	-
	150	203.1	91.7	226.2	2,189.1	1,068.1	1,813.1	-	-	-	-	-	-
	200	812.2	89.2	1,121.2	8,882.6	1,830.2	5,495.6	-	-	-	-	-	-
	250	1,616.3	361.3	1,869.6	-	-	-	-	-	-	-	-	-
	300	2,855.1	620.7	2,764.0	-	-	-	-	-	-	-	-	-
Drone Formation (Double)	50	-	-	-	-	-	-	-	-	-	-	-	-
	100	-	-	-	-	-	-	-	-	-	-	-	-
	150	-	-	-	-	-	-	-	-	-	-	-	-
	200	-	-	-	-	-	-	-	-	-	-	-	-
	250	-	-	-	-	-	-	-	-	-	-	-	-
	300	-	-	-	-	-	-	-	-	-	-	-	-

Table 4. Analyzing distributed asynchronous models.