HybridSynchAADL Manual

1 Introduction

The HybridSynchAADL tool is a formal modeling and anlaysis tool for virtually synchronous distributed cyber-physical systems with complex control programs, continous behaviors, and bounded clock skews, network delays, and execution times.

The tool provides the HybridSynchAADL modeling language for virtually synchronous distributed hybrid systems using the avionics modeling standard AADL [28]. The tool also provides a property specification language to specify bounded reachability and invariant properties of HybridSynchAADL models.

The HybridSynchAADL tool is implemented as an OSATE plugin which performs various formal analysis using Maude combined with SMT solving. It provides a symbolic reachability analysis and randomized simulation.

The architecture of the HybridSynchAADL tool is illustrated in Figure 1. The tool first statically checks whether a given model is a valid model that satisfies the syntactic constraints of HybridSynchAADL. 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.

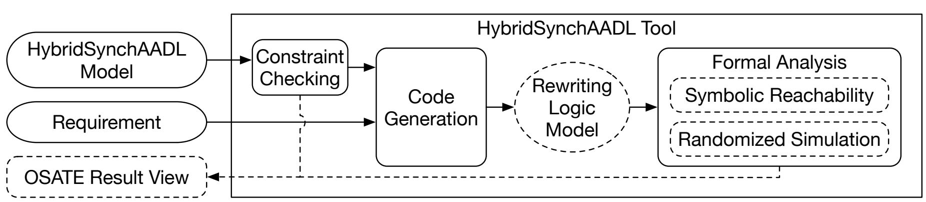


Fig. 1. The architecture of the HybridSynchAADL tool.

The tool is available at this link: `https://hybridsynhcaadl.github.io` which explains how to download the tool.

2 HybridSynchAADL Language

2.1 AADL Language.

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. The simple example of AADL component is illustrated in Figure 9.

An AADL model describes a system of hardware and software components. This manual 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.

2.2 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.

Property Set. 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.

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

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

Top-level System Component. 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;
```

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

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

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

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

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

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

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

Sometimes an environment component may include real-valued parameters or state variables that have the same constant values in each iteration, and can only be changed by a controller command; their dynamics can be specified as `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.

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].

Dispatch. The execution of an AADL thread is specified by the *dispatch protocol*. Since all "controller" components are executed in lock-step in HYBRID-SYNCHAADL, 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 3). In HybridSynchAADL, such unknown parameters can be declared using the following AADL property:

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

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. Therefore, two controller components can be connected only by data ports with delayed connections, declared by the connection property:

```
Timing => Delayed;
```

Connections Between Controller and Environment. In HybridSynchAADL, interactions between a controller and an environment occur *instantaneously* at the sampling and actuating times of the controller.[2] 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.

2.3 Property Specification Language.

The HybridSynchAADL's *property specification language* allows the user to easily specify invariant and reachability properties in an intuitive way, without having to understand an internal architecture. 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.

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

Atomic Propositions. Atomic propositions are given by AADL 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 using AADL Boolean expressions with an identifier as follows:

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

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

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.

Invariant 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}

Reachability Properties. 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 written as an invariant property by negating the goal condition.

reachability [*name*]: φ_{init} ==> φ_{goal} **in time** τ_{bound}

3 HybridSynchAADL Tool's Functionality

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.

3.1 Maude Preferences and PSPC Wizard.

Maude Preferences. The tool uses Maude with SMT to execute Maude code which represents the HybridSynchAADL model and properties. Open `Windows` ⇒ `Preferences` in the top menu. As illustrated in Figure 2, there is the Maude Preferences category in the left side of the window. Set the location of the Maude directory and the executable Maude file.

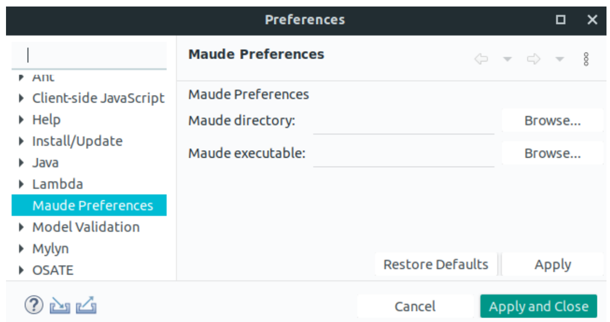


Fig. 2. Interface of the Maude Prefernces page

Property Specification Wizard. The tool provides a simple way to create a property specification (PSPC) language file. Open `New` ⇒ `Others` in the top menu. In the `New` window, click `HybridSynchAADL` ⇒ `HybridSynchAADL Property Specification Language`. As illustrated in Figure 3, Select the target instance.

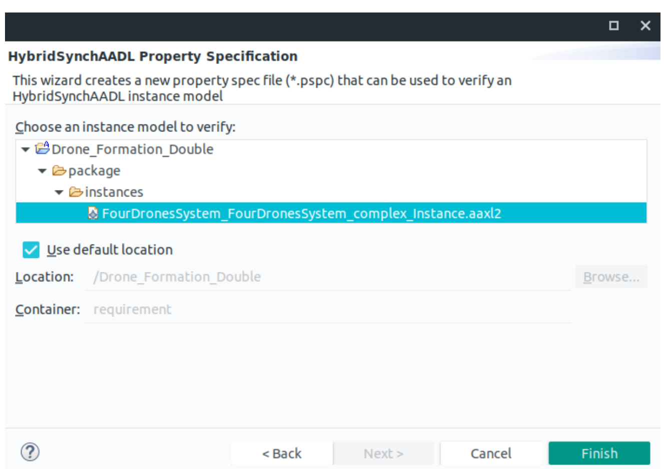


Fig. 3. The HybridSynchAADL property speicifcation language wizard.

3.2 Tool Interface.

Figure 4 shows the interface of our tool. The left editor shows the AADL code, 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 HybridSynchAADL Result view at the bottom displays the analysis results in a readable format.

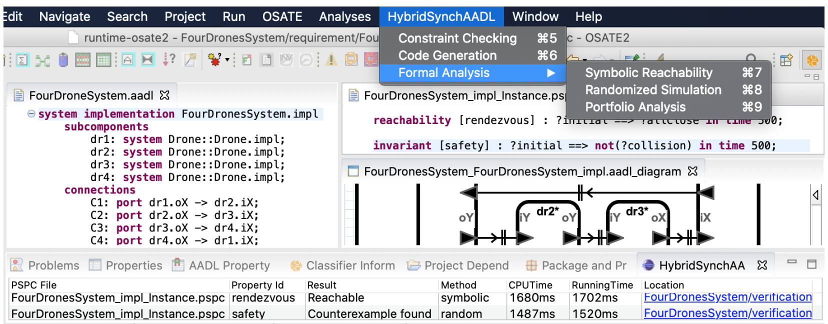


Fig. 4. Interface of the HybridSynchAADL tool.

Constraints Checking. 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.

Code Generation. The HybridSynchAADL tool synthesizes corresponding Maude code from the given model. During the process, when the error case occurs such as declaring a bus component (which is a hardware component), the tool shows an error message in the `Problem` view.

Formal Analysis. The HybridSynchAADL tool provides two different formal analysis methods: `randomized simulation` and `symbolic reachability anslysis`.

The `randomized simulation` repeatedly executes the model (using Maude) until a counterexample is found, by randomly choosing concrete sampling and actuating times, initial values of the state variables, nondeterministic transitions, etc. The randomized simulation is effective for finding "obvious" bugs.

The `symbolic reachability analysis` can verify that all possible behaviors—imposed by sensing and actuation times based on imprecise clocks—satisfy a given requirement;guarantee the absence of a counterexample.

The `portfolio analysis` combines symbolic reachability analysis and randomized simulation. HybridSynchAADL runs both analysis methods in parallel, and displays the result of the analysis that terminates first.

3.3 Analysis and Results.

HybridSynchAADL Analysis Configuration. As illustrated in Figure 5, the analysis method and corresponding parameters can be set in the `HybridSynchAADL Analysis` group in the `Run Configurations` window.

For the `randomized simulation`:

- `Random seed`: The random seed for the random function.
- `Default minimum bound of "param"`: The minimum bound of the paramaterized data component.
- `Default maximum bound of "param"`: The maximum bound of the parameterized data component.

For the `symbolic reachability analysis`:

- `Loop bound`: The maximal number of iterations in the loop statement specified in the behavior annex.
- `Transition bound`: The maximal number of transitions between states.

`PropSpec File` is for the path of the target PSPC file. `Timeout` is for the timeout value. When 'infinity' is written in `Timeout`, the tool analyzes properties until the results of the analysis come out.

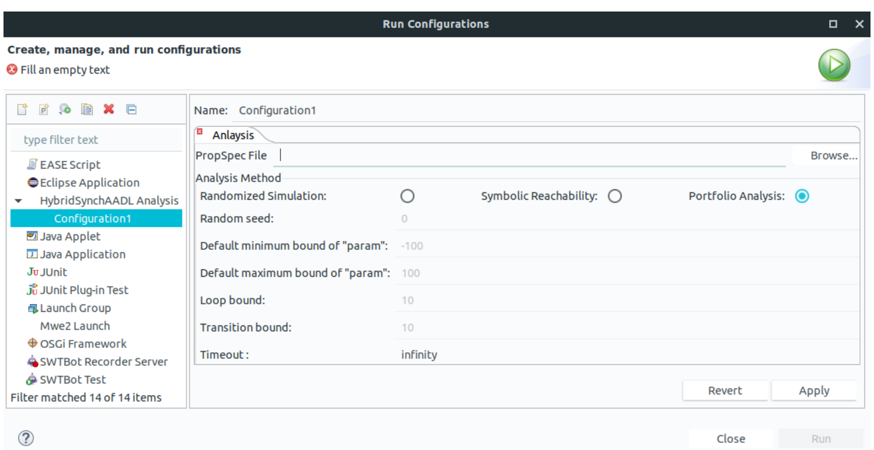


Fig. 5. Interface of the HybridSynchAADL run configuration.

HybridSynchAADL Result View. The tool shows the results of the analysis in the HybridSynchAADL Result view as illustrated in Figure 6. The meaning of each column is as follows:

- `PSPC File`: The analyzed PSPC file name.
- `Property Id`: The property name.
- `Result`: The analysis results.
- `Method`: The used method to get the result.
- `CPUTime`: The elappsed CPU time to get the result.
- `RunnignTime`: The elappsed running time.
- `Location`: The location of the result file.

In Figure 6, the concrete results of the analysis such as a concrete counterexample or witness is also shown in the editor as a sequence of states for synchronous steps. For example, the drone *dr3* has velocity $(-5126, 5682)$ at time 0 (i.e., in the initial state). You can see a counterexample/witness by clicking the link in `Location`.

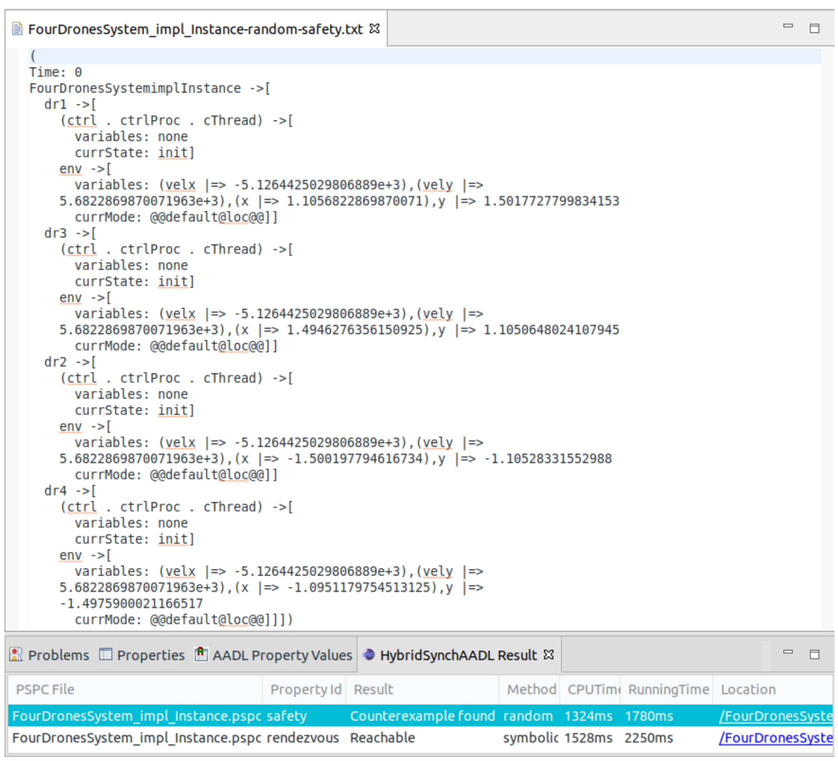


Fig. 6. HybridSynchAADL Analysis Results

4 Examples

We have developed a variety of HybridSynchAADL models for networked thermostat controllers, and 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`.

4.1 Networked Thermostat.

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 7 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).

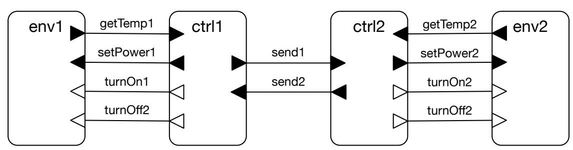


Fig. 7. A networked thermostat system.

The HybridSynchAADL Model.

Environment. Figure 9 gives an environment component `RoomEnv` for our networked thermostat system. Figure 8 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`.

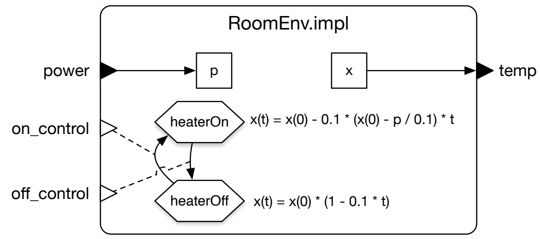


Fig. 8. An environment of the thermostat controller.

```
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. 9. A `RoomEnv` component.

Controller. Consider again our networked thermostat system. Figure 12 shows a controller system component. The system implementation `Thermostat.impl` includes the process component `thermProcess`. As shown in Figure 11 `thermProcess` again includes the thread component `thermThread` in `ThermostatProcess.impl`. The input and ouput port of a wrapper component are connected to the ports of the enclosed subcomponent.

Figure 10 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 eventually connected to an environment, and `tin` and `tout` are connected to another controller component (see Fig. 13). 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;
  properties
    Dispatch_Protocol => Periodic;
    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. 10. A simple thermostat thread.

```
process ThermostatProcess
  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;
end ThermostatProcess;
process implementation ThermostatProcess.impl
  subcomponents
    thermThread : thread ThermostatThread.impl;
  connections
    C1: port thermThread.on_control -> on_control;
    C2: port thermThread.off_control -> off_control;
    C3: port thermThread.set_power -> set_power;
    C4: port thermThread.tout -> tout;
    C5: port curr -> thermThread.curr;
    C6: port tin -> thermThread.tin;
end ThermostatProcess.impl;
```

Fig. 11. A simple thermostat process.

```
system Thermostat
  features
    on_control: out event port;    off_control: out event port;
    set_power: out data port Base_Types::Float
               {Data_Model::Initial_Value => ("0");};
    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");};
end Thermostat;
system implementation Thermostat.impl
  subcomponents
    thermProcess : process ThermostatProcess.impl;
  connections
    C1: port thermProcess.on_control -> on_control;
    C2: port thermProcess.off_control -> off_control;
    C3: port thermProcess.set_power -> set_power;
    C4: port thermProcess.tout -> tout;
    C5: port curr -> thermProcess.curr;
    C6: port tin -> thermProcess.tin;
end Thermostat.impl;
```

Fig. 12. A simple thermostat controller.

Top-Level Component. Figure 13 shows an implementation of a top-level system component `TwoThermostats` of our networked thermostat system, depicted in Figure 7. 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. 10, and the environment component `RoomEnv.impl` is given in Fig. 9. 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`.

```
system TwoThermostats
  properties
    Hybrid_SynchAADL::Synchronous => true;
end TwoThermostats;

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
    Period => 10 ms;
    Hybrid_SynchAADL::Max_Clock_Deviation => 1 ms;
    Timing => Delayed applies to send1, send2;
end TwoThermostats.impl;
```

Fig. 13. A top level component with two thermostat controllers.

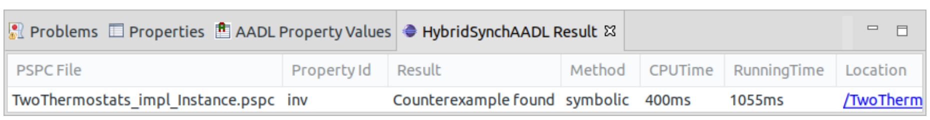
Property Specifications. Consider the thermostat system in Section 4.1 that consists of two thermostat controllers `ctrl1` and `ctrl2` and their environments `env1` and `env2`, respectively. The following declares two propositions `inRan1` and `inRan2` using the property specification language. For example, `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
```

As shown in Figure 14, there is a counterexample to `inv` up to bound 30. It takes 400ms in the CPU time and 1055ms in the running time to find a counterexample. The result is obtained by the symbolic analysis. Note that the result can also be obatined by the randomized simulation.



PSPC File	Property Id	Result	Method	CPUTime	RunningTime	Location
TwoThermostats_impl_Instance.pspc	inv	Counterexample found	symbolic	400ms	1055ms	/TwoTherm

Fig. 14. The analysis results of thermostat system.

4.2 Rendezvous Drones with Single-Integrator.

There are four distributed drones with rendezvous controller for single-integrator dynamics. Figure 15 illustrates the AADL architecture of the model. 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.

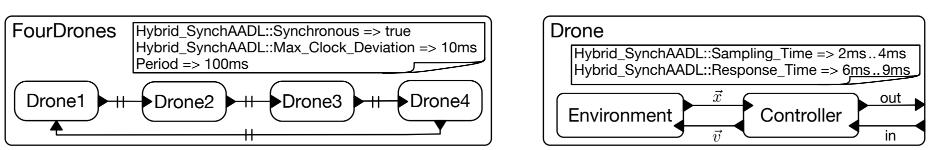


Fig. 15. The AADL architecture of four drones (left), and a drone component (right).

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.

The HybridSynchAADL Model.

Top-Level Component. The top-level component includes four `Drone` components (Fig. 16). 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`.

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

Drone Component. A drone component in Fig. 17 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.

```
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. 17. A drone component in HybridSynchAADL.

Environment. Figure 18 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 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 19 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 20, `ctrlProc` again includes the thread component

```
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. 18. An environment component in HybridSynchAADL.

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 21 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.

Property Specifications. 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 timesWe analyze them up to bound 500 ms.

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

```
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. 19. 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. 20. 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. 21. A controller thread in HybridSynchAADL

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. 18 and constrained by the condition `initial` and `velconst`. There are infinitely many initial states satisfying the proposition `initial`.

```
  (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;

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;
```

As shown in Figure 22, there is no counterexample to `safety` up to bound 500. The result is obtained by the symbolic analysis for `safety`, and by the randomized simulation for `rendezvous`.

Problems | Properties | AADL Property Values | HybridSynchAADL Result

PSPC File	Property Id	Result	Method	CPUTime	RunningTime	Location
FourDronesSystem_impl_Instance2.pspc	rendezvous	Reachable	random	92ms	512ms	/FourDrone
FourDronesSystem_impl_Instance2.pspc	safety	No counterexample found	symbolic	642068ms	642898ms	/FourDrone

Fig. 22. The AADL architecture of four drones (left), and a drone component (right).

References

1. Supplementary material: HybridSynchAADL technical report, semantics, benchmarks, and the tool, `https://hybridsynchaadl.github.io/`
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)