

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

Jaehun Lee¹, Sharon Kim¹, Kyungmin Bae¹, Peter Csaba Ölveczky², and

¹ Pohang University of Science and Technology

² University of Oslo

Abstract. We present the HYBRIDSYNCHAADL modeling language and formal analysis tool for virtually synchronous cyber-physical systems with complex control programs, continuous behaviors, and bounded clock skews, network delays, and execution times. We 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.

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.

This manual paper therefore defines the HYBRIDSYNCHAADL language for conveniently modeling virtually synchronous distributed hybrid systems using the avionics modeling standard AADL [28] (Section 2). 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.

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.

The rest of this manual paper is organized as follows. Section 2 presents the HYBRIDSYNCHAADL language. Section 3 presents the property specification language. Section 4 presents the HYBRIDSYNCHAADL tool. Section 5 presents case studies on virtually synchronous CPSs for controlling distributed drones. Section 6 presents the step-by-step process on how to exercise the benchmark model. Finally, Section 7 presents some concluding remarks.

2 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 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.³ The top-level system component declares the following properties to state that the model is a synchronous design and to declare the period of the system, respectively.

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

³ 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.

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).

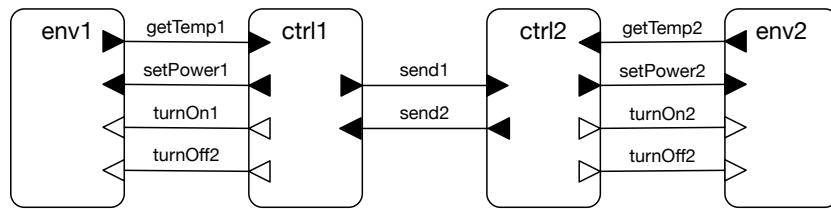


Fig. 1. A networked thermostat system.

2.1 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, \dots, x_n , say, $\frac{dx_i}{dt} = e_i(x_1, \dots, x_n)$ for $i = 1, \dots, n$, is written as a semicolon-separated string:

$d/dt(x_1) = e_1(x_1, \dots, x_n); \dots ; d/dt(x_n) = e_n(x_1, \dots, x_n);$

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 $x_1(0), \dots, x_n(0)$ of the variables x_1, \dots, x_n :

$x_1(t) = e_1(t, x_1(0), \dots, x_n(0)); \dots ; x_n(t) = e_n(t, x_1(0), \dots, x_n(0));$

Sometimes an environment component may include real-valued parameters or state variables that have the same constant values in each iteration, and can

only be changed by a controller command; their dynamics can be specified as $d/dt(x) = 0$ or $x(t) = x(0)$, and can be omitted in HYBRIDSYNCHAADL.

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

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

There are two modes `heaterOn` and `heaterOff` with their respective continuous dynamics, specified by `Hybrid_SynchAADL::ContinuousDynamics`, using continuous functions over time parameter t , where `heaterOff` is the initial mode. Because `p` is a constant, `p`’s dynamics $d/dt(p) = 0$ is omitted. The value `x` 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`.

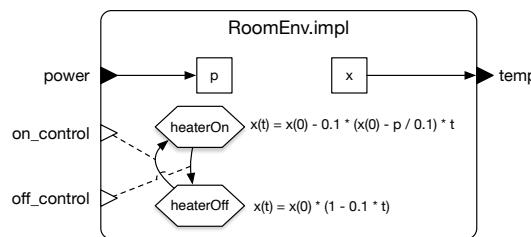


Fig. 2. An environment of the thermostat controller.

2.2 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;
```

```
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.

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.

2.3 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;

```

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.

```

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.

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.⁴ 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 More precisely, processing times and delays between environments and controllers are modeled using sampling and actuating times.

3 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 $path \mid exp$ 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]:  $\varphi_{init} ==> \varphi_{inv}$  in time  $\tau_{bound}$ 
```

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]:  $\varphi_{init} ==> \varphi_{goal}$  in time  $\tau_{bound}$ 
```

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

$$path \mid exp$$

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

Example 5. Consider the thermostat system in Section 2 that consists of two thermostat controllers `ctr11` and `ctr12` 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.

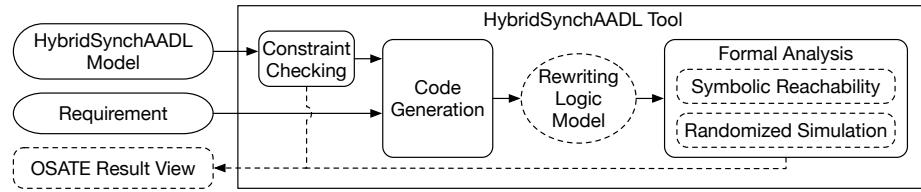


Fig. 6. The architecture of the HYBRIDSYNCHAADL tool.

```

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 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) synthesizes a rewriting logic model from a HYBRIDSYNCHAADL model, and (ii) performs various formal analyses using Maude combined with SMT solving.

4.1 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 in Section 2. It uses OSATE's code generation facilities to synthesize the corresponding Maude model from the validated model. Finally, our tool invokes Maude and an SMT solver to check whether the model satisfies given invariant and reachability requirements with respect to the formal semantics of HYBRIDSYNCHAADL. The Result view in OSATE displays the results of the analysis in a readable format.

By syntactically validating a HYBRIDSYNCHAADL model, we ensure that the model satisfies all the syntactic constraints of HYBRIDSYNCHAADL, and thus the corresponding Maude model is executable. For example, environment components (with `Hybrid_SynchAADL::isEnvironment`) can only contain data

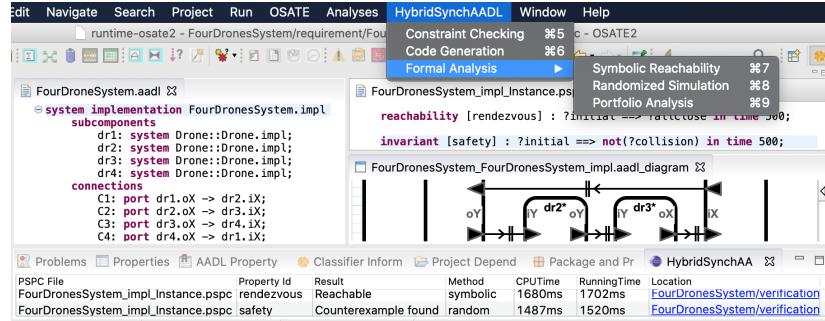


Fig. 7. Interface of the HYBRIDSYNCHAADL tool.

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;⁵ 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.

Figure 7 shows the interface of our tool that is fully integrated into OSATE. The left editor shows the code of `FourDroneSystem` 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.

4.2 Syntactic Constraints Checking

The HYBRIDSYNCHAADL tool checks whether a given model is a valid HYBRIDSYNCHAADL model that satisfies the syntactic constraints. For an erroneous model, the tool shows error messages in the ‘Problem’ view. The ‘Constraints

⁵ Symbolic analysis only supports (nonlinear) polynomial continuous dynamics, since the underlying SMT solver, Yices2, does not support general classes of ODEs.

'Checking' in the 'HybridSynchAADL' menu performs this process. Note that this process needs information of the target instance file. Therefore the 'Constraints Checking' menu button is enabled when the PSPC file editor which contains the location of the instance file is open.

The following lists shows which constraints are checked by the tool:

- The top-level system component must declare the property:
 $\text{Hybrid_SynchAADL} :: \text{Synrhonous} \Rightarrow \text{true};$
- All component except the data component must have the same period
- The system component connected to the other system component must initialize their output data port.
- The data component can only have boolean, float ,or the string value "param".
- The output data port is not allowed to be initialized except for boolean or float value.
- The type of data component in the environment component must be a float type.
- The environment component can only have data component as its subcomponent.
- The environment component must declare a
 $\text{Hybrid_SynchAADL} :: \text{ContinuousDynamics}$ property.
- Any component except the environment component must not declare a
 $\text{Hybrid_SynchAADL} :: \text{ContinuousDynamics}$ property.
- The data port value in the environment component can not be used directly in a $\text{Hybrid_SynchAADL} :: \text{ContinousDynamics}$ property.
- The $\text{Hybrid_SynchAADL} :: \text{ContinousDynamics}$ property must not have a syntax error.
- The target in $\text{Hybrid_SynchAADL} :: \text{ContinousDynamics}$ property must be a data component.
- Data components must be initialized by a $\text{Data_Model} :: \text{Initial_Value}$ property.
- The thread component must declare a sampling time property
 $\text{Hybrid_SynchAADL} :: \text{Sampling_Time}$
- The thread component must declare an actuation time property
 $\text{Hybrid_SynchAADL} :: \text{Response_Time}$
- The thread component must declare a max clock deviation property
 $\text{Hybrid_SynchAADL} :: \text{Max_Clock_Deviation}$
- The maximum value of the sampling time must be smaller than the minimum value of the actuation time.
- The maximum value of the actuation time must be less than the period.
- The connections between controller components must be delayed connection.
- The connections between the controller component and environment component must be immediate connection.
- The connections between environment components are not allowed.
- The output port must be connected to the input port.
- The input port must be connected to the output port.
- The output port in the environment component must be a data port.

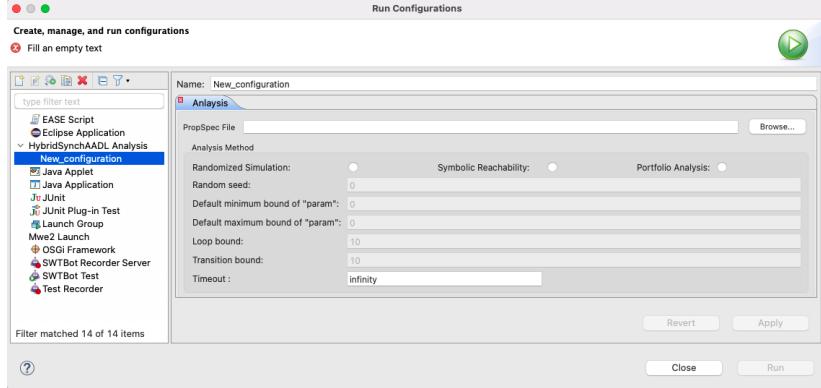


Fig. 8. Interface of the HYBRIDSYNCHAADL run configuration.

The tool shows an error message with a specific error location in the component and how to fix messages. When you double-click the error message in the 'Problem' view, the tool automatically locates the specific error component.

4.3 Code Generation

The HYBRIDSYNCHAADL tool synthesizes corresponding Maude code from the given model. During the process, when the error case occurs such as creating a bus component (which is a hardware component), the tool shows an error message in the 'Problem' view. The 'Code Generation' in the 'HybridSynchAADL' menu performs this process. The menu is enabled when the PSPC file editor file is opened. When the user clicks 'Code Generation', the tool firstly checks whether the given model is a valid HYBRIDSYNCHAADL model or not, which is the same process when the 'Constraints Checking' button is clicked and then synthesizes corresponding Maude code.

4.4 Formal Analysis

The HYBRIDSYNCHAADL tool provides two formal analysis methods: *Symbolic reachability analysis* and *Randomized Simulation*. The portfolio analysis uses both methods simultaneously and shows the results that terminate first. When you click the sub-menu in the 'Formal Analysis' button, 'Run Configuration' window is open.

When you perform an analysis for the first time, there is no configuration in the 'HybridSynchAADL Analysis' group. To create a configuration, double click 'HybridSynchAADL Analysis'. As illustrated in Figure 8, there are a lot of text fields in the configuration. Depending on the analysis method, some of the text fields are enabled and others are disabled:

For the randomized simulation:

- *Random seed*: The random seed for randomized simulation. It should be positive integer value.
- *Default minimum bound of "param"*: The minimum bound of the parameterized data component. It should be smaller than *Defualt maximum bound of "param"*.
- *Default maximum bound of "param"*: The maximum bound of the parameterized data component.

For the symbolic reachability analysis:

- *Loop bound*: The number of iterations in the loop statement specified in the behavior annex. It should be positive integer value.
- *Transition bound*: The number of transitions between states.

The *PropSpec File* text field is for the path of the PSPC file which contains properties to be verified. The *Timeout* text field is the timeout value. When 'infinity' is passed over, it never terminates until the results of the analysis come out. Note that when the user uses the portfolio analysis method, all text fields should be set to proper values.

4.5 HYBRIDSYNCHAADL Result View

The tool shows the results of the analysis in the HYBRIDSYNCHAADL Result view. The description of each column is as follows:

- *PSPC File*: The target PSPC file name to be verified
- *Property Id*: The property name
- *Result*: For invariant property, it shows "Counterexampe found" or "Counterexample not found". For reachability property, it shows "Reachable" or "Unreacahble".
- *Method*: The used method to get the result. In case of portfolio analysis, it shows the method which terminates first.
- *CPUTime*: The elappsed CPU time to get the result.
- *RunnignTime*: The elappsed actual running time.
- *Location*: The path of the result file which contains a counterexample or witness.

PSPC File	Property Id	Result	Method	CPUTime	RunningTime	Location
FourDronesSystemImpl_Instance.pspc	safety	Counterexample found	random	1669ms	2122ms	/FourDronesSystem/verification/result/F
FourDronesSystemImpl_Instance.pspc	rendezvous	Reachable	symbolic	12528ms	13288ms	/FourDronesSystem/verification/result/F

Fig. 9. Interface of the Maude Preferences page

The tool also supports exportation to CSV files of the given analysis results. Right click on the result columns and click 'Export into CSV files'.

4.6 Maude Preferences

The tool uses Maude with SMT to execute Maude code which corresponds to the given model and properties. The Maude with SMT can be downloaded on the following website: <https://maude-se.github.io>. To use Maude, the user must set the directory containing Maude and the executable Maude file. Open 'Windows' > 'Preferences' in the top menu. As illustrated in Figure 10, there is the Maude Preferences category in the 'Preferences' window. The user must be set the directory containing 'prelude.maude', 'smt.maude' ,and Maude executable file.

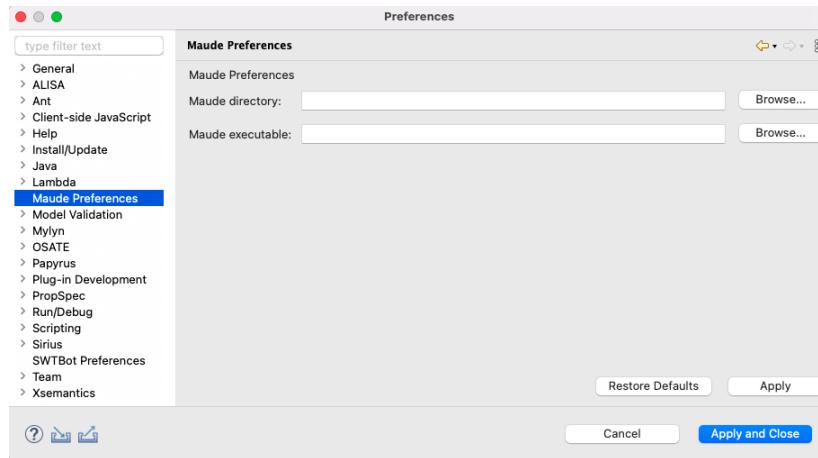


Fig. 10. Interface of the Maude Preferences page

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. 11. 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



Fig. 11. Rendezvous and formation control of distributed drones

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 $\ddot{\vec{x}}_i = \vec{a}_i$. 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:

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

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:

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

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

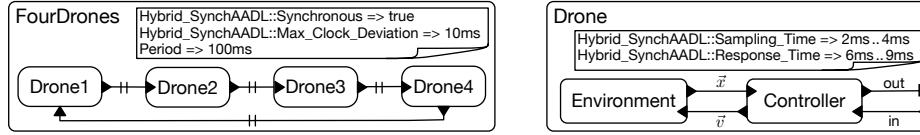


Fig. 12. The AADL architecture of four drones (left), and a drone component (right).

0, and the controller of a drone directly gives the value of velocity as a control input. For single-integrator dynamics, the following equations provide velocity \vec{v}_i for rendezvous and formation control, respectively [45]:

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

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

5.2 The HYBRIDSYNCHAADL Model

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

Figure 12 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.

```

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. 13. The top-level system component FourDronesSystem.

Top-Level Component. The top-level component includes four Drone components (Fig. 13). 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. 14 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 15 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 $y(t) = vel_y t +$

```

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. 14. A drone component in HYBRIDSYNCHAADL.

$y(0)$ 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 16 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 17, `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 18 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 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. 15. An environment component 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. 15 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

```

```

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. 16. 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.vY -> vY;
  end DroneControlProc.impl;

```

Fig. 17. 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. 18. A controller thread in HYBRIDSYNCHAADL

```

(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 19. 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.

We therefore modify **safety** and **rendezvous** below by adding such a velocity constraint **velconst** to the initial condition. As shown in Fig. 20, 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.

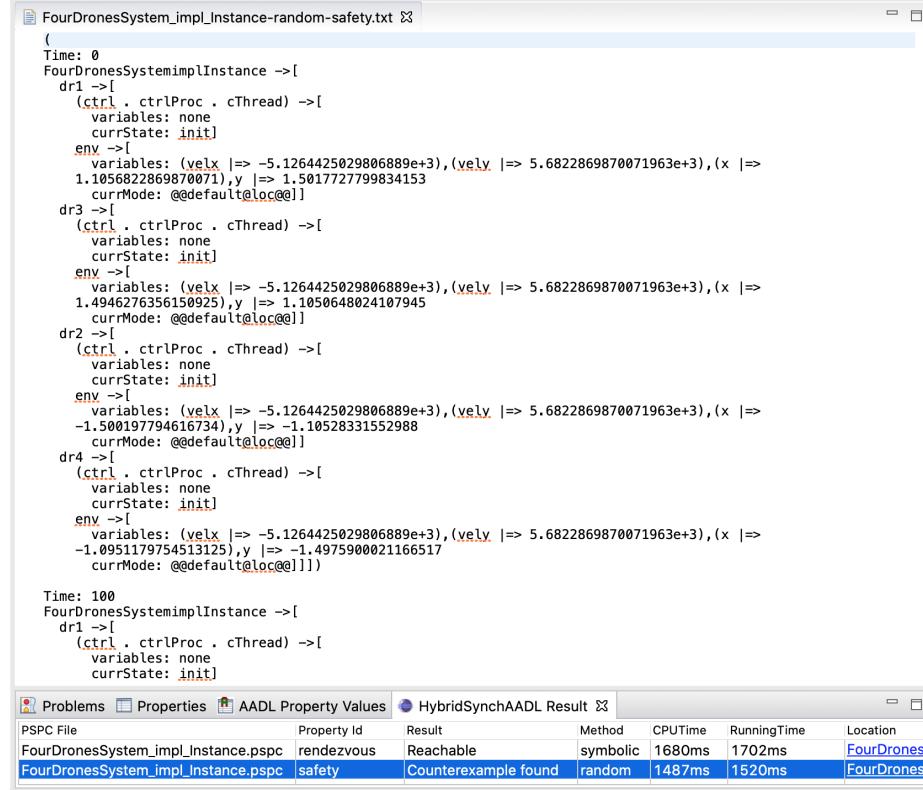
```

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.



The screenshot shows two windows from the OSATE2 IDE. The top window displays a concrete counterexample for the 'safety' property of the 'FourDronesSystemImpl_Instance.pspc' file. It lists four drone states (dr1, dr2, dr3, dr4) at Time: 0 and Time: 100, each with its control process, thread, variables, current state, and environment. The bottom window shows the 'HybridSynchAADL Result' table, which includes columns for PSPC File, Property Id, Result, Method, CPU Time, Running Time, and Location. It shows two rows: one for 'rendezvous' (Reachable, symbolic, 1680ms, 1702ms, FourDrones) and one for 'safety' (Counterexample found, random, 1487ms, 1520ms, FourDrones).

PSPC File	Property Id	Result	Method	CPUTime	RunningTime	Location
FourDronesSystemImpl_Instance.pspc	rendezvous	Reachable	symbolic	1680ms	1702ms	FourDrones
FourDronesSystemImpl_Instance.pspc	safety	Counterexample found	random	1487ms	1520ms	FourDrones

Fig. 19. Results for the safety and rendezvous properties.

6 Exercise: Analyzing Formation Control of Distributed Drones

This section shows step-by-step processes with screenshot images illustrating each step. We use the benchmark model: formation control of distributed drone system. The benchmark in this section can be downloaded on our website: <https://hybridsynchaadl.github.io>. The process starts by importing the benchmark project and finish by showing a concrete counterexample.

6.1 Importing the Benchmark Model

OSATE2 is based on Eclipse IDE so that the import process is the same as the process in Eclipse IDE. To load the benchmark model in the OSATE2, you need to import it into the OSATE2 workspace. To import a project stored on your local machine:

1. Click 'File' > 'Import' in the top menu.

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. 20. Results for the modified properties.

2. Click 'General' > 'Existing Projects into Workspace' (Figure 21a)
3. Click 'Browse', and select 'FourDronesSystem_Formalization' as a root directory. (Figure 21b)

You can edit the project directly in its original location or choose to create a copy of the project in the workspace.

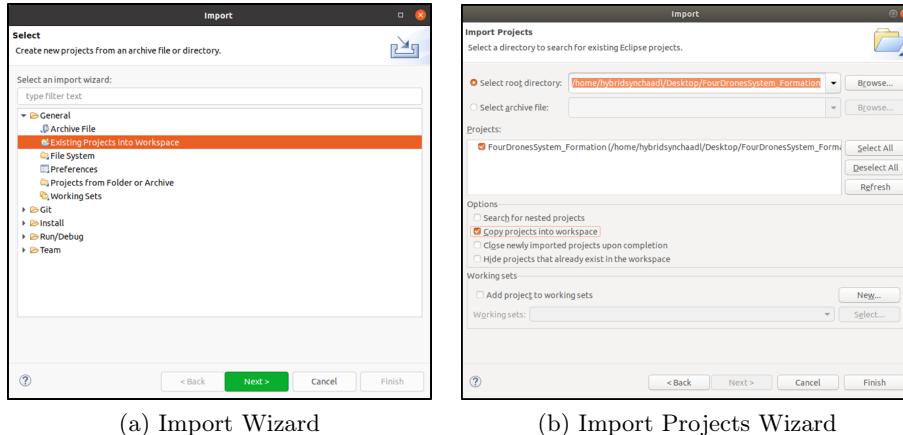
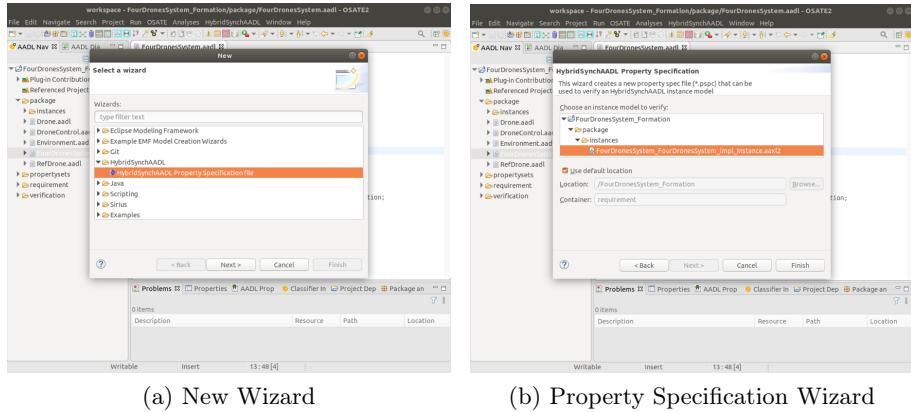
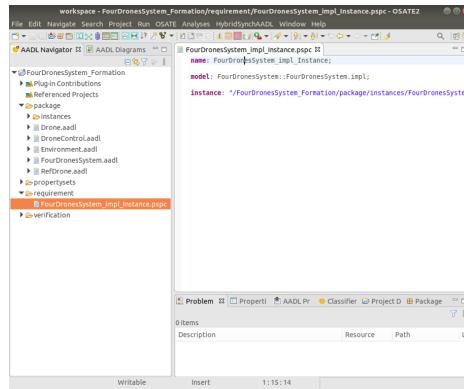


Fig. 21. Import an existing AADL project

6.2 Creating Property Specification Files(PSPC)

HYBRIDSYNCHAADL's property specification language can be written in a PSPC file. The PSPC file can be generated automatically from the AADL instance model file. To create an instance model from the AADL model, open the 'Outline' view by clicking 'Window' > 'Show View' > 'Outline' in the top menu. Right click on the top-level system implementation 'System impl FourDronesSystem.impl' and choose 'Instantiate'.

The created instance file is stored on the 'packages/instance' directory. From the instance file, the user can create a PSPC file. Click 'File' > 'New' > 'Other' in the top menu. Click 'HybridSynchAADL' > 'HybridSynchAADL Property Specification file' in the 'New' window as illustrated in Figure 22a. In the new wizard, click the generated instance file located in 'packages/instances'. The PSPC file is created in the 'requirement' directory as illustrated in Figure 22b.

**Fig. 22.** Create a PSPC file**Fig. 23.** Open a new PSPC file

After successfully create the PSPC File, the PSPC editor is opened as illustrated in Figure 23.

6.3 HybridSyncAADL Constraints Checker

The tool checks whether the AADL instance model is a syntactically validated HYBRIDSYNCHAADL model or not. Because the location information of target instance model is needed to check the validity of the model, the PSPC file editor which contains the location of the instance file should be activated. Note that in the imported benchmark model, there already exists the PSPC file.

To check whether the benchmark model is a valid HYBRIDSYNCHAADL model or not, click 'HybridSyncAADL' > 'Constraint Checking' in the top menu. The benchmark model is a valid > HYBRIDSYNCHAADL model so that the tool notifies that the model is valid.

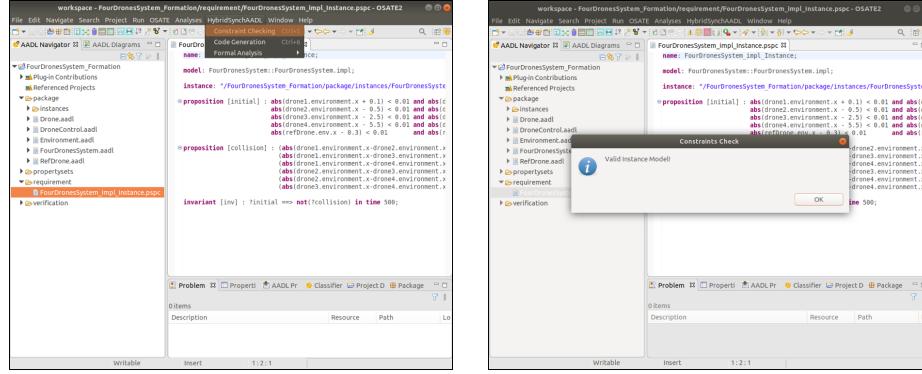


Fig. 24. Check a syntactically validated HYBRIDSYNCHAADL model

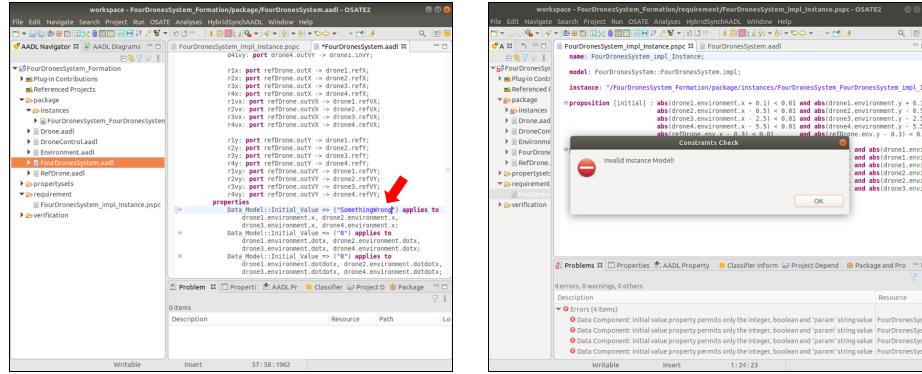


Fig. 25. Check an erroneous HYBRIDSYNCHAADL model

What if some HYBRIDSYNCHAADL constraints are not satisfied? As illustrated in Figure 25, add an invalid value to the data component. The initial value is the boolean value or floating number value. Before constraints checking, do not forget to instantiate the AADL model. In the case of the erroneous model, the tool shows an error message in the Problems view.

6.4 Maude Code Generation

The tool synthesizes Maude code from the valid instance HYBRIDSYNCHAADL model. Click 'HybridSynchAADL' > 'Code Generation' in the top menu. The tool automatically checks whether the given instance model is a valid HYBRIDSYNCHAADL model or not. If the instance model is valid, the tools generated Maude code in the 'verification/instance' directory as illustrated in Figure 26.

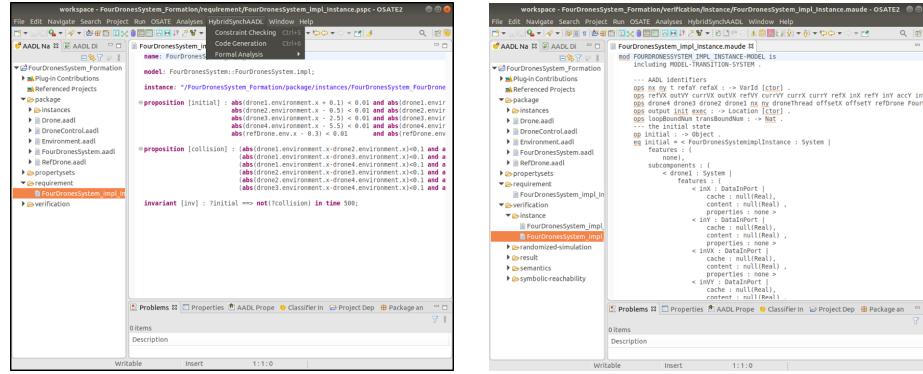


Fig. 26. Synthesize Maude code from the HYBRIDSYNCHAADL model

6.5 Formal Analysis

The tool provides three analysis methods: 'Symbolic Reachability', 'Randomized Simulation', and 'Portfolio Analysis'. Click 'Portfolio Analysis' to perform symbolic reachability and randomized simulation simultaneously using Maude with SMT. The 'Run Configuration' window contains no configuration file for our analysis. Create a new configuration file by double clicking the 'HybridSynchAADL Analysis' group as illustrated in Figure 27.

In the created configuration page, set 'PSPC File' to the PSPC file path located in the 'requirement' directory. To perform portfolio analysis, click the 'Portfolio Analysis' radio button. Because portfolio analysis uses symbolic reachability and randomized simulation, set random seed, default minimum and maximum bound of "param", loop bound, and trans bound, and then click 'Run'.

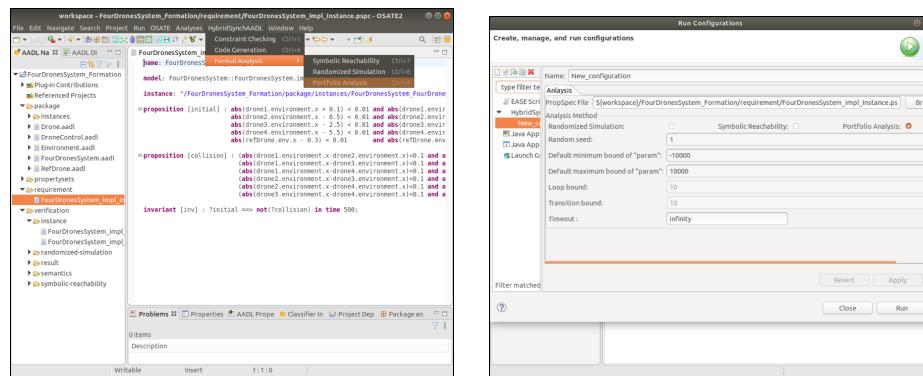


Fig. 27. Analyze the HYBRIDSYNCHAADL model using the portfolio method

The 'HybridSynchAADL Result' view shows the analysis results as illustrated in Figure 28. When clicking the path in the 'Location' column, the tool shows a concrete counterexample.

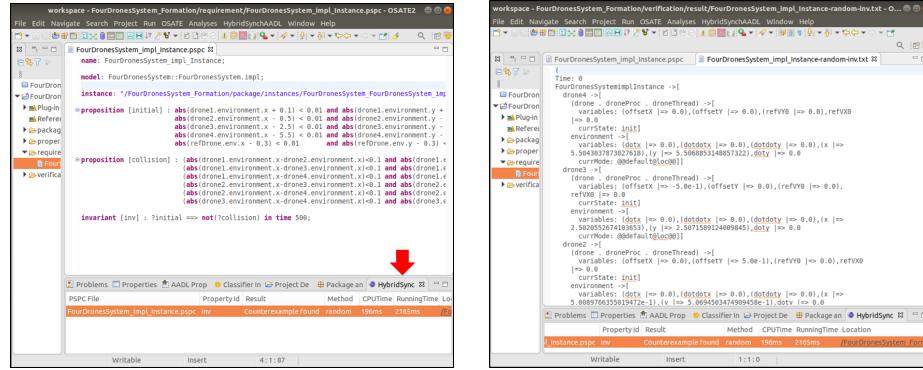


Fig. 28. Analysis results

7 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 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.

References

- Supplementary material: HybridSynchAADL technical report, semantics, benchmarks, and the tool, <https://hybridsynchaadl.github.io/>
- Abrial, J., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, LNCS, vol. 1165. Springer (1996)
- 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. Internetwork'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)