HybridSynchAADL Manual

1 Introduction

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

The tool provides the HybridSynchAADL modeling language using the avionics modeling standard AADL [28] and a property specification language to specify bounded reachability and invariant properties of HybridSynchAADL models. The 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 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.

The tool is available at [https://hybridsynchaadl.github.io/download](https://hybridsynchaadl.github.io/download) 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 sub-
components and a set of connections linking their ports. An AADL construct
may have properties describing its parameters, declared in property sets.

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. Software components include threads that model the applica-
tion 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 vari-
ables using AADL’s Behavior Annex [29]. The actions performed when a trans-
ition 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 cyber-physical systems in AADL. The language can specify environ-
ments with continuous dynamics, synchronous designs of distributed controllers,
and nontrivial interactions between controllers and environments with respect
to imprecise local clocks and sampling and actuation times.

The HybridSynchAADL language is a subset of AADL extended with the
following property set Hybrid_SynchAADL. We use a subset of AADL without
changing the meaning of AADL constructs or adding new a annex—the subset
has the same meaning for synchronous models and asynchronous distributed
implementations—so that AADL experts can easily develop and understand
HybridSynchAADL models.

<table>
<thead>
<tr>
<th>property set</th>
<th>Hybrid_SynchAADL is</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>inherit aadlboolean applies to (system, process, thread);</td>
</tr>
<tr>
<td>isEnvironment</td>
<td>inherit aadlboolean applies to (system);</td>
</tr>
<tr>
<td>ContinuousDynamics</td>
<td>aadlstring applies to (system);</td>
</tr>
<tr>
<td>Max_Clock_Deviation</td>
<td>inherit Time applies to (system, thread);</td>
</tr>
<tr>
<td>Sampling_Time</td>
<td>inherit Time_Range applies to (system, thread);</td>
</tr>
<tr>
<td>Response_Time</td>
<td>inherit Time_Range applies to (system, thread);</td>
</tr>
</tbody>
</table>
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, \(\frac{dx_i}{dt} = e_i(x_1, \ldots, x_n)\) for \(i = 1, \ldots, n\), is written as a semicolon-separated string:

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

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

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

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

An environment component interacts with discrete controllers by sending its state values, and by receiving actuator commands that may update the values of state variables or trigger mode (and hence trajectory) changes. This behavior is specified in HybridSynchAADL using connections between ports and data subcomponents. A connection from a data subcomponent inside the environment to an output data port of an environment component declares that the value of the data subcomponent is “sampled” by a controller through the output port of the environment component. A connection from an environment’s input port to a data subcomponent inside the environment declares that a controller command arrived at the input port and updates the value of the data subcomponent. When a discrete controller sends actuator commands, some input ports of the environment component may receive no value (more precisely, some “don’t care” value ⊥). 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 \[29\]. The hardware and scheduling features of AADL, which are not relevant to synchronous \textit{designs}, are not considered in HybridSynchAADL.

Dispatch. The execution of an AADL thread is specified by the \textit{dispatch protocol}. A thread with an \textit{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 \textit{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:

\begin{verbatim}
Dispatch_Protocol => Periodic;
\end{verbatim}

Timing Properties. A controller receives the state of the environment at some \textit{sampling time}, and sends a controller command to the environment at some \textit{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:

\begin{verbatim}
Hybrid_SynchAADL::Max_Clock_Deviation => time;
Hybrid_SynchAADL::Sampling_Time => lower bound .. upper bound;
Hybrid_SynchAADL::Response_Time => lower bound .. upper bound;
\end{verbatim}

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, \textit{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:

\begin{verbatim}
Data_Model::Initial_Value => ("value");
\end{verbatim}

Sometimes initial values can be \textit{parameters}, instead of concrete values. E.g., you can check whether a certain property holds from initial values satisfying a certain constraint for those parameters (see Section \[1\]). In HybridSynchAADL, such unknown parameters can be declared using the following AADL property:

\begin{verbatim}
Data_Model::Initial_Value => ("param");
\end{verbatim}
Communication. There are three kinds of ports: 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. 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.

2.3 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. 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

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

\[ \text{path} \mid \text{exp} \]

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

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

\[
\text{invariant [name]}: \; \varphi_{\text{init}} \Rightarrow \varphi_{\text{inv}} \text{ in time } \tau_{\text{bound}}
\]

**Reachability Properties.** A reachability property (the dual of an invariant) holds if a state satisfying \( \varphi_{\text{goal}} \) is reachable from some state satisfying the initial condition \( \varphi_{\text{init}} \) within the time bound \( \tau_{\text{bound}} \). It is worth noting that a reachability property can be written as an invariant property by negating the goal condition.

\[
\text{reachability [name]}: \; \varphi_{\text{init}} \Rightarrow \neg \varphi_{\text{goal}} \text{ in time } \tau_{\text{bound}}
\]

3 **HYBRIDSYNCHAADL Tool’s Functionality**

This section introduces the HYBRIDSYNCHAADL tool supporting the modeling and formal analysis of HYBRIDSYNCHAADL models. The tool is an OSAE plugin which: (i) provides the wizard which create a PSPC language file to specify properties of models, the Maude preference page to set up Maude. (ii) synthesizes a rewriting logic model from a HYBRIDSYNCHAADL model, and performs various formal analyses using Maude combined with SMT solving. (iii) shows the results of the analyses.

3.1 **Maude Preferences and PSPC Wizard**

**Maude Preferences.** The tool uses Maude with SMT to execute Maude code which is the formal representation of the models and properties. To use Maude, 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.
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 HybridSyncAADL ⇒ HybridSyncAADL Property Specification Language. As illustrated in Figure 3, Select an existing instance.

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 HYBRIDSYNC-AADL 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.

![HybridSynchAADL Interface](image)

**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.** HybridSynchAADL provides two formal analysis methods. *Symbolic reachability analysis* can verify that all possible behaviors—imposed by sensing and actuation times based on imprecise clocks—satisfy a given requirement; if not, a counterexample is generated. *Randomized simulation* repeatedly executes the model (using Maude) until a counterexample is found, by randomly choosing concrete sampling and actuating times, initial values of the state variables, 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.
3.3 Analysis Configuration and Result View

*HybridSynchAADL Analysis Configuration*. As illustrated in Figure 5 when the analysis method is selected in the HybridSynchAADL menu, the user can set the corresponding parameters in the HybridSynchAADL Analysis configuration in the Run Configurations window.

For the *randomized simulation*:

- **Random seed**: The initial random seed for the random operation.
- **Default minimum bound of "param"**: The minimum bound of the parameterized component.
- **Default maximum bound of "param"**: The maximum bound of the parameterized 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.

*PSPC 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.

![Image of Run Configurations window](image)

*Fig. 5*. Interface of the HybridSynchAADL analysis 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 target PSPC file name.
- **Property Id**: The analyzed property name.
- **Result**: The analysis result.
- **Method**: The used method to get the result.
- **CPUTime**: The elapsed CPU time.
- **RunningTime**: The elapsed running time.
- **Location**: The location of the result file.

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

![Image of analysis results](image.png)

**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 drones with respect to single-integrator and double-integrator dynamics. All these models are available at [https://hybridsynchaadl.github.io/benchmark](https://hybridsynchaadl.github.io/benchmark).

4.1 Networked Thermostat

The HYBRIDSYNCHAADL Model. 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.](image)

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 values.

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

The value of $x$ is sent to the controller through the output port $temp$, declared by the connection $port\ x \rightarrow temp$. When a discrete controller sends an actuation command through input ports $power$, $on\_control$, and $off\_control$, the mode changes according to the mode transitions, and the value of $p$ can be updated by the value of input port $power$, declared by the connection $port\ x \rightarrow temp$. 
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] the thread component thermThread is declared as subcomponents in Thermostat.impl. The input and output port of a wrapper component are connected to the ports of the enclosed subcomponent.

Figure [10] shows a thread component ThermThread 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 ThermThread
  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 ThermThread;

thread implementation ThermThread.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 ThermThread.impl;
```

Fig. 10. A simple thermostat thread.
process ThermProcess
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 ThermProcess;

process implementation ThermProcess.impl
subcomponents
    thermThread : thread ThermThread.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 ThermProcess.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 ThermProcess.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 components ctrl1 and ctrl2 are implemented using the system component Thermostat.impl in Figure 10 and the environment components env1 and env2 are given using RoomEnv.impl in Figure 9. Each discrete controller ctrli, for i = 1, 2, is connected to its environment component envi using four connections turnOni, turnOffi, setPoweri, and getTempi. 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 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.
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.

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 obtained by the randomized simulation.

Fig. 14. The analysis results of thermostat system.

4.2 Rendezvous Drones with Single-Integrator

The HybridSynchAADL Model. 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.

**Top-Level Component.** The top-level component includes four Drone components (Figure 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.

```plaintext
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 Figure 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.

```plaintext
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 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. An environment component in HYBRIDSYNCHAADL.

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 times. We analyze them up to bound 500 ms.

We define four atomic propositions collision, gather, initial, and velconst for four drones \(dr1\), \(dr2\), \(dr3\), and \(dr4\). Two drones collide if the distance between

```
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;
```
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
them is less than 0.1. All drones have gathered if the distance between each pair of drones is less than 1. The initial values of x, y, velx and vely are declared to be parametric in Figure 18 and constrained by the propositions initial and velconst. There are infinitely many states satisfying the initial condition.

**proposition [collision]:**

\[
\begin{align*}
& (\text{abs}(\text{dr1.env.x} - \text{dr2.env.x}) < 0.1 \text{ and abs}(\text{dr1.env.y} - \text{dr2.env.y}) < 0.1) \text{ or } \\
& (\text{abs}(\text{dr1.env.x} - \text{dr3.env.x}) < 0.1 \text{ and abs}(\text{dr1.env.y} - \text{dr3.env.y}) < 0.1) \text{ or } \\
& (\text{abs}(\text{dr1.env.x} - \text{dr4.env.x}) < 0.1 \text{ and abs}(\text{dr1.env.y} - \text{dr4.env.y}) < 0.1) \text{ or } \\
& (\text{abs}(\text{dr2.env.x} - \text{dr3.env.x}) < 0.1 \text{ and abs}(\text{dr2.env.y} - \text{dr3.env.y}) < 0.1) \text{ or } \\
& (\text{abs}(\text{dr2.env.x} - \text{dr4.env.x}) < 0.1 \text{ and abs}(\text{dr2.env.y} - \text{dr4.env.y}) < 0.1) \text{ or } \\
& (\text{abs}(\text{dr3.env.x} - \text{dr4.env.x}) < 0.1 \text{ and abs}(\text{dr3.env.y} - \text{dr4.env.y}) < 0.1); \\
\end{align*}
\]

**proposition [gather]:**

\[
\begin{align*}
& \text{abs}(\text{dr1.env.x} - \text{dr2.env.x}) < 1 \text{ and abs}(\text{dr1.env.y} - \text{dr2.env.y}) < 1 \text{ and } \\
& \text{abs}(\text{dr1.env.x} - \text{dr3.env.x}) < 1 \text{ and abs}(\text{dr1.env.y} - \text{dr3.env.y}) < 1 \text{ and } \\
& \text{abs}(\text{dr1.env.x} - \text{dr4.env.x}) < 1 \text{ and abs}(\text{dr1.env.y} - \text{dr4.env.y}) < 1 \text{ and } \\
& \text{abs}(\text{dr2.env.x} - \text{dr3.env.x}) < 1 \text{ and abs}(\text{dr2.env.y} - \text{dr3.env.y}) < 1 \text{ and } \\
& \text{abs}(\text{dr2.env.x} - \text{dr4.env.x}) < 1 \text{ and abs}(\text{dr2.env.y} - \text{dr4.env.y}) < 1 \text{ and } \\
& \text{abs}(\text{dr3.env.x} - \text{dr4.env.x}) < 1 \text{ and abs}(\text{dr3.env.y} - \text{dr4.env.y}) < 1; \\
\end{align*}
\]

**proposition [initial]:**

\[
\begin{align*}
& \text{abs}(\text{dr1.env.x} - 1.1) < 0.01 \text{ and abs}(\text{dr1.env.y} - 1.5) < 0.01 \text{ and } \\
& \text{abs}(\text{dr2.env.x} + 1.5) < 0.01 \text{ and abs}(\text{dr2.env.y} + 1.1) < 0.01 \text{ and } \\
& \text{abs}(\text{dr3.env.x} - 1.5) < 0.01 \text{ and abs}(\text{dr3.env.y} - 1.1) < 0.01 \text{ and } \\
& \text{abs}(\text{dr4.env.x} + 1.1) < 0.01 \text{ and abs}(\text{dr4.env.y} + 1.5) < 0.01; \\
\end{align*}
\]

**proposition [velconst]:**

\[
\begin{align*}
& \text{abs}(\text{dr1.env.velx}) \leq 0.01 \text{ and abs}(\text{dr1.env.vely}) \leq 0.01 \text{ and } \\
& \text{abs}(\text{dr2.env.velx}) \leq 0.01 \text{ and abs}(\text{dr2.env.vely}) \leq 0.01 \text{ and } \\
& \text{abs}(\text{dr3.env.velx}) \leq 0.01 \text{ and abs}(\text{dr3.env.vely}) \leq 0.01 \text{ and } \\
& \text{abs}(\text{dr4.env.velx}) \leq 0.01 \text{ and abs}(\text{dr4.env.vely}) \leq 0.01; \\
\end{align*}
\]

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.

Fig. 22. The analysis results of drone rendezvous models
4.3 Formation Drones with Double-Integrator

The HybridSynchAADL Model. There are four distributed drones with a formation controller for double-integrator dynamics. Compared to the rendezvous drones in Section 4.2 each drone sends its position and velocity to the connected drone. In each round, a controller determines a new acceleration. The controller obtains the position and velocity from its environment. The environment changes its position and velocity according to the acceleration indicated by its controller.

In the case of formation drones, all drones follow a reference drone which changes its acceleration in a predefined set of accelerations in each round. All drones try to keep formation based on the position of the reference drone and offset values.

Top-Level Component. The top-level component includes four Drone components and one RefDrone component as illustrated in Figure 23. Each drone sends its position and velocity through its output ports \( oX, oY, oVX, \) and \( oVY \), and receives the position and velocity of the other drone through its input ports \( iX, iY, iVX, \) and \( iVY \). Each drone also receives the position and velocity of the reference drone through its input ports \( rX, rY, rVX, \) and \( rVY \). To keep the formation, each drone has initialized offset values \( \text{offsetX} \) and \( \text{offsetY} \) in its thread component.

Drone Component. A drone component in Figure 24 has input ports \( iX, iY, iVX, iVY, rX \) and \( rY \) and output ports \( oX, oY, oVX, \) and \( oVY \). The controller \( \text{ctrl} \) obtains the current position and velocity from \( \text{env} \) via input ports \( cX, cY, cVX, \) and \( cVY \), and sends a new acceleration to \( \text{env} \) via output ports \( aX \) and \( aY \). The controller \( \text{ctrl} \) obtains the reference drone’s position and velocity through its input ports \( rX, rY, rVX, \) and \( rVY \) to calculate a proper acceleration.

Environment. Figure 25 shows an Environment component. It has two input ports \( aX, aY \) and output ports \( cX, cY, cVX, \) and \( cVY \). Data subcomponents \( x, y, velx, vely, accx, \) and \( accy \) represent the position, velocity and acceleration of the drone. The dynamics of \((x, y)\) is given as continuous functions \( x(t) = x(0) + \text{velx} t + \frac{1}{2} \text{accx} t^2 \) and \( y(t) = y(0) + \text{vely} t + \frac{1}{2} \text{accy} t^2 \) over time \( t \) in Hybrid_SynchAADL::ContinuousDynamics. The dynamics of \((\text{velx}, \text{vely})\) is also given as \( \text{velx}(t) = \text{velx}(0) + \text{accx} t \) and \( \text{vely}(t) = \text{vely}(0) + \text{accy} t \).

Drone Controller. Figure 26 shows a thread component for a drone controller. When the distance between the current position and the connected drone is too close, the new acceleration is set to negation of the current velocity. Otherwise, the new acceleration is set toward the connected drone according to a discretized version of the distributed consensus algorithm. At the output state, the thread component saves the current velocity of the reference drone.

Reference Drone Controller. Figure 27 shows a thread component for a reference drone. It saves its states using data subcomponents \( nx \) and \( ny \) which represents the current acceleration of the reference drone. Eventually, the acceleration of the reference drone is changed into one of \{0, 1, 2\} in each period.
Fig. 23. The top-level system component FourDronesSystem.
system Drone

features
iX: in data port Base_Types::Float; iY: in data port Base_Types::Float;
iVX: in data port Base_Types::Float; iVY: in data port Base_Types::Float;
rX: in data port Base_Types::Float; rY: in data port Base_Types::Float;
rVX: in data port Base_Types::Float; rVY: in data port Base_Types::Float;
oX: out data port Base_Types::Float {Data_Model::Initial_Value => ("0");}
oVX: out data port Base_Types::Float {Data_Model::Initial_Value => ("0");}
oY: out data port Base_Types::Float {Data_Model::Initial_Value => ("0");}
oVY: out data port Base_Types::Float {Data_Model::Initial_Value => ("0");}
end Drone;

system implementation Drone.impl

subcomponents
ctrl: system DroneControl::DroneControl.impl;
env: system Environment::Environment.impl;

connections
C1: port ctrl.oX -> oX; C10: port ctrl.oY -> oY;
C2: port ctrl.oVX -> oVX; C11: port ctrl.oVY -> oVY;
C3: port iX -> ctrl.iX; C12: port iY -> ctrl.iY;
C4: port iVX -> ctrl.iVX; C13: port iVY -> ctrl.iVY;
C5: port ctrl.aX -> env.accX; C14: port ctrl.aY -> env.accY;
C6: port env.cX -> ctrl.cX; C15: port env.cY -> ctrl.cY;
C7: port env.cVX -> ctrl.cVX; C16: port env.cVY -> ctrl.cVY;
C8: port rX -> ctrl.rX; C17: port rY -> ctrl.rY;
C9: port rVX -> ctrl.rVX; C18: port rVY -> ctrl.rVY;

properties
Hybrid_SynchAADL::Sampling_Time => 3 ms .. 5 ms;
Hybrid_SynchAADL::Response_Time => 20 ms .. 30 ms;
end Drone.impl;

Fig. 24. A drone component in HYBRIDSYNCHAADL.
system Environment

features
  cX: out data port Base_Types::Float;  cY: out data port Base_Types::Float;
  cVX: out data port Base_Types::Float;  cVY: out data port Base_Types::Float;
  aX: in data port Base_Types::Float;  aY: in data port Base_Types::Float;

properties
  Hybrid_SynchAADL::isEnvironment => true;
end Environment;

system implementation Environment.impl

subcomponents
  x : data Base_Types::Float {Data_Model::Initial_Value => ("param");};
  y : data Base_Types::Float {Data_Model::Initial_Value => ("param");};
  velx : data Base_Types::Float {Data_Model::Initial_Value => ("0");};
  vely : data Base_Types::Float {Data_Model::Initial_Value => ("0");};
  accx : data Base_Types::Float {Data_Model::Initial_Value => ("0");};
  accy : data Base_Types::Float {Data_Model::Initial_Value => ("0");};

connections
  C1: port velx -> cVX;  C4: port vely -> cVY;
  C2: port x -> cX;  C5: port y -> cY;
  C3: port aX -> accx;  C6: port aY -> accy;

properties
  Hybrid_SynchAADL::ContinuousDynamics =>
  "velx(t) = ((0.001) * accx * t) + velx(0);
  x(t) = (x(0) + (0.001 * velx(0) * t) + ((0.000001) * accx * t * t) / 2);
  vely(t) = ((0.001) * accy * t) + vely(0);
  y(t) = (y(0) + (0.001 * vely(0) * t) + ((0.000001) * accy * t * t) / 2);";
end Environment.impl;

Fig. 25. An environment component in HYBRIDSYNCHAADL.
thread DroneControlThread
features
cX: in data port Base_Types::Float; cY: in data port Base_Types::Float;
cVX: in data port Base_Types::Float; cVY: in data port Base_Types::Float;
iX: in data port Base_Types::Float; iY: in data port Base_Types::Float;
iVX: in data port Base_Types::Float; iVY: in data port Base_Types::Float;
oX: out data port Base_Types::Float; oY: out data port Base_Types::Float;
oVX: out data port Base_Types::Float; oVY: out data port Base_Types::Float;
aX: out data port Base_Types::Float; aY: out data port Base_Types::Float;
rX: in data port Base_Types::Float; rY: in data port Base_Types::Float;
rVX: in data port Base_Types::Float; rVY: in data port Base_Types::Float;
properties Dispatch_Protocol => Periodic;
end DroneControlThread;

thread implementation DroneControlThread.impl
subcomponents
offsetX: data Base_Types::Float;
offsetY: data Base_Types::Float;
rVX0: data Base_Types::Float {Data_Model::Initial_Value => ("0");};
rVY0: data Base_Types::Float {Data_Model::Initial_Value => ("0");};

annex behavior_specification (**
variables
nx, ny, raX, raY : Base_Types::Float;
states
init : initial complete state;
exec, output : state;
transitions
init -[on dispatch]--> exec;
exec -[abs(cX - iX) < 0.3 and abs(cY - iY) < 0.3]--> output{
  aX := -cVX; aY := -cVY;
}
exec -[otherwise]--> output{ 
  raX := (rVX - rVX0);
  nx := raX - #DroneSpec::alpha * 
    (cX - offsetX - rX + #DroneSpec::gamma * (cVX - rVX)) - 
    #DroneSpec::A * (cX - offsetX - iX + #DroneSpec::gamma * (cVX - iVX));
  raY := (rVY - rVY0);
  ny := raY - #DroneSpec::alpha * 
    (cY - offsetY - rY + #DroneSpec::gamma * (cVY - rVY)) - 
    #DroneSpec::A * (cY - offsetY - iY + #DroneSpec::gamma * (cVY - iVY));
  if (nx > 0.5)   aX := 40 
  elsif (nx > 0) aX := 0 
  else
    aX := -40 end if;
  if (ny > 0.5)   aY := 40 
  elsif (ny > 0) aY := 0 
  else
    aY := -40 end if 
};
output -[ ]--> init{ 
  oX := cX - offsetX; oY := cY - offsetY;
oVX := cVX; oVY := cVY;
rVX0 := rVX; rVY0 := rVY 
};
**);
end DroneControlThread.impl;

Fig. 26. A controller thread in HYBRIDSYNCHAADL
thread RefDroneThread
features
  aX: out data port Base_Types::Float; aY: out data port Base_Types::Float;
  oX: out data port Base_Types::Float; oY: out data port Base_Types::Float;
  oVX: out data port Base_Types::Float; oVY: out data port Base_Types::Float;
  cX: in data port Base_Types::Float; cY: in data port Base_Types::Float;
  cVX: in data port Base_Types::Float; cVY: in data port Base_Types::Float;
properties
  Dispatch_Protocol => Periodic;
end RefDroneThread;

thread implementation RefDroneThread.impl
subcomponents
  nx : data Base_Types::Float {Data_Model::Initial_Value => ("0");};
  ny : data Base_Types::Float {Data_Model::Initial_Value => ("0");};

annex behavior_specification {**
  states
    init : initial complete state;
    exec, output : state;
  transitions
    init -[ ]-> exec;
    exec -[ nx = 0 and ny = 0 ]-> output { nx := 1; ny := 1 };
    exec -[ nx = 1 and ny = 1 ]-> output { nx := 2; ny := 2 };
    exec -[ nx = 2 and ny = 2 ]-> output { nx := 0; ny := 0 };
    output -[ ]-> init { aX := nx; aY := ny; oX := cX; oY := cY; oVX := cVX; oVY := cVY };
**};
end RefDroneThread.impl;

Fig. 27. A reference drone controller in HYBRIDSYNCHAADL
Property Specifications. In the formation drone model, consider only one property: "drones do not collide". We analyze them up to bound 500 ms. Compared to the drone rendezvous model, we add the initial constraints of the reference drone in initial. Similar to the drone rendezvous model, there are many initial states satisfying the proposition initial.

| invariant [safety]: ?initial ==? collision in time 500; |
| proposition [collision]: |
| (abs(dr1.env.x - dr2.env.x) < 0.1 and abs(dr1.env.y - dr2.env.y) < 0.1) or |
| (abs(dr1.env.x - dr3.env.x) < 0.1 and abs(dr1.env.y - dr3.env.y) < 0.1) or |
| (abs(dr1.env.x - dr4.env.x) < 0.1 and abs(dr1.env.y - dr4.env.y) < 0.1) or |
| (abs(dr2.env.x - dr3.env.x) < 0.1 and abs(dr2.env.y - dr3.env.y) < 0.1) or |
| (abs(dr2.env.x - dr4.env.x) < 0.1 and abs(dr2.env.y - dr4.env.y) < 0.1) or |
| (abs(dr3.env.x - dr4.env.x) < 0.1 and abs(dr3.env.y - dr4.env.y) < 0.1); |

| proposition [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 and |
| abs(refDr.env.x - 0.0) < 0.01 and abs(refDr.env.y - 0.0) < 0.01; |

As shown in Figure 28, there is a counterexample to safety. The result is obtained by the randomized simulation method. The elapsed CPU time and running time to get the result are 216ms and 782ms respectively.

![Fig. 28. The analysis results of drone formation models](image)

References

1. Supplementary material: HybridSynchAADL technical report, semantics, benchmarks, and the tool [https://hybridsynchaadl.github.io/](https://hybridsynchaadl.github.io/)
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.
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 algo-
48. Tripakis, S., Pinello, C., Benveniste, A., Sangiovanni-Vincent, A., Caspi, P., Di Na-
tale, M.: Implementing synchronous models on loosely time triggered architectures.