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

No Author Given
No Institute Given

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][39][40]: 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 \cite{11} 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 \cite{11}, 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 \cite{28} (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 \cite{25} combined with SMT solving \cite{15,46} 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 \cite{27} 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](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 [5,39] 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 $\Gamma$ 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 [39]. 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 perform their transitions simultaneously, and outputs to other machines become inputs at the next iteration. PALS was extended to the multi-rate setting in [8], but, for simplicity of exposition, this paper focuses on the single-rate case.

Hybrid PALS [11] 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 \cite{39}. In its $i$th iteration, a controller $M$ samples the values of its environment at time $c_M(i \cdot p) + ts$, where $ts$ 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 $c_M(i \cdot p) + ta$, where $ta$ is the actuating time.

2.2 AADL

The Architecture Analysis & Design Language (AADL) \cite{28} 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\cite{1}. 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 \cite{29}. 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.

\footnote{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 [25] 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 [38] is a triple $R = (\Sigma, E, R)$, where $\Sigma$ is a signature (declaring sorts, subsorts, and function symbols) and $E$ a set of equations; and $R$ is a set of rewrite rules $l : t \rightarrow t'$ if $\text{cond}$, where $l$ is a label, $t$ and $t'$ are terms, and $\text{cond}$ is a conjunction of equations and rewrites. A rewrite $t \rightarrow^* t'$ holds if $t'$ is reachable from $t$ using the rewrite rules in $R$.

A declaration \text{class} C | \text{att}_1 : s_1, \ldots, \text{att}_n : s_n \text{ declares a class } C \text{ with attributes } \text{att}_1, \ldots, \text{att}_n \text{ of sorts } s_1, \ldots, s_n. \text{ An object } o \text{ of class } C \text{ is a term } < o : C | \text{att}_1 : v_1, \ldots, \text{att}_n : v_n >, \text{ of sort Object, where } v_i \text{ is the value of the attribute } \text{att}_i. \text{ 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 [15,46], (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 $\phi$ holds; i.e., given the underlying SMT theory $T$, we have:

$$\{ t(a_1, \ldots, a_n) \mid T \models \phi(a_1, \ldots, a_n) \}.$$

A symbolic rewrite $\phi_t \parallel t \rightarrow^* \phi_u \parallel u$ on constrained terms symbolically represents a (possibly infinite) set of system transitions sequences. For a symbolic rewrite $\phi_t \parallel t \rightarrow^* \phi_u \parallel u$, there is a “concrete” rewrite $t' \rightarrow^* u'$ with $t' \in \{ \phi_t \parallel t \}$ and $u' \in \{ \phi_u \parallel u \}$, and vice versa for each $t' \rightarrow^* u'$ 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 [27] and CVC4 [20].

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 \text{Hybrid_SynchAADL}. We use a subset of AADL without changing the meaning of AADL constructs or adding new a annex—the subset has the same meaning for synchronous models and asynchronous distributed implementations—so that AADL experts can easily develop and understand HYBRIDSYNCHAADL models.
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 \([9,12]\) of AADL. 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.

\[
\text{Hybrid\_SynchAADL}::\text{Synchronous} \Rightarrow \text{true}; \\
\text{Period} \Rightarrow \text{period};
\]

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 \(\text{ctrl}_i\) controls its environment \(\text{env}_i\) (using “connections” explained below).

Fig. 1. A networked thermostat system.

\(^2\) Just like Synchronous AADL can be extended to multi-rate controllers \([12]\), 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, ..., x_n`, say,\[ \frac{dx_i}{dt} = e_i(x_1, ..., x_n) \] for `i = 1, ..., n`, is written as a semicolon-separated string:

```
\frac{d}{dt}(x_1) = e_1(x_1, ..., x_n); ...; \frac{d}{dt}(x_n) = e_n(x_1, ..., 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), ..., x_n(0)` of the variables `x_1, ..., x_n`:

```
x_1(t) = e_1(t, x_1(0), ..., x_n(0)); ...; x_n(t) = e_n(t, x_1(0), ..., 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 `\frac{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 `⊥`). 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 \( \frac{d}{dt}(p) = 0 \) is omitted. The value \( x \) changes 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 \( x \rightarrow \text{temp} \). When a discrete controller sends an 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 \( x \rightarrow \text{temp} \).
3.2 Controller Components

Discrete controllers are usual AADL software components in the Synchronous AADL subset [10, 13]. 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 [29]. 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-triggered 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:

\[
\text{Dispatch Protocol} \Rightarrow \text{Periodic;}
\]

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:

\[
\text{Hybrid SynchAADL::Max Clock Deviation} \Rightarrow \text{time;}
\]
\[
\text{Hybrid SynchAADL::Sampling Time} \Rightarrow \text{lower bound .. upper bound;}
\]
\[
\text{Hybrid SynchAADL::Response Time} \Rightarrow \text{lower bound .. upper bound;}
\]

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 [11].

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:

\[
\text{Data Model::Initial Value} \Rightarrow \text{("value");}
\]

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 [1]). In HybridSynchAADL, such unknown parameters can be declared using the following AADL property:

\[
\text{Data Model::Initial Value} \Rightarrow \text{("param");}
\]
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 the 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.

```plaintext
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 [10,13], 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. 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 envi 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 combined with SMT solving.

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 \( \varphi_{\text{init}} \), an invariant condition \( \varphi_{\text{inv}} \), and a time bound \( \tau_{\text{bound}} \), where \( \varphi_{\text{init}} \) and \( \varphi_{\text{inv}} \) are in propositional logic. Intuitively, the invariant property holds if for every (initial) state satisfying the initial condition \( \varphi_{\text{init}} \), all states reachable within the time bound \( \tau_{\text{bound}} \) satisfy the invariant condition \( \varphi_{\text{inv}} \).

```
invariant [name]: \varphi_{\text{init}} \implies \varphi_{\text{inv}} \text{ in time } \tau_{\text{bound}}
```
A reachability property (the dual of an invariant) holds if a state satisfying \( \varphi_{\text{goal}} \) is reachable from some state satisfying the initial condition \( \varphi_{\text{init}} \) within the time bound \( \tau_{\text{bound}} \). It is worth noting that a reachability property can be expressed as an invariant property by negating the goal condition.

**reachability [name]:** \( \varphi_{\text{init}} \implies \varphi_{\text{goal}} \text{ in time } \tau_{\text{bound}} \)

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

\[
\text{path | exp}
\]

denotes that the component path of each identifier in the expression \( \text{exp} \) begins with \( \text{path} \). For example, \( \text{c}_1 \cdot \text{c}_2 | ((x_1 > x_2) \text{ and } (b_1 = b_2)) \) is equivalent to \( \text{c}_1 \cdot \text{c}_2 \cdot x_1 > \text{c}_1 \cdot \text{c}_2 \cdot x_2 \) and \( \text{c}_1 \cdot \text{c}_2 \cdot b_1 = \text{c}_1 \cdot \text{c}_2 \cdot b_2 \). These scopes can be nested so that one scope may include another scope. For example, \( \text{c}_1 | ((\text{c}_2 | (x > \text{c}_3 \cdot y)) = (\text{c}_4 | (\text{c}_5 | b))) \) is equivalent to the expression \( \text{c}_1 \cdot \text{c}_2 \cdot x > \text{c}_1 \cdot \text{c}_2 \cdot \text{c}_3 \cdot y ) = \text{c}_1 \cdot \text{c}_4 \cdot \text{c}_5 \cdot b \).

**Example 5.** Consider the thermostat system in Section 3 that consists of two thermostat controllers \( \text{ctrl1} \) and \( \text{ctrl2} \) and their environments \( \text{env1} \) and \( \text{env2} \), respectively. The following declares two propositions \( \text{inRan1} \) and \( \text{inRan2} \) using the property specification language. For example, \( \text{inRan1} \) holds if the value of \( \text{env1} \)'s data subcomponent \( x \) is between 10 and 25.

**proposition [inRan1]:** \( \text{env1} | (x > 10 \text{ and } x \leq 25) \)

**proposition [inRan2]:** \( \text{env2} | (x > 5 \text{ and } x \leq 10) \)

The following declares the invariant property \( \text{inv} \). The initial condition states that the value of \( \text{env1} \)'s data subcomponent \( x \) satisfies \( |x - 15| < 3 \) and the value of \( \text{env2} \)'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 \( \text{inRan1} \), \( \text{inRan2} \), and \( \text{env1}.x > \text{env2}.x \).

**invariant [inv]:** \( \text{abs(} \text{env1}.x - 15) < 3 \text{ and abs(} \text{env2}.x - 7) < 1 \implies \text{inRan1 and inRan2 and } (\text{env1}.x > \text{env2}.x) \text{ in time 30} \)

### 4.2 Tool Interface

Figure [4] 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 facilities 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.
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; 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.

---

4 Symbolic analysis only supports (nonlinear) polynomial continuous dynamics, since the underlying SMT solver, Yices2, does not support general classes of ODEs.
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 [14] 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 moving in 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 ordinary 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 [45] 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:

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

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 [45], 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:

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

where $\vec{e}_i = \vec{x}_i - \vec{o}_i$ with $\vec{o}_i$ an offset vector for the formation, and $\alpha$ and $\gamma$ are positive constants. The position $\vec{x}_i$ of the $i$-th drone eventually converges to $\vec{x}_N - \vec{a}_i$, 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 [45]:

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

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 HybridSynchAADL model requires much more text to specify connections. Nevertheless, we have developed a variety of HybridSynchAADL models for both rendezvous and formation control of different numbers of drones with respect to single-integrator and double-integrator dynamics. All these models are available at [https://hybridsynchaadl.github.io](https://hybridsynchaadl.github.io).
Figure 9 illustrates the AADL architecture of our model for rendezvous. There are four drone components. Each drone is connected with two other drones to exchange positions. For example, Drone 1 sends its position to Drone 2, and receives the position of Drone 4. A drone component consists of an environment and its controller. An environment component specifies the physical model of the drone, including position and velocity. A controller component interacts with the environment according to the sampling and actuating times. All controllers in the model have the same period.

In each round, a controller determines a new velocity to synchronize its movement with the other drones. The controller obtains the position \( \vec{x} \) from its environment according to the sampling time. The position of the connected drone is sent in the previous round, and is already available to the controller at the beginning of the round. The controller sends the current position \( \vec{x} \) through its output port. In the meantime, the environment changes its position according to the velocity indicated by its controller, where the new velocity \( \vec{v} \) from the controller becomes 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 ports \( oX \) and \( oY \), and receives the position of the other drone through its input ports \( iX \) and \( iY \). The component is declared to be synchronous with period 100 ms. Also, to meet the constraints of HYBRID_SYNCH_AADL, the connections between drone components are delayed and the output ports have some initial values. The maximal clock skew is given by Hybrid_SynchAADL::Max_Clock_Deviation.

**Drone Component.** A drone component in Fig. 11 has input ports \( iX \) and \( iY \) and output ports \( oX \) and \( oY \). Its implementation Drone.impl contains a controller ctrl and an environment env. The controller ctrl obtains the current position from env via input ports \( cX \) and \( cY \), and sends a new velocity to env via output ports \( vX \) and \( vY \), according to its sampling and actuating times.

**Environment.** Figure 12 shows an Environment component that specifies the physical model of the drone. It has two input ports \( vX \) and \( vY \) and two output ports \( cX \) and \( cY \). Data subcomponents \( x \), \( y \), velx and vely represent the position and velocity of the drone. The values of \( x \) and \( y \) are sent to the controller through the output ports \( cX \) and \( cY \). When a controller sends an actuation command to ports \( vX \) and \( vY \), the values of velx and vely are updated by the values of \( vX \) and \( vY \), or the mode changes according to the mode transitions. The dynamics
system FourDronesSystem
end FourDronesSystem;

system implementation FourDronesSystem.impl
subcomponents
dr1: system Drone::Drone.impl;  dr2: system Drone::Drone.impl;
dr3: system Drone::Drone.impl;  dr4: system Drone::Drone.impl;
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.impl;

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.cX -> ctrl.cX;  C6: port env.cY -> ctrl.cY;
C7: port ctrl.vX -> env.vX;  C8: port ctrl.vY -> env.vY;
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
cX: out data port Base_Types::Float;
cY: out data port Base_Types::Float;
vX: in data port Base_Types::Float;
vY: 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;
velx: data Base_Types::Float;
vely: data Base_Types::Float;
connections
  C1: port x -> cX;
  C2: port y -> cY;
  C3: port vX -> velx;
  C4: port vY -> vely;
properties
  Hybrid_SynchAADL::ContinuousDynamics =>
    "x(t) = velx * t + x(0); y(t) = vely * t + y(0);"
  Data_Model::Initial_Value => ("param") applies to x, y, velx, vely;
end Environment.impl;

Fig. 12. An environment component in HybridSynchAADL.

of \((x, y)\) is given as continuous functions \(x(t) = vel_x t + x(0)\) and \(y(t) = vel_y t + y(0)\) over time \(t\) in Hybrid_SynchAADL::ContinuousDynamics, which are actually equivalent to the ordinary differential equations \(\dot{x} = vel_x\) and \(\dot{y} = vel_y\).

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 \(cX, cY, vX,\) and \(vY\) for interacting with the environment. The system implementation DroneControl.impl includes the process component ctrlProc. As shown in Figure 14, ctrlProc again includes the thread component cThread in its implementation DroneControlProc.impl. The input and output ports of a wrapper component (e.g., ctrlProc) are connected to the ports of the enclosed subcomponent (e.g., cThread).

Figure 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 \((0, 0)\) and the close flag is set to true to avoid a collision. Otherwise, the new velocity is set toward the connected drone according to a discretized version of the distributed consensus algorithm. That is, the new velocity \((vX, vY)\) is chosen from a predefined set of velocities, according to the value \((nx, ny)\) obtained by the distributed consensus algorithm and the close flag. Finally, the current position is assigned to the output ports \(oX\) and \(oY\).
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;
    cX: in data port Base_Types::Float;
    cY: in data port Base_Types::Float;
    vX: out data port Base_Types::Float;
    vY: out data port Base_Types::Float;
end DroneControl;

system implementation DroneControl.impl
  subcomponents
    ctrlProc: process DroneControlProc.impl;
  connections
      C1: port ctrlProc.oX -> oX;  C2: port ctrlProc.oY -> oY;
      C3: port iX -> ctrlProc.iX;  C4: port iY -> ctrlProc.iY;
      C5: port cX -> ctrlProc.cX;  C6: port cY -> ctrlProc.cY;
      C7: port ctrlProc.vX -> vX;  C8: port ctrlProc.vY -> vY;
end DroneControl.impl;

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;
    cX: in data port Base_Types::Float;
    cY: in data port Base_Types::Float;
    vX: out data port Base_Types::Float;
    vY: out data port Base_Types::Float;
end DroneControlProc;

process implementation DroneControlProc.impl
  subcomponents
    cThread: process DroneControlThread.impl;
  connections
      C1: port cThread.oX -> oX;  C2: port cThread.oY -> oY;
      C3: port iX -> cThread.iX;  C4: port iY -> cThread.iY;
      C5: port cX -> cThread.cX;  C6: port cY -> cThread.cY;
      C7: port cThread.vX -> vX;  C8: port cThread.velY -> vY;
end DroneControlProc.impl;

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;
    cX: in data port Base_Types::Float;
    cY: in data port Base_Types::Float;
    vX: out data port Base_Types::Float;
    vY: out data port Base_Types::Float;
    properties
    Dispatch_Protocol => Periodic;
end DroneControlThread;

thread implementation DroneControlThread.impl
    subcomponents
    close: data Base_Types::Boolean
        (Data_Model::Initial_Value => ("false"););
    annex behavior_specification (**
        variables
        nx, ny: Base_Types::Float;
        states
        init: initial complete state; exec, output: state;
        transitions
        init -on dispatch-> exec;
        exec -abs(cX - iX) < 0.5 and abs(cY - iY) < 0.5]-> output {
            vX := 0; vY := 0; close := true
        };
        exec -otherwise-> output {
            nx := -#DroneSpec::A * (cX - iX);
            ny := -#DroneSpec::A * (cY - iY);
            if (nx > 0.3) vX := 2.5
                elsif (nx > 0.15)
                    if (close) vX := 1.5
                        else vX := 0.0
                        end if
                    else
                        vX := -2.5
                    end if;
                if (ny > 0.3) vY := 2.5
                    elsif (ny > 0.15)
                        if (close) vY := 1.5
                            else vY := 0.0
                            end if
                        else
                            vY := -2.5
                        end if;
                    close := false ;
                output -[]-> init { oX := cX; oY := cY }
            **)
        };
end DroneControlThread.impl;

Fig. 15. A controller thread in HYBRIDSYNCHAADL
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.

\[
\text{invariant [safety]: } \text{?initial} \implies \neg \text{?collision in time 500;}
\]
\[
\text{reachability [rendezvous]: } \text{?initial} \implies \text{?gather in time 500;}
\]

We define three atomic propositions collision, gather, and initial for four drones \text{dr1}, \text{dr2}, \text{dr3}, and \text{dr4}. Two drones collide if the distance between them is less than 0.1. All nodes have gathered if the distance between each pair of nodes is less than 1. The initial values of \text{x}, \text{y}, \text{velx} and \text{vely} are declared to be parametric in Fig. 12 and constrained by the condition initial. There are infinitely many initial states satisfying the proposition initial.

\[
\text{proposition [collision]:}
\]
\[
\text{abs(}\text{dr1.env.x} - \text{dr2.env.x}) < 0.1 \text{ and } \text{abs(}\text{dr1.env.y} - \text{dr2.env.y}) < 0.1 \text{ or }
\text{abs(}\text{dr1.env.x} - \text{dr3.env.x}) < 0.1 \text{ and } \text{abs(}\text{dr1.env.y} - \text{dr3.env.y}) < 0.1 \text{ or }
\text{abs(}\text{dr1.env.x} - \text{dr4.env.x}) < 0.1 \text{ and } \text{abs(}\text{dr1.env.y} - \text{dr4.env.y}) < 0.1 \text{ or }
\text{abs(}\text{dr2.env.x} - \text{dr3.env.x}) < 0.1 \text{ and } \text{abs(}\text{dr2.env.y} - \text{dr3.env.y}) < 0.1 \text{ or }
\text{abs(}\text{dr2.env.x} - \text{dr4.env.x}) < 0.1 \text{ and } \text{abs(}\text{dr2.env.y} - \text{dr4.env.y}) < 0.1 \text{ or }
\text{abs(}\text{dr3.env.x} - \text{dr4.env.x}) < 0.1 \text{ and } \text{abs(}\text{dr3.env.y} - \text{dr4.env.y}) < 0.1 \text{;}
\]

\[
\text{proposition [gather]:}
\]
\[
\text{abs(}\text{dr1.env.x} - \text{dr2.env.x}) < 1 \text{ and } \text{abs(}\text{dr1.env.y} - \text{dr2.env.y}) < 1 \text{ and }
\text{abs(}\text{dr1.env.x} - \text{dr3.env.x}) < 1 \text{ and } \text{abs(}\text{dr1.env.y} - \text{dr3.env.y}) < 1 \text{ and }
\text{abs(}\text{dr1.env.x} - \text{dr4.env.x}) < 1 \text{ and } \text{abs(}\text{dr1.env.y} - \text{dr4.env.y}) < 1 \text{ and }
\text{abs(}\text{dr2.env.x} - \text{dr3.env.x}) < 1 \text{ and } \text{abs(}\text{dr2.env.y} - \text{dr3.env.y}) < 1 \text{ and }
\text{abs(}\text{dr2.env.x} - \text{dr4.env.x}) < 1 \text{ and } \text{abs(}\text{dr2.env.y} - \text{dr4.env.y}) < 1 \text{ and }
\text{abs(}\text{dr3.env.x} - \text{dr4.env.x}) < 1 \text{ and } \text{abs(}\text{dr3.env.y} - \text{dr4.env.y}) < 1 \text{;}
\]

\[
\text{proposition [initial]:}
\]
\[
\text{abs(}\text{dr1.env.x} - 1.1) < 0.01 \text{ and } \text{abs(}\text{dr1.env.y} - 1.5) < 0.01 \text{ and }
\text{abs(}\text{dr2.env.x} + 1.5) < 0.01 \text{ and } \text{abs(}\text{dr2.env.y} + 1.1) < 0.01 \text{ and }
\text{abs(}\text{dr3.env.x} - 1.5) < 0.01 \text{ and } \text{abs(}\text{dr3.env.y} - 1.1) < 0.01 \text{ and }
\text{abs(}\text{dr4.env.x} + 1.1) < 0.01 \text{ and } \text{abs(}\text{dr4.env.y} + 1.5) < 0.01 ;
\]

The result of the analysis is shown in Figure 16. The HybridSynchAADL Result view displays a witness for rendezvous (by symbolic analysis in 1.7 seconds), and a counterexample for safety (by randomized simulation in 1.5 seconds). A concrete counterexample of safety is shown in the editor as a sequence of states for synchronous steps. For example, the drone \text{dr3} has velocity \((-5125,5682)\) 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.
Fig. 16. Results for the safety and rendezvous properties.

We therefore modify safety and rendezvous below by adding such a velocity constraint $\text{velconst}$ to the initial condition. As shown in Fig. 17 there is now no counterexample to safety up to bound 500. The result is obtained by symbolic analysis for safety, and by randomized simulation for rendezvous.

```
invariant [safety]: ?initial and \text{velconst} \implies \neg \text{collision} in time 500;
reachability [rendezvous]: ?initial and \text{velconst} \implies \text{gather} in time 500;
proposition [velconst]:
    abs(dr1.env.vx) <= 0.01 \text{ and } abs(dr1.env.vy) <= 0.01 \text{ and }
    abs(dr2.env.vx) <= 0.01 \text{ and } abs(dr2.env.vy) <= 0.01 \text{ and }
    abs(dr3.env.vx) <= 0.01 \text{ and } abs(dr3.env.vy) <= 0.01 \text{ and }
    abs(dr4.env.vx) <= 0.01 \text{ and } abs(dr4.env.vy) <= 0.01;
```

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, $\text{perturbed}$ 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 [10,13], 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 [11] 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 $\phi(x_1,\ldots,x_n) \parallel \text{obj}(x_1,\ldots,x_n)$ to symbolically represent an infinite number of objects, where an SMT constraint $\phi(x_1,\ldots,x_n)$ and an object “pattern” 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, 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 $\phi \parallel \text{conf}$ with $\text{conf}$ 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...
where PHI is a variable that denotes an SMT constraint, \( \varphi \) denotes a newly generated constraint, and check-sat checks the satisfiability of the accumulated constraints using the SMT solver.

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

### 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.

```plaintext
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.

```plaintext
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 \(\text{gen}\) generates a fresh variable of a given type:

```plaintext
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
of sort `BoolExp` represents the pattern \((1 + x_1 > 2 \times x_2 \text{ or } b_1) = (b_2 \text{ and } \text{true})\).

\[
(((1) + r(1) > [2] \times r(2)) \text{ or } b(1)) \implies (b(2) \text{ and } \text{true})
\]

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.

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

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

```plaintext
class Ensemble .
subclass Ensemble < Component .

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

\footnote{\([1]\) and \([2]\) are symbolic values of sort `RealValue`, \([\text{true}]\) 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`.}
Thread Components. Thread components are represented as object instances of the Thread class that contains extra attributes for thread behaviors:

```java
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.

```java
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_dynamics] where m is a mode and continuous_dynamics 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:

```java
class Data | value : DataContent .
subclass Data < Component .
```
In HYBRIDSYNCHAADL, a data component has either no value (more precisely, some “don’t care” value ⊥) 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., ⊥) if \( b \) is \( \text{false} \), and a value represented by the expression \( e \) if \( b \) is \( \text{true} \).

\[
\text{sort DataContent}.
\text{op } \_\#_ : \text{Exp} \times \text{BoolExp} \rightarrow \text{DataContent} \ [\text{ctor}] .
\]

**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.

\[
\begin{align*}
\text{class } & \text{Port } | \text{content} : \text{DataContent}, \text{properties} : \text{PropertyAssociation} . \\
\text{class } & \text{InPort} . \text{class } \text{OutPort} . \\
\text{subclass } & \text{InPort OutPort } < \text{Port} . \\
\end{align*}
\]

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 \( ⊥ \) (i.e., \( e \# \text{false} \)) in the latest dispatch, the thread can use the value in the `cache`, while the behavior annex expression \( p’\text{fresh} \) becomes \( \text{false} \)\(^6\)\([9,12]\).

\[
\begin{align*}
\text{class } & \text{DataPort} . \\
\text{subclass } & \text{DataPort } < \text{Port} . \\
\text{class } & \text{DataInPort } | \text{cache} : \text{DataContent} . \\
\text{class } & \text{DataOutPort} . \\
\text{subclass } & \text{DataInPort } < \text{InPort DataPort} . \\
\text{subclass } & \text{DataOutPort } < \text{OutPort DataPort} . \\
\end{align*}
\]

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.\(^6\) To symbolically encode the immediate communication, the attribute `envCache` contains a data content in the previous iteration (see Section 6.7).

\(^6\) This implies that no fan-out connection from an environment output port \( p \) exists, i.e., \( p \) can only be connected to one controller input port.
class 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 \( p_i \rightarrow p_o \), 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 \( c \ldots p \). A connection from an output port \( p_1 \) in \( c_1 \) to an input port \( p_2 \) in \( c_2 \) is written as \( c_1 \ldots p_1 \rightarrow c_2 \ldots p_2 \). 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 \( c \ldots p \rightarrow p' \) (resp., \( p' \rightarrow c \ldots p \)).

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 \( d \Rightarrow p \), and a connection from an input port \( p \) to a data subcomponent \( d \) for updating data is written as the term \( p \Rightarrow d \).

Example 6. An instance of the TwoThermostats.impl component in Fig. 5 is represented by the following object, where property values are enclosed by \{(...)}\.

< 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 ;
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},
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 \{...\} in flows.

```object
< 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.
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, `executeStep` 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.
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 \[12\], 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` non-deterministically assigns any possible computation result of the behavior transition system to the pattern $L' | FMAP' | DATA' | PHI' | GEN'$. The function `writeFeature` updates the content of each output port, and `writeData` updates the value of each data subcomponent. The function `check-sat` invokes an SMT solver to check whether the generated constraint is satisfiable. The constants `#loopbound#` and `#transbound#` denote a loop unrolling bound and a transition bound, respectively, for symbolic analysis.\[7\]

```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])
```

\[7\] `loopBound` limits the number of loop unrolling when symbolically executing behavior actions, and `transBound` limits the number of visiting the same behavior locations in one synchronous step when symbolically executing behavior transitions.
Behavior Configurations. We represent a group of “named” function arguments $id_1 : \text{arg}_1, id_2 : \text{arg}_2, \ldots, id_n : \text{arg}_n$ as a a multiset of behavior configuration items of the form $id_1(\text{arg}_1) \ id_2(\text{arg}_2) \ldots id_n(\text{arg}_n)$. 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.

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:

A feature map built by readFeature has different feature map contents for input and output ports, because a behavior annex expression $p'\text{fresh}$ 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).

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.

\[
\begin{align*}
\text{eq } \text{readFeature} & (P : \text{DataInPort} | \text{content} : E \ # B, \ \text{cache} : E' \ # B' > \text{PORTS}, \\
& \text{PORTS}' , \ \text{FMAP}) \\
& \quad = \text{readFeature}(\text{PORTS}, \\
& \quad \quad < P : \text{DataInPort} | \text{content} : E \ # [false], \\
& \quad \quad \quad \text{cache} : (B \ ? \ E : E') \ # (B \ or \ B') > \text{PORTS}', \\
& \quad \quad \quad \quad \text{insert}(P, ((B \ ? \ E : E') \ # (B \ or \ B')) : B, \ \text{FMAP}).
\end{align*}
\]

Finally, each output port is related to \(E \ # [false]\) in the resulting map \(\text{FMAP}\),
indicating \(\bot\) with the second item \([false]\), because behavior transitions cannot
read a value from output ports.

\[
\begin{align*}
\text{eq } \text{readFeature} & (P : \text{DataOutPort} | \text{content} : E \ # B > \text{PORTS}, \ \text{PORTS}', \ \text{FMAP}) \\
& \quad = \text{readFeature}(\text{PORTS}, < P : \text{DataOutPort} | > \text{PORTS}', \\
& \quad \quad \quad \quad \text{insert}(P, E \ # [false], \ \text{FMAP}).
\end{align*}
\]

The semantic function \(\text{writeFeature}\) replaces the content of each output port
\(P\) by the corresponding content \(D'\) in the map \(\text{FMAP}\).

\[
\begin{align*}
\text{eq } \text{writeFeature} & (\text{FMAP}, \ \text{PORTS}) = \text{writeFeature}(\text{FMAP}, \ \text{PORTS}, \ \text{none}) . \\
\text{eq } \text{writeFeature} & ((P \ |-> D'), \ \text{FMAP}), \\
& \quad < P : \text{DataOutPort} | \text{content} : D > \text{PORTS}, \ \text{PORTS}') \\
& \quad \quad \quad \quad \quad \text{writeFeature}(\text{FMAP}, \ \text{PORTS}, < P : \text{DataOutPort} | \text{content} : D' > \text{PORTS}') . \\
\text{eq } \text{writeFeature} & (\text{FMAP}, \ \text{PORTS}, \ \text{PORTS}') = \text{PORTS} \ \text{PORTS}' \ [\text{owise}] .
\end{align*}
\]

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

\[
\begin{align*}
\text{eq } \text{readData} & (\text{COMPS}) = \text{readData}(\text{COMPS}, \ \text{empty}) . \\
\text{eq } \text{readData} & (C : \text{Data} | \text{value} : D > \text{COMPS}, \ \text{DATA}) \\
& \quad \quad \quad \quad \quad \text{readData}(\text{COMPS}, \ \text{insert}(C, D, \ \text{DATA})) . \\
\text{eq } \text{readData} & (\text{none}, \ \text{DATA}) = \ \text{DATA} . \\
\text{eq } \text{writeData} & (\text{DATA}, \ \text{COMPS}) = \text{writeData}(\text{DATA}, \ \text{COMPS}, \ \text{none}) . \\
\text{eq } \text{writeData} & (C |-> D', \ \text{DATA}), < C : \text{Data} | \text{value} : D > \text{COMPS}, \ \text{COMPS}') \\
& \quad \quad \quad \quad \quad \text{writeData}(\text{DATA}, \ \text{COMPS}, \ \text{COMPS}' < C : \text{Data} | \text{value} : D' >) . \\
\text{eq } \text{writeData} & (\text{DATA}, \ \text{COMPS}, \ \text{COMPS}') = \text{COMPS} \ \text{COMPS}' \ [\text{owise}] .
\end{align*}
\]

**Executing Transitions.** The behavior of the semantic operation \(\text{execTrans}\) is
defined with respect to behavior configurations as follows.

\[
\begin{align*}
\text{crl } [\text{trans}]: \\
& \text{execTrans(location(SL) \ trans(TRS) \ tbound(s(N)) \ local(VAL) \ REST)} \\
& \quad => \\
& \quad \quad \text{execTStep}(\text{VAL}, \ \text{location(L')} \ \text{trans(TRS)} \ \text{tbound(N)}
\end{align*}
\]
A transition $L \xrightarrow{[GUARD]} L'$ 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. 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.

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.

---

Add constraint to configuration:

```plaintext
addConst(execAction(ACT, local(VAL) REST), B and B'))
```

```plaintext
if (L ->[GUARD] L') ; TRS' := TRS
\ / B := locConst(SL, L) \ / check-sat(B)
\ / B' := guardConst(GUARD, outTrs(L, TRS'), local(VAL) REST).
```

---

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.
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.

\[
\begin{align*}
eq_{\text{eval}}([[B:\text{Bool}]], \ BCF) &= [B:\text{Bool}] # [\text{true}] . \\
eq_{\text{eval}}([[I:\text{Int}]], \ BCF) &= [I:\text{Int}] # [\text{true}] . \\
eq_{\text{eval}}([[F:\text{Float}]], \ BCF) &= [\text{rat}(F:\text{Float})] # [\text{true}] .
\end{align*}
\]

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`.

\[
\begin{align*}
eq_{\text{eval}}(v[VI], \ local(VAL) \ REST) &= VAL[VI] . \\
eq_{\text{eval}}(f[PI], \ feature(FMAP) \ REST) &= \text{getData}(FMAP[PI]) . \\
eq_{\text{eval}}(c[C], \ data(DATA) \ REST) &= DATA[C] . \\
eq_{\text{eval}}(p[PR], \ prop(PROPS) \ REST) &= \text{eval}(\text{value}(PROPS[PR]), \ REST) . \\
\text{ceq}_{\text{eval}}(\text{fresh}(PI), \ feature(FMAP) \ REST) &= B \ # B1 \text{ if } E \ # B := \text{FMAP}[PI] .
\end{align*}
\]

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.

\[
\begin{align*}
eq_{\text{eval}}(AE1 + AE2, \ REST) &= \text{eval}(AE1, \ REST) + \text{eval}(AE2, \ REST) . \\
eq_{\text{eval}}((E1 \ # B1) + (E2 \ # B2) &= (E1 + E2) \ # (B1 \text{ and } B2) .
\end{align*}
\]

Executing Actions. The semantic operation `execAction` computes a behavior action based on a given behavior configuration, and returns the resulting behavior configuration. These behavior configurations contain symbolic expressions and constraints, and represent (possibly 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.

\[
\begin{align*}
\text{ceq}_{\text{execAction}}(v(VI) := \ AE, \ REST)
&= \text{local}(insert(VI, E \ # [\text{true}], VAL)) \text{ addConst}(REST, B) \\
&\text{ if } E \ # B := \text{eval}(AE,REST) . \\
\text{ceq}_{\text{execAction}}(f(PI) := \ AE, \ REST)
&= \text{feature}(insert(PI, E \ # [\text{true}], FMAP)) \text{ addConst}(REST, B) \\
&\text{ if } E \ # B := \text{eval}(AE,REST) . \\
\text{ceq}_{\text{execAction}}(c(C) := \ AE, \ REST)
&= \text{data}(insert(C, E \ # [\text{true}], DATA)) \text{ addConst}(REST, B) \\
&\text{ if } E \ # B := \text{eval}(AE,REST) .
\end{align*}
\]
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:

\[
\text{crl} \text{execAction(if (AE) AS end if, REST)} \\
\Rightarrow \text{execAction(AS, addConst(REST, E and B))} \\
\text{if } E \# B := \text{eval}(AE, REST) .
\]

\[
\text{crl} \text{execAction(if (AE) AS end if, REST)} \\
\Rightarrow \text{addConst(REST, E and B)} \\
\text{if } E \# B := \text{eval}(\text{not}(AE), REST) .
\]

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:

\[
\text{crl} \text{execAction(while (AE) {AS}, lbound(s(N)) REST)} \\
\Rightarrow \text{execAction({AS ; while (AE) {AS}}, lbound(N) addConst(REST, E and B))} \\
\text{if } E \# B := \text{eval}(AE, REST) .
\]

\[
\text{crl} \text{execAction(while (AE) {AS}, REST)} \Rightarrow \text{addConst(REST, E and B)} \\
\text{if } E \# B := \text{eval}(AE, REST) .
\]

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:

\[
\text{eq} \text{execAction({A ; ASQ}, REST)} = \text{execAction({ASQ}, execAction(A, REST))} .
\]

\[
\text{eq} \text{execAction({A}, REST)} = \text{execAction(A, REST)} .
\]

### 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 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 \( t^n_0 \) to controller `C_n`, and receives `C_n`’s command \( \alpha_n \) at time \( t^n_0 \) (and may also change its continuous dynamics by \( \alpha_n \)), according to the sampling and actuating times of `C_n`.

The semantics of environment components cannot be directly specified as synchronous composition. Indeed, the environment behavior is *asynchronous*, since the order of “interaction events” in a single iteration (e.g., `sampling(C_1)`, `sampling(C_2)`, `response(C_1)`, and `response(C_2)` in Fig. [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 [41], but it can lead to state-space explosion. In Hybrid PALS [11], 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 HYBRIDSYNCAADL, 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 is 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:

\[
(x_1, \ldots, x_n) \mapsto \phi(x_1, x_2, \ldots, x_n)
\]

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.
Fig. 19. The behavior of \texttt{readEnvFeature}.

\textit{Executing Symbolic Steps}. The following rule defines the behavior of environment components. The function \texttt{readEnvFeature} returns a map from each input port to its symbolic content, and \texttt{writeEnvFeature} updates the content of each output port. These functions also return extra constraints to encode the environment communication. The operation \texttt{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 \texttt{check-sat} then invokes an SMT solver to check whether the generated constraint is satisfiable.

\begin{verbatim}
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)
  \(\text{DATA} := \text{readData(COMPS)}\)
  \(\text{execEnv(feature(FMAP) data(DATA) prop(PROPS) vargen(GEN1) mode(L) time([0]) jumps(JUMPS) flows(FLOWS) sampling(STS) response RTS envcon(CONXS,PORTS) const([true]))} =\)
  \(L' | FMAP' | DATA' | PHI' | GEN2\)
  \(\text{PORTS',OPHI,GEN'} := \text{writeEnvFeature(FMAP', PORTS1, GEN2)}\)
  \(\text{check-sat(PHI and IPHI and PHI' and OPHI)}\).
\end{verbatim}

\textit{Environment Feature Operations}. Given a set of environment ports, the function \texttt{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 \texttt{envCache} that contains the symbolic variable for the previous round. Suppose that a controller sends a data content $d_i$ to an input port $p$ in the $i$-th iteration; the content of $p$ is then $d_{i-1}$ at the beginning.
of the $i$-th iteration. The function \texttt{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 $\text{PI}$ is related to a symbolic content $V \# BV$ with fresh variables $V$ and $BV$, the current content $E \# B$ and $\text{envCach} E' \# B'$ are declared to be identical as a constraint, the $\text{envCach}$ attribute is updated, and the \text{content} attribute is set absent. In the last equation, each output port is related to $E \# \text{false}$, indicating $\bot$ with the second item $[\text{false}]$.

\begin{verbatim}
  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,BV2) := 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).
\end{verbatim}

The function \texttt{writeFeature} replaces the content of each output port $P$ by a symbolic variable and declares that \textit{the data content sent in the previous round is identical to the current content} in the map \texttt{FMAP}. Thus, the corresponding input port of the controller receives the current content in the same iteration. Similarly, we use \texttt{envCache}, containing the symbolic variable sent in the previous round, to implement this behavior, as described in Fig. 20 where $x_0$ denotes the initial content. The function \texttt{writeFeature} is defined as follows.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\# Iteration & 0 & 1 & 2 & 3 \\
\hline
Environment & FMAP & $d_1$ & $d_2$ & $d_3$ \\
& envCache & $x_0$ & $x_1$ & $x_2$ \\
\hline
writeEnvFeature & content of \text{PI} & $x_0$ & $x_1$ & $x_2$ & $x_3$ \\
& constraint & $x_0 = d_1$ & $x_1 = d_2$ & $x_2 = d_3$ \\
\hline
Controller & & $x_0$ & $x_1$ & $x_2$ \\
\hline
\end{tabular}
\caption{The behavior of \texttt{writeEnvFeature}.

Fig. 20. The behavior of \texttt{writeEnvFeature}.}
\end{table}
eq writeEnvFeature(FMAP, PORTS, GEN) = writeEnvFeature(PORTS, none, FMAP, [true], GEN).

ceq writeFeature(< PI : EnvOutPort | content : D, envCache : E # B > PORTS, PORTS', FMAP, PHI, GEN) = writeFeature(PORTS, PORTS' < PI : EnvOutPort | content : V # BV, envCache : V # BV >, FMAP, PHI and E === E' and B === B', GEN2)

if E' # B' := FMAP[PI] /\ {V, GEN1} := gen(GEN, type(E)) /\ {BV, GEN2} := gen(GEN1, Boolean).

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

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

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

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

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

**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_{\text{period}}.
The environment time frame is “shifted” to the left from the global time frame \( [g(i), g(i + 1)] \) by a maximal clock skew \( \epsilon > 0 \). Because each controller \( C_n \), \( n = 1, 2 \), runs according to its local clock, the period of \( C_n \) begins at any time \( 0 < t^0_n < 2\epsilon \), and its sampling and actuation happen according to the sampling and actuating times of \( C_n \) with respect to \( t^0_n \). That is, in Fig. 21, \( t^*_n - t^0_n \) and \( t^*_n - t^0_n \) denote the sampling and actuating times declared by \( C_n \).

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

The following rule \( \text{env-cont} \) 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 \( \text{FLOW} \) is the same as the current mode \( \text{SL} \). The function \( \text{execFlow} \) builds symbolic values and constraints to encode the new environment states given by evolving the environment by time \( T' - T \). Finally the function \( \text{updateEnvData} \) updates the data subcomponents to have the symbolic values for time \( T' \).

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

The function \( \text{execFlow} \) computes the values of continuous dynamics for a given input \( T \). As mentioned, continuous dynamics in \( \text{HYBRIDSYNCHAADL} \) are specified using either ODEs or a set of continuous real functions. Currently, we only consider the case of continuous real functions\(^9\) a function of the form \( C(VI) = AE \) over an input argument \( VI \) is evaluated using the function \( \text{eval} \), while \( T \) is assigned to the local variable identifier \( VI \) as follows:

\[
\text{eq} \ \text{execFlow}([(\text{FUNCS}, T, \text{REST}) = \ \text{execFuncFlow}(\text{FUNCS}, T, \text{empty}, \text{REST})].
\]

\[
\text{ceq} \ \text{execFuncFlow}((C(VI) = AE) ; \ \text{FUNCS, T, DATA, REST}) \\
= \ \text{execFuncFlow}(\text{FUNCS, T, insert}(C, D, DATA), \text{REST}) \\
\text{if} & \ D := \ \text{eval}(AE, \ \text{local}(VI \rightarrow T \ # \ \text{true}), \text{REST}) .
\]

\[
\text{eq} \ \text{execFuncFlow}(\text{empty}, T, \text{DATA, REST}) = \text{DATA} .
\]

\(^9\) If ODEs have closed-form solutions, we can define them as continuous functions. It is possible to directly generate SMT constraints with ODEs, and solve them using a specialized SMT solver, such as \( \text{dReal} \). Because \( \text{dReal} \) has not been integrated with Maude, the current version of the semantics only supports continuous functions.
The function `updateEnvData` simply updates the behavior configuration item `data` with the result `DATA` of applying `execFlow`.

\[
\text{eq updateEnvData}(\langle C |\Rightarrow E \# B, DATA \rangle, data(DATA') \text{ REST}) \\
= \text{updateEnvData}(DATA, data(insert(C, E \# [true], DATA')) \text{ addConst(REST,B)}) .
\]

\[
\text{eq updateEnvData}(\text{empty}, \text{ REST}) = \text{ REST} .
\]

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`.

\[
\text{ceq envStep}(\text{time}(T) \text{ mode}(L) \text{ sampling}(\text{empty}) \text{ response}(\text{empty}) \text{ prop}(\text{PROPS}) \text{ REST}) \\
= \text{transResult}(L, \text{prop}(\text{PROPS}) \text{ addConst(REST, B \text{ and } (T \text{ === } \text{PER})))}.
\]

### 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.

\[
\text{crl [env-samp]}:
\text{envStep}(\text{time}(T) \text{ sampling}(\langle C :(LT,UT), SIT \rangle) \text{ REST}) \\
\Rightarrow \text{execEnv}(\text{time}(T) \text{ sampling}(SIT) \text{ addConst(REST',B)}) \\
\text{if } B := \text{timeConst}(T, LT, UT, \text{ REST}) \\
/\ REST' := \text{updateEnvFeature}(C, \text{ REST}) .
\]

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

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

The function `updateEnvFeature` updates the content of the output port `PI` using the content of state variable `CI`, if there is an internal connection `CI \Rightarrow PI ; CONXS, C, data(DATA) feature(FMAP) REST` from `CI` to `PI`, provided that `PI` is connected to the controller `C`.

\[
\text{eq updateEnvFeature}(C, \text{ envcon(CONXS,PORTS) REST}) \\
= \text{updateEnvFeature(CONXS, C, envcon(CONXS,PORTS) REST}) .
\]

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

\[
\text{eq updateEnvFeature}(\text{CONXS, C, REST}) = \text{ REST [owise]} .
\]
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 $C$ have 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 `_?_:_.

\[
\text{eq updateRespData}(C, \text{envcon(CONXS,PORTS) feature(FMAP) REST})
\]

\[
= \text{updateRespData(CONXS, C, FMAP, envcon(CONXS,PORTS) feature(FMAP) REST)} .
\]

\[
\text{ceq updateRespData((PI =>> CI) ; CONXS, C, FMAP, data(DATA) REST)}
\]

\[
= \text{updateRespData(CONXS, C, FMAP, data(DATA') REST')}
\]

\[
\text{if validTarget(PI, C, REST)}
\]

\[
/\ E \ # B : B' := \text{FMAP[PI]}
\]

\[
/\ E' \ # B' := \text{DATA[CI]}
\]

\[
/\ \text{DATA'} := \text{insert(CI, (B ? E : E') # (B or B')) , DATA}
\]

\[
/\ \text{REAT'} := \text{addConst(REST, B'' and (B or B'))} .
\]

\[
\text{eq updateRespData(CONXS, C, FMAP, REST) = REST [owise]}. 
\]

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 \(O((2^n)!/2^n)\) different symbolic execution results, according to non-deterministic orders of sampling and actuating events. The modular encoding can symbolically eliminate the interleavings of components, but cannot eliminate the nondeterminism in a component.

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

\[
(x_1 = u_1 \land \cdots \land x_n = u_n) \parallel t(x_1,\ldots,x_n),
\]

and it is semantically equivalent to \(t\) (i.e., \([\text{abs}(t)] = [[\text{true} \parallel t]] \) ?].

**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 \(\rho\) with \(t_1 = \rho t_2\) (i.e., \(t_1\) and \(t_2\) are equivalent up to renaming). In this case, the merged term is the constrained term

\[
(\phi_1 \lor \rho \phi_2) \parallel t_1.
\]

For example, \(y = 2 + x \parallel f(y)\) and \(z = 3 \parallel f(z)\) can be merged into the constrained term \((y = 2 + x \lor y = 3) \parallel f(y)\). We can easily show the following proposition that ensures the soundness and completeness of our method.
Algorithm 1: Semantic operation $f$ with state merging

Input: A constrained object $\phi \parallel t$

Output: A set of constrained objects

1. $\text{post} \leftarrow \{ (\phi' \parallel t') \mid f(\phi \parallel t) \sim \phi' \parallel t' \}$;
2. $\text{post} \leftarrow \{ (\phi' \land \psi \parallel t'') \mid (\psi \parallel t'') = \text{abs}(t'), \phi' \parallel t' \in \text{post} \}$;
3. while $\exists (\phi_1 \parallel t_1), (\phi_2 \parallel t_2) \in \text{post}$ with mergeable $t_1$ and $t_2$ do
4. \hspace{1em} $\varphi \parallel u$ is a merged term of $\phi_1 \parallel t_1$ and $\phi_2 \parallel t_2$;
5. \hspace{2em} $\text{post} \leftarrow (\text{post} \cup \{ \varphi \parallel u \}) \setminus \{ \phi_1 \parallel t_1, \phi_2 \parallel t_2 \}$;
6. return $\text{post}$;

Proposition 1. $\llbracket (\phi_1 \lor \rho \phi_2) \parallel t_1 \rrbracket = \llbracket \phi_1 \parallel t_1 \rrbracket \cup \llbracket \phi_2 \parallel t_2 \rrbracket$

Proof. By definition (Section 2), $u \in \llbracket (\phi_1 \lor \rho \phi_2) \parallel t_1 \rrbracket$ iff there is a substitution $\theta$ such that $u = \theta t_1$ and $T \models \theta(\phi_1 \lor \rho \phi_2)$. Because $t_1 = \rho t_2$, $u = \theta pt_2$. Also, $T \models \theta \phi_1$ or $T \models \theta \rho \phi_2$. Thus, one of the following cases must hold: (i) $u = \theta t_1$ and $T \models \theta \phi_1$, or (ii) $u = \theta pt_2$ and $T \models \theta \rho \phi_2$. By definition, $u \in \llbracket \phi_1 \parallel t_1 \rrbracket$ or $u \in \llbracket \rho (\phi_2 \parallel t_2) \rrbracket$. Because $\rho$ is a renaming substitution, $u \in \llbracket \rho (\phi_2 \parallel t_2) \rrbracket$ iff $u \in \llbracket \phi_2 \parallel t_2 \rrbracket$. Consequently, $\llbracket (\phi_1 \lor \rho \phi_2) \parallel t_1 \rrbracket = \llbracket \phi_1 \parallel t_1 \rrbracket \cup \llbracket \phi_2 \parallel t_2 \rrbracket$.

Algorithm 1 shows the new “merging” operation. It collects all the execution results by executeStep and merges all mergeable results. For our HYBRID-SYNCHAADL 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 [35], but has been generalized to deal with arbitrary constrained objects in HYBRID-SYNCHAADL.

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

```plaintext
ceq symAbs(E1, E2, GEN)
   = (X, X === E1, X === E2, GEN')
if (X,GEN') := gen(GEN,type(E1)) \land type(E1) == type(E2)
```

We define $\text{symAbs}$ 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 $\text{loc}(r)$ and data contents of the form $e \# b$ in our semantics. Using $\text{symAbs}$ for SMT expressions described above, we can easily define $\text{symAbs}$ for these cases as follows.

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

\[
\text{eq} \quad \text{execTransMerge}(\text{BCF}) = \text{symMerge}(\text{collectResults('execTrans[upTerm(BCF)])))}.
\]
\[
\text{eq} \quad \text{execEnvMerge}(\text{ECF}) = \text{symMerge}(\text{collectResults('execEnv[upTerm(ECF)])))}.
\]
\[
\text{ceq} \quad \text{symMerge}(\text{BTRS}, \text{GEN}) = \text{symMerge}(\text{BTRS}, \text{GEN}) \quad \text{if} \quad \text{GEN} := \maxGen(\text{BTRS}) .
\]
\[
\text{ceq} \quad \text{symMerge}((\text{ML} | \text{MFMAP} | \text{MDATA} | \text{MCS} | \text{GEN}) ;; \text{BTRS}, \text{GEN}) = \text{symMerge}((\text{ML} | \text{MFMAP} | \text{MDATA} | \text{MCS} | \text{GEN}) ;; \text{BTRS}, \text{GEN}) \quad \text{if} \quad \text{GEN} := \maxGen(\text{BTRS}) .
\]

### 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`.)

\[
\text{ceq} \quad [[ \text{COND} ]] \quad \text{OBJ} = \text{B and B'}.
\]

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.

\[
\text{eq} \quad \text{normal(E)} = \text{normal(nil, E)} .
\]
\[
\text{eq} \quad \text{normal(PATH, PATH' | E)} = \text{normal(PATH . PATH', E)} .
\]
\[
\text{eq} \quad \text{normal(PATH, VAR)} = \text{PATH . VAR} .
\]
\[
\text{eq} \quad \text{normal(PATH, VALUE)} = \text{VALUE} .
\]
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.

Reachability and invariant properties in our property specification language correspond to Maude’s search command. A reachability property of the form $\varphi_{init} \Rightarrow \varphi_{goal}$ in time $\tau_{bound}$ corresponds to the following search command to find its witness (i.e., the property holds if the search command finds a solution), where $N_{bound}$ is the quotient of $\tau_{bound}$ by the period of the model, and initState denotes the term representation of the entire model:

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

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

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

Example 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.
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 [7, 34, 44]), and rendezvous and formation control of distributed drones. Many variants of these models are considered: different numbers of components, different sampling and actuating times, different control logics, different continuous 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. The models and the experimental results are available in [1].

7.1 Comparison with CPS Analysis Tools

We compare the performance of HYBRIDSynchAADL’s symbolic analysis with four reachability analysis tools for hybrid automata, HyComp [24], SpaceEx [30], Flow* [23], and dReach [6]. For each model, we use two kinds of controllers with different complexity, as most of the other tools time out for more complex models. We consider two invariant properties for each model: \( \text{Inv}_\top \), which holds, and \( \text{Inv}_\bot \), which does not hold.

To use the other tools, we have “encoded” the synchronous designs of the HYBRIDSynchAADL models as networks of hybrid automata. Each component is modeled as a hybrid automaton with three modes: starting a new round, sampling, and controller transition/actuation. The behavior of a controller is encoded as single jumps. We use flat hybrid automata (obtained by HYST [16]) for Flow* and dReach, which do not support networks of hybrid automata.

For example, a component of “Drone Rendezvous” with single-integrator dynamics is modeled as a hybrid automaton in Figure 22. It begins with mode NewRound and local time \( t = 0 \). The value of \( x \) is sampled when a jump from NewRound to Sampling happens, and a new velocity is set when a jump from Sampling to Actuation happens. The behavior of the controller component is specified as the condition \( \bigwedge_{i=1}^{3} ((d_i = 1 \leq x_{in} - x_{sampled} \leq d_i) \rightarrow vel_x := c_i) \),
where $x_{in}$ is an input variable, and $c_i$ and $d_i$, $i = 1, 2, 3$, are real constants. Finally, all components are synchronized when a jump from `Actuation` to `NewRound` takes place, where the sampled value $x_{sampled}$ is available to the connected drone component in the next iteration. Notice that this automaton models the behavior of a `HybridSynchAADL` environment component in Sec 6.7.

We measure the execution times for analyzing the invariant properties up to bound 500 ms, with a timeout of 60 minutes. For SpaceEx, we use the PHAVer scenario for linear dynamics, and the STC scenario for nonlinear polynomial dynamics. For Flow*, we use adaptive steps with minimum 0.001 and maximum 0.005, TM orders 1 (for single) and 2 (for double). We use the default precision for dReach, and BMC for HyComp.

The experimental results are summarized in Table 1 and Table 2 as execution times (seconds) over time bounds ($B \cdot 100$ ms), with $N$ the number of components. The results for double-integrator dynamics (where control input is given by acceleration instead of velocity) do not include HyComp, which does not support nonlinear polynomial dynamics. Table 1 shows the largest time bound and time up to $B = 5$, for which the tool could prove the absence of counterexamples for $Inv^\top$. Often, tools timed out T/O when trying to verify that $Inv^\top$ holds up to time bound 500. Table 2 shows the smallest bound for which the tool found counterexamples for $Inv^\bot$.

As seen, `HybridSynchAADL` outperforms the other tools in most cases, in particular for complex models with larger $N$. Consider, e.g., “Rend (single)” with complex control logic for $N = 4$. For $Inv^\top$, `HybridSynchAADL` needs 33.2 seconds for $B = 5$, whereas SpaceEx needs 470.2 seconds for $B = 1$ and timed out for $B > 1$. For $Inv^\bot$, `HybridSynchAADL` found a counterexample at $B = 4$ in 66.7 seconds, whereas all other tools timed out. Flow* occasionally found (spurious) counterexamples at smaller bounds, because of over-approximation by the Taylor model flowpipe construction.

---

10 For STC, we use the following parameters: octagonal template directions, and time horizon 0.08. we also apply convex hull set aggregation.
### Table 1. Comparison of HybridSynchAADL and the other tools for $\text{Inv}_\tau$

<table>
<thead>
<tr>
<th>Model</th>
<th>Tool</th>
<th>Simplified Control Logic</th>
<th>Complex Control Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$N = 2$</td>
<td>$N = 3$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time $B$</td>
<td>Time $B$</td>
</tr>
<tr>
<td>Read (single)</td>
<td>HSADDL</td>
<td>2.0 5</td>
<td>3.9 5</td>
</tr>
<tr>
<td></td>
<td>HyComp</td>
<td>0.8 5</td>
<td>4.0 5</td>
</tr>
<tr>
<td></td>
<td>SpaceEx</td>
<td>8.0 5</td>
<td>2230.3 3</td>
</tr>
<tr>
<td></td>
<td>dReach</td>
<td>1382.7 3</td>
<td>107.1 1</td>
</tr>
<tr>
<td></td>
<td>Flow*</td>
<td>3552.8 4</td>
<td>2725.5 2</td>
</tr>
<tr>
<td>Form (single)</td>
<td>HSADDL</td>
<td>3.0 5</td>
<td>7.3 5</td>
</tr>
<tr>
<td></td>
<td>HyComp</td>
<td>13.3 5</td>
<td>41.3 5</td>
</tr>
<tr>
<td></td>
<td>SpaceEx</td>
<td>91.9 2</td>
<td>2.8 1</td>
</tr>
<tr>
<td></td>
<td>dReach</td>
<td>139.0 1</td>
<td>T/O -</td>
</tr>
<tr>
<td></td>
<td>Flow*</td>
<td>1464.7 2</td>
<td>873.4 1</td>
</tr>
<tr>
<td>Thermostat</td>
<td>HSADDL</td>
<td>2.7 5</td>
<td>4.7 5</td>
</tr>
<tr>
<td></td>
<td>HyComp</td>
<td>1.6 5</td>
<td>8.5 5</td>
</tr>
<tr>
<td></td>
<td>SpaceEx</td>
<td>2.3 5</td>
<td>696.4 3</td>
</tr>
<tr>
<td></td>
<td>dReach</td>
<td>341.6 3</td>
<td>57.5 1</td>
</tr>
<tr>
<td></td>
<td>Flow*</td>
<td>3196.4 5</td>
<td>1240.7 2</td>
</tr>
<tr>
<td>Water Tank</td>
<td>HSADDL</td>
<td>1.3 5</td>
<td>2.5 5</td>
</tr>
<tr>
<td></td>
<td>HyComp</td>
<td>0.5 5</td>
<td>6.0 5</td>
</tr>
<tr>
<td></td>
<td>SpaceEx</td>
<td>0.8 5</td>
<td>877.1 5</td>
</tr>
<tr>
<td></td>
<td>dReach</td>
<td>1080.8 3</td>
<td>52.8 1</td>
</tr>
<tr>
<td></td>
<td>Flow*</td>
<td>1070.1 3</td>
<td>96.4 1</td>
</tr>
<tr>
<td>Read (double)</td>
<td>HSADDL</td>
<td>3.7 4</td>
<td>37.8 4</td>
</tr>
<tr>
<td></td>
<td>SpaceEx</td>
<td>1147.6 3</td>
<td>81.1 1</td>
</tr>
<tr>
<td></td>
<td>dReach</td>
<td>2156.2 3</td>
<td>274.3 1</td>
</tr>
<tr>
<td></td>
<td>Flow*</td>
<td>2325.2 2</td>
<td>230.1 1</td>
</tr>
<tr>
<td>Form (double)</td>
<td>HSADDL</td>
<td>2.2 3</td>
<td>1.7 3</td>
</tr>
<tr>
<td></td>
<td>SpaceEx</td>
<td>70.8 1</td>
<td>T/O -</td>
</tr>
<tr>
<td></td>
<td>dReach</td>
<td>360.4 1</td>
<td>T/O -</td>
</tr>
<tr>
<td></td>
<td>Flow*</td>
<td>116.8 1</td>
<td>3411.5 1</td>
</tr>
</tbody>
</table>

### 7.2 Analyzing Invariant Properties

We evaluate the power of HybridSynchAADL for analyzing bounded invariants. We measure the time taken to find counterexamples, using HybridSynchAADL’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 $\epsilon$ the maximal clock skew. All models have period 100 (milliseconds). The timeout is 20 minutes.
Table 2. Comparison of HybridSynchAADL and the other tools for Inv

<table>
<thead>
<tr>
<th>Model</th>
<th>Tool</th>
<th>Simplified Control Logic</th>
<th>Complex Control Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>N = 2</td>
<td>N = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time B</td>
<td>Time B</td>
</tr>
<tr>
<td>Rend</td>
<td>HSADDL</td>
<td>2.4</td>
<td>4.2</td>
</tr>
<tr>
<td>(single)</td>
<td>HyComp</td>
<td>8.9</td>
<td>11.5</td>
</tr>
<tr>
<td></td>
<td>SpaceX</td>
<td>5.1</td>
<td>267.6</td>
</tr>
<tr>
<td></td>
<td>dReach</td>
<td>T/O</td>
<td>T/O</td>
</tr>
<tr>
<td></td>
<td>Flow*</td>
<td>167.3</td>
<td>380.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>N = 2</td>
<td>N = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time B</td>
<td>Time B</td>
</tr>
<tr>
<td>Form</td>
<td>HSADDL</td>
<td>15.5</td>
<td>2.5</td>
</tr>
<tr>
<td>(single)</td>
<td>HyComp</td>
<td>2.6</td>
<td>43.1</td>
</tr>
<tr>
<td></td>
<td>SpaceX</td>
<td>T/O</td>
<td>T/O</td>
</tr>
<tr>
<td></td>
<td>dReach</td>
<td>T/O</td>
<td>T/O</td>
</tr>
<tr>
<td></td>
<td>Flow*</td>
<td>18.5</td>
<td>2.6</td>
</tr>
<tr>
<td>Thermostat</td>
<td>HSADDL</td>
<td>1.8</td>
<td>3.1</td>
</tr>
<tr>
<td></td>
<td>HyComp</td>
<td>2.6</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td>SpaceX</td>
<td>T/O</td>
<td>T/O</td>
</tr>
<tr>
<td></td>
<td>dReach</td>
<td>T/O</td>
<td>T/O</td>
</tr>
<tr>
<td></td>
<td>Flow*</td>
<td>1.0</td>
<td>5.7</td>
</tr>
<tr>
<td>Water Tank</td>
<td>HSADDL</td>
<td>1.4</td>
<td>16.3</td>
</tr>
<tr>
<td></td>
<td>HyComp</td>
<td>0.1</td>
<td>3.1</td>
</tr>
<tr>
<td></td>
<td>SpaceX</td>
<td>0.2</td>
<td>7.4</td>
</tr>
<tr>
<td></td>
<td>dReach</td>
<td>T/O</td>
<td>T/O</td>
</tr>
<tr>
<td></td>
<td>Flow*</td>
<td>1.0</td>
<td>5.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Sampling Actuation ϵ</th>
<th>Invariant property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thermostat</td>
<td>20 ~ 30 60 ~ 70 5</td>
<td>temperatures are between 20 and 50</td>
</tr>
<tr>
<td>Water tank</td>
<td>30 ~ 40 70 ~ 80 5</td>
<td>water levels are above 30</td>
</tr>
<tr>
<td>Rendezvous</td>
<td>30 ~ 50 60 ~ 80 10</td>
<td>distance between drones greater than 0.5</td>
</tr>
<tr>
<td>Formation</td>
<td>30 ~ 50 60 ~ 80 10</td>
<td>distance between drones greater than 0.3</td>
</tr>
</tbody>
</table>

The experimental results are summarized in Fig. 23 as execution times (seconds, log scale) over time bounds (ms), with N the number of components. 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 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.

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 98 seconds, while randomized simulation timed out. For bound 400, randomized simulation found a counterexample in less than 12 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. 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.

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 57,200 and the number of SMT calls is 438. Without merging, the size of the constraints is about 38 million and the number of SMT calls is more than 260 thousand.

---

11 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. 23. HYBRIDSYNCHAADL portfolio analysis.
behaviors. The latter includes reachability analysis tools for hybrid automata whereas most formal tools are strong at analyzing either discrete or continuous models with the unrealistic assumptions. For the experiment, we consider highly simplified distributed models with the following assumptions: (i) all clocks are perfectly synchronized; (ii) there is no network delay; (iii) controllers (nondeterministically) choose one of the predefined values for sampling and actuating times; and (iv) controllers take zero time to perform transitions. We use floating-point arithmetic to compute the continuous dynamics of the environments given by polynomials.

Figure 25 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.

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 [11]. We measured the execution times for generating all reachable concrete states up to bounds in distributed asynchronous models. We set a timeout of 360 minutes.

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

Figure 25 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...
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 \cite{21,22,33,36}, GALS \cite{32,42}, approximate synchrony \cite{26}, time-triggered architectures \cite{47,48}, virtual synchrony \cite{11,40}, 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 \cite{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 \cite{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.

Hybrid Systems in AADL. The Hybrid Annex for AADL \cite{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 \cite{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 \cite{19}, 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 \cite{43}, and an AADL sublanguage, called AADL+, where continuous behaviors can be defined using stochastic differential equations is given in \cite{37}. Both approaches come with some kind of operational semantics and simulation, but with no formal analysis support.

PALS and AADL. Synchronous AADL \cite{10,14} and its multi-rate extension \cite{13} 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 \cite{11} 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 [11], 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 and 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 OSAITE 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 [31] and Flow* [23] 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 [8].

References

A HybridSynchAADL Models

This section shows the entire AADL codes of benchmark models. There are six benchmark models: (1) drone rendezvous with single-integrator dynamics, (2) drone rendezvous with double-integrator dynamics, (3) drone formation with single-integrator dynamics, (4) drone formation with double-integrator dynamics, (5) networked thermostat, and (6) networked watertank. The below AADL codes represent the benchmark models for $N = 4$. You can easily get AADL codes of the models for $N = 2$ and $3$ by modifying the Top-Level component.

A.1 Drone Rendezvous with Single-Integrator Dynamics

```aadl
--- Top-Level component
package FourDronesSystem
public
  with Drone;
  with RefDrone;
  with Data_Model;
  with Hybrid_SynchAADL;

system FourDronesSystem
  properties
    Hybrid_SynchAADL::Synchronous => true;
    Hybrid_SynchAADL::Max_Clock_Deviation => 5ms;
    Period => 100ms;
    Classifier_Substitution_Rule => Type_Extension;
end FourDronesSystem;

system implementation FourDronesSystem.rev1D
subcomponents
  drone1: system Drone::RevuDrone.impl;
  drone2: system Drone::RevuDrone.impl;
  drone3: system Drone::RevuDrone.impl;
  drone4: system Drone::RevuDrone.impl;
connections
  d12x: port drone1.outX -> drone2.inX;
  d23x: port drone2.outX -> drone3.inX;
  d34x: port drone3.outX -> drone4.inX;
```
d41x: \textbf{port} drone4.outX \rightarrow drone1.inX;

\textbf{properties}
Data\_Model::\textbf{Initial\_Value} => ("param") \textbf{applies to}
drone1.environment.x, drone2.environment.x,
drone3.environment.x, drone4.environment.x;
Data\_Model::\textbf{Initial\_Value} => ("0") \textbf{applies to}
drone1.environment.dotx, drone2.environment.dotx,
drone3.environment.dotx, drone4.environment.dotx;

Timing => \textbf{Delayed} \textbf{applies to}
d12x, d23x, d34x, d41x;

Hybrid\_SynchAADL::\textbf{Sampling\_Time} => 20ms .. 30ms \textbf{applies to}
drone1.drone, drone2.drone, drone3.drone, drone4.drone;

Hybrid\_SynchAADL::\textbf{Response\_Time} => 60ms .. 80ms \textbf{applies to}
drone1.drone, drone2.drone, drone3.drone, drone4.drone;

\textbf{end} FourDronesSystem.revu1D;

--- There is additional data subcomponents in Drone component for the y-axis.
--- You need to connect those components and initialize them.

\textbf{system implementation} FourDronesSystem.revu2D
\hspace{1em} \textbf{extends} FourDronesSystem.revu1D

\textbf{subcomponents}
drone1: \textbf{refined} \textbf{to system} Drone::RevuDrone2D.impl;
drone2: \textbf{refined} \textbf{to system} Drone::RevuDrone2D.impl;
drone3: \textbf{refined} \textbf{to system} Drone::RevuDrone2D.impl;
drone4: \textbf{refined} \textbf{to system} Drone::RevuDrone2D.impl;

\textbf{connections}
d12y: \textbf{port} drone1.outY \rightarrow drone2.inY;
d23y: \textbf{port} drone2.outY \rightarrow drone3.inY;
d34y: \textbf{port} drone3.outY \rightarrow drone4.inY;
d41y: \textbf{port} drone4.outY \rightarrow drone1.inY;

\textbf{properties}
Data\_Model::\textbf{Initial\_Value} => ("param") \textbf{applies to}
drone1.environment.y, drone2.environment.y,
drone3.environment.y, drone4.environment.y;
Data\_Model::\textbf{Initial\_Value} => ("0") \textbf{applies to}
drone1.environment.doty, drone2.environment.doty,
drone3.environment.doty, drone4.environment.doty;

Timing => \textbf{Delayed} \textbf{applies to}
d12y, d23y, d34y, d41y;
\textbf{end} FourDronesSystem.revu2D;
\textbf{end} FourDronesSystem;

--- Drone component

\textbf{package} Drone
\textbf{public}
\hspace{1em} \textbf{with} Environment;
with Base_Types;
with Data_Model;
with DroneControl;

system RevuDrone
features
  inX: in data port Base_Types::Float;
  outX: out data port Base_Types::Float
  {Data_Model::Initial_Value => ("0");};
end RevuDrone;

system implementation RevuDrone.impl
subcomponents
  drone: system DroneControl::RevuDroneControl.impl;
  environment: system Environment::Environment.impl;
connections
  C1: port drone.outX -> outX;
  C2: port inX -> drone.inX;
  C3: port drone.velX -> environment.velX;
  C4: port environment.currX -> drone.currX;
end RevuDrone.impl;

system RevuDrone2D extends RevuDrone
features
  inY: in data port Base_Types::Float;
  outY: out data port Base_Types::Float
  {Data_Model::Initial_Value => ("0");};
properties
  Classifier_Substitution_Rule => Type_Extension;
end RevuDrone2D;

system implementation RevuDrone2D.impl extends RevuDrone.impl
subcomponents
  drone: refined to system DroneControl::RevuDrone2DControl.impl;
  environment: refined to system Environment::Environment2D.impl;
connections
  C5: port drone.outY -> outY;
  C6: port inY -> drone.inY;
  C7: port drone.velY -> environment.velY;
  C8: port environment.currY -> drone.currY;
end RevuDrone2D.impl;

end Drone;

--- Controller component
package DroneControl
public
  with Base_Types;
  with DroneSpec;
  with Data_Model;

system RevuDroneControl
features
currX: in data port Base_Types::Float;
inX: in data port Base_Types::Float;
outX : out data port Base_Types::Float
    {Data_Model::Initial_Value => ("0");};
velX : out data port Base_Types::Float
    {Data_Model::Initial_Value => ("0");};
end RevuDroneControl;

system implementation RevuDroneControl.impl
subcomponents
droneProc: process RevuDroneControlProc.impl;
connections
C1: port currX -> droneProc.currX;
C2: port inX -> droneProc.inX;
C3: port droneProc.velX -> velX;
C4: port droneProc.outX -> outX;
end RevuDroneControl.impl;

system RevuDrone2DControl extends RevuDroneControl
features
currY : in data port Base_Types::Float;
inY : in data port Base_Types::Float;
outY : out data port Base_Types::Float
    {Data_Model::Initial_Value => ("0");};
velY : out data port Base_Types::Float
    {Data_Model::Initial_Value => ("0");};
properties
Classifier_Substitution_Rule => Type_Extension;
end RevuDrone2DControl;

system implementation RevuDrone2DControl.impl extends RevuDroneControl.impl
subcomponents
droneProc: refined to process RevuDrone2DControlProc.impl;
connections
C5: port currY -> droneProc.currY;
C6: port inY -> droneProc.inY;
C7: port droneProc.velY -> velY;
C8: port droneProc.outY -> outY;
end RevuDrone2DControl.impl;

process RevuDroneControlProc
features
currX: in data port Base_Types::Float;
inX: in data port Base_Types::Float;
outX : out data port Base_Types::Float;
velX : out data port Base_Types::Float;
end RevuDroneControlProc;

process implementation RevuDroneControlProc.impl
subcomponents
droneThread: thread RevuDroneControlThread.impl;
connections
C1: port currX -> droneThread.currX;
C2: port inX -> droneThread.inX;
C3: port droneThread.velX -> velX;
C4: port droneThread.outX -> outX;
end RevuDroneControlThread;

process RevuDrone2DControlProc extends RevuDroneControlProc
features
currY: in data port Base_Types::Float;
inY: in data port Base_Types::Float;
outY : out data port Base_Types::Float;
velY : out data port Base_Types::Float;
properties
Classifier_Substitution_Rule => Type_Extension;
end RevuDrone2DControlProc;

process implementation RevuDrone2DControlProc.impl
extends RevuDrone2DControlProc.impl
subcomponents
  droneThread: refined to thread RevuDrone2DControlThread.impl;
connections
  C5: port currY -> droneThread.currY;
  C6: port inY -> droneThread.inY;
  C7: port droneThread.velY -> velY;
  C8: port droneThread.outY -> outY;
end RevuDrone2DControlProc.impl;

thread RevuDroneControlThread
features
currX: in data port Base_Types::Float;
inX: in data port Base_Types::Float;
outX : out data port Base_Types::Float;
velX : out data port Base_Types::Float;
properties
Dispatch_Protocol => Periodic;
end RevuDroneControlThread;

thread implementation RevuDroneControlThread.impl

annex behavior_specification (**
variables
  nx : Base_Types::Float;
states
  init : initial complete state;
  exec, output : state;
transitions
  init -[on dispatch]-> exec;
  exec -[abs(currX - inX) < 0.1]-> output {
    velX := 0
  };

exec -[otherwise]-> output {
    nx := - #DroneSpec::A * (currX - inX);
    if (nx > 1) velX := 2
    elsif (nx > 0.1) velX := 1
    elsif (nx > 0) velX := 0
    elsif (nx > -0.1) velX := -1
    else velX := -2
    end if
};
output -[ ]-> init {
    outX := currX
};
**};
end RevuDroneControlThread.impl;

--- RevuDrone thread with complex control logic has 5 transitions for each axis.

thread RevuDrone2DControlThread extends RevuDroneControlThread
features
    currY: in data port Base_Types::Float;
    inY: in data port Base_Types::Float;
    outY : out data port Base_Types::Float;
    velY : out data port Base_Types::Float;
end RevuDrone2DControlThread;
thread implementation RevuDrone2DControlThread.impl
    extends RevuDroneControlThread.impl
    annex behavior_specification (**
        variables
            nx : Base_Types::Float;
            ny : Base_Types::Float;
        states
            init : initial complete state;
            exec, output : state;
        transitions
            init -[on dispatch]-> exec;
            exec -[abs(currX - inX) < 0.1 and abs(currY - inY) < 0.1]-> output {
                velX := 0; velY := 0
            };
            exec -[otherwise]-> output {
                nx := - #DroneSpec::A * (currX - inX);
                if (nx > 1) velX := 2
                elsif (nx > 0.1) velX := 1
                elsif (nx > 0) velX := 0
                elsif (nx > -0.1) velX := -1
                else velX := -2
                end if;
                ny := - #DroneSpec::A * (currY - inY);
                if (ny > 1) velY := 2
                elsif (ny > 0.1) velY := 1
                elsif (ny > 0) velY := 0
                elsif (ny > -0.1) velY := -1
                else velY := -2
                end if;
            };
        })
end RevuDrone2DControlThread.
};

output -[ ]-> init {
  outX := currX;
  outY := currY
};

**};
end RevuDrone2DControlThread.impl;

--- Environment component
package Environment
public
  with Hybrid_SynchAADL;
  with Base_Types;
  with Data_Model;

system Environment
  features
    currX : out data port Base_Types::Float
      {Data_Model::Initial_Value => ("0");};
    velX : in data port Base_Types::Float;
  properties
    Hybrid_SynchAADL::isEnvironment => true;
end Environment;

system implementation Environment.impl
  subcomponents
    x : data Base_Types::Float;
    dotx : data Base_Types::Float;
  connections
    C1: port velX -> dotx;
    C2: port x -> currX;
  properties
    Hybrid_SynchAADL::ContinuousDynamics =>
      "x(t) = ((0.001) * dotx * t) + x(0);";
end Environment.impl;

system Environment2D extends Environment
  features
    currY : out data port Base_Types::Float;
    velY : in data port Base_Types::Float;
end Environment2D;

system implementation Environment2D.impl
  subcomponents
    y : data Base_Types::Float;
    doty : data Base_Types::Float;
  connections
    C3: port velY -> doty;
    C4: port y -> currY;
  properties
    Hybrid_SynchAADL::ContinuousDynamics =>
      "x(t) = ((0.001) * dotx * t) + x(0);";
\[ y(t) = (0.001) \cdot \dot{x} \cdot t + y(0); \]

\textbf{A.2 Drone Formation with Single-Integrator Dynamics}

--- Top-Level component

\textbf{package} FourDronesSystem
\textbf{public}
\textbf{with} Drone;
\textbf{with} RefDrone;
\textbf{with} Data_Model;
\textbf{with} Hybrid_SynchAADL;

\textbf{system} FourDronesSystem
\textbf{properties}
\hspace{1em} Hybrid_SynchAADL::Synchronous \Rightarrow true;
\hspace{1em} Hybrid_SynchAADL::Max_Clock_Deviation \Rightarrow 5ms;
\hspace{1em} Period \Rightarrow 100ms;
\hspace{1em} Classifier_Substitution_Rule \Rightarrow Type_Extension;
\textbf{end} FourDronesSystem;

\textbf{system implementation} FourDronesSystem.formID
\textbf{extends} FourDronesSystem.revu1D

--- Because drone formation extends drone rendezvous, it needs \textbf{to} only refine four drone \textbf{subcomponents}.
--- Unlike drone rendezvous, drone formation has a reference drone --- The other drones move along with the reference drone.

\textbf{subcomponents}
\hspace{1em} drone1: \textbf{refined to} system Drone::FormDrone.impl;
\hspace{1em} drone2: \textbf{refined to} system Drone::FormDrone.impl;
\hspace{1em} drone3: \textbf{refined to} system Drone::FormDrone.impl;
\hspace{1em} drone4: \textbf{refined to} system Drone::FormDrone.impl;
\hspace{1em} refDrone: \textbf{system} RefDrone::RefDroneControl.formation;

\textbf{connections}
\hspace{1em} r1x: \textbf{port} refDrone.outX \rightarrow drone1.refX;
\hspace{1em} r2x: \textbf{port} refDrone.outX \rightarrow drone2.refX;
\hspace{1em} r3x: \textbf{port} refDrone.outX \rightarrow drone3.refX;
\hspace{1em} r4x: \textbf{port} refDrone.outX \rightarrow drone4.refX;

\textbf{properties}
\hspace{1em} Data_Model::Initial_Value \Rightarrow (-0.5) \textbf{applies to}
\hspace{1em} drone1.drone.droneProc.droneThread.offsetX,
\hspace{1em} drone3.drone.droneProc.droneThread.offsetX;
\hspace{1em} Data_Model::Initial_Value \Rightarrow (0) \textbf{applies to}
\hspace{1em} drone2.drone.droneProc.droneThread.offsetX,
\hspace{1em} drone4.drone.droneProc.droneThread.offsetX;
\hspace{1em} Data_Model::Initial_Value \Rightarrow (\text{param}) \textbf{applies to}
\hspace{1em} refDrone.env.x;
Timing => Delayed applies to
    r1x, r2x, r3x, r4x;

Hybrid_SynchAADL::Sampling_Time => 20ms .. 30ms
    applies to refDrone.refDrone;

Hybrid_SynchAADL::Response_Time => 60ms .. 80ms
    applies to refDrone.refDrone;

end FourDronesSystem.form1D;

system implementation FourDronesSystem.form2D
    extends FourDronesSystem.revu2D

subcomponents
    drone1: refined to system Drone::FormDrone2D.impl;
    drone2: refined to system Drone::FormDrone2D.impl;
    drone3: refined to system Drone::FormDrone2D.impl;
    drone4: refined to system Drone::FormDrone2D.impl;
    refDrone: system RefDrone::RefDrone2DControl.formation;

connections
    r1y: port refDrone.outY -> drone1.refY;
    r2y: port refDrone.outY -> drone2.refY;
    r3y: port refDrone.outY -> drone3.refY;
    r4y: port refDrone.outY -> drone4.refY;

properties
    Data_Model::Initial_Value => ("0.5") applies to
        drone1.drone.droneProc.droneThread.offsetY,
        drone2.drone.droneProc.droneThread.offsetY;
    Data_Model::Initial_Value => ("-0.5") applies to
        drone1.drone.droneProc.droneThread.offsetX,
        drone2.drone.droneProc.droneThread.offsetX;
    Data_Model::Initial_Value => ("0") applies to
        drone3.drone.droneProc.droneThread.offsetX,
        drone3.drone.droneProc.droneThread.offsetY,
        drone4.drone.droneProc.droneThread.offsetX,
        drone4.drone.droneProc.droneThread.offsetY;
    Data_Model::Initial_Value => ("param") applies to
        refDrone.env.x;

Timing => Delayed applies to
    r1x, r2x, r3x, r4x, r1y, r2y, r3y, r4y;

Hybrid_SynchAADL::Sampling_Time => 20ms .. 30ms
    applies to refDrone.refDrone;

Hybrid_SynchAADL::Response_Time => 60ms .. 80ms
    applies to refDrone.refDrone;

end FourDronesSystem.form2D;
end FourDronesSystem;

--- Drone component
--- Each drone component uses the inherited environment component
--- from drone rendezouvs component
package Drone
public
with Environment;
with Base_Types;
with Data_Model;
with DroneControl;

system FormDrone extends RevuDrone
features
  refX: in data port Base_Types::Float;
end FormDrone;

system implementation FormDrone.impl extends RevuDrone.impl
subcomponents
  drone: refined to system DroneControl::FormDroneControl.impl;
connections
  C5: port refX -> drone.refX;
end FormDrone.impl;

system FormDrone2D extends RevuDrone2D
features
  refX: in data port Base_Types::Float;
  refY: in data port Base_Types::Float;
end FormDrone2D;

system implementation FormDrone2D.impl extends RevuDrone2D.impl
subcomponents
  drone: refined to system DroneControl::FormDrone2DControl.impl;
connections
  C9: port refX -> drone.refX;
  C10: port refY -> drone.refY;
end FormDrone2D.impl;
end Drone;

--- Because drone formation extends drone rendezvous,
--- You need to connect ports between the reference drone and the original drone
package DroneControl
public
with Base_Types;
with DroneSpec;
with Data_Model;

system FormDroneControl extends RevuDroneControl
features
  refX: in data port Base_Types::Float;
properties
  Classifier_Substitution_Rule => Type_Extension;
FormDroneControl.impl extends RevuDroneControl.impl
subcomponents
  droneProc: refined to process FormDroneControlProc.impl;
connections
  C5: port refX -> droneProc.refX;
end FormDroneControl.impl;

system FormDrone2DControl extends RevuDrone2DControl
features
  refX: in data port Base_Types::Float;
  refY: in data port Base_Types::Float;
properties
  Classifier_Substitution_Rule => Type_Extension;
end FormDrone2DControl;
system implementation FormDrone2DControl.impl
  extends RevuDrone2DControl.impl
subcomponents
  droneProc: refined to process FormDrone2DControlProc.impl;
connections
  C9: port refX -> droneProc.refX;
  C10: port refY -> droneProc.refY;
end FormDrone2DControl.impl;

process FormDroneControlProc extends RevuDroneControlProc
features
  refX: in data port Base_Types::Float;
properties
  Classifier_Substitution_Rule => Type_Extension;
end FormDroneControlProc;
process implementation FormDroneControlProc.impl
  extends RevuDroneControlProc.impl
subcomponents
  droneThread: refined to thread FormDroneControlThread.impl;
connections
  C5: port refX -> droneThread.refX;
end FormDroneControlProc.impl;

process FormDrone2DControlProc extends RevuDrone2DControlProc
features
  refX: in data port Base_Types::Float;
  refY: in data port Base_Types::Float;
properties
  Classifier_Substitution_Rule => Type_Extension;
end FormDrone2DControlProc;
process implementation FormDrone2DControlProc.impl
  extends RevuDrone2DControlProc.impl
subcomponents
    droneThread: refined to thread FormDrone2DControlThread.impl;
connections
    C9: port refX -> droneThread.refX;
    C10: port refY -> droneThread.refY;
end FormDrone2DControlProc.impl;

thread FormDroneControlThread extends RevuDroneControlThread
features
    refX: in data port Base_Types::Float;
properties
    Dispatch_Protocol => Periodic;
end FormDroneControlThread;
thread implementation FormDroneControlThread.impl
    extends RevuDroneControlThread.impl
subcomponents
    offsetX: data Base_Types::Float;
    refX0: data Base_Types::Float {Data_Model::Initial_Value => ("0");};
annex behavior_specification (**
    variables
        nx, refvX : Base_Types::Float;
    states
        init : initial complete state;
        exec, output : state;
    transitions
        init -[on dispatch]-> exec;
        exec -[abs(currX - inX) < 0.1]-> output {
            velX := 0
        };
        exec -[otherwise]-> output {
            refvX := (refX-refX0);
            nx := refvX - #DroneSpec::alpha * (currX - offsetX - refX)
                - #DroneSpec::A * (currX - offsetX - inX);
            if (nx > 1) velX := 2
            elsif (nx > -0.5) velX := 0
            else velX := -2
            end if
        };
        output -[ ]-> init {
            outX := currX - offsetX;
            refX0 := refX
        };
**);
end FormDroneControlThread.impl;

thread FormDrone2DControlThread extends RevuDrone2DControlThread
features
    refX: in data port Base_Types::Float;
refY: in data port Base_Types::Float;

properties
  Dispatch_Protocol => Periodic;
end FormDrone2DControlThread;

thread implementation FormDrone2DControlThread.impl
  extends RevuDrone2DControlThread.impl

subcomponents
  offsetX: data Base_Types::Float;
  offsetY: data Base_Types::Float;
  refX0: data Base_Types::Float (Data_Model::Initial_Value => ("0"));
  refY0: data Base_Types::Float (Data_Model::Initial_Value => ("0"));

annex behavior_specification (**

variables
  nx, refvX : Base_Types::Float;
  ny, refvY : Base_Types::Float;

states
  init : initial complete state;
  exec, output : state;

transitions
  init -[on dispatch]-> exec;
  exec -[abs(currX - inX) < 0.1 and abs(currY - inY) < 0.1]-> output {
    velX := 0;
    velY := 0
  };
  exec -[otherwise]-> output {
    refvX := (refX-refX0);
    nx := refvX - #DroneSpec::alpha * (currX - offsetX - refX)
         - #DroneSpec::A * (currX - offsetX - inX);
    refvY := (refX-refX0);
    ny := refvX - #DroneSpec::alpha * (currY - offsetY - refY)
         - #DroneSpec::A * (currY - offsetY - inY);
    if (nx > 1) velX := 2
      elsif (nx > 0.5) velX := 1
      elsif (nx > 0) velX := 0
      elsif (nx > -0.5) velX := -1
      else velX := -2
    end if;
    if (ny > 1) velY := 2
      elsif (ny > 0.5) velY := 1
      elsif (ny > 0) velY := 0
      elsif (ny > -0.5) velY := -1
      else velY := -2
    end if
  };
  output -[ ]-> init {
    outX := currX - offsetX;
    refX0 := refX;
    outY := currY - offsetY;
    refY0 := refY
  };

**};
end FormDrone2DControlThread.impl;
end DroneControl;

--- For drone formation, it needs to declare reference drone component for other drones to follow
package RefDrone
public
with Base_Types;
with Environment;
with Data_Model;

system RefDroneControl
features
  outX: out data port Base_Types::Float
    {Data_Model::Initial_Value => ("0");};
end RefDroneControl;

system implementation RefDroneControl.formation
subcomponents
  refDrone: system RefDrone.formation;
  env: system Environment::Environment.impl;
connections
  C1: port refDrone.velX -> env.velX;
  C2: port env.currX -> refDrone.currX;
  C3: port refDrone.outX -> outX;
end RefDroneControl.formation;


define system RefDrone
features
  velX: out data port Base_Types::Float;
end RefDrone;
end
system implementation RefDrone形成
subcomponents
  refProc: process refProc.formation;
connections
  C1: port refProc.velX -> velX;
  C2: port currX -> refProc.currX;
  C3: port refProc.outX -> outX;
end RefDrone.formation;

system RefDrone2D extends RefDrone
features
  velY : out data port Base_Types::Float;
  outY : out data port Base_Types::Float;
  currY : in data port Base_Types::Float;
properties
  Classifier_Substitution_Rule => Type_Extension;
end RefDrone2D;

system implementation RefDrone2D.formation extends RefDrone.formation
subcomponents
  refProc: refined to process RefProc2D.formation;
connections
  C4: port refProc.velY -> velY;
  C5: port currY -> refProc.currY;
  C6: port refProc.outY -> outY;
end RefDrone2D.formation;

process RefProc
features
  velX: out data port Base_Types::Float;
  outX : out data port Base_Types::Float;
  currX: in data port Base_Types::Float;
end RefProc;

process implementation RefProc.formation
subcomponents
  refThread: thread RefThread.formation;
connections
  C1: port refThread.velX -> velX;
  C2: port currX -> refThread.currX;
  C3: port refThread.outX -> outX;
end RefProc.formation;

process RefProc2D extends RefProc
features
  velY : out data port Base_Types::Float;
  outY : out data port Base_Types::Float;
currY: in data port Base_Types::Float;

properties
Classifier_Substitution_Rule => Type_Extension;
end RefProc2D;

process implementation RefProc2D.formation
extends RefProc.formation

subcomponents
refThread: refined to thread RefThread2D.formation;

connections
C4: port refThread.velY -> velY;
C5: port currY -> refThread.currY;
C6: port refThread.outY -> outY;
end RefProc2D.formation;

thread RefThread
features
velX: out data port Base_Types::Float;
outX: out data port Base_Types::Float;
currX: in data port Base_Types::Float;

properties
Dispatch_Protocol => Periodic;
end RefThread;

thread implementation RefThread.formation

subcomponents
nx: data Base_Types::Float
(Data_Model::Initial_Value => ("0");)

annex behavior_specification {**
states
init: initial complete state;
exec, output: state;

transitions
init -[on dispatch]-> exec;
exec -[nx = 0]-> output {
    nx := 1
};
exec -[nx = 1]-> output {
    nx := 2
};
exec -[nx = 2]-> output {
    nx := 0
};
output -[ ]-> init {
    velX := nx;
    outX := currX
};
**};
end RefThread.formation;

thread RefThread2D extends RefThread
features
  velY: out data port Base_Types::Float;
  outY: out data port Base_Types::Float;
  currY: in data port Base_Types::Float;
properties
  Dispatch_Protocol => Periodic;
end RefThread2D;

thread implementation RefThread2D.formation extends RefThread.formation
subcomponents
  ny: data Base_Types::Float {Data_Model::Initial_Value => ("0");}

annex behavior_specification (**
  states
    init: initial complete state;
    exec, output: state;
  transitions
    init -[ on dispatch ]-> exec;
    exec -[ nx = 0 ]-> output {
      nx := 1; ny := 1
    };
    exec -[ nx = 1 ]-> output {
      nx := 2; ny := 2
    };
    exec -[ nx = 2 ]-> output {
      nx := 0; ny := 0
    };
    output -[ ]-> init {
      velX := nx; velY := ny;
      outX := currX; outY := currY
    };
**};
end RefThread2D.formation;
end RefDrone;

A.3 Drone Rendezvous with Double-Integrator Dynamics

--- Top-Level component
package FourDronesSystem
public
  with Drone;
  with RefDrone;
  with Data_Model;
  with Hybrid_SynchAADL;

system FourDronesSystem
properties
  Hybrid_SynchAADL::Synchronous => true;
  Hybrid_SynchAADL::Max_Clock_Deviation => 5ms;
  Period => 100ms;
  Classifier_Substitution_Rule => Type_Extension;
end FourDronesSystem;
system implementation FourDronesSystem.revu1D
subcomponents
drone1: system Drone::RevuDrone.impl;
drone2: system Drone::RevuDrone.impl;
drone3: system Drone::RevuDrone.impl;
drone4: system Drone::RevuDrone.impl;
connections
d12x: port drone1.outX -> drone2.inX;
d23x: port drone2.outX -> drone3.inX;
d34x: port drone3.outX -> drone4.inX;
d41x: port drone4.outX -> drone1.inX;
d12vx: port drone1.outVX -> drone2.inVX;
d23vx: port drone2.outVX -> drone3.inVX;
d34vx: port drone3.outVX -> drone4.inVX;
d41vx: port drone4.outVX -> drone1.inVX;
properties
Data_Model::Initial_Value => ("param") applies to drone1.environment.x, drone2.environment.x,
                        drone3.environment.x, drone4.environment.x;
Data_Model::Initial_Value => ("0") applies to drone1.environment.dotx, drone2.environment.dotx,
                        drone3.environment.dotx, drone4.environment.dotx;
Data_Model::Initial_Value => ("0") applies to drone1.environment.dotdotx, drone2.environment.dotdotx,
                        drone3.environment.dotdotx, drone4.environment.dotdotx;
Timing => Delayed applies to d12x, d23x, d34x, d41x, d12vx, d23vx, d34vx, d41vx;
Hybrid_SynchAADL::Sampling_Time => 20ms .. 30ms applies to drone1.drone, drone2.drone, drone3.drone, drone4.drone;
Hybrid_SynchAADL::Response_Time => 60ms .. 80ms applies to drone1.drone, drone2.drone, drone3.drone, drone4.drone;
end FourDronesSystem.revu1D;

system implementation FourDronesSystem.revu2D
    extends FourDronesSystem.revu1D
subcomponents
drone1: refined to system Drone::RevuDrone2D.impl;
drone2: refined to system Drone::RevuDrone2D.impl;
drone3: refined to system Drone::RevuDrone2D.impl;
drone4: refined to system Drone::RevuDrone2D.impl;
connections
d12y: port drone1.outY -> drone2.inY;
d23y: port drone2.outY -> drone3.inY;
d34y: port drone3.outY -> drone4.inY;
d41y: port drone4.outY -> drone1.inY;
d12vy: port drone1.outVY -> drone2.inVY;
d23vy: port drone2.outVY -> drone3.inVY;
d34vy: port drone3.outVY -> drone4.inVY;
d41vy: port drone4.outVY -> drone1.inVY;

properties
Data_Model::Initial_Value => ("param") applies to
drone1.environment.y, drone2.environment.y,
drone3.environment.y, drone4.environment.y;
Data_Model::Initial_Value => ("0") applies to
drone1.environment.doty, drone2.environment.doty,
drone3.environment.doty, drone4.environment.doty;
Data_Model::Initial_Value => ("0") applies to
drone1.environment.dotdoty, drone2.environment.dotdoty,
drone3.environment.dotdoty, drone4.environment.dotdoty;

Timing => Delayed applies to
d12y, d23y, d34y, d41y, d12vy, d23vy, d34vy, d41vy;

end FourDronesSystem.revu2D;
end FourDronesSystem;

--- Drone component
package Drone
public
with Environment;
with Base_Types;
with Data_Model;
with DroneControl;

system RevuDrone
features
inX: in data port Base_Types::Float;
inVX: in data port Base_Types::Float;
outX: out data port Base_Types::Float
  (Data_Model::Initial_Value => ("0"));
outVX: out data port Base_Types::Float
  (Data_Model::Initial_Value => ("0"));
end RevuDrone;

system implementation RevuDrone.impl
subcomponents
drone: system DroneControl::RevuDroneControl.impl;
environment: system Environment::Environment.impl;

connections
C1: port drone.outX -> outX;
C2: port drone.outVX -> outVX;
C3: port inX -> drone.inX;
C4: port inVX -> drone.inVX;
C5: port drone.accX -> environment.accX;
C6: port environment.currX -> drone.currX;
C7: port environment.currVX -> drone.currVX;
end RevuDrone.impl;
system RevuDrone2D extends RevuDrone
features
inY: in data port Base_Types::Float;
inVY: in data port Base_Types::Float;
outY: out data port Base_Types::Float
(Data_Model::Initial_Value => ("0"););
outVY: out data port Base_Types::Float
(Data_Model::Initial_Value => ("0"););
properties
Classifier_Substitution_Rule => Type_Extension;
end RevuDrone2D;

system implementation RevuDrone2D.impl extends RevuDrone.impl
subcomponents
  drone: refined to system DroneControl::RevuDrone2DControl.impl;
  environment: refined to system Environment::Environment2D.impl;
connections
  C8 : port drone.outY -> outY;
  C9 : port drone.outVY -> outVY;
  C10: port inY -> drone.inY;
  C11: port inVY -> drone.inVY;
  C12: port drone.accY -> environment.accY;
  C13: port environment.currY -> drone.currY;
  C14: port environment.currVY -> drone.currVY;
end RevuDrone2D.impl;

end Drone;

--- Controller component
package DroneControl
public
  with Base_Types;
  with DroneSpec;
  with Data_Model;

system RevuDroneControl
features
  currX: in data port Base_Types::Float;
  currVX: in data port Base_Types::Float;
inX: in data port Base_Types::Float;
inVX: in data port Base_Types::Float;
outX: out data port Base_Types::Float
(Data_Model::Initial_Value => ("0"););
outVX: out data port Base_Types::Float
(Data_Model::Initial_Value => ("0"););
accX: out data port Base_Types::Float
(Data_Model::Initial_Value => ("0"););
end RevuDroneControl;

system implementation RevuDroneControl.impl
subcomponents
droneProc: process RevuDroneControlProc.impl;
connections
C1: port currX -> droneProc.currX;
C2: port curr VX -> droneProc.curr VX;
C3: port inX -> droneProc.inX;
C4: port inVX -> droneProc.inVX;
C5: port droneProc.outX -> outX;
C6: port droneProc.outVX -> outVX;
C7: port droneProc.accX -> accX;
end RevuDroneControl.impl;

system RevuDrone2DControl extends RevuDroneControl
features
currY: in data port Base_Types::Float;
currVY: in data port Base_Types::Float;
inY: in data port Base_Types::Float;
inVY: in data port Base_Types::Float;
outY : out data port Base_Types::Float
    {Data_Model::Initial_Value => ("0");};
outVY: out data port Base_Types::Float
    {Data_Model::Initial_Value => ("0");};
accY: out data port Base_Types::Float
    {Data_Model::Initial_Value => ("0");};
properties
Classifier_Substitution_Rule => Type_Extension;
end RevuDrone2DControl;

system implementation RevuDrone2DControl.impl extends RevuDroneControl.impl
subcomponents
droneProc: refined to process RevuDrone2DControlProc.impl;
connections
C8 : port currY -> droneProc.currY;
C9 : port currVY -> droneProc.currVY;
C10: port inY -> droneProc.inY;
C11: port inVY -> droneProc.inVY;
C12: port droneProc.outY -> outY;
C13: port droneProc.outVY -> outVY;
C14: port droneProc.accY -> accY;
end RevuDrone2DControl.impl;

process RevuDroneControlProc
features
currX: in data port Base_Types::Float;
currVX: in data port Base_Types::Float;
inX: in data port Base_Types::Float;
inVX: in data port Base_Types::Float;
outX : out data port Base_Types::Float;
outVX: out data port Base_Types::Float;
accX: out data port Base_Types::Float;
end RevuDroneControlProc;

process implementation RevuDroneControlProc.impl
subcomponents
  droneThread: thread RevuDroneControlThread.impl;
connections
  C1: port currX -> droneThread.currX;
  C2: port currVX -> droneThread.currVX;
  C3: port inX -> droneThread.inX;
  C4: port inVX -> droneThread.inVX;
  C5: port droneThread.outX -> outX;
  C6: port droneThread.outVX -> outVX;
  C7: port droneThread.accX -> accX;
end RevuDroneControlProc.impl;

process RevuDrone2DControlProc extends RevuDroneControlProc
features
  currY: in data port Base_Types::Float;
  currVY: in data port Base_Types::Float;
  inY: in data port Base_Types::Float;
  inVY: in data port Base_Types::Float;
  outY : out data port Base_Types::Float;
  outVY: out data port Base_Types::Float;
  accY: out data port Base_Types::Float;
properties
  Classifier_Substitution_Rule => Type_Extension;
end RevuDrone2DControlProc;

process implementation RevuDrone2DControlProc.impl
  extends RevuDroneControlProc.impl
subcomponents
  droneThread: refined to thread RevuDrone2DControlThread.impl;
connections
  C8 : port currY -> droneThread.currY;
  C9 : port currVY -> droneThread.currVY;
  C10: port inY -> droneThread.inY;
  C11: port inVY -> droneThread.inVY;
  C12: port droneThread.outY -> outY;
  C13: port droneThread.outVY -> outVY;
  C14: port droneThread.accY -> accY;
end RevuDrone2DControlProc.impl;

thread RevuDroneControlThread
features
  currX: in data port Base_Types::Float;
  currVX: in data port Base_Types::Float;
  inX: in data port Base_Types::Float;
  inVX: in data port Base_Types::Float;
  outX : out data port Base_Types::Float;
  outVX: out data port Base_Types::Float;
  accX: out data port Base_Types::Float;
properties
  Dispatch_Protocol => Periodic;
end RevuDroneControlThread;

thread implementation RevuDroneControlThread.impl
  annex behavior_specification (**
    variables
      nx : Base_Types::Float;
    states
      init : initial complete state;
      exec, output : state;
    transitions
      init -[on dispatch]--> exec;
      exec -[abs(currX - inX) < 0.3]--> output{
        accX := -currVX
      };
      exec -[otherwise]--> output{
        nx := -#DroneSpec::A * (currX - inX
        + #DroneSpec::gamma * (currVX - inVX));
        if (nx > 0.5) accX := 40
        elsif (nx > 0) accX := 0
        else accX := -40
      end if
    };
    output -[]--> init{
      outX := currX;
      outVX := currVX
    };
  )**);
end RevuDroneControlThread.impl;

thread RevuDrone2DControlThread extends RevuDroneControlThread
  features
    currY: in data port Base_Types::Float;
    currVY: in data port Base_Types::Float;
    inY: in data port Base_Types::Float;
    inVY: in data port Base_Types::Float;
    outY: out data port Base_Types::Float;
    outVY: out data port Base_Types::Float;
    accY: out data port Base_Types::Float;
end RevuDrone2DControlThread;

thread implementation RevuDrone2DControlThread.impl
  extends RevuDroneControlThread.impl
  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 - inX) < 0.3 and abs(currY - inY) < 0.3] -> output{
    accX := -currVX;
    accY := -currVY
  };
  exec -[otherwise] -> output{
    nx := - #DroneSpec::A * (currX - inX + #DroneSpec::gamma * (currVX - inVX));
    ny := - #DroneSpec::A * (currY - inY + #DroneSpec::gamma * (currVY - inVY));
    if (nx > 0.5) accX := 40
      elsif (nx > 0) accX := 0
      else accX := -40
    end if;
    if (ny > 0.5) accY := 40
      elsif (ny > 0) accY := 0
      else accY := -40
    end if
  };
  output -[ ]-> init{
    outX := currX;
    outVX := currVX;
    outY := currY;
    outVY := currVY
  };
**};
end RevuDrone2DControlThread.impl;
end DroneControl;
--- Environment component
package Environment
public
  with Hybrid_SynchAADL;
  with Base_Types;
  with Data_Model;

system Environment
features
  currX : out data port Base_Types::Float
    (Data_Model::Initial_Value => ("0"););
  currVX : out data port Base_Types::Float
    (Data_Model::Initial_Value => ("0"););
  accX : in data port Base_Types::Float;
properties
  Hybrid_SynchAADL::isEnvironment => true;
end Environment;

system implementation Environment.impl
subcomponents
  x : data Base_Types::Float;
  dotx : data Base_Types::Float;
dotdotx : data Base_Types::Float;

connections
C1: port currVX -> dotx;
C2: port x -> currX;
C3: port accX -> dotdotx;

properties
Hybrid_SynchAADL::ContinuousDynamics =>
"dotx(t) = ((0.001) * dotdotx * t) + dotx(0);
  x(t) = (x(0) + (0.001 * dotx(0) * t)
   + ((0.000001) * dotdotx * t * t) / 2) ; ";
end Environment.impl;

system Environment2D extends Environment
features
  currY : out data port Base_Types::Float
    (Data_Model::Initial_Value => ("0"););
  currVY : out data port Base_Types::Float
    (Data_Model::Initial_Value => ("0"););
  accY : in data port Base_Types::Float;
end Environment2D;

system implementation Environment2D.impl extends Environment.impl
subcomponents
  y : data Base_Types::Float;
  doty : data Base_Types::Float;
  dotdoty : data Base_Types::Float;
connections
C4: port currVY -> doty;
C5: port y -> currY;
C6: port accY -> dotdoty;
properties
Hybrid_SynchAADL::ContinuousDynamics =>
"dotx(t) = ((0.001) * dotdotx * t) + dotx(0);
  x(t) = (x(0) + (0.001 * dotx(0) * t)
   + ((0.000001) * dotdotx * t * t) / 2);
  doty(t) = ((0.001) * dotdoty * t) + doty(0);
  y(t) = (y(0) + (0.001 * doty(0) * t)
   + ((0.000001) * dotdoty * t * t) / 2);";
end Environment2D.impl;
end Environment;

A.4 Drone Formation with Dingle-Integrator Dynamics

--- Top-Level component
package FourDronesSystem
public
  with Drone;
  with RefDrone;
  with Data_Model;
  with Hybrid_SynchAADL;
system FourDronesSystem
properties
  Hybrid_SynchAADL::Synchronous => true;
  Hybrid_SynchAADL::Max_Clock_Deviation => 5ms;
  Period => 100ms;
  Classifier_Substitution_Rule => Type_Extension;
end FourDronesSystem;

system implementation FourDronesSystem.form1D
  extends FourDronesSystem.revu1D
  subcomponents
    drone1: refined to system Drone::FormDrone.impl;
    drone2: refined to system Drone::FormDrone.impl;
    drone3: refined to system Drone::FormDrone.impl;
    drone4: refined to system Drone::FormDrone.impl;
    refDrone: system RefDrone::RefDroneControl.formation;
  connections
    r1x: port refDrone.outX -> drone1.refX;
    r2x: port refDrone.outX -> drone2.refX;
    r3x: port refDrone.outX -> drone3.refX;
    r4x: port refDrone.outX -> drone4.refX;
  properties
    Data_Model::Initial_Value => (-0.5) applies to
      drone1.drone.droneProc.droneThread.offsetX,
      drone3.drone.droneProc.droneThread.offsetX;
    Data_Model::Initial_Value => (0) applies to
      drone2.drone.droneProc.droneThread.offsetX,
      drone4.drone.droneProc.droneThread.offsetX;
    Data_Model::Initial_Value => (param) applies to
      refDrone.env.x;
    Timing => Delayed applies to
      r1x, r2x, r3x, r4x;
    Hybrid_SynchAADL::Sampling_Time => 20ms .. 30ms
      applies to refDrone.refDrone;
    Hybrid_SynchAADL::Response_Time => 60ms .. 80ms
      applies to refDrone.refDrone;
end FourDronesSystem.form1D;

system implementation FourDronesSystem.form2D
  extends FourDronesSystem.revu2D
  subcomponents
    drone1: refined to system Drone::FormDrone2D.impl;
    drone2: refined to system Drone::FormDrone2D.impl;
    drone3: refined to system Drone::FormDrone2D.impl;
    drone4: refined to system Drone::FormDrone2D.impl;
    refDrone: system RefDrone::RefDrone2DControl.formation;
connections
r1x: port refDrone.outX -> drone1.refX;
r2x: port refDrone.outX -> drone2.refX;
r3x: port refDrone.outX -> drone3.refX;
r4x: port refDrone.outX -> drone4.refX;
r1y: port refDrone.outY -> drone1.refY;
r2y: port refDrone.outY -> drone2.refY;
r3y: port refDrone.outY -> drone3.refY;
r4y: port refDrone.outY -> drone4.refY;

properties
Data_Model::Initial_Value => ("0.5") applies to
drone1.drone.droneProc.droneThread.offsetY,
drone2.drone.droneProc.droneThread.offsetY;
Data_Model::Initial_Value => ("-0.5") applies to
drone1.drone.droneProc.droneThread.offsetX,
drone3.drone.droneProc.droneThread.offsetX;
Data_Model::Initial_Value => ("0") applies to
drone2.drone.droneProc.droneThread.offsetX,
drone3.drone.droneProc.droneThread.offsetY,
drone4.drone.droneProc.droneThread.offsetX,
drone4.drone.droneProc.droneThread.offsetY;
Data_Model::Initial_Value => ("param") applies to
refDrone.env.x;

Timing => Delayed applies to
r1x, r2x, r3x, r4x, r1y, r2y, r3y, r4y;

Hybrid_SynchAADL::Sampling_Time => 20ms .. 30ms
    applies to refDrone.refDrone;
Hybrid_SynchAADL::Response_Time => 60ms .. 80ms
    applies to refDrone.refDrone;

end FourDronesSystem.form2D;
end FourDronesSystem;

--- Drone component
package Drone
public
with Environment;
with Base_Types;
with Data_Model;
with DroneControl;

system FormDrone extends RevuDrone
features
    refX: in data port Base_Types::Float;
    refVX: in data port Base_Types::Float;
end FormDrone;

drawdroned.raw.data.model
system implementation FormDrone.impl extends RevuDrone.impl
```
subcomponents
  drone: refined to system DroneControl::FormDroneControl.impl;
  environment: refined to system Environment::Environment.impl;
connections
  C8: port refX -> drone.refX;
  C9: port refVX -> drone.refVX;
end FormDrone.impl;

system FormDrone2D extends RevuDrone2D
features
  refX: in data port Base_Types::Float;
  refY: in data port Base_Types::Float;
  refVX: in data port Base_Types::Float;
  refVY: in data port Base_Types::Float;
end FormDrone2D;

system implementation FormDrone2D.impl extends RevuDrone2D.impl
subcomponents
  drone: refined to system DroneControl::FormDrone2DControl.impl;
  environment: refined to system Environment::Environment2D.impl;
connections
  C15: port refX -> drone.refX;
  C16: port refY -> drone.refY;
  C17: port refVX -> drone.refVX;
  C18: port refVY -> drone.refVY;
end FormDrone2D.impl;
end Drone;

--- Controller component
package DroneControl
public
with Base_Types;
with DroneSpec;
with Data_Model;

system FormDroneControl extends RevuDroneControl
features
  refX: in data port Base_Types::Float;
  refVX: in data port Base_Types::Float;
properties
  Classifier_Substitution_Rule => Type_Extension;
end FormDroneControl;

system implementation FormDroneControl.impl extends RevuDroneControl.impl
subcomponents
  droneProc: refined to process FormDroneControlProc.impl;
connections
  C8: port refX -> droneProc.refX;
  C9: port refVX -> droneProc.refVX;
end FormDroneControl.impl;
```
system FormDrone2DControl extends RevuDrone2DControl
features
  refX: in data port Base_Types::Float;
  refY: in data port Base_Types::Float;
  refVX: in data port Base_Types::Float;
  refVY: in data port Base_Types::Float;
properties
  Classifier_Substitution_Rule => Type_Extension;
end FormDrone2DControl;

system implementation FormDrone2DControl.impl extends RevuDrone2DControl.impl

subcomponents
  droneProc: refined to process FormDrone2DControlProc.impl;
connections
  C15: port refX -> droneProc.refX;
  C16: port refY -> droneProc.refY;
  C17: port refVX -> droneProc.refVX;
  C18: port refVY -> droneProc.refVY;
end FormDrone2DControl.impl;

process FormDroneControlProc extends RevuDroneControlProc
features
  refX: in data port Base_Types::Float;
  refVX: in data port Base_Types::Float;
properties
  Classifier_Substitution_Rule => Type_Extension;
end FormDroneControlProc;

process implementation FormDroneControlProc.impl extends RevuDroneControlProc.impl

subcomponents
  droneThread: refined to thread FormDroneControlThread.impl;
connections
  C8: port refX -> droneThread.refX;
  C9: port refVX -> droneThread.refVX;
end FormDroneControlProc.impl;

process FormDrone2DControlProc extends RevuDrone2DControlProc
features
  refX: in data port Base_Types::Float;
  refY: in data port Base_Types::Float;
  refVX: in data port Base_Types::Float;
  refVY: in data port Base_Types::Float;
properties
  Classifier_Substitution_Rule => Type_Extension;
end FormDrone2DControlProc;

process implementation FormDrone2DControlProc.impl extends RevuDrone2DControlProc.impl
subcomponents
   droneThread: refined to thread FormDrone2DControlThread.impl;
connections
   C15: port refX -> droneThread.refX;
   C16: port refY -> droneThread.refY;
   C17: port refVX -> droneThread.refVX;
   C18: port refVY -> droneThread.refVY;
end FormDrone2DControlProc.impl;

thread FormDroneControlThread extends RevuDroneControlThread
   features
      refX: in data port Base_Types::Float;
      refVX: in data port Base_Types::Float;
   properties
      Dispatch_Protocol => Periodic;
end FormDroneControlThread;
thread implementation FormDroneControlThread.impl
   extends RevuDroneControlThread.impl
   subcomponents
      offsetX: data Base_Types::Float;
      refVX0: data Base_Types::Float
         (Data_Model::Initial_Value => ("0"););
   annex behavior_specification (**
      variables
         nx, refaX : Base_Types::Float;
      states
         init : initial complete state;
         exec, output : state;
      transitions
         init -[on dispatch]-> exec;
         exec -[abs(currX - inX) < 0.3]-> output{
            accX := -currVX
         }
         exec -[otherwise]-> output {
            refax := (refVX - refVX0);
            nx := refax - #DroneSpec::alpha * (currX - offsetX - refX
               + #DroneSpec::gamma * (currVX - refVX))
               - #DroneSpec::A * (currX - offsetX - inX
               + #DroneSpec::gamma * (currVX - inVX));
            if (nx > 0.5) accX := 40
            elsif (nx > 0) accX := 0
            else accX := -40
         end if
      }
      output -[ ]-> init {
         outX := currX - offsetX;
         outVX := currVX;
         refVX0 := refVX
      };
}
thread FormDrone2DControlThread extends RevuDrone2DControlThread
features
  refX: in data port Base_Types::Float;
  refY: in data port Base_Types::Float;
  refVX: in data port Base_Types::Float;
  refVY: in data port Base_Types::Float;
properties
  Dispatch_Protocol => Periodic;
end FormDrone2DControlThread;

thread implementation FormDrone2DControlThread.impl
  extends RevuDrone2DControlThread.impl
subcomponents
  offsetX: data Base_Types::Float;
  offsetY: data Base_Types::Float;
  refVX0: data Base_Types::Float
    (Data_Model::Initial_Value => ("0"););
  refVY0: data Base_Types::Float
    (Data_Model::Initial_Value => ("0"););
annex behavior_specification (**
variables
  nx, refaX : Base_Types::Float;
  ny, refaY : Base_Types::Float;
states
  init : initial complete state;
  exec, output : state;
transitions
  init -[on dispatch]-> exec;
  exec -[abs(currX - inX) < 0.3 and abs(currY - inY) < 0.3]-> output{
    accX := -currVX;
    accY := -currVY;
  };
  exec -[otherwise]-> output {
    refaX := (refVX - refVX0);
    nx := refaX - #DroneSpec::alpha * (currX - offsetX - refX
      + #DroneSpec::gamma * (currVX - refVX))
      - #DroneSpec::A * (currX - offsetX - inX
      + #DroneSpec::gamma * (currVX - inVX));
    refaX := (refVX - refVX0);
    ny := refaY - #DroneSpec::alpha * (currY - offsetY - refY
      + #DroneSpec::gamma * (currVY - refVY))
      - #DroneSpec::A * (currY - offsetY - inY
      + #DroneSpec::gamma * (currVY - inVY));
    if (nx > 0.5) accX := 40
    elsif (nx > 0.1) accX := 20
    elsif (nx > 0) accX := 0
    elsif (nx > -0.1) accX := 20
    accY := -currVY;
  };
end FormDrone2DControlThread.impl;
else accX := -40
end if;
if (ny > 0.5) accY := 40
elsif (ny > 0.1) accY := 20
elsif (ny > 0) accY := 0
elsif (ny > -0.1) accY := 20
else accY := -40
end if
);
output -[ ]-> init {
  outX := currX - offsetX;
  outVX := currVX;
  refVX0 := refVX;
  outY := currY - offsetY;
  outVY := currVY;
  refVY0 := refVY
};
**};
end FormDrone2DControlThread.impl;
end DroneControl;

--- Reference drone component
package RefDrone
public
  with Base_Types;
  with Environment;
  with Data_Model;

  system RefDroneControl
  features
    outX: out data port Base_Types::Float
      {Data_Model::Initial_Value => ("0");};
    outVX: out data port Base_Types::Float
      {Data_Model::Initial_Value => ("0");};
  end RefDroneControl;

  system implementation RefDroneControl.formation
  subcomponents
    refDrone: system RefDrone.formation;
    env: system Environment::Environment.impl;
  connections
    C1: port refDrone.accX -> env.accX;
    C2: port env.currX -> refDrone.currX;
    C3: port refDrone.outX -> outX;
    C4: port env.currVX -> refDrone.currVX;
    C5: port refDrone.outVX -> outVX;
  end RefDroneControl.formation;

  system RefDrone2DControl extends RefDroneControl
  features
outY: out data port Base_Types::Float
    (Data_Model::Initial_Value => ("0"););
outVY: out data port Base_Types::Float
    (Data_Model::Initial_Value => ("0"););

properties
 Classifier_Substitution_Rule => Type_Extension;
end RefDrone2DControl;

system implementation RefDrone2DControl.formation
    extends RefDroneControl.formation
subcomponents
    refDrone: refined to system RefDrone2D.formation;
    env: refined to system Environment::Environment2D.impl;
connections
    C6: port refDrone.accY -> env.accY;
    C7: port env.currY -> refDrone.currY;
    C8: port refDrone.outY -> outY;
    C9: port env.currVY -> refDrone.currVY;
    C10: port refDrone.outVY -> outVY;
end RefDrone2DControl.formation;

system RefDrone
features
    accX: out data port Base_Types::Float;
    outX : out data port Base_Types::Float;
    outVX :out data port Base_Types::Float;
    currX: in data port Base_Types::Float;
    currVX: in data port Base_Types::Float;
end refDrone;

system implementation RefDrone.formation
subcomponents
    refProc: process refProc.formation;
connections
    C1: port refProc.accX -> accX;
    C2: port currX -> refProc.currX;
    C3: port currVX -> refProc.currVX;
    C4: port refProc.outX -> outX;
    C5: port refProc.outVX -> outVX;
end RefDrone.formation;

system RefDrone2D extends RefDrone
features
    accY: out data port Base_Types::Float;
    outY : out data port Base_Types::Float;
    outVY :out data port Base_Types::Float;
    currY: in data port Base_Types::Float;
    currVY: in data port Base_Types::Float;
properties
    Classifier_Substitution_Rule => Type_Extension;
end RefDrone2D;

system implementation RefDrone2D.formation extends RefDrone.formation
subcomponents
  refProc: refined to process RefProc2D.formation;
connections
  C6: port refProc.accY -> accY;
  C7: port currY -> refProc.currY;
  C8: port currVY -> refProc.currVY;
  C9: port refProc.outY -> outY;
  C10: port refProc.outVY -> outVY;
end RefDrone2D.formation;

process RefProc
  features
    accX: out data port Base_Types::Float;
    outX: out data port Base_Types::Float;
    outVX: out data port Base_Types::Float;
    currX: in data port Base_Types::Float;
    currVX: in data port Base_Types::Float;
end RefProc;

process implementation RefProc.formation
subcomponents
  refThread: thread RefThread.formation;
connections
  C1: port refThread.accX -> accX;
  C2: port currX -> refThread.currX;
  C3: port currVX -> refThread.currVX;
  C4: port refThread.outX -> outX;
  C5: port refThread.outVX -> outVX;
end RefProc.formation;

process RefProc2D extends RefProc
  features
    accY: out data port Base_Types::Float;
    outY: out data port Base_Types::Float;
    outVY: out data port Base_Types::Float;
    currY: in data port Base_Types::Float;
    currVY: in data port Base_Types::Float;
  properties
    Classifier_Substitution_Rule => Type_Extension;
end RefProc2D;

process implementation RefProc2D.formation extends RefProc.formation
subcomponents
  refThread: refined to thread RefThread2D.formation;
connections
  C6: port refThread.accY -> accY;
  C7: port currY -> refThread.currY;
  C8: port currVY -> refThread.currVY;
  C9: port refThread.outY -> outY;
C10: port refThread.outVY -> outVY;
end RefProc2D.formation;

thread RefThread
  features
  accX: out data port Base_Types::Float;
  outX : out data port Base_Types::Float;
  outVX : out data port Base_Types::Float;
  currX: in data port Base_Types::Float;
  currVX: in data port Base_Types::Float;
  properties
  Dispatch_Protocol => Periodic;
end RefThread;

thread implementation RefThread.formation
subcomponents
  nx : data Base_Types::Float
  {Data_Model::Initial_Value => ("0");};
  array: data Base_Types::Float[10];
  annex behavior_specification (**
  states
  init : initial complete state;
  exec, output : state;
  transitions
  init -> exec;
  exec -> output {
  nx := 1;
  };
  exec -> output {
  nx := 2
  };
  exec -> output {
  nx := 0
  };
  output -> init {
  accX := nx;
  outX := currX;
  outVX := currVX
  }
  **
end RefThread.formation;

thread RefThread2D extends RefThread
  features
  accY: out data port Base_Types::Float;
  outY : out data port Base_Types::Float;
  outVY : out data port Base_Types::Float;
  currY: in data port Base_Types::Float;
  currVY: in data port Base_Types::Float;
  properties
Dispatch_Protocol => Periodic;
end RefThread2D;

thread implementation RefThread2D.formation extends RefThread.formation

subcomponents
ny : data Base_Types::Float
    {Data_Model::Initial_Value => ("0");};

annex behavior_specification (**

states
    init : initial complete state;
    exec, output : state;

transitions
    init -[ on dispatch ]-> exec;
    exec -[ nx = 0 and ny = 0]-> output {
        nx := 1;
        ny := 1
    };
    exec -[ nx = 1 and ny = 1]-> output {
        nx := 2;
        ny := 2
    };
    exec -[ nx = 2 and ny = 2]-> output {
        nx := 0;
        ny := 0
    };
    output -[ ]-> init {
        accX := nx;
        outX := currX;
        outVX := currVX;
        accY := ny;
        outY := currY;
        outVY := currVY
    };
**});
end RefThread2D.formation;
end RefDrone;

A.5 Networked WaterTank

--- Top-Level component
package FourWaterTankSystem

public
    with Hybrid_SynchAADL;
    with WaterTank;
    with Environment;
    with LeftWaterTank;
    with LeftEnvironment;
    with RightWaterTank;
    with RightEnvironment;
    with Data_Model;
system FourWaterTankSystem
properties
  Hybrid_SynchAADL::Synchronous => true;
end FourWaterTankSystem;

system implementation FourWaterTankSystem.simple
subcomponents
  waterTank1: system LeftWaterTank::LeftWaterTank.simple;
  waterTank2: system WaterTank::WaterTank.simple;
  waterTank3: system WaterTank::WaterTank.simple;
  waterTank4: system RightWaterTank::RightWaterTank.simple;
  env1: system LeftEnvironment::LeftEnvironment.impl;
  env2: system Environment::Environment.impl;
  env3: system Environment::Environment.impl;
  env4: system RightEnvironment::RightEnvironment.impl;
connections
  C11: port waterTank1.on_control -> env1.on_control;
  C12: port waterTank1.off_control -> env1.off_control;
  C13: port waterTank1.power -> env1.power;
  C14: port env1.decreased_water -> waterTank1.decreased_water;
  C15: port env1.curr_water -> waterTank1.curr_water;
  C16: port waterTank1.released_water -> waterTank2.incoming_water;
  C21: port waterTank2.on_control -> env2.on_control;
  C22: port waterTank2.off_control -> env2.off_control;
  C23: port waterTank2.increased_water -> env2.increased_water;
  C24: port waterTank2.power -> env2.power;
  C25: port env2.decreased_water -> waterTank2.decreased_water;
  C26: port env2.curr_water -> waterTank2.curr_water;
  C27: port waterTank2.released_water -> waterTank3.incoming_water;
  C31: port waterTank3.on_control -> env3.on_control;
  C32: port waterTank3.off_control -> env3.off_control;
  C33: port waterTank3.increased_water -> env3.increased_water;
  C34: port waterTank3.power -> env3.power;
  C35: port env3.decreased_water -> waterTank3.decreased_water;
  C36: port env3.curr_water -> waterTank3.curr_water;
  C37: port waterTank3.released_water -> waterTank4.incoming_water;
  C41: port waterTank4.on_control -> env4.on_control;
  C42: port waterTank4.off_control -> env4.off_control;
  C43: port waterTank4.power -> env4.power;
  C44: port env4.curr_water -> waterTank4.curr_water;
  C45: port waterTank4.increased_water -> env4.increased_water;

properties
  Period => 100ms;

  Timing => Delayed applies to C16, C27, C37;
Hybrid_SynchAADL::Max_Clock_Deviation => 5ms;

Hybrid_SynchAADL::Sampling_Time => 20ms .. 30ms applies to
  waterTank1, waterTank2, waterTank3, waterTank4;

Hybrid_SynchAADL::Response_Time => 60ms .. 80ms applies to
  waterTank1, waterTank2, waterTank3, waterTank4;

Data_Model::Initial_Value => ("param") applies to
  env1.water, env2.water, env3.water, env4.water;
Data_Model::Initial_Value => ("0") applies to
  env1.decrease, env2.decrease, env3.decrease, env4.decrease;
Data_Model::Initial_Value => ("0") applies to
  env1.pump_power, env2.pump_power, env3.pump_power, env4.pump_power;
Data_Model::Initial_Value => ("0") applies to
  env2.increase, env3.increase, env4.increase;

end FourWaterTankSystem.simple;

system implementation FourWaterTankSystem.complex
  extends FourWaterTankSystem.simple
  subcomponents
    waterTank1: refined to system LeftWaterTank::LeftWaterTank.complex;
    waterTank2: refined to system WaterTank::WaterTank.complex;
    waterTank3: refined to system WaterTank::WaterTank.complex;
    waterTank4: refined to system RightWaterTank::RightWaterTank.complex;
  end FourWaterTankSystem.complex;
end FourWaterTankSystem;

--- Watertank controller component
package WaterTank
public
  with Base_Types;
  with Data_Model;

system WaterTank
  features
    on_control: out event port;
    off_control: out event port;
    curr_water: in data port Base_Types::Float;
    incoming_water: in data port Base_Types::Float;
    increased_water: out data port Base_Types::Float
      {Data_Model::Initial_Value => ("0");};
    released_water: out data port Base_Types::Float
      {Data_Model::Initial_Value => ("0");};
    power: out data port Base_Types::Float
      {Data_Model::Initial_Value => ("0");};
    decreased_water: in data port Base_Types::Float;
end WaterTank;

system implementation WaterTank.simple
subcomponents
  ctrlProc: process WaterTankProc.simple;
connections
  C1: port ctrlProc.on_control -> on_control;
  C2: port ctrlProc.off_control -> off_control;
  C3: port curr_water -> ctrlProc.curr_water;
  C4: port incoming_water -> ctrlProc.incoming_water;
  C5: port ctrlProc.increased_water -> increased_water;
  C6: port ctrlProc.released_water -> released_water;
  C7: port ctrlProc.power -> power;
  C8: port decreased_water -> ctrlProc.decreased_water;
end WaterTank.simple;

system implementation WaterTank.complex extends WaterTank.simple
subcomponents
  ctrlProc: refined to process WaterTankProc.complex;
end WaterTank.complex;

process WaterTankProc
features
  on_control: out event port;
  off_control: out event port;
  curr_water: in data port Base_Types::Float;
  incoming_water: in data port Base_Types::Float;
  increased_water: out data port Base_Types::Float;
  released_water: out data port Base_Types::Float;
  decreased_water: in data port Base_Types::Float;
  power: out data port Base_Types::Float;
end WaterTankProc;

process implementation WaterTankProc.simple
subcomponents
  ctrlThread: thread WaterTankThread.simple;
connections
  C1: port ctrlThread.on_control -> on_control;
  C2: port ctrlThread.off_control -> off_control;
  C3: port curr_water -> ctrlThread.curr_water;
  C4: port incoming_water -> ctrlThread.incoming_water;
  C5: port ctrlThread.increased_water -> increased_water;
  C6: port ctrlThread.released_water -> released_water;
  C7: port ctrlThread.power -> power;
  C8: port decreased_water -> ctrlThread.decreased_water;
end WaterTankProc.simple;

process implementation WaterTankProc.complex extends WaterTankProc.simple
subcomponents
  ctrlThread: refined to thread WaterTankThread.complex;
end WaterTankProc.complex;

thread WaterTankThread
  features
    on_control: out event port;
    off_control: out event port;
    curr_water: in data port Base_Types::Float;
    incoming_water: in data port Base_Types::Float;
    increased_water: out data port Base_Types::Float;
    released_water: out data port Base_Types::Float;
    power: out data port Base_Types::Float;
    decreased_water: in data port Base_Types::Float;
  properties
    Dispatch_Protocol => Periodic;
end WaterTankThread;

thread implementation WaterTankThread.simple
  annex behavior_specification(**
    states
      init : initial complete state;
      exec : state;
    transitions
      init -[ on dispatch ]-> exec {
        increased_water := incoming_water;
        released_water := decreased_water
      };
      exec -[ true ]-> init {
        if (curr_water <= 33)
          power := 0.5;
          on_control!
        elsif (curr_water <= 39)
          power := 0.3;
          on_control!
        else {
          power := 0.0;
          off_control!
        }
        end if
      };
    **};
end WaterTankThread.simple;

thread implementation WaterTankThread.complex
  extends WaterTankThread.simple
  annex behavior_specification(**
    states
      init : initial complete state;
      exec : state;
    transitions
      init -[ on dispatch ]-> exec {

increased_water := incoming_water;
released_water := decreased_water
);
exec -[curr_water <= 42]-> init {
if (curr_water <= 30)
  power := 0.5
elsif (curr_water <= 33)
  power := 0.4
elsif (curr_water <= 36)
  power := 0.3
elsif (curr_water <= 39)
  power := 0.2
else {
  power := 0.1
}
end if;
on_control!
};
exec -[otherwise]-> init {
  off_control!
};
**};
end WaterTankThread.complex;
end WaterTank;

--- Watertank controller component
package LeftWaterTank
public
with Base_Types;
with Data_Model;

system LeftWaterTank
features
  on_control: out event port;
  off_control: out event port;
  released_water: out data port Base_Types::Float
    {Data_Model::Initial_Value => ("0");};
  power: out data port Base_Types::Float
    {Data_Model::Initial_Value => ("0");};
  decreased_water: in data port Base_Types::Float;
  curr_water: in data port Base_Types::Float;
end LeftWaterTank;

system implementation LeftWaterTank.simple
subcomponents
  ctrlProc: process LeftWaterTankProc.simple;
connections
  C1: port ctrlProc.on_control -> on_control;
C2: port ctrlProc.off_control -> off_control;
C4: port ctrlProc.released_water -> released_water;
C5: port ctrlProc.power -> power;
C6: port decreased_water -> ctrlProc.decreased_water;
C7: port curr_water -> ctrlProc.curr_water;
end LeftWaterTank.simple;

system implementation LeftWaterTank.complex extends LeftWaterTank.simple
subcomponents
ctrlProc: refined to process LeftWaterTankProc.complex;
end LeftWaterTank.complex;

process LeftWaterTankProc
features
  on_control: out event port;
  off_control: out event port;
  released_water: out data port Base_Types::Float;
  power: out data port Base_Types::Float;
  decreased_water: in data port Base_Types::Float;
  curr_water: in data port Base_Types::Float;
end LeftWaterTankProc;

process implementation LeftWaterTankProc.simple
subcomponents
  ctrlThread: thread LeftWaterTankThread.simple;
connections
  C1: port ctrlThread.on_control -> on_control;
  C2: port ctrlThread.off_control -> off_control;
  C4: port ctrlThread.released_water -> released_water;
  C5: port ctrlThread.power -> power;
  C6: port decreased_water -> ctrlThread.decreased_water;
  C7: port curr_water -> ctrlThread.curr_water;
end LeftWaterTankProc.simple;

process implementation LeftWaterTankProc.complex
   extends LeftWaterTankProc.simple
subcomponents
  ctrlThread: refined to thread LeftWaterTankThread.complex;
end LeftWaterTankProc.complex;

thread LeftWaterTankThread
features
  on_control: out event port;
  off_control: out event port;
  released_water: out data port Base_Types::Float;
  decreased_water: in data port Base_Types::Float;
  curr_water: in data port Base_Types::Float;
  power: out data port Base_Types::Float;
properties
Dispatch_Protocol => Periodic;
end LeftWaterTankThread;

thread implementation LeftWaterTankThread.simple
annex behavior_specification(**
states
   init : initial complete state;
   exec : state;
transitions
   init -[ on dispatch ]-> exec {
      released_water := decreased_water
   };
   exec -[ true ]-> init {
      if (curr_water <= 33)
         power := 0.5;
         on_control!
      elsif (curr_water <= 39)
         power := 0.3;
         on_control!
      else {
         power := 0.0;
         off_control!
      }
   } end if
   );
**); end LeftWaterTankThread.simple;

thread implementation LeftWaterTankThread.complex
   extends LeftWaterTankThread.simple
annex behavior_specification(**
states
   init : initial complete state;
   exec : state;
transitions
   init -[ on dispatch ]-> exec {
      released_water := decreased_water
   };
   exec -[curr_water <= 42]-> init {
      if (curr_water <= 30)
         power := 0.5
      elsif (curr_water <= 33)
         power := 0.4
      elsif (curr_water <= 36)
         power := 0.3
      elsif (curr_water <= 39)
         power := 0.2
      else {
         power := 0.1
      }
   }
```plaintext
end if;
on_control!
);
exec -[otherwise]-> init {
  off_control!
});
**};
end LeftWaterTankThread.complex;
end LeftWaterTank;

--- Watertank controller component
package RightWaterTank
public
with Base_Types;
with Data_Model;

system RightWaterTank
  features
    on_control: out event port;
    off_control: out event port;
    curr_water: in data port Base_Types::Float;
    incoming_water: in data port Base_Types::Float;
    increased_water: out data port Base_Types::Float
      {Data_Model::Initial_Value => ("0");};
    power: out data port Base_Types::Float
      {Data_Model::Initial_Value => ("0");};
end RightWaterTank;

system implementation RightWaterTank.simple
  subcomponents
    ctrlProc: process RightWaterTankProc.simple;
  connections
    C1: port ctrlProc.on_control -> on_control;
    C2: port ctrlProc.off_control -> off_control;
    C3: port curr_water -> ctrlProc.curr_water;
    C4: port incoming_water -> ctrlProc.incoming_water;
    C5: port ctrlProc.increased_water -> increased_water;
    C6: PORT ctrlProc.power -> power;
end RightWaterTank.simple;

system implementation RightWaterTank.complex extends RightWaterTank.simple
  subcomponents
    ctrlProc: refined to process RightWaterTankProc.complex;
end RightWaterTank.complex;

process RightWaterTankProc
  features
    on_control: out event port;
```
off_control: out event port;
curr_water: in data port Base_Types::Float;
incoming_water: in data port Base_Types::Float;
increased_water: out data port Base_Types::Float;
power: out data port Base_Types::Float;
end RightWaterTankProc;

process implementation RightWaterTankProc.simple
  subcomponents
  ctrlThread: thread RightWaterTankThread.simple;
  connections
  C1: port ctrlThread.on_control -> on_control;
  C2: port ctrlThread.off_control -> off_control;
  C3: port curr_water -> ctrlThread.curr_water;
  C4: port incoming_water -> ctrlThread.incoming_water;
  C5: port ctrlThread.increased_water -> increased_water;
  C6: port ctrlThread.power -> power;
end RightWaterTankProc.simple;

process implementation RightWaterTankProc.complex
  extends RightWaterTankProc.simple
  subcomponents
  ctrlThread: refined to thread RightWaterTankThread.complex;
end RightWaterTankProc.complex;

thread RightWaterTankThread
  features
  on_control: out event port;
  off_control: out event port;
  curr_water: in data port Base_Types::Float;
incoming_water: in data port Base_Types::Float;
increased_water: out data port Base_Types::Float;
power: out data port Base_Types::Float;
  properties
  Dispatch_Protocol => Periodic;
end RightWaterTankThread;

thread implementation RightWaterTankThread.simple
  annex behavior_specification(**
    states
    init : initial complete state;
    exec : state;
    transitions
    init -[ on dispatch ]-> exec {
      increased_water := incoming_water
    };
    exec -[ true ]-> init {
      if (curr_water <= 33)
        power := 0.5;
        on_control!
elsif (curr_water <= 39)
    power := 0.3;
    on_control!
else {
    power := 0.0;
    off_control!
}
end if

thread implementation RightWaterTankThread.complex
extends RightWaterTankThread.simple

annex behavior_specification(**

states
    init : initial complete state;
    exec : state;

transitions
    init -[ on dispatch ]-> exec {
        increased_water := incoming_water
    };
    exec -[curr_water <= 42]-> init {
        if (curr_water <= 30)
            power := 0.5
        elsif (curr_water <= 33)
            power := 0.4
        elsif (curr_water <= 36)
            power := 0.3
        elsif (curr_water <= 39)
            power := 0.2
        else {
            power := 0.1
        }
        end if;
        on_control!
    };
    exec -[otherwise]-> init {
        off_control!
    };
**
); end RightWaterTankThread.complex;

end RightWaterTankThread.

--- Environment component
package Environment
public
with Base_Types;
with Data_Model;
with Hybrid_SynchAADL;

system Environment
features
  curr_water : out data port Base_Types::Float
  {Data_Model::Initial_Value => ("0");};
  increased_water : in data port Base_Types::Float;
  power: in data port Base_Types::Float;
  decreased_water : out data port Base_Types::Float
  {Data_Model::Initial_Value => ("0");};
  on_control : in event port;
  off_control : in event port;
properties
  Hybrid_SynchAADL::isEnvironment => true;
end Environment;

system implementation Environment.impl
subcomponents
  water: data Base_Types::Float;
  decrease: data Base_Types::Float;
  increase: data Base_Types::Float;
  pump_power: data Base_Types::Float;
connections
  C1: port water -> curr_water;
  C2: port increased_water -> increase;
  C3: port decrease -> decreased_water;
  C4: port power -> pump_power;
modes
  off: initial mode;
  on: mode;

  off -[on_control]--> on;
  off -[off_control]--> off;

  on -[on_control]--> on;
  on -[off_control]--> off;
properties
  Hybrid_SynchAADL::ContinuousDynamics =>
  "decrease(t) = (50 * 0.001 * t) ;
  water(t) = water(0) + pump_power - (50 * 0.001 * t)
  + increase" in modes(on),
  "decrease(t) = (50 * 0.001 * t) ;
  water(t) = water(0) - (50 * 0.001 * t) + increase" in modes(off);
end Environment.impl;
end Environment;

A.6 Networked Thermostat
--- Top-Level component
package FourRoomTherm
public
   with Thermostat;
   with Thermostat2side;
   with Hybrid_SynchAADL;
   with RoomEnv;
   with Data_Model;
end FourRoomTherm;

system implementation FourRoomTherm.simple
subcomponents
   therm1: system Thermostat::Thermostat.simple;
   therm2: system Thermostat2side::Thermostat2side.simple;
   therm3: system Thermostat2side::Thermostat2side.simple;
   therm4: system Thermostat::Thermostat.simple;
   env1: system RoomEnv::RoomEnv.impl;
   env2: system RoomEnv::RoomEnv.impl;
   env3: system RoomEnv::RoomEnv.impl;
   env4: system RoomEnv::RoomEnv.impl;
connections
   C11: port therm1.on_control -> env1.on_control;
C12: port therm1.off_control -> env1.off_control;
C13: port therm1.power -> env1.power;
C14: port therm1.t_out -> therm2.t_in1;
C15: port env1.temp -> therm1.curr_temp;

   C21: port therm2.on_control -> env2.on_control;
C22: port therm2.off_control -> env2.off_control;
C23: port therm2.power -> env2.power;
C24: port therm2.t_out -> therm1.t_in;
C25: port therm2.t_out -> therm3.t_in;
C26: port env2.temp -> therm2.curr_temp;

   C31: port therm3.on_control -> env3.on_control;
C32: port therm3.off_control -> env3.off_control;
C33: port therm3.power -> env3.power;
C34: port therm3.t_out -> therm2.t_in2;
C35: port therm3.t_out -> therm4.t_in;
C36: port env3.temp -> therm3.curr_temp;

   C41: port therm4.on_control -> env4.on_control;
C42: port therm4.off_control -> env4.off_control;
C43: port therm4.power -> env4.power;
C44: port therm4.t_out -> therm3.t_in2;
C45: \texttt{port} env4.temp $\rightarrow$ therm4.curr_temp;

\textbf{properties}

\begin{itemize}
\item \texttt{Period} $\Rightarrow$ 100 ms;
\item \texttt{Data\_Model::Initial\_Value} $\Rightarrow$ ("param") \texttt{applies to env1.x, env2.x, env3.x, env4.x};
\item \texttt{Timing} $\Rightarrow$ Delayed \texttt{applies to C14, C24, C25, C34, C35, C44};
\item \texttt{Hybrid\_SynchAADL::Max\_Clock\_Deviation} $\Rightarrow$ 5ms;
\item \texttt{Hybrid\_SynchAADL::Sampling\_Time} $\Rightarrow$ 20ms .. 30ms \texttt{applies to therml, therm3, therm2, therm4};
\item \texttt{Hybrid\_SynchAADL::Response\_Time} $\Rightarrow$ 60ms .. 70ms \texttt{applies to therml, therm3, therm2, therm4};
\end{itemize}

\texttt{end} FourRoomTherm.simple;

\textbf{system implementation} FourRoomTherm.complex \texttt{extends} FourRoomTherm.simple

\textbf{subcomponents}

\begin{itemize}
\item \texttt{therm1: refined to system Thermostat::Thermostat.complex;}
\item \texttt{therm2: refined to system Thermostat2side::Thermostat2side.complex;}
\item \texttt{therm3: refined to system Thermostat2side::Thermostat2side.complex;}
\item \texttt{therm4: refined to system Thermostat::Thermostat.complex;}
\end{itemize}

\texttt{end} FourRoomTherm.complex;

\texttt{end} FourRoomTherm;

--- Thermostat controller component

\textbf{package} Thermostat

\textbf{public}

\textbf{with} Base\_Types;

\textbf{with} Data\_Model;

\textbf{system} Thermostat

\textbf{features}

\textbf{features}

\begin{itemize}
\item \texttt{on\_control: out event port;}
\item \texttt{off\_control: out event port;}
\item \texttt{power: out data port Base\_Types::Float}
\item \texttt{t\_out: out data port Base\_Types::Float}
\item \texttt{t\_in: in data port Base\_Types::Float;}
\item \texttt{curr\_temp: in data port Base\_Types::Float;}
\end{itemize}

\texttt{end} Thermostat;

\textbf{system implementation} Thermostat.simple

\textbf{subcomponents}
ctrlProc: process ThermostatProc.simple;
connections
  O1: port ctrlProc.on_control -> on_control;
  O2: port ctrlProc.off_control -> off_control;
  O3: port ctrlProc.power -> power;
  O4: port ctrlProc.t_out -> t_out;
  I1: port t_in -> ctrlProc.t_in;
  I2: port curr_temp -> ctrlProc.curr_temp;
end Thermostat.simple;

system implementation Thermostat.complex extends Thermostat.simple
subcomponents
  ctrlProc: refined to process ThermostatProc.complex;
end Thermostat.complex;

process ThermostatProc
features
  on_control: out event port;
  off_control: out event port;
  power : out data port Base_Types::Float;
  t_out : out data port Base_Types::Float;
  t_in : in data port Base_Types::Float;
  curr_temp : in data port Base_Types::Float;
end ThermostatProc;

process implementation ThermostatProc.simple
subcomponents
  ctrlThread: thread ThermostatThread.simple;
connections
  O1: port ctrlThread.on_control -> on_control;
  O2: port ctrlThread.off_control -> off_control;
  O3: port ctrlThread.power -> power;
  O4: port ctrlThread.t_out -> t_out;
  I1: port t_in -> ctrlThread.t_in;
  I2: port curr_temp -> ctrlThread.curr_temp;
end ThermostatProc.simple;

process implementation ThermostatProc.complex extends ThermostatProc.simple
subcomponents
  ctrlThread: refined to thread ThermostatThread.complex;
end ThermostatProc.complex;

thread ThermostatThread
features
  on_control: out event port;
  off_control: out event port;
  power : out data port Base_Types::Float;
  t_out : out data port Base_Types::Float;
  t_in : in data port Base_Types::Float;
  curr_temp : in data port Base_Types::Float;
properties
  Dispatch_Protocol => Periodic;
end ThermostatThread;

thread implementation ThermostatThread.simple
subcomponents
temp_min : Data_Base_Types::Float
  (Data_Model::Initial_Value => ("20"););
temp_max : Data_Base_Types::Float
  (Data_Model::Initial_Value => ("50"););
annex behavior_specification(**
  variables
    a : Base_Types::Float;
  states
    init : initial complete state;
    exec : state;
  transitions
    init -[ on dispatch ]-> exec {
      t_out := curr_temp
    };
    exec -[ true ]-> init {
      a := -(curr_temp - t_in);
      if(a > 10){
        power := 5;
        on_control!
      }
      elsif(a > 3){
        power := 3;
        on_control!
      }
      else{
        power := 0;
        off_control!
      }
      end if
    };
**);
end ThermostatThread.simple;

thread implementation ThermostatThread.complex
  extends ThermostatThread.simple
annex behavior_specification(**
  variables
    a : Base_Types::Float;
  states
    init : initial complete state;
    exec : state;
  transitions
init -[ on dispatch ]-> exec {
    t_out := curr_temp
};

eexec -[ curr_temp > temp_max ]-> init {
    off_control!
    power := 0
};

eexec -[ curr_temp < temp_min ]-> init {
    on_control!
    power := 10
};

eexec -[ otherwise ]-> init {
    a := -(curr_temp - t_in);
    if(a > 10){
        power := 5;
        on_control!
    }
    elsif(a > 5){
        power := 4;
        on_control!
    }
    elsif(a > 3){
        power := 3;
        on_control!
    }
    elsif(a > 0){
        power := 2;
        on_control!
    }
    else{
        power := 0;
        off_control!
    }
    end if
};

end ThermostatThread.complex;
end Thermostat;

--- Thermostat controller component
package Thermostat2side
public
with Base_Types;
with Data_Model;
system Thermostat2side
features
  power: out data port Base_Types::Float
  t_out: out data port Base_Types::Float
  t_in1: in data port Base_Types::Float;
  t_in2: in data port Base_Types::Float;
  curr_temp: in data port Base_Types::Float;
  on_control: out event port;
  off_control: out event port;
end Thermostat2side;

system implementation Thermostat2side.simple
subcomponents
  ctrlProc: process ThermostatProc.simple;
connections
  O1: port ctrlProc.on_control -> on_control;
  O2: port ctrlProc.off_control -> off_control;
  O3: port ctrlProc.power -> power;
  O4: port ctrlProc.t_out -> t_out;
  I1: port t_in1 -> ctrlProc.t_in1;
  I2: port t_in2 -> ctrlProc.t_in2;
  I3: port curr_temp -> ctrlProc.curr_temp;
end Thermostat2side.simple;

system implementation Thermostat2side.complex extends Thermostat2side.simple
subcomponents
  ctrlProc: refined to process ThermostatProc.complex;
end Thermostat2side.complex;

process ThermostatProc
features
  power: out data port Base_Types::Float;
  t_out: out data port Base_Types::Float;
  t_in1: in data port Base_Types::Float;
  t_in2: in data port Base_Types::Float;
  curr_temp: in data port Base_Types::Float;
  on_control: out event port;
  off_control: out event port;
end ThermostatProc;

process implementation ThermostatProc.simple
subcomponents
  ctrlThread: thread ThermostatThread.simple;
connections
  O1: port ctrlThread.on_control -> on_control;
  O2: port ctrlThread.off_control -> off_control;
  O3: port ctrlThread.power -> power;
  O4: port ctrlThread.t_out -> t_out;
I1: port t_in1 -> ctrlThread.t_in1;
I2: port t_in2 -> ctrlThread.t_in2;
I3: port curr_temp -> ctrlThread.curr_temp;
end ThermostatProc.simple;

process implementation ThermostatProc.complex extends ThermostatProc.simple
subcomponents
  ctrlThread: refined to thread ThermostatThread.complex;
end ThermostatProc.complex;

thread ThermostatThread
  features
    power: out data port Base_Types::Float;
    t_out: out data port Base_Types::Float;
    t_in1: in data port Base_Types::Float;
    t_in2: in data port Base_Types::Float;
    curr_temp: in data port Base_Types::Float;
    on_control: out event port;
    off_control: out event port;
  properties
    Dispatch_Protocol -> Periodic;
end ThermostatThread;

thread implementation ThermostatThread.simple
subcomponents
  temp_min: Data Base_Types::Float {Data_Model::Initial_Value => ("20");};
  temp_max: Data Base_Types::Float {Data_Model::Initial_Value => ("50");};
annex behavior_specification (**
  variables
    a : Base_Types::Float;
  states
    init : initial complete state;
    exec : state;
  transitions
    init -[on dispatch]-> exec {
      t_out := curr_temp
    };
    exec -[ true ]-> init {
      a := -(curr_temp - t_in1);
      a := a - (curr_temp - t_in2);
      if(a > 10){
        power := 5;
        on_control!
      }
      elsif(a > 3){
        power := 3;
        on_control!
      }
      else{
        power := 0;
      }
ThermostatThread.simple;

**};

end ThermostatThread.simple;

thread implementation ThermostatThread.complex
  extends ThermostatThread.simple

annex behavior_specification {**
  variables
    a : Base_Types::Float;
  states
    init : initial complete state;
    exec : state;
  transitions
    init -[on dispatch] -> exec {
      t_out := curr_temp
    };
    exec -[curr_temp > temp_max] -> init {
      off_control !;
      power := 0
    };
    exec -[curr_temp < temp_min] -> init {
      on_control !;
      power := 10
    };
    exec -[otherwise] -> init {
      a := -(curr_temp - t_in1);
      a := a - (curr_temp - t_in2);
      if(a > 10) {
        power := 5;
        on_control!
      }
      elsif(a > 5) {
        power := 4;
        on_control!
      }
      elsif(a > 3) {
        power := 3;
        on_control!
      }
      elsif(a > 0) {
        power := 2;
        on_control!
      }
      else {
        power := 0;
        off_control!
      }
    };

end ThermostatThread.complex;

off_control!
}
end if
};
**};
end ThermostatThread.complex;
end Thermostat2side;

--- Environemnt component
package RoomEnv
public
with Base_Types;
with Hybrid_SynchAADL;
with Data_Model;

system RoomEnv
features
  temp : out data port Base_Types::Float
        {Data_Model::Initial_Value => ("0");};
  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;
  p: data Base_Types::Float {Data_Model::Initial_Value => ("0");};
connections
  C: port x -> temp;
  P: port power -> p;
modes
  off: initial mode;
  on: mode;

  off -[on_control]--> on;
  off -[off_control]--> off;

  on -[on_control]--> on;
  on -[off_control]--> off;

properties
  Hybrid_SynchAADL::ContinuousDynamics =>
  "x(t) = x(0) + (0.01 * p) * t; " in modes (on),
  "x(t) = x(0) - (0.001 * t);" in modes (off);
end RoomEnv.impl;
end RoomEnv;
B Hybrid Automata Components

This section shows the hybrid automata components of the benchmark models in Figures 26–37. Because we use two kinds of the controllers with different complexity, there are two different hybrid automata components for each model. Each model with the simple controller has 3 controller transitions labeled by ControllerTransition. However, For drone models with complex controller, there are 25 controller transitions and additional variables for the y-axis. For thermostat and wateratank models with complex controller, there are 5 controller transitions.

Fig. 26. Hybrid automaton component for drone rendezvous with simple control logic and double-integrator dynamics

C More Details on Experimental Evaluation

This section shows the entire tables used in Section 7. All the tables show the time results in seconds. Tables 3–6 for Section 7.1 show the results for all bound. Table
Fig. 27. Hybird automaton component for drone rendezvous with complex control logic and double-integrator dynamics

used in Section 7.2 provides additional information about standard deviations for five repeated experiments. Tables 8 for Section 7.3 and 9 for Section 7.4 show detailed data for all cases of the benchmark models including those shown in Figures 24 and 25 respectively.
Fig. 28. Hybrid automaton component for drone formation with simple control logic and double-integrator dynamics.
Fig. 29. Hybrid automaton component for drone formation with complex control logic and double-integrator dynamics
Fig. 30. Hybrid automaton component for drone rendezvous with simple control logic and single-integrator dynamics

Fig. 31. Hybrid automaton component for drone rendezvous with complex control logic and single-integrator dynamics
Sampling
\[ \text{lb}_{\text{samp}} \leq \text{localTime} \leq \text{ub}_{\text{samp}} + 2 \times \text{clockSkew} \]
\[ x_{\text{samp}} := x \]

New Round
\[ \text{localTime} \leq \text{ub}_{\text{samp}} + 2 \times \text{clockSkew} \]
\[ x' = v_x \& \]
\[ v_x' = 0 \& \]
\[ x_{\text{samp}}' = 0 \& \]
\[ x_{\text{out}}' = 0 \& \]
\[ \text{refX}_{\text{prev in}}' = 0 \]

Sampling
\[ \text{localTime} \leq \text{ub}_{\text{resp}} + 2 \times \text{clockSkew} \]
\[ x' = v_x \& \]
\[ v_x' = 0 \& \]
\[ x_{\text{samp}}' = 0 \& \]
\[ x_{\text{out}}' = 0 \& \]
\[ \text{refX}_{\text{prev in}}' = 0 \]

Controller Transition 1
\[ \text{lb}_{\text{resp}} \leq \text{localTime} \leq \text{ub}_{\text{resp}} + 2 \times \text{clockSkew} \]
\[ (\text{refX}_{\text{in}} - \text{refX}_{\text{prev in}}) - (\text{DroneSpec}_{\text{Alpha}} \times (x_{\text{samp}} - \text{offsetX} - \text{refX}_{\text{in}})) - (\text{DroneSpec}_{A} \times (x_{\text{samp}} - \text{offsetX} - x_{\text{in}})) > 1 \]
\[ v_x := 2 \]

Controller Transition 3

Actuation
\[ \text{localTime} \leq \text{period} \]
\[ x' = v_x \& \]
\[ v_x' = 0 \& \]
\[ x_{\text{samp}}' = 0 \& \]
\[ x_{\text{out}}' = 0 \& \]
\[ \text{refX}_{\text{prev in}}' = 0 \]

Synch
\[ \text{localTime} = \text{period} \]
\[ x_{\text{out}} := x_{\text{samp}} \]

**Fig. 32.** Hybrid automaton component for drone formation with simple control logic and single-integrator dynamics
Fig. 33. Hybrid automaton component for drone formation with complex control logic and single-integrator dynamics
Fig. 34. Hybrid automaton component for networked thermostat with simple control logic

Fig. 35. Hybrid automaton component for networked thermostat with complex control logic
Fig. 36. Hybrid automaton component for networked waterTank with simple control logic

Sampling
lb_samp <= localTime <= ub_samp + 2 * clockSkew
water_samp := decreased &
water = water + water_in

NewRound
localTime <= ub_samp + 2 * clockSkew
water' = -50 &
Water_samp' = 0 &
decreased' = 50 &
water_out' = 0

Synch
localTime == period
x_out := x_samp

Sampling
localTime <= ub_resp + 2 * clockSkew
water' = -50 &
Water_samp' = 0 &
decreased' = 50 &
water_out' = 0

ControllerTransition1
lb_resp <= localTime <= ub_resp + 2 * clockSkew &
water <= 33
water := water + 0.5
...
ControllerTransition3

Actuation
localTime <= period
water' = -50 &
Water_samp' = 0 &
decreased' = 50 &
water_out' = 0

Fig. 37. Hybrid automaton component for networked waterTank with complex control logic
<table>
<thead>
<tr>
<th>Model</th>
<th>N Tool</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drone Formation (Single)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSADDL</td>
<td>0.39 0.45 0.28 0.83 1.62</td>
<td>1.88 1.37 0.39 0.75 1.59</td>
<td>0.84 0.52 0.23 0.52 0.84</td>
<td>1.30 0.30 0.29 0.75 1.19</td>
<td>1.30 0.30 0.29 0.75 1.19</td>
</tr>
<tr>
<td>HyComp</td>
<td>0.05 0.17 0.28 0.83 1.62</td>
<td>1.88 1.37 0.39 0.75 1.59</td>
<td>0.84 0.52 0.23 0.52 0.84</td>
<td>1.30 0.30 0.29 0.75 1.19</td>
<td>1.30 0.30 0.29 0.75 1.19</td>
</tr>
<tr>
<td>SpaceEx</td>
<td>0.05 0.29 0.75 1.19 2.33</td>
<td>1.88 1.37 0.39 0.75 1.59</td>
<td>0.84 0.52 0.23 0.52 0.84</td>
<td>1.30 0.30 0.29 0.75 1.19</td>
<td>1.30 0.30 0.29 0.75 1.19</td>
</tr>
<tr>
<td>dReach</td>
<td>1.30 26.00 341.58 - -</td>
<td>1.88 1.37 0.39 0.75 1.59</td>
<td>0.84 0.52 0.23 0.52 0.84</td>
<td>1.30 0.30 0.29 0.75 1.19</td>
<td>1.30 0.30 0.29 0.75 1.19</td>
</tr>
<tr>
<td>Flow*</td>
<td>2.82 245.26 1070.07 . - .</td>
<td>1.88 1.37 0.39 0.75 1.59</td>
<td>0.84 0.52 0.23 0.52 0.84</td>
<td>1.30 0.30 0.29 0.75 1.19</td>
<td>1.30 0.30 0.29 0.75 1.19</td>
</tr>
<tr>
<td>Drone Formation (Double)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSADDL</td>
<td>0.36 0.66 1.29 2.51 4.72</td>
<td>3.08 1240.68 . - .</td>
<td>7.97 1.90 3.76 7.79</td>
<td>3.07 0.68 2.00 7.12 17.23</td>
<td>5.53 1.88 5.84 - .</td>
</tr>
<tr>
<td>HyComp</td>
<td>0.09 0.40 1.24 3.97 8.52</td>
<td>3.08 1240.68 . - .</td>
<td>7.97 1.90 3.76 7.79</td>
<td>3.07 0.68 2.00 7.12 17.23</td>
<td>5.53 1.88 5.84 - .</td>
</tr>
<tr>
<td>SpaceEx</td>
<td>0.96 103.68 696.38 . - .</td>
<td>3.08 1240.68 . - .</td>
<td>7.97 1.90 3.76 7.79</td>
<td>3.07 0.68 2.00 7.12 17.23</td>
<td>5.53 1.88 5.84 - .</td>
</tr>
<tr>
<td>dReach</td>
<td>5.75 . - . - .</td>
<td>3.08 1240.68 . - .</td>
<td>7.97 1.90 3.76 7.79</td>
<td>3.07 0.68 2.00 7.12 17.23</td>
<td>5.53 1.88 5.84 - .</td>
</tr>
<tr>
<td>Flow*</td>
<td>96.36 . - . - .</td>
<td>3.08 1240.68 . - .</td>
<td>7.97 1.90 3.76 7.79</td>
<td>3.07 0.68 2.00 7.12 17.23</td>
<td>5.53 1.88 5.84 - .</td>
</tr>
<tr>
<td>WaterTank</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSADDL</td>
<td>0.41 0.88 1.90 3.76 7.79</td>
<td>3.08 1240.68 . - .</td>
<td>7.97 1.90 3.76 7.79</td>
<td>3.07 0.68 2.00 7.12 17.23</td>
<td>5.53 1.88 5.84 - .</td>
</tr>
<tr>
<td>HyComp</td>
<td>0.20 0.84 5.69 11.23 37.87</td>
<td>3.08 1240.68 . - .</td>
<td>7.97 1.90 3.76 7.79</td>
<td>3.07 0.68 2.00 7.12 17.23</td>
<td>5.53 1.88 5.84 - .</td>
</tr>
<tr>
<td>SpaceEx</td>
<td>0.47 . - . - .</td>
<td>3.08 1240.68 . - .</td>
<td>7.97 1.90 3.76 7.79</td>
<td>3.07 0.68 2.00 7.12 17.23</td>
<td>5.53 1.88 5.84 - .</td>
</tr>
<tr>
<td>dReach</td>
<td>. - . - . - .</td>
<td>3.08 1240.68 . - .</td>
<td>7.97 1.90 3.76 7.79</td>
<td>3.07 0.68 2.00 7.12 17.23</td>
<td>5.53 1.88 5.84 - .</td>
</tr>
<tr>
<td>Flow*</td>
<td>27.96 . - . - .</td>
<td>3.08 1240.68 . - .</td>
<td>7.97 1.90 3.76 7.79</td>
<td>3.07 0.68 2.00 7.12 17.23</td>
<td>5.53 1.88 5.84 - .</td>
</tr>
</tbody>
</table>

Table 3. Comparison of HybridSynchAADL and the other tools for $\text{InUT}$ at all bounds in using simplified control logic. The '-' represents a timeout.
<table>
<thead>
<tr>
<th>Model</th>
<th>N</th>
<th>Tool</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>HSADDL</td>
<td>0.32</td>
<td>0.52</td>
<td>0.88</td>
<td>1.56</td>
<td>7.61</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>HyComp</td>
<td>0.07</td>
<td>0.30</td>
<td>0.76</td>
<td>1.79</td>
<td>2.63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>SpaceEx</td>
<td>0.45</td>
<td>0.29</td>
<td>0.67</td>
<td>1.19</td>
<td>2.16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drone Rendezvous (Double)</td>
<td>4</td>
<td>Flow*</td>
<td>2.05</td>
<td>12.08</td>
<td>15.50</td>
<td>8.41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpaceEx</td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Thermostat</td>
<td></td>
<td>HSADDL</td>
<td>0.38</td>
<td>0.70</td>
<td>1.41</td>
<td>2.71</td>
<td>15.27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>HyComp</td>
<td>0.22</td>
<td>1.59</td>
<td>5.35</td>
<td>12.71</td>
<td>15.50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>SpaceEx</td>
<td>0.96</td>
<td>100.19</td>
<td>708.21</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HyComp</td>
<td>0.17</td>
<td>0.84</td>
<td>1.86</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SpaceEx</td>
<td>0.88</td>
<td>13.48</td>
<td>74.37</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>HSADDL</td>
<td>0.45</td>
<td>0.94</td>
<td>2.11</td>
<td>10.65</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SpaceEx</td>
<td>3.27</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>1718.11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSADDL</td>
<td>0.46</td>
<td>0.86</td>
<td>4.82</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SpaceEx</td>
<td>40.37</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>28.53</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drone Rendezvous (Single)</td>
<td>1</td>
<td>HSADDL</td>
<td>0.30</td>
<td>0.46</td>
<td>2.36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>HyComp</td>
<td>0.16</td>
<td>5.89</td>
<td>8.86</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>SpaceEx</td>
<td>0.02</td>
<td>0.38</td>
<td>5.08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Flow*</td>
<td>1.87</td>
<td>12.58</td>
<td>167.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>41.84</td>
<td>1586.32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>15.54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSADDL</td>
<td>0.45</td>
<td>2.54</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>45.30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SpaceEx</td>
<td>2.51</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>291.12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSADDL</td>
<td>0.52</td>
<td>5.16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SpaceEx</td>
<td>105.87</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>20.30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSADDL</td>
<td>2.64</td>
<td>20.30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SpaceEx</td>
<td>2.51</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HSADDL</td>
<td>105.87</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SpaceEx</td>
<td>105.87</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4. Comparison of HybridSynchAADL and the other tools for Inv_1 at all bounds in using simplified control logic. The ‘-’ means that a counterexample is not found before a timeout.
<table>
<thead>
<tr>
<th>Model</th>
<th>N</th>
<th>Tool</th>
<th>B = 1</th>
<th>B = 2</th>
<th>B = 3</th>
<th>B = 4</th>
<th>B = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpaceEx</td>
<td>2</td>
<td>HyComp</td>
<td>0.05</td>
<td>0.13</td>
<td>0.40</td>
<td>0.91</td>
<td>3.97</td>
</tr>
<tr>
<td>HyComp</td>
<td></td>
<td>SpaceEx</td>
<td>0.03</td>
<td>0.30</td>
<td>1.76</td>
<td>2.81</td>
<td>4.72</td>
</tr>
<tr>
<td>dReach</td>
<td></td>
<td>Flow*</td>
<td>1.56</td>
<td>9.07</td>
<td>41.97</td>
<td>318.79</td>
<td></td>
</tr>
<tr>
<td>HyComp</td>
<td></td>
<td>dReach</td>
<td>12.62</td>
<td>1341.60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpaceEx</td>
<td></td>
<td>Flow*</td>
<td>3.72</td>
<td>12.43</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td></td>
<td>Flow*</td>
<td>1.65</td>
<td>5.31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Formation</td>
<td></td>
<td>Time</td>
<td>1.54</td>
<td>3.61</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Single)</td>
<td></td>
<td>Time</td>
<td>2.33</td>
<td>7.70</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drone</td>
<td></td>
<td>Time</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rendezvous</td>
<td></td>
<td>Time</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Double)</td>
<td></td>
<td>Time</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.** Comparison of HybridSynchAADL and the other tools for Inout at all bounds in using complex control logic. The '-' represents a timeout.
<table>
<thead>
<tr>
<th>Model</th>
<th>N</th>
<th>Tool</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Drone</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Formation (Single)</td>
<td>0.32</td>
<td>0.71</td>
<td>1.11</td>
<td>1.80</td>
<td>9.57</td>
<td>2.85</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Drone</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Formation (Double)</td>
<td>0.07</td>
<td>0.31</td>
<td>0.66</td>
<td>1.97</td>
<td>3.05</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Thermostat</td>
<td>0.03</td>
<td>0.31</td>
<td>1.71</td>
<td>2.73</td>
<td>4.30</td>
<td>0.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Drone</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rendezvous (Single)</td>
<td>1.62</td>
<td>1.97</td>
<td>45.53</td>
<td>19.04</td>
<td>19.34</td>
<td>12.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Drone</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rendezvous (Double)</td>
<td>0.40</td>
<td>0.80</td>
<td>1.64</td>
<td>2.73</td>
<td>19.04</td>
<td>1.39</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HyComp</td>
<td>0.31</td>
<td>1.46</td>
<td>4.57</td>
<td>19.34</td>
<td>28.35</td>
<td>12.71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SpaceEx</td>
<td>0.23</td>
<td>95.67</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SpaceEx</td>
<td>2.65</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dReach</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow*</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tool</td>
<td>10.16</td>
<td>1.64</td>
<td>3.72</td>
<td>84.84</td>
<td>13.3</td>
<td>15.36</td>
</tr>
</tbody>
</table>

Table 6. Comparison of HybridSynchAADL and the other tools for Inv_1 at all bounds in using complex control logici. The '-' means that a counterexample is not found before a timeout.
<table>
<thead>
<tr>
<th>Model</th>
<th>Method</th>
<th>B = 1</th>
<th>B = 2</th>
<th>B = 3</th>
<th>B = 4</th>
<th>B = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AVG</td>
<td>STDEV</td>
<td>AVG</td>
<td>STDEV</td>
<td>AVG</td>
<td>STDEV</td>
</tr>
<tr>
<td>random</td>
<td>0.32</td>
<td>0.01</td>
<td>0.57</td>
<td>0.01</td>
<td>1.08</td>
<td>0.02</td>
</tr>
<tr>
<td>symbolic</td>
<td>0.32</td>
<td>0.01</td>
<td>0.57</td>
<td>0.01</td>
<td>1.08</td>
<td>0.02</td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>0.39</td>
<td>0.00</td>
<td>0.82</td>
<td>0.01</td>
<td>1.67</td>
<td>0.03</td>
</tr>
<tr>
<td>portfolio</td>
<td>0.39</td>
<td>0.00</td>
<td>0.82</td>
<td>0.01</td>
<td>1.67</td>
<td>0.03</td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>0.47</td>
<td>0.01</td>
<td>1.11</td>
<td>0.03</td>
<td>2.67</td>
<td>0.01</td>
</tr>
<tr>
<td>portfolio</td>
<td>0.47</td>
<td>0.01</td>
<td>1.11</td>
<td>0.03</td>
<td>2.67</td>
<td>0.01</td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WaterTank</td>
<td>0.33</td>
<td>0.00</td>
<td>0.53</td>
<td>0.01</td>
<td>0.85</td>
<td>0.02</td>
</tr>
<tr>
<td>symbolic</td>
<td>0.33</td>
<td>0.00</td>
<td>0.53</td>
<td>0.01</td>
<td>0.85</td>
<td>0.02</td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>0.43</td>
<td>0.01</td>
<td>0.80</td>
<td>0.01</td>
<td>1.43</td>
<td>0.04</td>
</tr>
<tr>
<td>symbolic</td>
<td>0.43</td>
<td>0.01</td>
<td>0.80</td>
<td>0.01</td>
<td>1.43</td>
<td>0.04</td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>0.54</td>
<td>0.01</td>
<td>1.09</td>
<td>0.01</td>
<td>2.14</td>
<td>0.05</td>
</tr>
<tr>
<td>symbolic</td>
<td>0.54</td>
<td>0.01</td>
<td>1.09</td>
<td>0.01</td>
<td>2.14</td>
<td>0.05</td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>0.69</td>
<td>0.01</td>
<td>1.70</td>
<td>0.05</td>
<td>3.83</td>
<td>0.10</td>
</tr>
<tr>
<td>portfolio</td>
<td>0.69</td>
<td>0.01</td>
<td>1.70</td>
<td>0.05</td>
<td>3.83</td>
<td>0.10</td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>0.98</td>
<td>0.02</td>
<td>3.00</td>
<td>0.06</td>
<td>13.06</td>
<td>0.43</td>
</tr>
<tr>
<td>symbolic</td>
<td>0.98</td>
<td>0.02</td>
<td>3.00</td>
<td>0.06</td>
<td>13.06</td>
<td>0.43</td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>1.32</td>
<td>0.03</td>
<td>5.66</td>
<td>0.06</td>
<td>17.63</td>
<td>0.22</td>
</tr>
<tr>
<td>symbolic</td>
<td>1.32</td>
<td>0.03</td>
<td>5.66</td>
<td>0.06</td>
<td>17.63</td>
<td>0.22</td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>0.94</td>
<td>0.01</td>
<td>2.50</td>
<td>0.02</td>
<td>36.38</td>
<td>0.77</td>
</tr>
<tr>
<td>portfolio</td>
<td>0.94</td>
<td>0.01</td>
<td>2.50</td>
<td>0.02</td>
<td>36.38</td>
<td>0.77</td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>1.32</td>
<td>0.03</td>
<td>10.64</td>
<td>0.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>1.32</td>
<td>0.03</td>
<td>10.64</td>
<td>0.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>1.70</td>
<td>0.01</td>
<td>6.18</td>
<td>0.08</td>
<td>98.39</td>
<td>2.56</td>
</tr>
<tr>
<td>symbolic</td>
<td>1.70</td>
<td>0.01</td>
<td>6.18</td>
<td>0.08</td>
<td>98.39</td>
<td>2.56</td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>1.47</td>
<td>0.00</td>
<td>11.41</td>
<td>0.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>1.47</td>
<td>0.00</td>
<td>11.41</td>
<td>0.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>1.89</td>
<td>0.00</td>
<td>16.78</td>
<td>0.08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>1.89</td>
<td>0.00</td>
<td>16.78</td>
<td>0.08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>1.41</td>
<td>0.01</td>
<td>15.22</td>
<td>0.26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>1.41</td>
<td>0.01</td>
<td>15.22</td>
<td>0.26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>2.01</td>
<td>0.03</td>
<td>20.23</td>
<td>0.40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>2.01</td>
<td>0.03</td>
<td>20.23</td>
<td>0.40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>random</td>
<td>.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>2.81</td>
<td>0.05</td>
<td>26.28</td>
<td>0.53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>symbolic</td>
<td>2.81</td>
<td>0.05</td>
<td>26.28</td>
<td>0.53</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7. HybridSynchAADL portfolio analysis. The ‘-’ 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 bounds \( T' \geq T \) are grayed out.
<table>
<thead>
<tr>
<th>Model</th>
<th>N</th>
<th>Symbolic with merging</th>
<th>Symbolic without merging</th>
<th>Model</th>
<th>N</th>
<th>Symbolic with merging</th>
<th>Symbolic without merging</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>(k)</td>
<td># SMT call (k)</td>
<td>State</td>
<td>Time</td>
<td>(k)</td>
</tr>
<tr>
<td>Drone</td>
<td>2</td>
<td>1</td>
<td>0.2</td>
<td>3.5</td>
<td>61.0</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.4</td>
<td>14.5</td>
<td>144.0</td>
<td>3</td>
<td>25.6</td>
<td>3.1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.7</td>
<td>35.8</td>
<td>227.0</td>
<td>4</td>
<td>544.3</td>
<td>54.6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.6</td>
<td>67.2</td>
<td>310.0</td>
<td>5</td>
<td>3675.7</td>
<td>324.4</td>
</tr>
<tr>
<td>Thermostat</td>
<td>1</td>
<td>1.2</td>
<td>0.2</td>
<td>5.3</td>
<td>91.0</td>
<td>2</td>
<td>6.0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.6</td>
<td>24.6</td>
<td>215.0</td>
<td>3</td>
<td>2315.3</td>
<td>270.6</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.3</td>
<td>65.8</td>
<td>339.0</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3.7</td>
<td>128.8</td>
<td>463.0</td>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.3</td>
<td>7.2</td>
<td>121.0</td>
<td>2</td>
<td>67.6</td>
<td>6.1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.8</td>
<td>36.2</td>
<td>286.0</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2.3</td>
<td>103.3</td>
<td>451.0</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7.2</td>
<td>208.2</td>
<td>616.0</td>
<td>5</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 8. Symbolic analysis with merging and without merging. | Const | denotes the accumulated size of SMT constraints in thousands.
<table>
<thead>
<tr>
<th>Model</th>
<th>B</th>
<th>N = 2</th>
<th>N = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time sample = 1</td>
<td>Time sample = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Average (s)</td>
<td>STDEV</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>8.6</td>
<td>9.9</td>
</tr>
<tr>
<td>Thermostat</td>
<td>100</td>
<td>8.5</td>
<td>10.0</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>10.3</td>
<td>12.5</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>9.5</td>
<td>13.2</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>13.6</td>
<td>18.7</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>14.5</td>
<td>20.0</td>
</tr>
<tr>
<td>Water Tank</td>
<td>50</td>
<td>7.8</td>
<td>9.8</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>8.2</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>9.3</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>9.0</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>9.7</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>10.2</td>
<td>3.8</td>
</tr>
<tr>
<td>Rendezvous</td>
<td>50</td>
<td>8.8</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>8.8</td>
<td>5.5</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>10.1</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>15.4</td>
<td>8.5</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>15.6</td>
<td>9.2</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>20.7</td>
<td>10.2</td>
</tr>
<tr>
<td>Bond Formation</td>
<td>50</td>
<td>7,163.6</td>
<td>207.5</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>7,854.2</td>
<td>169.3</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>8,196.7</td>
<td>265.8</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Dose Formation</td>
<td>50</td>
<td>138.8</td>
<td>96.5</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>154.3</td>
<td>4.6</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>203.1</td>
<td>91.7</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>812.2</td>
<td>89.3</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>1,616.3</td>
<td>361.3</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>2,855.1</td>
<td>620.7</td>
</tr>
</tbody>
</table>

Table 9. Analyzing distributed asynchronous models. \( \text{Time sample} \) is the number of the predefined values for sampling and actuating times, and the number of concrete states is written in thousands.