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* [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 [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 [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 [25] combined with SMT solving [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 [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 HYBRIDSYNCH-AADL tool is available at `https://hybridsynchaadl.github.io`.

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

2 Preliminaries

2.1 PALS and Hybrid PALS

When the infrastructure guarantees bounds on clock skews, network delays, and execution times, the PALS pattern [5,39] 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 Γ on clock skews, network delays, and execution times, and a period p of each round, the PALS transformation gives us the asynchronous distributed real-time system $PALS(SD, \Gamma, p)$, which is stuttering bisimilar to SD.

The synchronous design SD is formalized as the synchronous composition of an *ensemble* of state machines with input and output ports [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 $c_M : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, 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 [39]. In its ith iteration, a controller M samples the values of its environment at time $c_M(i \cdot p) + t_s$, where t_s is the *sampling time*, and then executes a transition (based on the sampled values, the values received from other controllers, and the controller's own state). As a result, the new control command is received by the environment at time $c_M(i \cdot p) + t_a$, where t_a is the *actuating time*.

2.2   AADL

The *Architecture Analysis & Design Language* (AADL) [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*.[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* [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.

---

[1] 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 $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory with Σ a signature (declaring sorts, subsorts, and function symbols) and E a set of equations; and R is a set of rewrite rules $l : t \longrightarrow t'$ **if** *cond*, where l is a label, t and t' are terms, and *cond* is a conjunction of equations and rewrites. A *rewrite* $t \longrightarrow^* t'$ holds if t' is reachable from t using the rewrite rules in $\mathcal{R}$.

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

$$< o : C \mid att_1 : v_1, \ldots, att_n : v_n >,$$

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

In rewriting modulo SMT [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 ϕ holds: i.e., given the underlying SMT theory $\mathcal{T}$, we have:

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

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

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

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

There are two kinds of components in HYBRIDSYNCHAADL: *continuous* environments and *discrete* controllers. Environments are specified as system components whose continuous dynamics is specified using continuous functions or ordinary differential equations. Discrete controllers are usual AADL software components in the Synchronous AADL subset [9, 12] of AADL.[2] The top-level system component declares the following properties to state that the model is a synchronous design and to declare the period of the system, respectively.

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

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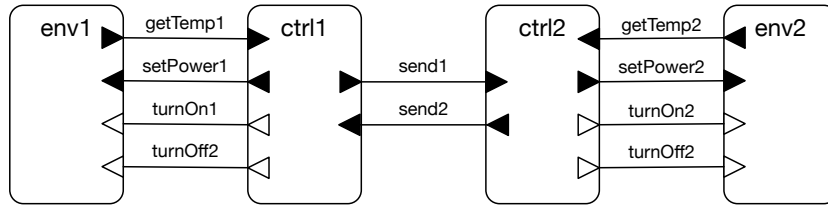

Fig. 1. A networked thermostat system.

---

[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 =>
      "dynamics₁" in modes (mode₁), ..., "dynamicsₙ" in modes (modeₙ);
```

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

```
d/dt(x₁) = e₁(x₁,...,xₙ); ... ; d/dt(xₙ) = eₙ(x₁,...,xₙ);
```

If a closed-form solution of ODEs is known, we can directly specify concrete continuous functions, which are parameterized by a time parameter t and the initial values $x_1(0), \ldots, x_n(0)$ of the variables $x_1, \ldots, x_n$:

```
x₁(t) = e₁(t,x₁(0),...,xₙ(0)); ... ; xₙ(t) = eₙ(t,x₁(0),...,xₙ(0));
```

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

An environment component interacts with discrete controllers by sending its state values, and by receiving actuator commands that may update the values of state variables or trigger mode (and hence trajectory) changes. This behavior is specified in HYBRIDSYNCHAADL using *connections between ports and data subcomponents*. A connection from a data subcomponent inside the environment to an output data port of an environment component declares that the value of the data subcomponent is "sampled" by a controller through the output port of the environment component. A connection from an environment's input port to a data subcomponent inside the environment declares that a controller command arrived at the input port and updates the value of the data subcomponent. When a discrete controller sends actuator commands, some input ports of the environment component may receive no value (more precisely, some "don't care" value $\perp$). In this case, the behavior of the environment is unchanged.

Example 2. Figure 3 gives an environment component `RoomEnv` for our networked thermostat system. Figure 2 shows its architecture. It has data output port `temp`, data input port `power`, and event input ports `on_control` and `off_control`. The implementation of `RoomEnv` has two data subcomponents `x` and `p` to denote the temperature of the room and the heater's power, respectively. They represent the state variables of `RoomEnv` with the specified initial values.

There are two modes `heaterOn` and `heaterOff` with their respective continuous dynamics, specified by `Hybrid_SynchAADL::ContinuousDynamics`, using continuous functions over time parameter t, where `heaterOff` is the initial mode. Because `p` is a constant, `p`'s dynamics `d/dt(p) = 0` is omitted. The value `x` 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 -> 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 -> temp`.
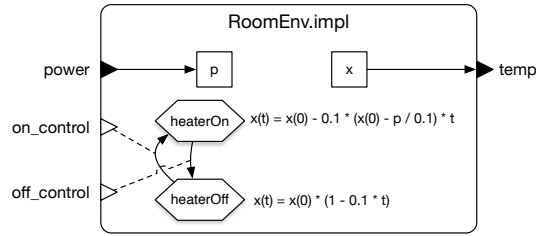
Fig. 2. An environment of the thermostat controller.

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

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

Fig. 3. A `RoomEnv` component.

3.2 Controller Components

Discrete controllers are usual AADL software components in the Synchronous AADL subset [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:

```
Dispatch_Protocol => Periodic;
```

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

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

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

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

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

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

Example 3. Consider again our networked thermostat system. Figure 4 shows a thread component `ThermostatThread` that turns the heater on or off depending on the average value `avg` of the current temperatures of the two rooms. It has event output ports `on_control` and `off_control`, data input ports `curr` and `tin`, and data output ports `set_power` and `tout`. The ports `on_control`, `off_control`, `set_power`, and `curr` are connected to an environment, and `tin` and `tout` are connected to another controller component (see Fig. 5). The implementation has 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.

```
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.[3] Because an environment does not "actively" send data for sampling, every output port of an environment must be a *data* port, whereas its input ports could be of any kind.

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

Example 4. Figure 5 shows an implementation of a top-level system component `TwoThermostats` of our networked thermostat system, depicted in Figure 1. This component has no ports and contains two thermostats and their environments. The controller system component `Thermostat.impl` is implemented using the thread component `ThermostatThread.impl` in Fig. 4, and the environment component `RoomEnv.impl` is given in Fig. 3. Each discrete controller `ctrl`i, for $i = 1, 2$, is connected to its environment component `env`i using four connections `turnOn`i, `turnOff`i, `setPower`i, and `getTemp`i. The controllers `ctrl1` and `ctrl2` are connected with each other using delayed data connections `send1` and `send2`.

4 The HYBRIDSYNCHAADL Tool

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

---

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

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

Fig. 5. A top level component with two thermostat controllers.

4.1   Property Specification Language

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

Atomic propositions are given by Boolean expressions in the AADL Behavior Annex syntax. Each identifier is fully qualified with its component path in the AADL syntax. A *scoped expression* of the form *path | exp* denotes that each component path of each identifier in the expression *exp* begins with *path*. A "named" atomic proposition can be declared with an identifier as follows:

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

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

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

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

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

```
reachability [name]: φ_init ==> φ_goal in time τ_bound
```

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

$$path \mid exp$$

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

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

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

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

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

4.2   Tool Interface

Figure 6 depicts the architecture of the HYBRIDSYNCHAADL tool. The tool first statically checks whether a given model is a valid model that satisfies the syntactic constraints of HYBRIDSYNCHAADL in Section 3. It uses OSATE's code generation 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.
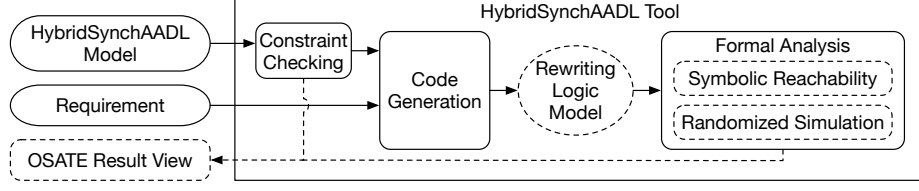
Fig. 6. The architecture of the HybridSynchAADL tool.
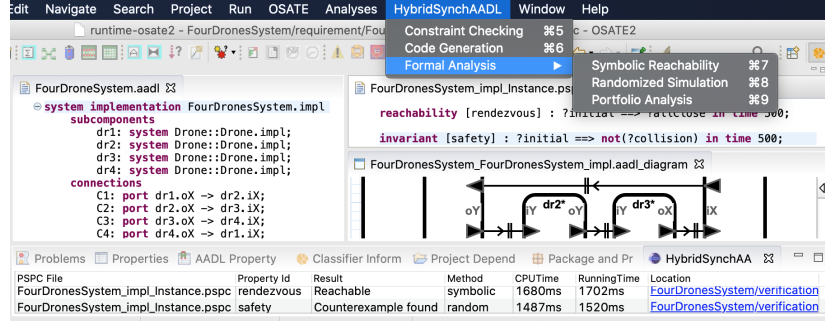
Fig. 7. Interface of the HybridSynchAADL tool.

By syntactically validating a HybridSynchAADL model, we ensure that the model satisfies all the syntactic constraints of HybridSynchAADL, and thus the corresponding Maude model is executable. For example, environment components (with `Hybrid_SynchAADL::isEnvironment`) can only contain data subcomponents of type `Base_Types::Float`, and must declare the continuous dynamics using `Hybrid_SynchAADL::ContinuousDynamics`. The tool checks other "trivial" constraints that are assumed in the semantics of HybridSynchAADL; e.g., all input ports are connected to some output ports.

HybridSynchAADL provides two formal analysis methods. *Symbolic reachability analysis* can verify that all possible behaviors—imposed by sensing and actuation times based on imprecise clocks—satisfy a given requirement;[4] 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.

Fig. 8. Rendezvous and formation control of distributed drones

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

5 Case Study: Collaborating Autonomous Drones

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

5.1 Distributed Consensus Algorithms

We use distributed consensus algorithms [45] 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:

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

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

In formation control problems [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:

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

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

For both cases, a simplified model for drones with *single-integrator* dynamics is also considered, assuming acceleration is negligible. Acceleration $\vec{a}_i$ is always 0, and the controller of a drone directly gives the value of velocity as a control input. For single-integrator dynamics, the following equations provide velocity $\vec{v}_i$ for rendezvous and formation control, respectively [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 HYBRIDSYNCH-AADL 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`.
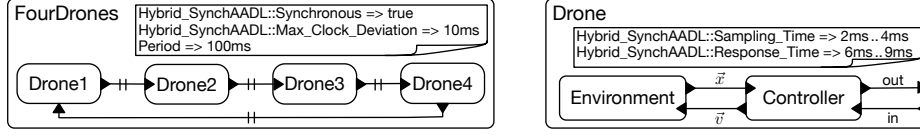
Fig. 9. The AADL architecture of four drones (left), and a drone component (right).

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 HYBRIDSYNCHAADL, the connections between drone components are delayed and the output ports have some initial values. The maximal clock skew is given by `Hybrid_SynchAADL::Max_Clock_Deviation`.

Drone Component. 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 HYBRID-SYNCHAADL portfolio analysis.

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

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

We define three atomic propositions `collision`, `gather`, and `initial` for four drones `dr1`, `dr2`, `dr3`, and `dr4`. Two drones collide if the distance between them is less than 0.1. All nodes have gathered if the distance between each pair of nodes is less than 1. The initial values of x, y, velx and 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`.

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

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

proposition [initial]:
  abs(dr1.env.x - 1.1) < 0.01 and abs(dr1.env.y - 1.5) < 0.01 and
  abs(dr2.env.x + 1.5) < 0.01 and abs(dr2.env.y + 1.1) < 0.01 and
  abs(dr3.env.x - 1.5) < 0.01 and abs(dr3.env.y - 1.1) < 0.01 and
  abs(dr4.env.x + 1.1) < 0.01 and abs(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 *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 `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 ?velconst ==> not ?collision in time 500;
reachability [rendezvous]: ?initial and ?velconst ==> ?gather in time 500;

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

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

PSPC File	Property Id	Result	Method	CPUTime	RunningTime	Location
FourDronesSystem_impl_Instance2.pspc	safety	No counterexample found	symbolic	837345ms	867321ms	FourDronesSystem/verific:
FourDronesSystem_impl_Instance2.pspc	rendezvous	Reachable	random	116ms	134ms	FourDronesSystem/verific:

Fig. 17. Results for the modified properties.

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 HYBRID-SYNCHAADL 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 $\phi(x_1, \ldots, x_n) \parallel obj(x_1, \ldots, x_n)$ 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 $\phi \parallel conf$ with *conf* a multiset of object patterns.

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

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

Notice that f can be *nondeterministic*; e.g., there could be many constrained terms $\phi_1' \parallel t_1', \ldots, \phi_n' \parallel t_n'$ with $f(\phi \parallel t) \rightsquigarrow \phi_i' \parallel t_i'$ for $1 \leq i \leq n$. In this paper, we often call f a *semantic function* if f is a deterministic semantic operation.

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

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

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

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

6.2 SMT Expressions

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

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

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

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

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

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

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

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

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

of sort `BoolExp` represents the pattern $(1 + x_1 > 2 * x_2 \ or \ b_1) = (b_2 \ and \ true)$.[5]

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

sort BoolExp .
subsorts BoolValue SMTBoolVar < BoolExp < Exp .
op not_ : BoolExp -> BoolExp [prec 53] .
op _and_ : BoolExp BoolExp -> BoolExp [assoc comm prec 55] .
...
sort RealExp.
subsorts RealValue SMTRealVar < RealExp < Exp .
op -_ : RealExp -> RealExp .
op _+_ : RealExp RealExp -> RealExp [assoc comm prec 33] .
...
sort UnitExp .
subsorts UnitValue < UnitExp < Exp .
op * : -> UnitValue [ctor] .
op _===_ : UnitExp UnitExp -> BoolExp [gather (e E) prec 51] .
```

6.3 Representing HYBRIDSYNCHAADL Models

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

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

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

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

```
class Ensemble .
subclass Ensemble < Component .

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

---

[5] [1] and [2] are symbolic values of sort `RealValue`, [*true*] is a symbolic value of sort
`BoolValue`, $r(1)$ and $r(2)$ are symbolic variables of sort `RealVar`, and $b(1)$ and $b(2)$
are symbolic variables of sort `BoolVar`.

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

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

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

A transition system of a thread is represented as a (semi-colon-separated) set of transitions of the form: s -[*guard*]-> s' {*actions*}, where s is a source state, s' is a destination state, *guard* is a Boolean condition, and {*actions*} is a behavior action block.

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

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

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

A continuous dynamics in flows is represented as a (semi-colon-separated) set of continuous dynamics assignments of the form m [*continuous_dynacmis*]where m is a mode and *continuous_dynacmis* is either ODEs or continuous functions as explained in Section 3.

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

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

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

In HYBRIDSYNCHAADL, a data component has either no value (more precisely, some "don't care" value $\perp$) 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., $\perp$) if b is *false*, and a value represented by the expression e if b is *true*.

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

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

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

We distinguish between the ports of discrete components and the ports of environment components, because their behaviors are significantly different. As mentioned in Section 3, the communication between discrete components is *delayed*, i.e., outputs generated in one iteration are available at the destination discrete components in the next iteration. On the other hand, the communication between discrete and environment components is *immediate*.

Ports of discrete components are represented as instances of the `DataPort` class. An input data port contains the extra attribute `cache` to keep the previously received value; if an input port p has received $\perp$ (i.e., e # *false*) in the latest dispatch, the thread can use the value in the `cache`, while the behavior annex expression p'`fresh` becomes *false* [9, 12].

```
class DataPort .
subclass DataPort < Port .

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

Ports of environment components are represented as instances of the `EnvPort` class, containing two extra attributes `target` and `envCache`. Because the port communication depends on the sampling and actuating times of the connected controller, an environment port keeps the identifier of the `target` component.[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 `-->` p_o, where p_i denotes the source port name and p_o denotes the target port name. The name of a port p in a subcomponent c is written as a term c `..` p. A connection from an output port p_1 in c_1 to an input port p_2 in c_2 is written as c_1 `..` p_1 `-->` c_2 `..` p_2. A level-up (resp., level-down) connection, connecting a port p in a subcomponent c to the port p' in the "current" component is written as the term c `..` p `-->` p' (resp., p' `-->` c `..` p).

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

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

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

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

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

```
< TwoThermostatsimplInstance : System |
  features : none,
  subcomponents : < ctrl1 : System | ... >
                  < ctrl2 : System | ... >
                  < env1 : Env | ... >
                  < env2 : Env | ... >,
  connections : ctrl1 .. oncontrol --> env1 .. oncontrol ;
                ctrl1 .. offcontrol --> env1 .. offcontrol ;
                ctrl1 .. setpower --> env1 .. power ;
                env1 .. temp --> ctrl1 .. curr ;
```

```
                ctrl1 .. tou --> ctrl2 .. tin ;
                ctrl2 .. oncontrol --> env2 .. oncontrol ;
                ctrl2 .. offcontrol --> env2 .. offcontrol ;
                ctrl2 .. setpower --> env2 .. power ;
                env2 .. temp --> ctrl2 .. curr ;
                ctrl2 .. tou --> ctrl1 .. tin,
  properties : TimingProperties::Period => {{10}} ;
                HybridSynchAADL::Synchronous => {{true}} >
```

Example 7. An instance of `ThermostatThread.impl` in Fig. 4 can be represented by the following object, where the initial value of `avg` is a symbolic variable $r(0)$. For parsing purposes, syntactic values are enclosed by [[...]], component identifiers are enclosed by c{...}, and port identifiers are enclosed by f{...} in transitions.

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

```
    currState : init,
    completeStates : init,
    varGen : < 1, 0 >
>
```

Example 8. An instance of `RoomEnv.impl` in Fig. 3 can be represented by the following object. The contents of environment ports include symbolic variables, such as $r(1), b(0),$ and $b(3)$, to symbolically encode the immediate communication (see Section 6.7). For parsing purposes, syntactic values are enclosed by [[...]] and component identifiers are enclosed by c{...} in flows.

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

6.4   Symbolic Synchronous Steps

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

```
op executeStep : ConstObject ~> ConstObject .

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

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

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

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

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

6.5 Ensemble Behavior

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

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

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

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

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

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

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

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

```
              location(L) complete(LS) trans(TRS)  local(defaultVal(VIS))
              lbound(#loopbound#)  tbound(#transbound#)    vargen(GEN))
   => L' | FMAP' | DATA' | PHI' | GEN'
/\ check-sat(PHI and PHI') .
```

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

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

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

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

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

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

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

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

Consider a port P with a content E # B and a cached content E' # B'. If the content is present (i.e., B is *true*), P corresponds to the pair (E # B) : true in the resulting map FMAP; otherwise, P corresponds to the pair (E' # B') : false

using the cached value. The resulting content can be compactly represented as `((B ? E : E') # (B or B'))` using the conditional operator. Then, the `cache` attribute is updated, and the content is set absent.

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

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

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

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

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

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

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

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

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

```
crl [trans]:
    execTrans(location(SL) trans(TRS) tbound(s(N)) local(VAL) REST)
 =>
    execTStep(VAL, location(L') trans(TRS) tbound(N)
```

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

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

If the next state L' is a complete state, the operation ends with a result,
provided that the constraint PHI is satisfiable.[8] Otherwise, execTrans is applied
again with the new configuration. The number of such iterations is limited by
the bound tbound (in the rule trans) to avoid infinite symbolic computation.

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

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

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

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

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

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

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

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

---

[8] 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.

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

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

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

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

```
eq eval(AE1 + AE2, REST) = eval(AE1, REST) + eval(AE2, REST) .
eq (E1 # B1) + (E2 # B2) = (E1 + E2) # (B1 and B2) .
```

Executing Actions. The semantic operation `execAction` computes a behavior action based 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.

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

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

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

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

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

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

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

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

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

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

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

6.7   Environment Behavior

In HYBRIDSYNCHAADL, an environment component interacts with each of its controllers *in a single iteration*. For example, consider an environment E that is connected to two controllers C_1 and C_2. Figure 18 shows a timeline of their interactions. Initially, the state variables of E have values $\vec{v}_0$ and change over time according to E's continuous dynamics. For $i = 1, 2$, environment E sends the state values $\vec{v}_n$ at time t_n^s to controller C_n, and receives C_n's command α_n at time t_n^a (and may also change its continuous dynamics by α_n), according to the sampling and actuating times of C_n.

The semantics of environment components cannot be directly specified as synchronous composition. Indeed, the environment behavior is *asynchronous*, since the order of "interaction events" in a single iteration (e.g., *sampling*(C_1), *sampling*(C_2), *response*(C_1), and *response*(C_2) in Fig. 18) can lead to different behaviors. The synchronous semantics requires that the interactions between components must be *delayed*, but the interactions between environment and controller components are *immediate*. Hence, any *concrete semantics* of HYBRID-SYNCHAADL is not likely definable as synchronous composition.
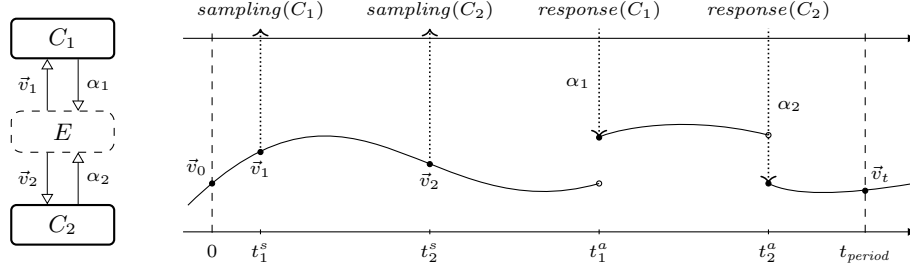
Fig. 18. Interactions between an environment E and two controllers C_1 and C_2

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 HYBRIDSYNCHAADL, because a controller may include arbitrarily complex (hierarchical) subcomponents. Defining environment restrictions for generic AADL components is thus very difficult.

In this paper we present an alternative approach to symbolically encode asynchronous interactions in a *modular* way. We encode the values of input and output ports at different sampling and actuating times into symbolic variables, and perform executeStep of each component *independently*. The correspondence between the input and output ports 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:

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

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

Therefore, the behavior of environment components is also specified in the operation executeStep in our symbolic semantics. The operation executeStep builds *constrained objects* with logical constraints to encode the environment behavior. All information required for interaction with discrete controllers—including the values of input and output ports at different sampling and actuating times—is encoded as a set of symbolic variables. The immediate communication between environment and controller components is also encoded as symbolic constraints. As a result, the semantics of ensemble components with environment components is specified in the same way as in Section 6.5.

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

Fig. 19. The behavior of readEnvFeature.

Executing Symbolic Steps. The following rule defines the behavior of environment components. The function readEnvFeature returns a map from each input port to its symbolic content, and writeEnvFeature updates the content of each output port. These functions also return extra constraints to encode the environment communication. The operation execEnv builds logical constraints to encode the behavior of the environment in one-step iteration, and each of them is assigned to the pattern L' | FMAP' | DATA' | PHI' | GEN2. The function check-sat then invokes an SMT solver to check whether the generated constraint is satisfiable.

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

Environment Feature Operations. Given a set of environment ports, the function readEnvFeature builds a map from each port identifier to a symbolic variable denoting the value *sent from the controller in the same iteration*. As described in Fig. 19, we use an extra attribute envCache that contains the symbolic variable 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

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

Fig. 20. The behavior of writeEnvFeature.

of the i-th iteration. The function readEnvFeature relates the port identifier p to a fresh variable x_i, and generates the constraint $x_{i-1} = d_{i-1}$.

This function is defined by the following equations using an auxiliary function with extra arguments to carry intermediate results. In the third equation, each input port PI is related to a symbolic content V # BV with fresh variables V and BV, the current content E # B and envCach E' # B' are declared to be identical as a constraint, the envCach attribute is updated, and the content attribute is set absent. In the last equation, each output port is related to E # [false], indicating ⊥ with the second item [false].

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

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

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

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

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

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


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


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

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

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

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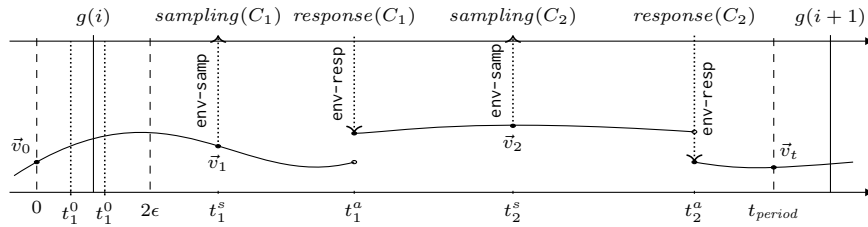

Fig. 21. The behavior of an environment interacting with two controllers.

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 $g(i+1) - g(i) = t_{period}$.

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

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

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

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

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

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

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

---

[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 dReal [31]. Because dReal has not been integrated with Maude, the current version of the semantics only supports continuous functions.

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

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

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

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

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

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

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

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

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 ==> PI from CI to PI, provided that PI is connected to the controller C.

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

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

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

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

Environment Actuation. The rules with label env-resp specify the behavior of actuation operations. There are two cases: either a mode change is triggered or not. In the first rule, there is a mode transition from the current mode L, where one of its triggers, PI, which is connected to the controller C (validTarget), has received a content (isPortPresent). In the second rule, all input ports in the mode transitions from L that are connected with C 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 `_?_:_`.

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

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

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

6.8 Merging Symbolic States

This section presents a state-space reduction method for our symbolic semantics of HYBRIDSYNCHAADL. Recall that the behavior of threads and environments is specified by using two operations `execTrans` and `execEnv`. Even for one component, `executeStep` can produce many different execution results. In particular, for an environment interacting with n controllers, the rules in Section 6.7 can generate $O((2n)!/2^n)$ different symbolic execution results, according to nondeterministic orders of sampling and actuating events. The modular encoding can symbolically eliminate the interleavings of components, but cannot eliminate the nondeterminism in a component.

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

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

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

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

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

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

Algorithm 1: Semantic operation f with state merging

    Input: A constrained object $\phi \parallel t$
    Output: A set of constrained objects

1 $post \longleftarrow \{(\phi' \parallel t') \mid f(\phi \parallel t) \rightsquigarrow \phi' \parallel t'\};$

2 $\widehat{post} \longleftarrow \{(\phi' \wedge \psi \parallel t'') \mid (\psi \parallel t'') = abs(t'),\ \phi' \parallel t' \in post\};$

3 **while** $\exists(\varphi_1 \parallel t_1), (\varphi_2 \parallel t_2) \in \widehat{post}$ *with mergeable* t_1 *and* t_2 **do**

4 $\varphi \parallel u$ is a merged term of $\varphi_1 \parallel t_1$ and $\varphi_2 \parallel t_2$;

5 $\widehat{post} \longleftarrow (\widehat{post} \cup \{\varphi \parallel u\}) \setminus \{\varphi_1 \parallel t_1,\ \varphi_2 \parallel t_2\};$

6 **return** $\widehat{post}$;

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

Proof. By definition (Section 2), $u \in [\![(\phi_1 \vee \rho\phi_2) \parallel t_1]\!]$ iff there is a substitution θ such that $u = \theta t_1$ and $\mathcal{T} \models \theta(\phi_1 \vee \rho\phi_2)$. Because $t_1 = \rho t_2$, $u = \theta\rho t_2$. Also, $\mathcal{T} \models \theta\phi_1$ or $\mathcal{T} \models \theta\rho\phi_2$. Thus, one of the following cases must hold: (i) $u = \theta t_1$ and $\mathcal{T} \models \theta\phi_1$, or (ii) $u = \theta\rho t_2$ and $\mathcal{T} \models \theta\rho\phi_2$. By definition, $u \in [\![\phi_1 \parallel t_1]\!]$ or $u \in [\![\rho(\phi_2 \parallel t_2)]\!]$. Because ρ is a renaming substitution, $u \in [\![\rho(\phi_2 \parallel t_2)]\!]$ iff $u \in [\![\phi_2 \parallel t_2]\!]$. Consequently, $[\![(\phi_1 \vee \rho\phi_2) \parallel t_1]\!] = [\![\phi_1 \parallel t_1]\!] \cup [\![\phi_2 \parallel t_2]\!]$. ∎

Algorithm 1 shows the new "merging" operation. It collects all the execution results by executeStep and merges all mergeable results. For our 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 HYBRIDSYNCHAADL.

In order to obtain abstractions of built-ins for two terms t_1 and t_2, we define a function symAbs(t_1, t_2) that returns a triple (u, ϕ_1, ϕ_2), where $\phi_1 \parallel u$ and $\phi_2 \parallel u$ are abstractions of t_1 and t_2, respectively, with the same set of fresh SMT variables. (We do not need to perform extra renaming by using the same fresh variables.) For example, symAbs(e_1, e_2) for two SMT expressions e_1 and e_2 is a triple $(x, x = e_1, x = e_2)$ with a fresh variable x, specified using the following equation, where an extra arguments GEN is used to generate fresh variables.

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

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

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

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

The new operations execTransMerge and execEnvMerge find all the execution results obtained from execTrans and execEnv, respectively, and merge them into a single term using Algorithm 1. The function collectResults uses Maude's reflective features to compute a ;;-separated list of execution results, each of which has the form L | FMAP | DATA | PHI | GEN, as explained in Sections 6.6 and 6.7. The function symMerge syntactically merges those terms into a single term, by means of the abstraction function symAbs for each pattern.

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

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

6.9 Property Specification Language Semantics

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

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

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

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

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

The function evalPS(E, COMPS) evaluates a normalized expression E to its data content $e \# b$, given a set of components COMPS. The following equations specify evalPS, where BehComponent is a superclass of both classes Thread and Env. Notice that evalPS uses eval as a subroutine, which is defined in Section 6.6 to evaluate expressions with respect to behavior configurations.

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

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

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

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

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

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

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

```
search [N_bound] {([[φ_init]] initState) || initState}
            =>* {COND || OBJ}
      such that check-sat(COND and finalConst(OBJ) and not([[φ_inv]] OBJ)) .
```

Example 9. Consider the requirement inv in Example 5. This requirement inv corresponds to the following search command in Maude. The constant initState is replaced by the term representation of the entire model, and inRan1 and inRan2 are replaced by the related Boolean expressions.

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

7 Experimental Evaluation

This section evaluates the HYBRIDSYNCHAADL tool by addressing the following questions: (1) How effective is our tool compared to state-of-the-art CPS analysis tools? (2) How effective is our portfolio analysis method for finding bugs? (3) How effective is our novel state merging technique for symbolic analysis? (4) How effective is the Hybrid PALS methodology for reducing the complexity of analyzing virtually synchronous CPSs?

To answer these questions, we have analyzed HYBRIDSYNCHAADL models of networked thermostat and water tank systems (adapted from [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: $Inv_\top$, which holds, and $Inv_\bot$, 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 $\bigwedge_{i=1}^{3}((d_{i=1} \leq x_{in} - x_{sampled} \leq d_i) \rightarrow vel_x := c_i)$,
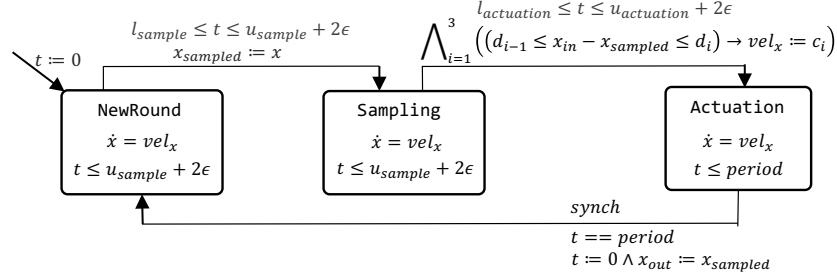
Fig. 22. A hybrid automaton component

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 [10] 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 $Inv_\top$

Model	Tool	Simplified Control Logic			Complex Control Logic		
		$N = 2$	$N = 3$	$N = 4$	$N = 2$	$N = 3$	$N = 4$
		Time B	Time B	Time B	Time B	Time B	Time B
Rend (single)	HSADDL	2.0 5	3.9 5	5.8 5	10.6 5	18.9 5	32.2 5
	HyComp	0.8 5	4.0 5	17.2 5	6.24 5	108.8 5	150.2 5
	SpaceEx	8.0 5	2230.3 3	4.5 1	1551.4 2	16.6 1	470.2 1
	dReach	1382.7 3	107.1 1	T/O -	982.7 1	T/O -	T/O -
	Flow*	3552.8 4	2725.5 2	1205.2 1	1267.8 2	422.6 1	T/O -
Form (single)	HSAADL	3.0 5	7.3 5	7.9 5	17.3 5	63.5 5	67.0 5
	HyComp	13.3 5	41.3 5	182.1 5	1586.1 5	652.2 4	844.4 4
	SpaceEx	91.9 2	2.8 1	114.8 1	27.7 1	501.4 1	T/O -
	dReach	139.0 1	T/O -	T/O -	1939.0 1	T/O -	T/O -
	Flow*	1464.7 2	873.4 1	T/O -	749.9 1	T/O -	T/O -
Thermostat	HSAADL	2.7 5	4.7 5	7.8 5	3.9 5	7.0 5	11.8 5
	HyComp	1.6 5	8.5 5	37.9 5	4.0 5	12.1 5	51.7 5
	SpaceEx	2.3 5	696.4 3	34.5 1	4.7 5	98.9 2	2.5 1
	dReach	341.6 3	57.5 1	T/O -	1341.6 2	T/O -	T/O -
	Flow*	3196.4 5	1240.7 2	977.7 1	318.8 4	1246.8 2	T/O -
Water Tank	HSAADL	1.3 5	2.5 5	4.2 5	1.6 5	3.3 5	5.6 5
	HyComp	0.5 5	6.0 5	37.1 5	4.3 5	14.9 5	98.8 5
	SpaceEx	0.8 5	877.1 5	28.0 1	1.4 5	331.9 2	3181.8 2
	dReach	1080.8 3	52.8 1	T/O 1	980.8 1	T/O -	T/O -
	Flow*	1070.1 3	96.4 1	T/O -	96.5 2	180.2 1	T/O -
Rend (double)	HSAADL	3.7 4	37.8 4	6.9 4	7.7 3	12.4 3	5.3 2
	SpaceEx	1147.6 3	81.1 1	T/O -	58.8 1	T/O -	T/O -
	dReach	2156.2 3	274.3 1	T/O -	2156.2 1	T/O -	T/O -
	Flow*	232.5 2	230.1 1	T/O -	T/O -	T/O -	T/O -
Form (double)	HSAADL	2.2 3	1.7 3	2.5 3	8.1 3	5.5 2	7.7 2
	SpaceEx	70.8 1	T/O -	T/O -	T/O -	T/O -	T/O -
	dReach	360.4 1	T/O -	T/O -	3124.2 1	T/O -	T/O -
	Flow*	116.8 1	3411.5 1	T/O -	T/O -	T/O -	T/O -

7.2 Analyzing Invariant Properties

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

Table 2. Comparison of HybridSynchAADL and the other tools for $Inv_\perp$

Model	Tool	Simplified Control Logic						Complex Control Logic					
		$N=2$		$N=3$		$N=4$		$N=2$		$N=3$		$N=4$	
		Time	B	Time	B	Time	B	Time	B	Time	B	Time	B
Rend (single)	HSADDL	2.4	3	4.2	3	5.9	3	84.8	4	17.5	3	66.7	4
	HyComp	8.9	3	11.5	3	192.6	3	462.7	4	357.2	3	T/O	-
	SpaceEx	5.1	3	2676.6	3	T/O	-	T/O	-	T/O	-	T/O	-
	dReach	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-
	Flow*	167.3	3	380.4	2	838.0	3	T/O	-	96.1	2	T/O	-
Form (single)	HSAADL	15.5	4	2.5	2	5.2	2	33.2	3	10.5	2	88.5	3
	HyComp	T/O	-	2.6	2	20.3	2	648.0	3	32.3	2	T/O	-
	SpaceEx	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-
	dReach	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-
	Flow*	T/O	-	45.3	1	291.3	2	T/O	-	1034.2	1	T/O	-
Thermostat	HSAADL	7.6	5	15.3	5	10.7	4	9.6	5	19.0	5	15.7	4
	HyComp	2.6	5	15.5	5	43.1	4	3.1	5	28.3	5	30.2	4
	SpaceEx	2.2	5	T/O	-	T/O	-	4.3	5	T/O	-	T/O	-
	dReach	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-
	Flow*	15.5	3	1718.1	4	T/O	-	45.5	3	T/O	-	T/O	-
Water Tank	HSAADL	1.8	3	3.1	3	4.8	3	5.8	5	11.9	5	11.7	4
	HyComp	0.1	3	1.9	3	4.2	3	3.6	5	61.8	5	1261.8	4
	SpaceEx	0.2	3	74.4	3	T/O	-	1.8	5	T/O	-	T/O	-
	dReach	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-
	Flow*	1.0	1	5.7	1	28.9	1	13.8	2	51.7	2	13.1	2
Rend (double)	HSAADL	1.4	2	16.3	2	2.8	2	6.7	2	11.4	2	16.7	2
	SpaceEx	15.2	2	T/O	-	T/O	-	881.6	2	T/O	-	T/O	-
	dReach	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-
	Flow*	2.2	2	25.4	2	2613.8	1	T/O	-	T/O	-	T/O	-
Form (double)	HSAADL	2.2	2	2.7	2	3.8	2	15.4	2	20.4	2	26.5	2
	SpaceEx	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-
	dReach	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-
	Flow*	14.0	2	1586.9	1	T/O	-	T/O	-	T/O	-	T/O	-

Model	Sampling	Actuation	ϵ	Invariant property
Thermostat	$20 \sim 30$	$60 \sim 70$	5	temperatures are between 20 and 50
Water tank	$30 \sim 40$	$70 \sim 80$	5	water levels are above 30
Rendezvous	$30 \sim 50$	$60 \sim 80$	10	distance between drones greater than 0.5
Formation	$30 \sim 50$	$60 \sim 80$	10	distance between drones greater than 0.3

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.[11] 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.
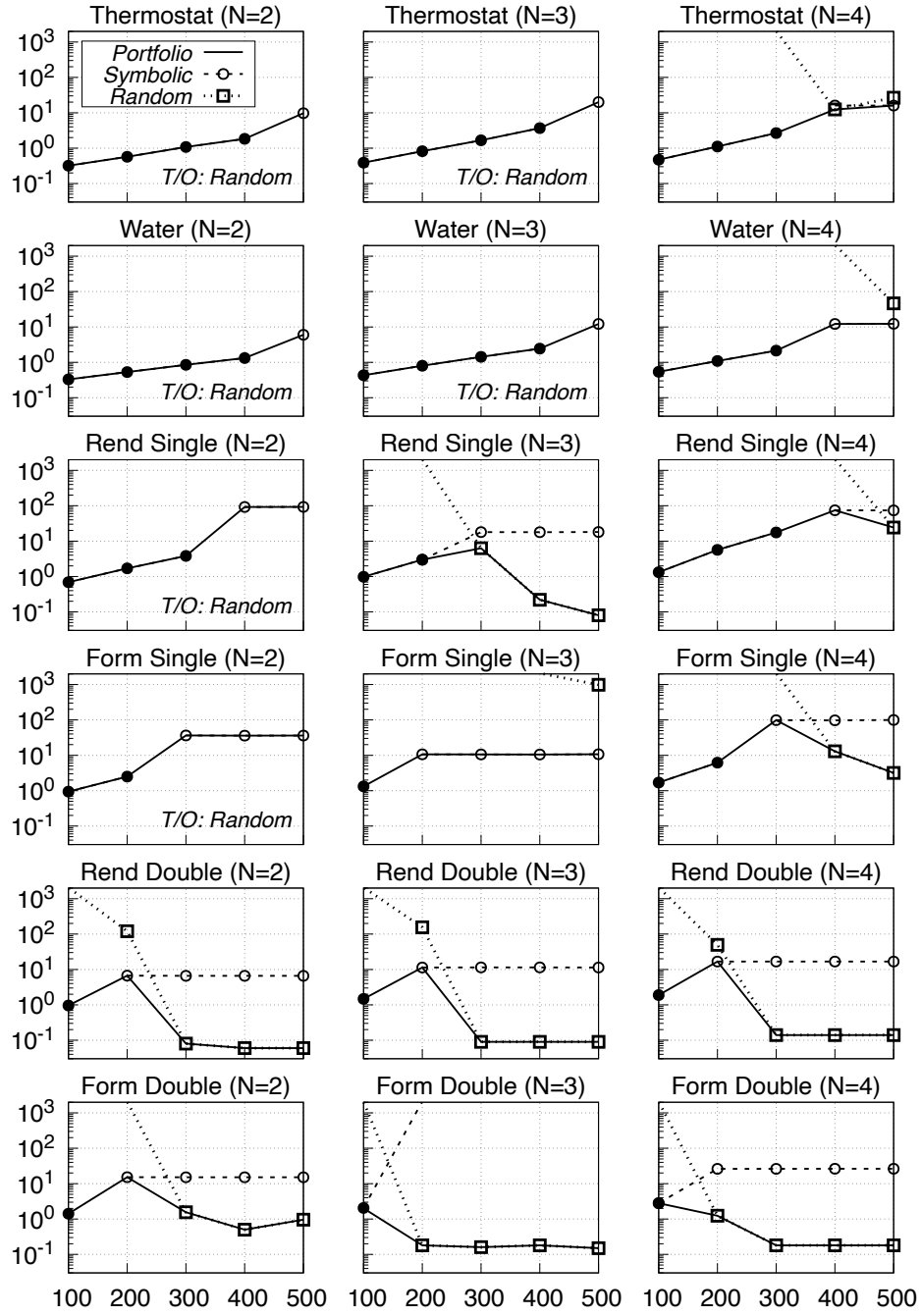
Fig. 23. HYBRIDSYNCHAADL portfolio analysis.
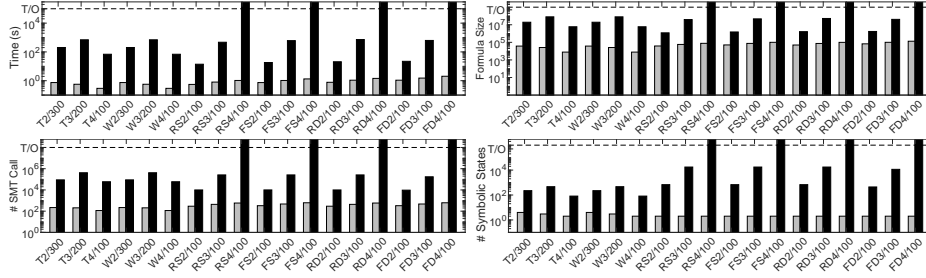
Fig. 24. Symbolic analysis with merging and without merging.
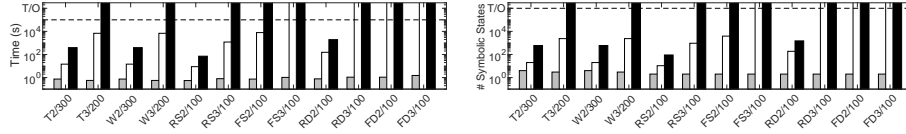
Fig. 25. Comparing Hybrid PALS models and distributed models

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 [41]. 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

[17, 24, 30], 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 [21, 22, 33, 36], GALS [32, 42], approximate synchrony [26], time-triggered architectures [47, 48], virtual synchrony [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 [4] allows specifying continuous behaviors in AADL, and its developers provide a theorem proving support for proving properties in Hoare Logic combined with Duration Calculus [3]. Controller behaviors are defined in Hybrid CSP. Only a "synchronous" subset without message delays is considered, and clock skews, etc., are not taken into account. In contrast: we analyze models specified using AADL's expressive Behavior Annex, we provide automatic model checking analysis instead of interactive theorem proving, and we consider (virtually synchronous) CPSs—with clock skews, network delays, etc.

In [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 [43], and an AADL sublanguage, called AADL+, where continuous behaviors can be defined using stochastic differential equations is given in [37]. Both approaches come with some kind of operational semantics and simulation, but with no formal analysis support.

PALS and AADL. Synchronous AADL [10, 14] and its multi-rate extension [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 [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 in the well-known modeling standard AADL. Our tool provides randomized simulation and symbolic reachability analysis (using Maude combined with SMT), and is fully integrated into the OSATE modeling environment for AADL. We have developed and implemented a number of optimization techniques to improve the performance of the analysis. We demonstrate the efficiency of our tool on a number of distributed hybrid systems, including collaborating drones, and show that in most cases our tool outperforms state-of-the-art hybrid systems reachability analysis tools.

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

1. Supplementary material: HybridSynchAADL technical report, semantics, benchmarks, and the tool, https://hybridsynchaadl.github.io/tacas21/
2. Abrial, J., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, LNCS, vol. 1165. Springer (1996)
3. Ahmad, E., Dong, Y., Wang, S., Zhan, N., Zou, L.: Adding formal meanings to AADL with Hybrid Annex. In: Proc. FACS. LNCS, vol. 8997. Springer (2015)
4. Ahmad, E., Larson, B.R., Barrett, S.C., Zhan, N., Dong, Y.: Hybrid Annex: an AADL extension for continuous behavior and cyber-physical interaction modeling. In: Proc. ACM SIGAda annual conference on High integrity language technology (HILT'14). ACM (2014)
5. Al-Nayeem, A., Sun, M., Qiu, X., Sha, L., Miller, S.P., Cofer, D.D.: A formal architecture pattern for real-time distributed systems. In: Proc. RTSS. IEEE (2009)
6. Kong, S., Gao, S., Chen, W., Clarke, E.M.: dReach: δ-reachability analysis for hybrid systems. In: Proc. TACAS. Lecture Notes in Computer Science, vol. 7898. Springer (2015)

7. Bae, K., Gao, S.: Modular SMT-based analysis of nonlinear hybrid systems. In: Proc. FMCAD. pp. 180–187. IEEE (2017)
8. Bae, K., Meseguer, J., Ölveczky, P.C.: Formal patterns for multirate distributed real-time systems. Science of Computer Programming **91**, 3–44 (2014)
9. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous aadl and its formal analysis in real-time maude. In: International Conference on Formal Engineering Methods. pp. 651–667. Springer (2011)
10. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous AADL and its formal analysis in Real-Time Maude. In: Proc. ICFEM'11. LNCS, vol. 6991. Springer (2011)
11. Bae, K., Ölveczky, P.C., Kong, S., Gao, S., Clarke, E.M.: SMT-based analysis of virtually synchronous distributed hybrid systems. In: Proc. HSCC. ACM (2016)
12. Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of multirate synchronous aadl. In: Proc. FM. Lecture Notes in Computer Science, vol. 8442, pp. 94–109. Springer (2014)
13. Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of Multirate Synchronous AADL. In: Proc. FM'14. LNCS, vol. 8442. Springer (2014)
14. Bae, K., Ölveczky, P.C., Meseguer, J., Al-Nayeem, A.: The SynchAADL2Maude tool. In: Proc. FASE'12. LNCS, vol. 7212. Springer (2012)
15. Bae, K., Rocha, C.: Symbolic state space reduction with guarded terms for rewriting modulo SMT. Science of Computer Programming **178**, 20–42 (2019)
16. Bak, S., Bogomolov, S., Johnson, T.T.: Hyst: a source transformation and translation tool for hybrid automaton models. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control. pp. 128–133 (2015)
17. Bak, S., Duggirala, P.S.: Hylaa: A tool for computing simulation-equivalent reachability for linear systems. In: Proc. HSCC. pp. 173–178 (2017)
18. Bak, S., Tran, H.D., Johnson, T.T.: Numerical verification of affine systems with up to a billion dimensions. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. pp. 23–32 (2019)
19. Bao, Y., Chen, M., Zhu, Q., Wei, T., Mallet, F., Zhou, T.: Quantitative performance evaluation of uncertainty-aware Hybrid AADL designs using statistical model checking. IEEE Transactions on CAD of Integrated Circuits and Systems **36**(12), 1989–2002 (2017)
20. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. pp. 171–177. Springer (2011)
21. Baudart, G., Bourke, T., Pouzet, M.: Soundness of the quasi-synchronous abstraction. In: Proc. FMCAD. pp. 9–16. IEEE (2016)
22. Caspi, P., Mazuet, C., Paligot, N.R.: About the design of distributed control systems: The quasi-synchronous approach. In: International Conference on Computer Safety, Reliability, and Security. Springer (2001)
23. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Proc. CAV. pp. 258–263. Springer (2013)
24. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: HyComp: An SMT-based model checker for hybrid systems. In: Proc. TACAS. LNCS, vol. 9035. Springer (2015)
25. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude – A High-Performance Logical Framework, Lecture Notes in Computer Science, vol. 4350. Springer (2007)
26. Desai, A., Seshia, S.A., Qadeer, S., Broman, D., Eidson, J.C.: Approximate synchrony: An abstraction for distributed almost-synchronous systems. In: Proc. CAV'15. LNCS, vol. 9207. Springer (2015)

27. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV. LNCS, vol. 8559, pp. 737–744. Springer (July 2014)
28. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language. Addison-Wesley (2012)
29. França, R., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL Behaviour Annex - experiments and roadmap. In: Proc. ICECCS'07. IEEE (2007)
30. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Proc. CAV. LNCS, vol. 6806. Springer (2011)
31. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: Proc. CADE. Lecture Notes in Computer Science, vol. 7898. Springer (2013)
32. Girault, A., Ménier, C.: Automatic production of globally asynchronous locally synchronous systems. In: International Workshop on Embedded Software. pp. 266–281. Springer (2002)
33. Halbwachs, N., Mandel, L.: Simulation and verification of asynchronous systems by means of a synchronous model. In: Sixth International Conference on Application of Concurrency to System Design (ACSD'06). pp. 3–14. IEEE (2006)
34. Henzinger, T.: The theory of hybrid automata. In: Verification of Digital and Hybrid Systems, NATO ASI Series, vol. 170, pp. 265–292. Springer (2000)
35. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. Acm Sigplan Notices **47**(6), 193–204 (2012)
36. Larrieu, R., Shankar, N.: A framework for high-assurance quasi-synchronous systems. In: 2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE). pp. 72–83. IEEE (2014)
37. Liu, J., Li, T., Ding, Z., Qian, Y., Sun, H., He, J.: AADL+: a simulation-based methodology for cyber-physical systems. Frontiers Comput. Sci. **13**(3), 516–538 (2019)
38. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)
39. Meseguer, J., Ölveczky, P.C.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. Theoretical Computer Science **451**, 1–37 (2012)
40. Miller, S., Cofer, D., Sha, L., Meseguer, J., Al-Nayeem, A.: Implementing logical synchrony in integrated modular avionics. In: Proc. IEEE/AIAA 28th Digital Avionics Systems Conference. IEEE (2009)
41. Ölveczky, P.C., Boronat, A., Meseguer, J.: Formal semantics and analysis of behavioral aadl models in real-time maude. In: Formal Techniques for Distributed Systems, pp. 47–62. Springer (2010)
42. Potop-Butucaru, D., Caillaud, B.: Correct-by-construction asynchronous implementation of modular synchronous specifications. Fundamenta Informaticae **78**(1), 131–159 (2007)
43. Qian, Y., Liu, J., Chen, X.: Hybrid AADL: a sublanguage extension to AADL. In: Proc. Internetware'13. ACM (2013)
44. Raisch, J., Klein, E., Meder, C., Itigin, A., O'Young, S.: Approximating automata and discrete control for continuous systems — two examples from process control. In: Hybrid systems V. pp. 279–303. Springer (1999)
45. Ren, W., Beard, R.W.: Distributed consensus in multi-vehicle cooperative control. Springer (2008)

46. Rocha, C., Meseguer, J., Muñoz, C.: Rewriting modulo SMT and open system analysis. Journal of Logical and Algebraic Methods in Programming **86**(1), 269–297 (2017)
47. Rushby, J.: Systematic formal verification for fault-tolerant time-triggered algorithms. IEEE Transactions on Software Engineering **25**(5), 651–660 (1999)
48. Tripakis, S., Pinello, C., Benveniste, A., Sangiovanni-Vincent, A., Caspi, P., Di Natale, M.: Implementing synchronous models on loosely time triggered architectures. IEEE Transactions on Computers **57**(10), 1300–1314 (2008)

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

```
--- 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;
    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;

      Timing => Delayed applies to
        d12x,  d23x,  d34x, d41x;

      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;


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

      Timing => Delayed applies to
          d12y,  d23y,  d34y, d41y;
  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;
      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 RevuDroneControlProc.impl;


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 RevuDroneControlProc.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 := 0
          else        velX := -2
          end if
        };
        output -[ ]-> init {
          outX := currX
        };
**};
end RevuDroneControlThread.impl;


--- RevuDrone thread with complex control logic has 5 transtions 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;
```

```
            };
        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 extends Environment.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) * dotx * t) + y(0);";
end Environment2D.impl;
end Environment;
```

A.2 Drone Formation with Single-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
    --- Because drone formation extends drone rendezvous,
        it needs to only refine four drone subcomponents.
    --- Unlike drone rendezvous, drone formation has a reference drone
    --- The other drones move along with the reference drone.
    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
    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
--- 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;
```

```
end FormDroneControl;
system implementation 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;


  system RefDrone2DControl extends RefDroneControl
    features
      outY: 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
      C4: port refDrone.velY -> env.velY;
      C5: port env.currY -> refDrone.currY;
      C6: port refDrone.outY -> outY;
  end RefDrone2DControl.formation;


  system RefDrone
    features
      velX: out data port Base_Types::Float;
```

```
    outX : out data port Base_Types::Float;
    currX: in data port Base_Types::Float;
end refDrone;
system implementation RefDrone.formation
  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 currVX -> droneProc.currVX;
    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;
  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
        };
```

```
**};
end FormDroneControlThread.impl;


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
```

```
        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 -[ on dispatch ]-> exec;
            exec -[ nx = 0 ]-> output {
                nx := 1;
              };
              exec -[ nx = 1 ]-> output {
                nx := 2
              };
              exec -[ nx = 2 ]-> 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
                  }
```

```
                    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
      };
  **};
end RightWaterTankThread.simple;

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 RightWaterTank;


--- 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;

 system FourRoomTherm
    properties
      Hybrid_SynchAADL::Synchronous => true;
 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_in1;
      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: port env4.temp -> therm4.curr_temp;

  properties
    Period => 100 ms;

    Data_Model::Initial_Value => ("param") applies to
     env1.x, env2.x, env3.x, env4.x;

    Timing => Delayed applies to
     C14, C24, C25, C34, C35, C44;

    Hybrid_SynchAADL::Max_Clock_Deviation => 5ms;

    Hybrid_SynchAADL::Sampling_Time => 20ms .. 30ms applies to
     therm1, therm3, therm2, therm4;
    Hybrid_SynchAADL::Response_Time => 60ms .. 70ms applies to
     therm1, therm3, therm2, therm4;

end FourRoomTherm.simple;

system implementation FourRoomTherm.complex extends FourRoomTherm.simple
    subcomponents
      therm1: refined to system Thermostat::Thermostat.complex;
      therm2: refined to system Thermostat2side::Thermostat2side.complex;
      therm3: refined to system Thermostat2side::Thermostat2side.complex;
      therm4: refined to system Thermostat::Thermostat.complex;
end FourRoomTherm.complex;
end FourRoomTherm;


--- Thermostat controller component
package Thermostat
public
 with Base_Types;
 with Data_Model;

 system Thermostat
  features
   on_control: out event port;
   off_control: out event port;
   power : out data port Base_Types::Float
                   {Data_Model::Initial_Value => ("0");};
   t_out : out data port Base_Types::Float
                   {Data_Model::Initial_Value => ("0");};
   t_in: in data port Base_Types::Float;
   curr_temp : in data port Base_Types::Float;
 end Thermostat;

 system implementation Thermostat.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_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
    };

    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_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
                        {Data_Model::Initial_Value => ("0");};
  t_out: out data port Base_Types::Float
                        {Data_Model::Initial_Value => ("0");};
  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;
```

```
                        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
          };
          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 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 labeld 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.
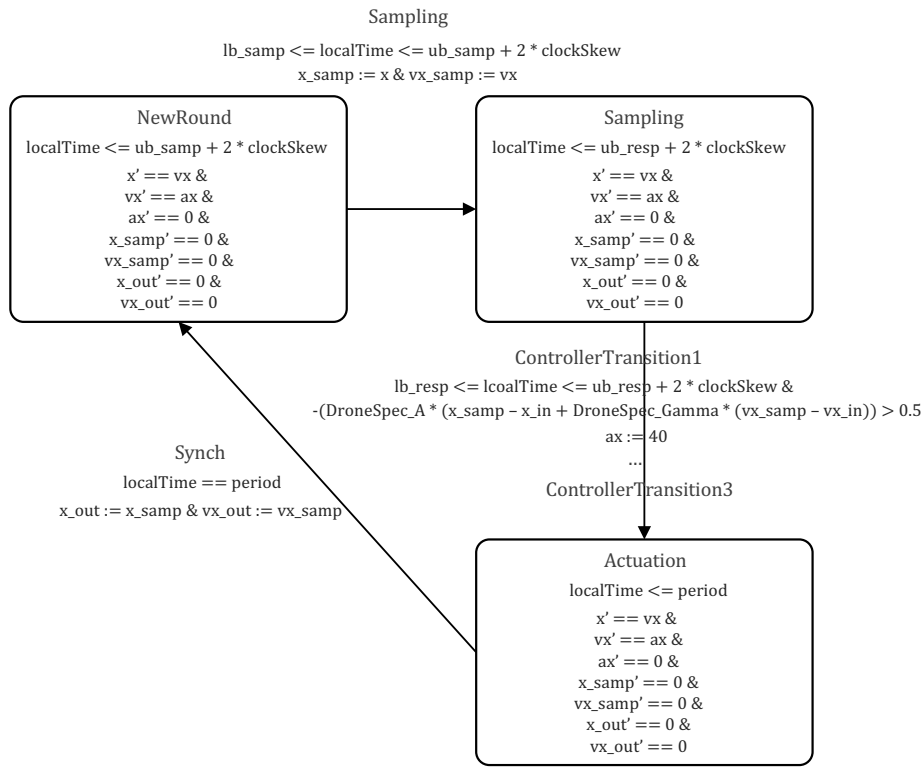
Fig. 26. Hybtid automaton component for drone rendezvous with simple control logic and double-integrator dynamics

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

Sampling

lb_samp <= localTime <= ub_samp + 2 * clockSkew

x_samp := x & vx_samp := vx &

y_samp := y & vy_samp := vy

NewRound

localTime <= ub_samp + 2 * clockSkew

x' == vx & y' == vy &

vx' == ax & vy' == ay &

ax' == 0 & ay' == 0 &

x_samp' == 0 & y_samp' == 0 &

vx_samp' == 0 & vy_samp' == 0 &

x_out' == 0 & y_out' == 0 &

vx_out' == 0 & vy_out'

Sampling

localTime <= ub_resp + 2 * clockSkew

x' == vx & y' == vy &

vx' == ax & vy' == ay &

ax' == 0 & ay' == 0 &

x_samp' == 0 & y_samp' == 0 &

vx_samp' == 0 & vy_samp' == 0 &

x_out' == 0 & y_out' == 0 &

vx_out' == 0 & vy_out'

ControllerTransition1

lb_resp <= lcoalTime <= ub_resp + 2 * clockSkew &

-(DroneSpec_A * (x_samp – x_in + DroneSpec_Gamma * (vx_samp – vx_in)) > 0.5 &

-(DroneSpec_A * (y_samp – y_in + DroneSpec_Gamma * (vy_samp – vy_in)) > 0.5

ax := 40 & ay := 40

...

ControllerTransition25

Synch

localTime == period

x_out := x_samp & vx_out := vx_samp &

y_out := y_samp & vy_out := vy_samp

Actuation

localTime <= period

x' == vx & y' == vy &

vx' == ax & vy' == ay &

ax' == 0 & ay' == 0 &

x_samp' == 0 & y_samp' == 0 &

vx_samp' == 0 & vy_samp' == 0 &

x_out' == 0 & y_out' == 0 &

vx_out' == 0 & vy_out'

Fig. 27. Hybtid automaton component for drone rendezvous with complex control logic and double-integrator dynamics

7 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.

Sampling

lb_samp <= localTime <= ub_samp + 2 * clockSkew

x_samp := x & vx_samp := vx

NewRound

localTime <= ub_samp + 2 * clockSkew

x' == vx &
vx' == ax &
ax' == 0 &
x_samp' == 0 &
vx_samp' == 0 &
x_out' == 0 &
vx_out' == 0 &
refVX_prev_in' == 0

Sampling

localTime <= ub_resp + 2 * clockSkew

x' == vx &
vx' == ax &
ax' == 0 &
x_samp' == 0 &
vx_samp' == 0 &
x_out' == 0 &
vx_out' == 0 &
refVX_prev_in' == 0

ControllerTransition1

lb_resp <= lcoalTime <= ub_resp + 2 * clockSkew &

-((refVX_in - refVX_prev_in) – (DroneSpec_Alpha * (x_samp – offsetX – refX_in

+ DroneSpec_Gamma * (vx_samp – refVX_in)) – (DroneSpec_A * (x_samp – offsetX – x_in

+ DroneSpec_Gamma * (vx_samp – vx_in)))) > 0.5

ax := 40

...

ControllerTransition3

Synch

localTime == period

x_out := x_samp & vx_out := vx_samp

Actuation

localTime <= period

x' == vx &
vx' == ax &
ax' == 0 &
x_samp' == 0 &
vx_samp' == 0 &
x_out' == 0 &
vx_out' == 0 &
refVX_prev_in' == 0

Fig. 28. Hybtid automaton component for drone formation with simple control logic and double-integrator dynamics

Sampling

lb_samp <= localTime <= ub_samp + 2 * clockSkew
x_samp := x & vx_samp := vx &
y_samp := y & vy_samp := vy

NewRound

localTime <= ub_samp + 2 * clockSkew
x' == vx & y' == vy &
vx' == ax & vy' == ay &
ax' == 0 & ay' == 0 &
x_samp' == 0 & y_samp' == 0 &
vx_samp' == 0 & vy_samp' == 0 &
x_out' == 0 & y_out' == 0 &
vx_out' == 0 & vy_out' == 0 &
refVX_prev_in' == 0 & refVY_prev_in' == 0

Sampling

localTime <= ub_resp + 2 * clockSkew
x' == vx & y' == vy &
vx' == ax & vy' == ay &
ax' == 0 & ay' == 0 &
x_samp' == 0 & y_samp' == 0 &
vx_samp' == 0 & vy_samp' == 0 &
x_out' == 0 & y_out' == 0 &
vx_out' == 0 & vy_out' == 0 &
refVX_prev_in' == 0 & refVY_prev_in' == 0

ControllerTransition1

lb_resp <= lcoalTime <= ub_resp + 2 * clockSkew &
-((refVX_in - refVX_prev_in) – (DroneSpec_Alpha * (x_samp – offsetX – refX_in
+ DroneSpec_Gamma * (vx_samp – refVX_in)) – (DroneSpec_A * (x_samp – offsetX – x_in
+ DroneSpec_Gamma * (vx_samp – vx_in)))) > 0.5 &

-((refVY_in - refVY_prev_in) – (DroneSpec_Alpha * (y_samp – offsetY – refY_in
+ DroneSpec_Gamma * (vy_samp – refVY_in)) – (DroneSpec_A * (y_samp – offsetY – y_in
+ DroneSpec_Gamma * (vy_samp – vy_in)))) > 0.5

ax := 40 & ay:=40
...
ControllerTransition25

Synch

localTime == period

x_out := x_samp & vx_out := vx_samp &
y_out := y_samp & vy_out := vy_samp

Actuation

localTime <= period
x' == vx & y' == vy &
vx' == ax & vy' == ay &
ax' == 0 & ay' == 0 &
x_samp' == 0 & y_samp' == 0 &
vx_samp' == 0 & vy_samp' == 0 &
x_out' == 0 & y_out' == 0 &
vx_out' == 0 & vy_out' == 0 &
refVX_prev_in' == 0 & refVY_prev_in' == 0

Fig. 29. Hybtid automaton component for drone formation with complex control logic and double-integrator dynamics

Sampling

lb_samp <= localTime <= ub_samp + 2 * clockSkew

x_samp := x

NewRound

localTime <= ub_samp + 2 * clockSkew

x' == vx &
vx' == 0 &
x_samp' == 0 &
x_out' == 0 &

Sampling

localTime <= ub_resp + 2 * clockSkew

x' == vx &
vx' == 0 &
x_samp' == 0 &
x_out' == 0 &

ControllerTransition1

lb_resp <= lcoalTime <= ub_resp + 2 * clockSkew &

-(DroneSpec_A * (x_samp – x_in)) > 1

vx := 2

...

ControllerTransition3

Synch

localTime == period

x_out := x_samp

Actuation

localTime <= period

x' == vx &
vx' == 0 &
x_samp' == 0 &
x_out' == 0 &

Fig. 30. Hybtid automaton component for drone rendezvous with simple control logic and single-integrator dynamics

Sampling

lb_samp <= localTime <= ub_samp + 2 * clockSkew

x_samp := x & y_samp := y

NewRound

localTime <= ub_samp + 2 * clockSkew

x' == vx & y' == vy &
vx' == 0 & vy' == 0 &
x_samp' == 0 & y_samp' == 0 &
x_out' == 0 & y_out' == 0

Sampling

localTime <= ub_resp + 2 * clockSkew

x' == vx & y' == vy &
vx' == 0 & vy' == 0 &
x_samp' == 0 & y_samp' == 0 &
x_out' == 0 & y_out' == 0

ControllerTransition1

lb_resp <= lcoalTime <= ub_resp + 2 * clockSkew &

-(DroneSpec_A * (x_samp – x_in)) > 1 &

-(DroneSpec_A * (y_samp – y_in)) > 1

vx := 2 & vy := 2

...

ControllerTransition25

Synch

localTime == period

x_out := x_samp &
y_out := y_samp

Actuation

localTime <= period

x' == vx & y' == vy &
vx' == 0 & vy' == 0 &
x_samp' == 0 & y_samp' == 0 &
x_out' == 0 & y_out' == 0

Fig. 31. Hybtid automaton component for drone rendezvous with complex control logic and single-integrator dynamics

Sampling

lb_samp <= localTime <= ub_samp + 2 * clockSkew
x_samp := x

NewRound	Sampling
localTime <= ub_samp + 2 * clockSkew	localTime <= ub_resp + 2 * clockSkew
x' == vx &	x' == vx &
vx' == 0 &	vx' == 0 &
x_samp' == 0 &	x_samp' == 0 &
x_out' == 0 &	x_out' == 0 &
refX_prev_in' == 0	refX_prev_in' == 0

ControllerTransition1

lb_resp <= lcoalTime <= ub_resp + 2 * clockSkew &

(refX_in – refX_prev_in) – (DroneSpec_Alpha * (x_samp – offsetX – refX_in))
– (DroneSpec_A * (x_samp – offsetX – x_in)) > 1

vx := 2
...

ControllerTransition3

Synch

localTime == period
x_out := x_samp

Actuation

localTime <= period

x' == vx &
vx' == 0 &
x_samp' == 0 &
x_out' == 0 &
refX_prev_in' == 0

Fig. 32. Hybtid automaton component for drone formation with simple control logic and single-integrator dynamics

Fig. 33. Hybtid automaton component for drone formation with complex control logic and single-integrator dynamics

Sampling

lb_samp <= localTime <= ub_samp + 2 * clockSkew

x_samp := x

NewRound

localTime <= ub_samp + 2 * clockSkew

x' == 10 * power &
power' == 0 &
x_samp' == 0 &
x_out' == 0 &

Sampling

localTime <= ub_resp + 2 * clockSkew

x' == 10 * power &
power' == 0 &
x_samp' == 0 &
x_out' == 0 &

ControllerTransition1

lb_resp <= lcoalTime <= ub_resp + 2 * clockSkew &

- (x_samp – x_in)) > 10

power := 5

...

ControllerTransition3

Synch

localTime == period

x_out := x_samp

Actuation

localTime <= period

x' == 10 & power &
power' == 0 &
x_samp' == 0 &
x_out' == 0 &
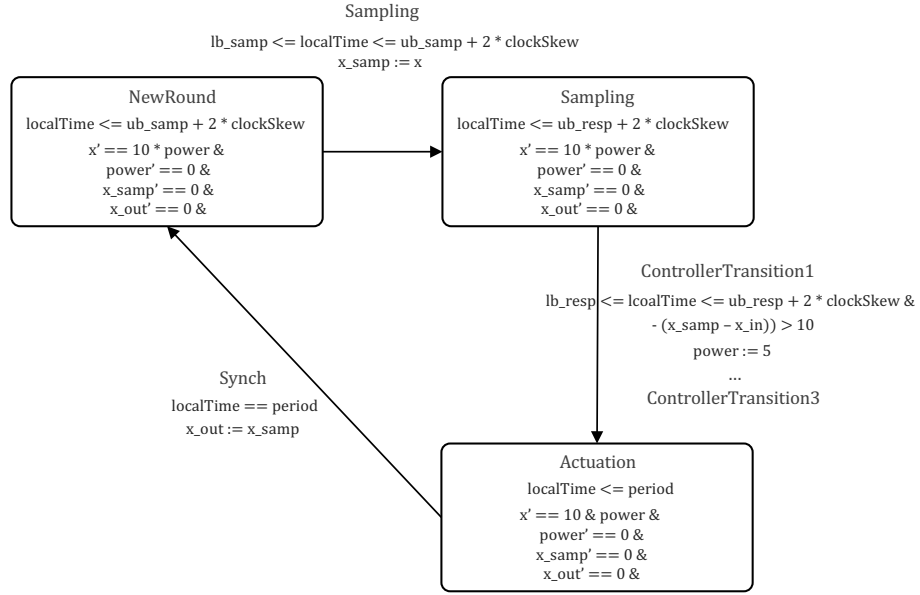
Fig. 34. Hybtid automaton component for networked thermostat with simple control logic

Sampling

lb_samp <= localTime <= ub_samp + 2 * clockSkew

x_samp := x

NewRound

localTime <= ub_samp + 2 * clockSkew

x' == 10 * power &
power' == 0 &
x_samp' == 0 &
x_out' == 0 &

Sampling

localTime <= ub_resp + 2 * clockSkew

x' == 10 * power &
power' == 0 &
x_samp' == 0 &
x_out' == 0 &

ControllerTransition1

lb_resp <= lcoalTime <= ub_resp + 2 * clockSkew &

- (x_samp – x_in)) > 10

power := 5

...

ControllerTransition6

Synch

localTime == period

x_out := x_samp

Actuation

localTime <= period

x' == 10 & power &
power' == 0 &
x_samp' == 0 &
x_out' == 0 &
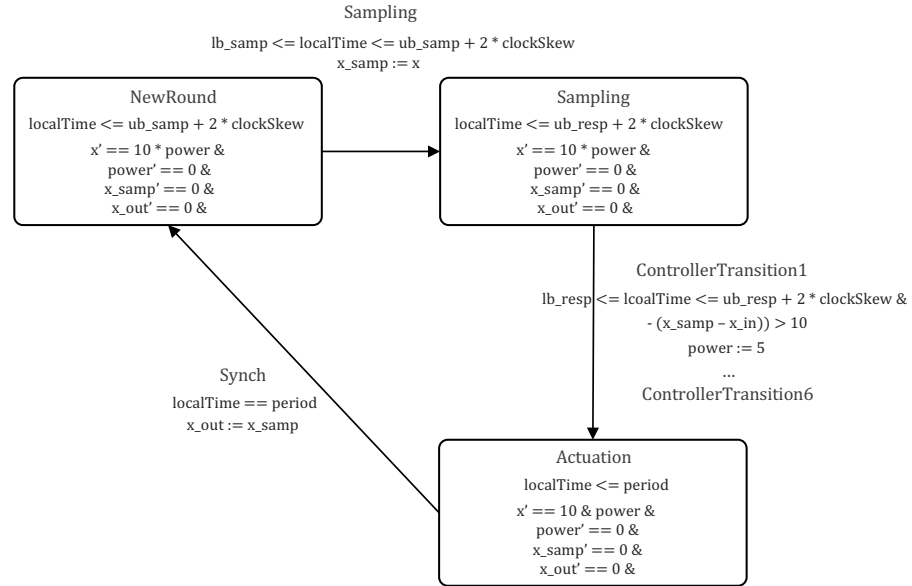
Fig. 35. Hybtid automaton component for networked thermostat with complex control logic

Sampling

lb_samp <= localTime <= ub_samp + 2 * clockSkew

water_samp := decreased &
water = water + water_in

```
NewRound                          Sampling
localTime <= ub_samp + 2 * clockSkew    localTime <= ub_resp + 2 * clockSkew

water' == -50 &                   water' == -50 &
Water_samp' == 0 &                Water_samp' == 0 &
decreased' == 50 &                decreased' == 50 &
water_out' == 0                   water_out' == 0
```

ControllerTransition1

lb_resp <= lcoalTime <= ub_resp + 2 * clockSkew &

water <= 33

water := water + 0.5

...

ControllerTransition3

Synch

localTime == period
x_out := x_samp

```
Actuation
localTime <= period

water' == -50 &
Water_samp' == 0 &
decreased' == 50 &
water_out' == 0
```

Fig. 36. Hybtid 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                          Sampling
localTime <= ub_samp + 2 * clockSkew    localTime <= ub_resp + 2 * clockSkew

water' == -50 &                   water' == -50 &
Water_samp' == 0 &                Water_samp' == 0 &
decreased' == 50 &                decreased' == 50 &
water_out' == 0                   water_out' == 0
```

ControllerTransition1

lb_resp <= lcoalTime <= ub_resp + 2 * clockSkew &

water <= 33

water := water + 0.5

...

ControllerTransition6

Synch

localTime == period
x_out := x_samp

```
Actuation
localTime <= period

water' == -50 &
Water_samp' == 0 &
decreased' == 50 &
water_out' == 0
```

Fig. 37. Hybtid automaton component for networked waterTank with complex control logic

Model	N	Tool	B = 1 Time	B = 2 Time	B = 3 Time	B = 4 Time	B = 5 Time	Model	N	Tool	B = 1 Time	B = 2 Time	B = 3 Time	B = 4 Time	B = 5 Time
Thermostat	2	HSADDL	0.30	0.48	0.82	1.46	2.71	WaterTank	2	HSADDL	0.30	0.44	0.66	0.96	1.34
		HyComp	0.05	0.17	0.28	0.83	1.62			HyComp	0.03	0.06	0.14	0.27	0.53
		SpaceEx	0.05	0.29	0.75	1.19	2.33			SpaceEx	0.05	0.14	0.34	0.60	0.81
		dReach	1.30	26.00	341.58	-	-			dReach	1.66	53.07	1080.83	-	-
		Flow*	1.91	10.71	48.35	336.85	3196.42			Flow*	2.82	45.26	1070.07	-	-
	3	HSADDL	0.36	0.66	1.29	2.51	4.72		3	HSADDL	0.39	0.63	1.02	1.62	2.52
		HyComp	0.09	0.40	1.24	3.97	8.52			HyComp	0.08	0.36	0.91	1.97	5.99
		SpaceEx	0.96	103.68	696.38	-	-			SpaceEx	0.76	20.75	97.50	335.50	877.17
		dReach	57.52	-	-	-	-			dReach	52.82	-	-	-	-
		Flow*	32.08	1240.68	-	-	-			Flow*	96.36	-	-	-	-
	4	HSADDL	0.41	0.88	1.90	3.76	7.79		4	HSADDL	0.45	0.80	1.48	2.55	4.16
		HyComp	0.20	0.84	5.69	11.23	37.87			HyComp	0.20	1.01	2.94	8.80	37.12
		SpaceEx	34.47	-	-	-	-			SpaceEx	27.96	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	977.71	-	-	-	-			Flow*	-	-	-	-	-
Drone Rendezvous (Single)	2	HSADDL	0.25	0.39	0.65	1.12	1.99	Drone Formation (Single)	2	HSADDL	0.34	0.53	0.90	1.61	3.01
		HyComp	0.04	0.09	0.23	0.52	0.84			HyComp	0.08	0.30	1.42	5.37	13.28
		SpaceEx	0.02	0.53	4.98	6.60	7.96			SpaceEx	0.21	91.91	-	-	-
		dReach	1.97	56.26	1382.67	-	-			dReach	139.03	-	-	-	-
		Flow*	1.83	18.64	205.94	3552.78	-			Flow*	41.60	1464.69	-	-	-
	3	HSADDL	0.29	0.51	0.97	1.93	3.90		3	HSADDL	0.38	0.66	1.31	2.81	7.30
		HyComp	0.08	0.33	0.79	2.05	3.97			HyComp	0.17	0.89	6.18	15.93	41.31
		SpaceEx	0.21	34.33	2230.25	-	-			SpaceEx	2.77	-	-	-	-
		dReach	107.05	-	-	-	-			dReach	-	-	-	-	-
		Flow*	36.65	2725.51	-	-	-			Flow*	873.41	-	-	-	-
	4	HSADDL	0.33	0.65	1.37	2.90	5.84		4	HSADDL	0.43	0.83	1.77	3.82	7.90
		HyComp	0.15	0.68	2.00	7.12	17.23			HyComp	0.23	2.73	21.89	52.24	182.10
		SpaceEx	4.53	-	-	-	-			SpaceEx	114.79	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	1205.17	-	-	-	-			Flow*	-	-	-	-	-
Drone Rendezvous (Double)	2	HSADDL	0.30	0.46	0.79	3.68	-	Drone Formation (Double)	2	HSADDL	0.42	0.66	2.18	-	-
		SpaceEx	1.19	15.07	1147.62	-	-			SpaceEx	70.79	-	-	-	-
		dReach	4.20	128.23	2156.23	-	-			dReach	360.40	-	-	-	-
		Flow*	10.20	232.53	-	-	-			Flow*	116.81	-	-	-	-
	3	HSADDL	0.35	0.63	1.24	37.76	-		3	HSADDL	0.50	0.85	1.75	-	-
		SpaceEx	81.12	-	-	-	-			SpaceEx	-	-	-	-	-
		dReach	274.25	-	-	-	-			dReach	-	-	-	-	-
		Flow*	230.12	-	-	-	-			Flow*	3411.46	-	-	-	-
	4	HSADDL	0.41	0.79	1.88	6.87	-		4	HSADDL	0.57	1.09	2.50	-	-
		SpaceEx	-	-	-	-	-			SpaceEx	-	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	-	-	-	-	-			Flow*	-	-	-	-	-

Table 3. Comparison of HybridSynchAADL and the other tools for $Inv_\top$ at all bounds in using simplified control logic. The '-' represents a timeout

Left half — Thermostat, Drone Rendezvous (Single), Drone Rendezvous (Double)

Model	N	Tool	B = 1 Time	B = 2 Time	B = 3 Time	B = 4 Time	B = 5 Time
Thermostat	2	HSADDL	0.32	0.52	0.88	1.56	7.61
		HyComp	0.07	0.30	0.76	1.79	2.63
		SpaceEx	0.45	0.29	0.67	1.19	2.16
		dReach	-	-	-	-	-
		Flow*	2.05	12.08	15.50	8.41	
	3	HSADDL	0.38	0.70	1.41	2.71	15.27
		HyComp	0.22	1.55	5.35	12.71	15.50
		SpaceEx	0.96	100.19	708.21	-	-
		dReach	-	-	-	-	-
		Flow*	36.22	1287.38	-	1718.11	
	4	HSADDL	0.45	0.94	2.11	10.65	
		HyComp	0.60	5.67	31.62	43.06	
		SpaceEx	32.72	-	-	-	-
		dReach	-	-	-	-	-
		Flow*	1072.26	-	-	-	-
Drone Rendezvous (Single)	2	HSADDL	0.30	0.46	2.36		
		HyComp	0.16	5.89	8.86		
		SpaceEx	0.02	0.38	5.08		
		dReach	-	-	-	-	-
		Flow*	1.87	12.58	167.25		
	3	HSADDL	0.34	0.72	4.16		
		HyComp	0.18	2.06	11.52		
		SpaceEx	0.17	35.91	2676.59		
		dReach	-	-	-	-	-
		Flow*	62.52	380.44			
	4	HSADDL	0.46	1.32	5.94		
		HyComp	0.78	106.79	192.58		
		SpaceEx	4.48	-	-	-	-
		dReach	-	-	-	-	-
		Flow*	2187.61	-	838.04		
Drone Rendezvous (Double)	2	HSADDL	0.34	1.36			
		SpaceEx	1.16	15.21		-	-
		dReach	-	-	-	-	-
		Flow*	5.76	2.17			
	3	HSADDL	0.49	16.33			
		SpaceEx	51.12	-	-	-	-
		dReach	-	-	-	-	-
		Flow*	119.46	25.39			
	4	HSADDL	0.60	2.83			
		SpaceEx	-	-	-	-	-
		dReach	-	-	-	-	-
		Flow*	-	-	-	-	-

Right half — WaterTank, Drone Formation (Single), Drone Formation (Double)

Model	N	Tool	B = 1 Time	B = 2 Time	B = 3 Time	B = 4 Time	B = 5 Time
WaterTank	2	HSADDL	0.32	0.48	1.81		
		HyComp	0.04	0.08	0.12		
		SpaceEx	0.04	0.10	0.18		
		dReach	-	-	-	-	-
		Flow*	0.99				
	3	HSADDL	0.40	0.68	3.09		
		HyComp	0.17	0.84	1.86		
		SpaceEx	0.88	13.48	74.37		
		dReach	-	-	-	-	-
		Flow*	5.73				
	4	HSADDL	0.46	0.86	4.82		
		HyComp	0.41	1.95	4.19		
		SpaceEx	40.37	-	-	-	-
		dReach	-	-	-	-	-
		Flow*	28.93				
Drone Formation (Single)	2	HSADDL	0.35	0.62	4.22	15.54	
		HyComp	4.64	16.36	-	-	-
		SpaceEx	0.20	93.96	-	-	-
		dReach	-	-	-	-	-
		Flow*	43.84	1586.32	-	-	-
	3	HSADDL	0.45	2.54			
		HyComp	0.43	2.59			
		SpaceEx	2.51	-	-	-	-
		dReach	-	-	-	-	-
		Flow*	45.30				
	4	HSADDL	0.52	5.16			
		HyComp	2.64	20.30			
		SpaceEx	105.87	-	-	-	-
		dReach	-	-	-	-	-
		Flow*	-	291.32			
Drone Formation (Double)	2	HSADDL	0.45	2.18			
		SpaceEx	70.19	-	-	-	-
		dReach	-	-	-	-	-
		Flow*	136.90	14.02			
	3	HSADDL	0.63	2.69			
		SpaceEx	-	-	-	-	-
		dReach	-	-	-	-	-
		Flow*	1586.93				
	4	HSADDL	0.76	3.79			
		SpaceEx	-	-	-	-	-
		dReach	-	-	-	-	-
		Flow*	-	-	-	-	-

Table 4. Comparison of HybridSynchAADL and the other tools for $Inv_\perp$ at all bounds in using simplified control logic. The '-' means that a counterexample is not found before a timeout.

128

Model	N	Tool	B=1 Time	B=2 Time	B=3 Time	B=4 Time	B=5 Time	Model	N	Tool	B=1 Time	B=2 Time	B=3 Time	B=4 Time	B=5 Time
Thermostat	2	HSADDL	0.31	0.56	1.02	1.88	3.40	WaterTank	2	HSADDL	0.32	0.51	0.77	1.15	1.67
		HyComp	0.05	0.13	0.40	0.91	3.97			HyComp	0.05	0.15	0.47	2.01	4.34
		SpaceEx	0.03	0.30	1.76	2.81	4.72			SpaceEx	0.05	0.17	0.33	0.65	1.36
		dReach	12.62	1341.60	-	-	-			dReach	980.80	-	-	-	-
		Flow*	1.56	9.07	41.97	318.79	-			Flow*	2.99	96.50	-	-	-
	3	HSADDL	0.37	0.77	1.56	3.08	6.00		3	HSADDL	0.43	0.74	1.26	2.10	3.28
		HyComp	0.13	0.45	1.61	4.27	12.09			HyComp	0.10	0.53	1.77	5.55	14.95
		SpaceEx	0.23	98.91	-	-	-			SpaceEx	3.88	331.97	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	332.68	1246.84	-	-	-			Flow*	180.26	-	-	-	-
	4	HSADDL	0.45	1.02	2.30	4.88	10.84		4	HSADDL	0.51	1.01	1.89	3.41	5.62
		HyComp	0.26	1.15	5.99	32.04	51.68			HyComp	0.21	1.15	5.45	20.26	98.76
		SpaceEx	2.49	-	-	-	-			SpaceEx	43.78	3181.84	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	-	-	-	-	-			Flow*	-	-	-	-	-
Drone Rendezvous (Single)	2	HSADDL	0.64	1.58	3.16	5.44	10.62	Drone Formation (Single)	2	HSADDL	0.90	2.28	4.70	8.89	16.24
		HyComp	0.09	0.20	1.22	1.74	6.24			HyComp	0.18	1.26	7.44	18.59	1586.12
		SpaceEx	0.47	1551.43	-	-	-			SpaceEx	27.74	-	-	-	-
		dReach	1982.70	-	-	-	-			dReach	930.90	-	-	-	-
		Flow*	12.29	1267.75	-	-	-			Flow*	749.97	-	-	-	-
	3	HSADDL	0.90	2.50	5.39	9.86	18.86		3	HSADDL	1.23	3.42	7.90	16.31	60.41
		HyComp	0.19	1.03	2.02	18.86	108.81			HyComp	0.33	3.99	101.91	652.21	-
		SpaceEx	16.59	-	-	-	-			SpaceEx	501.35	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	422.58	-	-	-	-			Flow*	-	-	-	-	-
	4	HSADDL	1.16	3.48	8.10	15.95	32.23		4	HSADDL	1.54	4.74	11.71	23.81	62.24
		HyComp	0.32	2.23	21.01	120.69	150.23			HyComp	0.59	5.94	91.42	844.42	-
		SpaceEx	470.15	-	-	-	-			SpaceEx	-	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	-	-	-	-	-			Flow*	-	-	-	-	-
Drone Rendezvous (Double)	2	HSADDL	0.93	2.40	7.74	-	-	Drone Formation (Double)	2	HSADDL	1.36	3.61	8.13	-	-
		SpaceEx	59.81	-	-	-	-			SpaceEx	-	-	-	-	-
		dReach	3156.20	-	-	-	-			dReach	2124.20	-	-	-	-
		Flow*	-	-	-	-	-			Flow*	-	-	-	-	-
	3	HSADDL	1.27	3.72	12.43	-	-		3	HSADDL	1.83	5.52	-	-	-
		SpaceEx	-	-	-	-	-			SpaceEx	-	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	-	-	-	-	-			Flow*	-	-	-	-	-
	4	HSADDL	1.65	5.31	-	-	-		4	HSADDL	2.33	7.70	-	-	-
		SpaceEx	-	-	-	-	-			SpaceEx	-	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	-	-	-	-	-			Flow*	-	-	-	-	-

Table 5. Comparison of HybridSynchAADL and the other tools for $Inv_\top$ at all bounds in using complex control logic. The '-' represents a timeout

Model	N	Tool	B = 1 Time	B = 2 Time	B = 3 Time	B = 4 Time	B = 5 Time	Model	N	Tool	B = 1 Time	B = 2 Time	B = 3 Time	B = 4 Time	B = 5 Time
Thermostat	2	HSADDL	0.32	0.57	1.11	1.80	9.57	WaterTank	2	HSADDL	0.34	0.55	0.88	1.36	5.81
		HyComp	0.07	0.31	0.66	1.97	3.05			HyComp	0.06	0.24	0.64	2.25	3.59
		SpaceEx	0.03	0.31	1.71	2.73	4.30			SpaceEx	0.05	0.15	0.38	0.81	1.83
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	1.62	9.78	45.53					Flow*	2.74	13.86			
	3	HSADDL	0.40	0.80	1.64	2.73	19.04		3	HSADDL	0.44	0.78	1.39	2.37	11.96
		HyComp	0.31	1.46	4.57	19.34	28.35			HyComp	0.20	1.08	3.28	39.76	61.80
		SpaceEx	0.23	95.67	-	-	-			SpaceEx	0.99	21.95	1218.34	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	34.06	-	-	-	-			Flow*	259.13	51.76			
	4	HSADDL	0.48	1.09	2.78	15.76			4	HSADDL	0.52	1.08	2.09	11.76	
		HyComp	0.81	5.45	22.35	30.20				HyComp	0.53	3.69	20.64	1261.85	
		SpaceEx	2.65	-	-	-	-			SpaceEx	645.10	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	1016.63	-	-	-	-			Flow*	-	13.13			
Drone Rendezvous (Single)	2	HSADDL	0.68	1.64	3.72	84.84		Drone Formation (Single)	2	HSADDL	0.94	2.49	33.22		
		HyComp	1.33	15.36	295.24	408.83				HyComp	1.61	991.75	648.03		
		SpaceEx	0.46	12.84	-	-	-			SpaceEx	25.05	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	12.71	195.14	-	-	-			Flow*	886.24	-	-	-	-
	3	HSADDL	0.97	2.92	17.56				3	HSADDL	1.29	10.52			
		HyComp	9.37	310.20	357.23					HyComp	4.50	32.27			
		SpaceEx	12.06	-	-	-	-			SpaceEx	380.90	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	696.93	96.11	108.53					Flow*	1034.18	-	-	-	-
	4	HSADDL	1.30	5.56	17.02	66.71			4	HSADDL	1.78	6.15	88.46		
		HyComp	198.82	-	-	-	-			HyComp	26.14	-	-	-	-
		SpaceEx	694.82	-	-	-	-			SpaceEx	-	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	-	-	-	-	-			Flow*	-	-	-	-	-
Drone Rendezvous (Double)	2	HSADDL	0.96	6.65				Drone Formation (Double)	2	HSADDL	1.41	15.44			
		SpaceEx	22.62	881.64						SpaceEx	-	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	-	-	-	-	-			Flow*	-	-	-	-	-
	3	HSADDL	1.46	11.36					3	HSADDL	2.02	20.44			
		SpaceEx	-	-	-	-	-			SpaceEx	-	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	-	-	-	-	-			Flow*	-	-	-	-	-
	4	HSADDL	1.90	16.71					4	HSADDL	2.85	26.49			
		SpaceEx	-	-	-	-	-			SpaceEx	-	-	-	-	-
		dReach	-	-	-	-	-			dReach	-	-	-	-	-
		Flow*	-	-	-	-	-			Flow*	-	-	-	-	-

Table 6. Comparison of HybridSynchAADL and the other tools for $Inv_\perp$ at all bounds in using complex control logici. The '-' means that a counterexample is not found before a timeout.

130

Model	N	Method	B=1 Time AVG	STDEV	B=2 Time AVG	STDEV	B=3 Time AVG	STDEV	B=4 Time AVG	STDEV	B=5 Time AVG	STDEV
Thermostat	2	random	-	-	-	-	-	-	-	-	-	-
		symbolic	0.32	0.01	0.57	0.01	1.08	0.02	1.84	0.04	9.76	0.09
		portfolio	0.32	0.01	0.57	0.01	1.08	0.02	1.84	0.04	9.76	0.09
	3	random	-	-	-	-	-	-	-	-	-	-
		symbolic	0.39	0.00	0.82	0.01	1.67	0.03	3.68	0.07	20.00	0.44
		portfolio	0.39	0.00	0.82	0.01	1.67	0.03	3.68	0.07	20.00	0.44
	4	random	-	-	-	-	-	-	12.42	14.63	26.71	29.10
		symbolic	0.47	0.01	1.11	0.03	2.67	0.01	16.06	0.37		
		portfolio	0.47	0.01	1.11	0.03	2.67	0.01	12.42	14.63	16.01	0.34
WaterTank	2	random	-	-	-	-	-	-	-	-	-	-
		symbolic	0.33	0.00	0.53	0.01	0.85	0.02	1.32	0.03	6.00	0.06
		portfolio	0.33	0.00	0.53	0.01	0.85	0.02	1.32	0.03	6.00	0.06
	3	random	-	-	-	-	-	-	-	-	-	-
		symbolic	0.43	0.01	0.80	0.01	1.43	0.04	2.45	0.07	12.16	0.09
		portfolio	0.43	0.01	0.80	0.01	1.43	0.04	2.45	0.07	12.16	0.09
	4	random	-	-	-	-	-	-	-	-	46.40	49.23
		symbolic	0.54	0.01	1.09	0.01	2.14	0.05	12.15	0.12		
		portfolio	0.54	0.01	1.09	0.01	2.14	0.05	12.15	0.12	12.27	0.28
Drone rendezvous (single)	2	random	-	-	-	-	-	-	-	-	-	-
		symbolic	0.69	0.01	1.70	0.05	3.83	0.10	92.45	1.00		
		portfolio	0.69	0.01	1.70	0.05	3.83	0.10	92.45	1.00	93.06	2.36
	3	random	-	-	-	-	6.31	5.98	0.22	0.14	0.08	0.01
		symbolic	0.98	0.02	3.00	0.06	18.06	0.43				
		portfolio	0.98	0.02	3.00	0.06	6.31	5.98	0.22	0.14	0.08	0.01
	4	random	-	-	-	-	-	-	-	-	24.30	21.30
		symbolic	1.32	0.03	5.66	0.06	17.63	0.22	74.50	1.47		
		portfolio	1.32	0.03	5.66	0.06	17.63	0.22	74.50	1.47	24.30	21.30
Drone formation (single)	2	random	-	-	-	-	-	-	-	-	-	-
		symbolic	0.94	0.01	2.50	0.02	36.38	0.77				
		portfolio	0.94	0.01	2.50	0.02	36.38	0.77	35.98	0.81	36.26	0.77
	3	random	-	-	-	-	-	-	-	-	986.46	477.48
		symbolic	1.32	0.03	10.64	0.22						
		portfolio	1.32	0.03	10.64	0.22	10.57	0.09	10.54	0.04	10.71	0.20
	4	random	-	-	-	-	-	-	12.94	4.56	3.18	4.03
		symbolic	1.70	0.01	6.18	0.08	98.39	2.56	12.94	4.56	3.18	4.03
		portfolio	1.70	0.01	6.18	0.08	98.39	2.56	12.94	4.56	3.18	4.03
Drone rendezvous (double)	2	random	-	-	120.98	127.02	0.08	0.05	0.06	0.01	0.06	0.01
		symbolic	0.96	0.01	6.68	0.02						
		portfolio	0.96	0.01	6.68	0.02	0.08	0.05	0.06	0.01	0.06	0.01
	3	random	-	-	155.62	149.80	0.09	0.00	0.09	0.00	0.09	0.00
		symbolic	1.47	0.00	11.41	0.07						
		portfolio	1.47	0.00	11.41	0.07	0.09	0.00	0.09	0.00	0.09	0.00
	4	random	-	-	49.52	46.25	0.14	0.00	0.14	0.00	0.14	0.00
		symbolic	1.89	0.00	16.79	0.08						
		portfolio	1.89	0.00	16.79	0.08	0.14	0.00	0.14	0.00	0.14	0.00
Drone formation (double)	2	random	-	-	-	-	1.55	1.49	0.50	0.19	0.96	1.17
		symbolic	1.41	0.01	15.22	0.26						
		portfolio	1.41	0.01	15.22	0.26	1.55	1.49	0.50	0.19	0.96	1.17
	3	random	-	-	-	-	0.16	0.06	0.18	0.09	0.15	0.03
		symbolic	2.01	0.03	20.23	0.40						
		portfolio	2.01	0.03	20.23	0.40	0.16	0.06	0.18	0.09	0.15	0.03
	4	random	-	-	1.24	0.97	0.18	0.03	0.18	0.03	0.18	0.03
		symbolic	2.81	0.05	26.28	0.53						
		portfolio	2.81	0.05	1.24	0.97	0.18	0.03	0.18	0.03	0.18	0.03

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.

Model	N	B	Symbolic with merging				Symbolic without merging				Model	N	B	Symbolic with merging				Symbolic without merging			
			Time	|Const| (k)	# SMT call	# State	Time	|Const| (M)	# SMT call(k)	# State				Time	|Const| (k)	# SMT call	# State	Time	|Const| (M)	# SMT call(k)	# State
Thermostat	2	1	0.2	3.5	61.0	2	0.5	0.1	0.7	10	WaterTank	2	1	0.2	5.8	57.0	2	2.7	0.3	2.8	37
		2	0.4	14.5	144.0	3	25.6	3.1	20.7	86			2	0.4	22.2	135.0	3	148.5	18.4	102.1	181
		3	0.7	35.8	227.0	4	544.3	54.6	225.0	331			3	0.7	50.5	213.0	4	994.3	129.4	499.3	791
		4	1.6	67.2	310.0	5	3675.7	324.4	982.7	1055			4	1.2	90.9	291.0	5	5721.4	776.3	2181.5	3075
	3	1	0.2	5.3	91.0	2	6.0	0.6	6.6	28		3	1	0.3	8.8	84.0	2	68.0	5.8	51.8	217
		2	0.6	24.6	215.0	3	2315.3	270.6	1312.0	976			2	0.6	37.4	200.0	3	-	-	-	-
		3	1.3	65.8	339.0	4	-	-	-	-			3	1.2	90.1	316.0	4	-	-	-	-
		4	3.7	128.8	463.0	5	-	-	-	-			4	2.2	167.6	432.0	5	-	-	-	-
	4	1	0.3	7.2	121.0	2	67.6	6.1	59.1	82		4	1	0.4	11.8	111.0	2	1523.0	124.7	940.4	1297
		2	0.8	36.2	286.0	3	-	-	-	-			2	0.9	54.9	265.0	3	-	-	-	-
		3	2.3	103.3	451.0	4	-	-	-	-			3	2.0	138.7	419.0	4	-	-	-	-
		4	7.2	208.2	616.0	5	-	-	-	-			4	3.5	264.0	573.0	5	-	-	-	-
Drone Rendezvous (Single)	2	1	0.5	38.1	293.0	2	14.4	1.2	10.0	677	Drone Formation (Single)	2	1	0.7	50.7	328.0	2	18.6	1.5	10.0	677
		2	1.5	133.9	585.0	3	6099.0	463.8	2185.8	21003			2	2.1	184.6	655.0	3	-	-	-	-
		3	2.8	299.6	877.0	4	-	-	-	-			3	4.2	437.2	982.0	4	-	-	-	-
		4	4.6	534.2	1169.0	5	-	-	-	-			4	8.0	814.1	1309.0	5	-	-	-	-
	3	1	0.7	57.2	438.0	2	467.3	38.8	260.1	17577		3	1	1.0	75.6	473.0	2	600.6	47.0	260.1	17577
		2	2.2	227.2	875.0	3	-	-	-	-			2	3.1	302.9	945.0	3	-	-	-	-
		3	5.0	543.0	1312.0	4	-	-	-	-			3	7.0	756.0	1417.0	4	-	-	-	-
		4	8.9	1000.5	1749.0	5	-	-	-	-			4	17.3	1446.3	1889.0	5	-	-	-	-
	4	1	1.0	76.2	583.0	2	-	-	-	-		4	1	1.3	100.5	618.0	2	-	-	-	-
		2	3.2	338.2	1165.0	3	-	-	-	-			2	4.3	440.4	1235.0	3	-	-	-	-
		3	7.6	848.8	1747.0	4	-	-	-	-			3	10.4	1148.3	1852.0	4	-	-	-	-
		4	14.2	1599.7	2329.0	5	-	-	-	-			4	-	-	-	-	-	-	-	-
Drone Rendezvous (Double)	2	1	0.7	49.2	293.0	2	15.7	1.3	8.3	485	Drone Formation (Double)	2	1	1.1	67.2	328.0	2	21.5	1.7	9.6	443
		2	2.2	175.9	585.0	3	-	-	-	-			2	3.3	248.4	655.0	3	-	-	-	-
		3	4.6	399.0	877.0	4	-	-	-	-			3	8.1	601.0	982.0	4	-	-	-	-
		4	8.9	719.9	1169.0	5	-	-	-	-			4	-	193.5	652.0		-	-	-	-
	3	1	1.1	73.8	438.0	2	456.9	33.8	183.5	10649		3	1	1.5	100.2	473.0	2	645.6	41.1	173.1	11493
		2	3.4	298.2	875.0	3	-	-	-	-			2	5.2	406.2	945.0	3	-	-	-	-
		3	7.6	722.7	1312.0	4	-	-	-	-			3	-	324.4	942.0		-	-	-	-
		4	15.9	1347.3	1749.0	5	-	-	-	-			4	-	324.4	942.0		-	-	-	-
	4	1	1.4	98.4	583.0	2	-	-	-	-		4	1	2.0	133.3	618.0	2	-	-	-	-
		2	4.9	443.5	1165.0	3	-	-	-	-			2	7.4	589.7	1235.0	3	-	-	-	-
		3	11.5	1129.1	1747.0	4	-	-	-	-			3	-	-	-	-	-	-	-	-
		4	24.4	2152.9	2329.0	5	-	-	-	-			4	-	-	-	-	-	-	-	-

Table 8. Symbolic analysis with merging and without merging. $|Const|$ denotes the accumulated size of SMT constraints in thousands.

Model	B	N = 2						N = 3					
		|Time sample| = 1			|Time sample| = 2			|Time sample| = 1			|Time sample| = 2		
		Time		# States (k)	Time		# States (k)	Time		# States (k)	Time		# States (k)
		Average (s)	STDEV		Average (s)	STDEV		Average (s)	STDEV		Average (s)	STDEV	
Thermostat	50	8.6	4.7	9.9	69.4	37.2	93.0	5,027.4	2,533.6	1,808.9	-	-	-
	100	8.5	4.7	10.0	74.2	42.5	99.8	4,232.3	249.2	1,812.7	-	-	-
	150	10.3	4.8	12.5	127.7	74.5	187.0	6,820.4	1,830.1	2,365.1	-	-	-
	200	9.5	4.1	13.2	149.1	76.4	250.7	-	-	-	-	-	-
	250	13.6	5.8	18.7	332.9	176.5	506.2	-	-	-	-	-	-
	300	14.5	8.2	20.0	391.6	233.9	596.4	-	-	-	-	-	-
WaterTank	50	7.8	2.4	9.8	79.7	35.3	91.1	5,036.6	1,221.9	1,793.2	-	-	-
	100	8.2	1.7	9.8	60.8	4.0	96.5	5,106.2	1,327.5	1,794.1	-	-	-
	150	9.3	1.5	10.9	84.7	8.7	139.6	5,173.0	925.1	1,876.0	-	-	-
	200	9.0	1.4	11.0	103.1	45.7	158.6	5,364.1	1,258.0	1,878.1	-	-	-
	250	9.7	1.2	12.0	183.4	120.3	235.3	5,162.0	100.7	1,952.5	-	-	-
	300	10.2	3.8	12.1	163.1	80.2	254.2	5,462.6	933.5	1,960.9	-	-	-
Drone Rendezvous (Single)	50	8.8	4.5	9.3	67.7	38.0	84.0	1,141.2	648.5	886.1	-	-	-
	100	8.8	5.5	10.7	70.9	43.3	90.5	1,195.2	667.0	939.7	-	-	-
	150	10.1	4.5	11.5	75.9	38.6	97.8	1,229.2	664.5	988.9	-	-	-
	200	15.4	8.5	19.2	92.4	47.5	134.4	3,015.8	1,424.6	2,253.9	-	-	-
	250	15.6	9.2	23.4	109.7	56.5	171.4	3,884.3	1,505.3	2,902.1	-	-	-
	300	20.7	10.2	30.9	131.4	69.5	207.3	5,935.4	230.1	3,923.7	-	-	-
Drone Formation (Single)	50	7,163.6	207.5	3,540.3	-	-	-	-	-	-	-	-	-
	10	7,854.2	169.3	3,888.4	-	-	-	-	-	-	-	-	-
	150	8,196.7	265.8	4,037.5	-	-	-	-	-	-	-	-	-
	200	-	-	-	-	-	-	-	-	-	-	-	-
	250	-	-	-	-	-	-	-	-	-	-	-	-
	300	-	-	-	-	-	-	-	-	-	-	-	-
Drone Rendezvous (Double)	50	158.8	96.5	148.7	1,700.5	892.2	1,337.2	-	-	-	-	-	-
	100	154.3	4.6	188.2	1,927.1	1,063.6	1,500.6	-	-	-	-	-	-
	150	203.1	91.7	226.2	2,189.1	1,068.1	1,813.1	-	-	-	-	-	-
	200	812.2	89.2	1,121.2	8,882.6	1,830.2	5,495.6	-	-	-	-	-	-
	250	1,616.3	361.3	1,869.6	-	-	-	-	-	-	-	-	-
	300	2,855.1	620.7	2,764.0	-	-	-	-	-	-	-	-	-
Drone Formation (Double)	50	-	-	-	-	-	-	-	-	-	-	-	-
	100	-	-	-	-	-	-	-	-	-	-	-	-
	150	-	-	-	-	-	-	-	-	-	-	-	-
	200	-	-	-	-	-	-	-	-	-	-	-	-
	250	-	-	-	-	-	-	-	-	-	-	-	-
	300	-	-	-	-	-	-	-	-	-	-	-	-

Table 9. Analyzing distributed asynchronous models. |*Time sample*| is the number of the predefined values for sampling and actuating times, and the number of concrete states is written in thousands.