# Modeling and Formal Analysis of Virtually Synchronous Cyber-Physical Systems in AADL

Jaehun Lee[1], Kyungmin Bae[1*], Peter Csaba Ölveczky[2], Sharon Kim[3] and Minseok Kang[1]

[1*] Pohang University of Science and Technology, Pohang, South Korea.
[2] University of Oslo, Oslo, Norway.
[3] Korea Shipbuilding & Offshore Engineering, Seoul, South Korea.

*Corresponding author(s). E-mail(s): kmbae@postech.ac.kr;
Contributing authors: thkighie1224@postech.ac.kr; peterol@ifi.uio.no; saron.kim@ksoe.co.kr; masonkang@postech.ac.kr;

**Abstract**

This paper presents the HybridSynchAADL modeling language and formal analysis tool for virtually synchronous cyber-physical systems with complex control programs, continuous behaviors, and bounded clock skews, network delays, and execution times. We leverage the Hybrid PALS methodology, so that it is sufficient to model and verify the much simpler underlying synchronous designs. We define the HybridSynchAADL language as a sublanguage of the avionics modeling standard AADL for modeling such synchronous designs in AADL. We define the formal semantics of HybridSynchAADL using Maude with SMT solving, which allows us to represent advanced control programs and communication features in Maude, while capturing timing uncertainties and continuous behaviors symbolically with SMT solving. We have developed new general methods for optimizing the performance of such symbolic rewriting, which makes the analysis of HybridSynchAADL models feasible. We have integrated the formal modeling and analysis of HybridSynchAADL models into the OSATE tool environment for AADL. Finally, we demonstrate the effectiveness of the Hybrid PALS methodology and HybridSynchAADL on a number of applications, including autonomous drones that collaborate to achieve common goals, and compare the performance of our tool with other state-of-the-art formal tools for hybrid systems.

**Keywords:** cyber-physical systems, virtual synchrony, AADL, formal methods, model checking, Maude, SMT

## 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 [1–3], networked medical devices [4, 5], and other distributed control systems such as the steam-boiler controller benchmark [6].

The underlying infrastructure of such critical systems typically guarantees bounds on clock skews, network delays, and execution times.

Virtually synchronous CPSs are notoriously hard to design—due to race conditions, message buffering, etc.—and to model check, because of the state space explosion caused by asynchronous communication. Motivated by an avionics application developed at Rockwell Collins, the *PALS* ("physically asynchronous, logically synchronous") formal pattern reduces the difficulty of modeling and verifying *distributed real-time* systems to the much easier tasks of modeling and verifying their underlying synchronous designs, as long as the infrastructure provides bounds on network delays, clock skews, and execution times [7–9]. The key point of PALS is that a *synchronous* design *SD*—where all components execute in lockstep and there is no asynchronous message passing, clock skews, or execution times—is stuttering bisimilar to, and therefore satisfies the same properties as, the corresponding *asynchronous* distributed "implementation" $PALS(SD)$. The paper [9] shows the effectiveness of the PALS "synchronizer" on the above avionics case study: the synchronous model had 185 reachable states and could be model checked in less than a second, whereas model checking even a very simplified corresponding asynchronous system—with perfect local clocks, no execution times, and message delays being either 0 or 1—was infeasible.[1]

PALS abstracts from the time when an event takes place, as long as it happens in a certain time interval. However, many virtually synchronous CPSs are networks of *hybrid* components with *continuous* behaviors combined with sophisticated controllers. In such systems, we can no longer abstract from the time when a controller reads a continuous value or sends an actuator command. *Hybrid PALS* [10] extends PALS to such virtually synchronous distributed hybrid systems, taking into account sensing and actuating times that depend on imprecise local clocks.

This paper targets the following main challenges for using the Hybrid PALS methodology to formally model and analyze industrial CPSs with continuous behaviors:

[1] If, in addition, all message delays are 0, then the asynchronous system has more than 3 million reachable states, and its model checking takes more than 30 minutes.

1. To enable formal analysis of (virtually) synchronous[2] industrial hybrid CPSs, the modeling language for such CPSs should be well-known for CPS developers.
2. Providing a formal semantics and efficient formal analysis methods for such a language, with complex discrete control programs interacting with continuous environments at times depending on imprecise local clocks.
3. Both modeling and intuitive automatic formal analysis should be integrated into industrial modeling environments.

To address Challenge (1), we identify a "synchronous" subset of the industrial modeling standard AADL [11] for embedded systems, and define the HybridSynchAADL language for conveniently modeling (virtually) synchronous distributed hybrid systems. HybridSynchAADL is a subset of AADL and its Behavior Annex [12], where the meaning of the constructs in HybridSynchAADL is the same as in AADL, which means that HybridSynchAADL should be an easy and intuitive language for the AADL modeler.

Concerning Challenge (2), providing a formal semantics to such models—with control programs written in AADL's expressive Behavior Annex, with continuous behaviors, and having to cover all possible sampling and actuation times based on imprecise clocks—is challenging. The semantics of control programs can be specified using existing frameworks for defining programming language semantics, but these techniques usually do not consider continuous behaviors. On the other hand, typical semantic frameworks for continuous behaviors, such as hybrid automata, are not good at defining the discrete semantics of conventional programming languages like AADL's Behavior Annex.

In this paper, we use the rewriting-logic-based tool Maude [13] combined with SMT solving [14, 15] to *symbolically* encode continuous behaviors with all possible sampling and actuating times depending on imprecise clocks, and provide a Maude-with-SMT semantics for HybridSynchAADL. The discrete semantics of control

[2] Although we present HybridSynchAADL in the context of Hybrid PALS, our language and tool can more generally be used to model and formally analyze any "synchronous" CPSs with continuous local environments that are sampled/actuated based on imprecise local clocks.

programs is specified using Maude, and the continuous semantics of physical environments is encoded using SMT. Nontrivial interactions between controllers and environments are precisely defined in our Maude-with-SMT semantics.

As usual in symbolic approaches [16], symbolic analysis with Maude-with-SMT quickly encounters the *symbolic state-space explosion* problem. We address this problem in two ways to make symbolic analysis feasible. First, we propose a *modular encoding* to symbolically eliminate the interleavings of different components due to interactions between environments and controllers. Second, our semantics is optimized with a novel state-space reduction technique, adapted from [15], which merges symbolic states to significantly improve the performance of symbolic analysis. This state merging optimization is especially effective for complex programs with many branches and guarded transitions.

Regarding Challenge (3), the HybridSynchAADL tool supports the modeling and formal analysis of HybridSynchAADL models inside the OSATE tool environment for AADL. Our tool provides an intuitive property specification language for specifying bounded reachability properties. HybridSynchAADL invokes Maude combined with the SMT solver Yices2 [17] to provide symbolic reachability analysis, randomized simulation, and multithreaded portfolio analysis for analyzing bounded reachability properties of HybridSynchAADL models with polynomial continuous dynamics.

We use our tool to model and verify a number of CPS applications, including networks of thermostats and of water tanks, as well as distributed drones that communicate to reach the "same" location, or fly in formation, without crashing into each other. We evaluate, and demonstrate, the effectiveness of the HybridSynchAADL tool by addressing the following questions:

- How effective is our tool compared to state-of-the-art formal CPS analysis tools?
- How effective is our portfolio analysis method in finding bugs, compared to randomized simulation and symbolic reachability analysis?
- How effective is our state merging technique?
- How effective is Hybrid PALS in model checking virtually synchronous CPSs?

HybridSynchAADL is one of few, if any, tools—certainly in an AADL context—that can formally analyze the important class of virtually synchronous CPSs with typical CPS features such as complex control programs, continuous behaviors, and arbitrary but bounded communication delays, clock skews, and execution times. This is made possible by:

- Hybrid PALS, which reduces the formal analysis of a virtually synchronous CPS to that of its synchronous design—albeit having to consider clock skews and sensing and actuating times.
- The integration of Maude with SMT solving. Maude is suitable to analyze complex control programs, with user-definable data types, distributed objects, asynchronous communication, and so on, whereas SMT solving allows us to symbolically analyze continuous behaviors and imprecise clocks.

HybridSynchAADL combines these techniques to provide an expressive and user-friendly formal modeling and analysis framework for virtually synchronous CPSs that is optimized to make formal analysis feasible. The HybridSynchAADL tool, the full semantics, and the benchmarks are available at the tool web page https://hybridsynchaadl.github.io.

Apart from providing significantly more detail, this paper extends our conference tool paper [18] as follows:

- It presents the formal semantics of HybridSynchAADL, including discrete/continuous behaviors and modular encoding.
- It describes our symbolic state space merging technique, which makes our symbolic analysis feasible.
- It defines the semantics of the property specification language and the analysis commands.
- It includes additional experiments to evaluate the effectiveness of HybridSynchAADL.

The rest of this paper is organized as follows. Section 2 gives some background on Hybrid PALS, AADL, and Maude with SMT. Section 3 introduces the HybridSynchAADL modeling language. Section 4 explains how HybridSynchAADL components can be symbolically represented as Maude-with-SMT terms. Sections 5 and 6 define the semantics of, respectively, discrete and continuous behaviors of HybridSynchAADL models in Maude-with-SMT. Section 6 also explains the modular encoding to symbolically eliminate the controller-environment interleavings. Section 7 presents the HybridSynchAADL tool and its user interface, its property specification language, and explains how Maude and SMT solving can verify HybridSynchAADL models. As part of this, it also explains our symbolic state merging technique for making such symbolic analysis efficient. Section 8 shows a case study of how virtually synchronous CPSs for controlling distributed drones can be modeled and analyzed using HybridSynchAADL. Section 9 presents the experimental evaluation of our tool, including the comparison with other formal CPS analysis tools. Finally, Section 10 discusses related work, and Section 11 gives some concluding remarks.

## 2 Preliminaries

This section provides some necessary background to Hybrid PALS, AADL, and Maude combined with SMT solving.

### 2.1 Hybrid PALS

A number of "synchronizers for CPSs"—such as TTA [19], PALS [7, 9], and their generalization MSYNC [20]—and similar methods, such as quasi-synchrony [21] and LTTA [22], aim at reducing the model checking complexity, caused by interleaving of the different communicating components, of cyber-physical systems (CPSs) when the underlying infrastructure guarantees bounds on clock skews and network delays.

In this paper we focus on PALS, which has shorter period than TTA, has an extension to the multi-rate setting [23], and is, to the best of our knowledge, the only synchronizer for CPSs that has been extended to CPSs with continuous dynamics. We refer to [20, 24] for a comparison of TTA and PALS, and to [9] for a discussion relating PALS to other kinds of synchronizers.

When the infrastructure guarantees bounds on clock skews, network delays, and execution times, the PALS ("physically asynchronous, logically synchronous") pattern [7–9], reduces the problems of designing and verifying virtually synchronous distributed real-time systems to the much easier problems of designing and verifying their underlying synchronous designs. Formally, given a synchronous system design $SD$, bounds $\Gamma$ on clock skews, network delays, and execution times, and a period $p$ of each round,[3] the PALS transformation generates the asynchronous distributed real-time system $PALS(SD, \Gamma, p)$, which is stuttering bisimilar to $SD$. The simple and easy-to-model-check synchronous design $SD$ therefore satisfies the same CTL$^*$ formulas not involving the "next" operator as the (asynchronous) distributed "implementation" $PALS(SD, \Gamma, p)$.

The synchronous design $SD$ is formalized as the synchronous composition of an *ensemble* of state machines with input and output ports [9]. In each iteration (i.e., at the beginning of each "period"), each machine performs a transition based on its current state and its inputs, proceeds to the next state, and generates outputs. All machines perform their transitions simultaneously, and outputs become inputs at the *next* iteration.

*Hybrid PALS* [10] extends PALS to virtually synchronous CPSs with physical environments that exhibit continuous behaviors. The *physical environment* $E_M$ of a machine $M$ has real-valued parameters $\vec{x} = (x_1, \ldots, x_l)$. The continuous behaviors of $\vec{x}$ are modeled by a set of ordinary differential equations (ODEs) that specify different *trajectories* on $\vec{x}$. $E_M$ also defines *which* trajectory the environment follows, as a function of the last *control command* received by $E_M$.

The local clock of a machine $M$ can be seen as a function $c_M : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, where $c_M(t)$ is the value of the local clock at time $t$, with $\forall t \in \mathbb{R}_{\geq 0},\ |c_M(t) - t| < \epsilon$ for $\epsilon > 0$ the maximal clock skew [9]. In its $i$th iteration, a controller $M$ has received all inputs from the other components at time $c_M(i \cdot p)$, when it starts the execution of its $i$th iteration. The controller $M$ then samples the values of its environment at time $c_M(i \cdot p) + t_s$, where $t_s$ is the *sampling time*. It then executes a transition (based on the sampled values, the values received from other controllers, and the controller's own state). As a result, the new control command is received by the environment at time $c_M(i \cdot p) + t_a$, where $t_a$ is the *actuating time*.

In PALS, the time when an event takes place does not matter, as long as it happens within a certain time interval. However, in hybrid systems, we cannot abstract from the time when

[3] Given performance bounds $\Gamma$, PALS can find the shortest period $p$ that allows all nodes to read the messages in the correct "rounds."

a continuous value is read or an actuator command is given, and therefore those times must be included also in the *synchronous* Hybrid PALS models. Furthermore, since these events are triggered by imprecise local clocks, we must also take into account those clocks in the synchronous models. Although Hybrid PALS cannot abstract away local clocks and timings, it nevertheless abstracts from message buffering and asynchronous communication between controllers, networks delays, execution times, and so on.

We refer to [10] for the formal definitions of Hybrid PALS synchronous and asynchronous models, and for the precise relationship between these models.

## 2.2 AADL

The *Architecture Analysis & Design Language* (AADL) [11] 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. AADL model development is supported by OSATE[4] (Open Source AADL Tool Environment), which provides a modeling environment for AADL as a set of Eclipse plug-ins.

An AADL component is defined by its *type* and its *implementation*. A component type specifies the component's *interface* in terms of *features* (e.g., ports) and *properties* (e.g., periods). A component implementation specifies the component's internal structure as a set of *subcomponents*, a set of *connections* linking their ports, and a set of *modes* that represent operational states of components.

An AADL construct may have *properties* describing its parameters and other information. A property has a name, a type, and a value, and can be assigned a value using a property association declaration (*property* `=>` *value*`;`). The value of a property may depend on the current *mode* of the component (*property* `=>` $value_1$ `in modes` ($mode_1$), ..., $value_k$ `in modes` ($mode_k$)`;`). A number of properties are predefined in AADL. Additional domain-specific properties can be declared using *property sets*.

[4] https://osate.org/

An AADL model describes a system of hardware and software components. This paper focuses on the software components, because we use AADL to specify *synchronous designs*.[5] Software components include *threads* that model the application software to be executed; *process* components defining protected memory that can be accessed by its thread and data subcomponents; and *data* components representing data types. *System* components are the top-level components.

A port is either a *data* port, an *event* port, or an *event data* port. Event ports and event data ports support queuing of, respectively, "events" and message data, while *data* ports only keep the latest data. Furthermore, a port is either an *input* or an *output* port. A connection between two ports can be either *immediate* or *delayed*; the semantics is somewhat subtle (see [11] for details), but an immediate connection roughly means that the source thread sends the event/data as soon as it can, whereas in a delayed connection the source thread sends it at the end of its period.

*Modes* represent the operational states of components. A component can have mode-specific property values, subcomponents, connections, etc. A *mode transition* $m_1$ `-[`$e$`]->` $m_2$ from mode $m_1$ to mode $m_2$ is triggered by the event $e$; that is, by the arrival of an event at port $e$.

Thread behavior is modeled as a guarded transition system with local variables using AADL's *Behavior Annex* [12] standard. Given finite sets of states and local variables, the behavior of a thread is defined by a set of state transitions of the form $s$ `-[`*guard*`]->` $s'$ `{`*actions*`}`, where $s$ and $s'$ are states, and where *guard* is a Boolean condition on the values of the local variables and/or the presence of events or data in the thread's input ports. The *actions* performed when a transition is applied may update the local variables, call methods, generate new outputs, and/or suspend the thread. Actions are built from basic actions using sequencing, conditionals, and finite loops. When a thread is activated, an enabled transition is nondeterministically selected and applied; if the resulting state $s'$ is not a *complete* state, another transition is applied, until a complete state is reached.

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

The *dispatch protocol* of a thread determines when a thread is executed: a *periodic* thread is activated at fixed time intervals (given by its property `Period`), and an *aperiodic* thread is activated when it receives an event.

## 2.3 Maude with SMT

Maude [13] is a rewriting-logic-based executable formal specification language and analysis tool for concurrent systems. In Maude, system states are modeled as elements of algebraic data types, specified using *equational specifications*. Transitions between states are specified using (possibly conditional) *rewrite rules*. Maude provides a range of formal analysis methods, including rewriting for simulation, (explicit-state) search for reachability analysis, LTL model checking, and various forms of theorem proving [25]. Maude recently has been integrated with SMT solving to perform symbolic reachability analysis of infinite-state systems [26].

### 2.3.1 Rewriting Logic and Maude

An (order-sorted) *equational logic signature* $\Sigma$ is a triple $(S, \leq, \Sigma)$, where $S$ is a set (of *sorts*), $\leq$ is a partial order, the *subsort* relation, on $S$, and $\Sigma = \Sigma = \{\Sigma_{w,s}\}_{(w,s)\in S^* \times S}$ an order-sorted signature (i.e., $f \in \Sigma_{s_1\ s_2, s_3}$ denotes a function $f : s_1 \times s_2 \to s_3$). The set $T_{\Sigma,s}$ denotes the set of ground $\Sigma$-terms with sort $s$, and $T_\Sigma(X)_s$ denotes the set of $\Sigma$-terms with sort $s$ over the set $X$ of sorted variables, so that any term of sort $s$ is also a term of sort $s'$ if $s \leq s'$.

An (order-sorted) *equational theory* $(\Sigma, E)$ is a pair $(\Sigma, E)$ with $\Sigma$ a signature and $E$ a finite set of conditional equations of the form

$$(\forall X)\ t = t' \ \textbf{if} \ \bigwedge_i p_i = q_i$$

where $t$ and $t'$ are terms having sorts in the same connected component of $(S, \leq)$. In Maude, an individual equation in the condition may also be a *matching equation* $p_l$ `:=` $q_l$, which is mathematically interpreted as an ordinary equation. However, operationally, the new variables occurring in the term $p_l$ become instantiated by matching the term $p_l$ against the canonical form of the instance of $q_l$ (see [13] for further explanations).

A Maude module specifies a *rewrite theory* [27] of the form $(\Sigma, E \cup A, R)$, where:

1. $(\Sigma, E \cup A)$ is an equational logic theory specifying the system's state space as an algebraic data type with $A$ a set of equational axioms (such as a combination of associativity, commutativity, and identity axioms), to perform equational deduction with the equations $E$ (oriented from left to right) *modulo* the axioms $A$, and
2. $R$ is a set of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form:

$$l : q \longrightarrow r \ \textbf{if} \ \bigwedge_i p_i = q_i \ \wedge \ \bigwedge_j t_j \longrightarrow t_j',$$

where $l$ is a *label*, and $q, r$ are $\Sigma$-terms with sorts in the same connected component. Intuitively, such a rule specifies a *one-step transition* from a substitution instance of $q$ to the corresponding substitution instance of $r$, *provided* the condition holds.[6]

We briefly summarize the syntax of Maude and refer to [13] for more details. Sorts and subsort relations are declared using the keywords `sort` and `subsort`. Operators (or function symbols) are introduced with the `op` keyword:

```
op f : s1 ... sn -> s .
```

where $s_1\ \ldots\ s_n$ are the sorts of its arguments, and $s$ is its *sort*. Operators can have user-definable syntax, with underbars '`_`' marking each of the argument positions, such as in `_+_`. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating that the operator is, respectively, associative, commutative, and has a certain identity element. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. An operator can also be declared to be a *constructor* (`ctor`) that defines the data elements of a sort.

(Unconditional and conditional) *equations* and *rewrite rules*, respectively, are introduced with the following syntax:

```
eq u = v .                ceq u = v if condition .

rl [l]: u => v .          crl [l]: u => v if condition .
```

[6] A rewrite condition $t_j \longrightarrow t_j'$ holds if (a substitution instance of) $t_j'$ is reachable from (the substitution instance of) $t_j$ in zero or more steps.

An equation $f(t_1,\ldots,t_n) = t$ with the `owise` (for "otherwise") attribute can be applied to a term $f(\ldots)$ only if no other equation with left-hand side $f(u_1,\ldots,u_n)$ can be applied. The mathematical variables in rules and equations are either explicitly declared with the keywords `var` and `vars`, or can be introduced on the fly without being declared previously, in which case they have the form *var* : *sort*. Finally, a comment is preceded by '`***`' or '`---`' and lasts till the end of the line.

In *object-oriented* Maude specifications, a *class* declaration `class` $C$ | $att_1 : s_1$, ..., $att_n : s_n$ declares a class $C$ of objects with attributes $att_1$ to $att_n$ of sorts $s_1$ to $s_n$. An *object instance* of class $C$ is represented as a term

$$\texttt{<}\ O : C \ \texttt{|}\ att_1 : v_1,\ \ldots,\ att_n : v_n\ \texttt{>},$$

where $O$, of sort `Oid`, is the object's *identifier*, and where $val_1$ to $val_n$ are the current values of the attributes $att_1$ to $att_n$. A `subclass` inherits the attributes and rewrite rules of its superclasses. A *message* is a term of sort `Msg`. A system state is modeled as a term of the sort `Configuration`, and has the structure of a *multiset* made up of objects and messages, where multiset union is denoted by juxtaposition (empty syntax).

The dynamic behavior of a system is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule (labeled `l`)

```
rl [l] :  < O : C | a1 : 0, a2 : y, a3 : z >
       =>
         < O : C | a1 : z+5, a2 : y, a3 : z > .
```

defines a family of transitions in which an object `O` of class `C` updates the value of its attribute `a1` to `z+5` when it reaches `0`. Attributes whose values do not change and do not affect the next state, such as `a3` and `a2`, need not be mentioned in a rule, so that the above can also be written

```
rl [l] :  < O : C | a1 : 0, a3 : z >
       =>
         < O : C | a1 : z+5 > .
```

*Reachability Analysis in Maude.* Maude provides a number of high-performance analysis methods, including rewriting for simulation purposes, explicit-state reachability analysis, and linear temporal logic (LTL) model checking. In this paper, we use rewriting for randomized simulations and reachability analysis to model check invariant and reachability properties. Given an initial state *init*, a state pattern *pattern* and an (optional) condition *cond*, Maude's `search` command searches the reachable state space from *init* in a breadth-first manner for states that match *pattern* such that *cond* holds:

```
search [bound, depth] init =>* pattern such that cond .
```

*bound* and *depth* provide upper bounds on, respectively, the number of solutions to be found and the number of rewrite steps from *init*. If *bound* or *depth* is unbounded, we write `search [,`*depth*`]` ... and `search [`*bound*`]` ..., respectively.

### 2.3.2 Rewriting Modulo SMT

In rewriting modulo SMT [14, 15], (possibly infinite) sets of system states can be *symbolically* represented using *constrained terms*. A constrained term is a pair $\phi \parallel t$ of a constraint $\phi(x_1,\ldots,x_n)$ and a term $t(x_1,\ldots,x_n)$ over SMT variables $x_1,\ldots,x_n$, representing the set of all instances of the pattern $t$ such that $\phi$ holds: i.e., given the underlying SMT theory $\mathcal{T}$, we have:

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

A *symbolic rewrite* $\phi_t \parallel t \rightsquigarrow^* \phi_u \parallel u$ on constrained terms symbolically represents a (possibly infinite) set of system transitions sequences. For a symbolic rewrite $\phi_t \parallel t \rightsquigarrow^* \phi_u \parallel u$, there exists a "concrete" rewrite $t' \longrightarrow^* u'$ (i.e., $t'$ rewrites to $u'$ in zero or more steps) with $t' \in [\![\phi_t \parallel t]\!]$ and $u' \in [\![\phi_u \parallel u]\!]$, and vice versa for each $t' \longrightarrow^* u'$ with $t' \in [\![\phi_t \parallel t]\!]$. Such symbolic rewrites can be "implemented" using ordinary rewrite rules on constrained terms of the following form [14]:

$$l : \phi \parallel q \longrightarrow \phi' \parallel r \ \textbf{if}\ cond \wedge (\mathcal{T} \models \phi \wedge \phi')$$

In addition to its explicit-state analysis methods for concrete states (ground terms), Maude provides SMT solving and *symbolic reachability analysis* for constrained terms, using connections to Yices2 [17] and CVC4 [28]. Maude supports SMT theories for Booleans, integers, and reals in the SMT-LIB standard [29], and provides the `check` command to invoke the SMT solver. We have implemented a function `check-sat` to check the satisfiability of a given formula using `check`:

```
op check-sat : BoolExp -> Bool .
```

```
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;
```

**Figure 1** The Hybrid_SynchAADL property set.

## 3 The HybridSynchAADL Modeling Language

This section presents the HybridSynchAADL language for modeling virtually synchronous CPSs in AADL. HybridSynchAADL is declared as a sublanguage of AADL extended with the property set Hybrid_SynchAADL shown in Figure 1. We use a subset of AADL without changing the meaning of AADL constructs or adding a new annex—the subset has the same meaning for synchronous designs and distributed implementations—so that AADL experts can easily develop and understand HybridSynchAADL models.

There are two ways of extending the language in AADL: using *property sets* and using *annexes* [11]. A property set can introduce new properties (or annotations) within the standard AADL syntax, whereas an annex allows defining a language extension with new syntax. There are AADL extensions for CPSs using the annex approach, such as [12, 30, 31]. We use the property set approach because annotations are sufficient to define most of the language features of HybridSynchAADL. The only exception is continuous dynamics, which are declared using AADL strings. Nonetheless, our tool can check the syntactic constraints, including continuous dynamics strings (see Section 7.3). This choice of not introducing a new annex is also in the spirit of HybridSynchAADL: it should be as easy as possible to use our language for the AADL modeler; we try to achieve this by minimizing the extension and having each AADL construct have the same meaning in both AADL and HybridSynchAADL.

HybridSynchAADL can specify synchronous designs of distributed controllers (Section 3.1), environments with continuous dynamics (Section 3.2), and interactions between controllers and environments involving imprecise local clocks and sampling and actuating times (Section 3.3). We use a networked thermostat system to illustrate HybridSynchAADL (Section 3.4).

Before we start with controllers and environments, the top-level system component declares the following properties to specify that the entire system is a synchronous design with period $T$:

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

### 3.1 Controller Components

Discrete controllers are usual software components in the Synchronous AADL subset [32, 33]. 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 [12]. The hardware and scheduling features of AADL, which are not relevant to synchronous *designs*, are not considered in HybridSynchAADL.

The execution of a thread is specified by the *dispatch protocol.* A thread with an *event-triggered* dispatch (such as aperiodic dispatch) is dispatched when it receives an event. Since all "controller" components are executed in lock-step in HybridSynchAADL, each thread must have *periodic* dispatch so that it 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;
```

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 => lbound .. ubound;
```

```
Hybrid_SynchAADL::Response_Time => lbound .. ubound;
```

The upper sampling bound must be strictly smaller than the upper actuating bound, and the lower actuating bound must be strictly greater than the lower sampling bound. The upper bounds of both sampling and actuating times must be strictly smaller than the maximal execution time to meet the (Hybrid) PALS constraints [10].

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*, not concrete values. E.g., it can be required that some property must hold for any initial value of a data subcomponent that satisfies certain constraints. Such unknown parameters are declared using the predefined value `param` (see Figure 5 for an example).

## 3.2 Environment Components

An *environment component* models real-valued state variables that continuously change over time. Such real-valued variables are specified using data subcomponents of type `Base_Types::Float`. The values of these data subcomponents change according to their continuous dynamics, declared using AADL constructs with the property set `Hybrid_SynchAADL`. Each environment component declares the component 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 (a data subcomponent). The continuous dynamics in each mode is specified using either ODEs or continuous real functions as follows:

```
Hybrid_SynchAADL::ContinuousDynamics =>
        "continuous_dynamics_1" in modes (mode_1),
        ...
        "continuous_dynamics_n" in modes (mode_n);
```

Modes and mode transitions are declared in the usual AADL way. An actuator command from a controller component through an input port of an environment can trigger a mode transition of the environment, and can change the continuous dynamics of the environment accordingly.

In HybridSynchAADL, a set of ODEs over $n$ variables $x_1, \ldots, x_n$, say, $\frac{\mathrm{d}x_i}{\mathrm{d}t} = e_i(x_1, \ldots, x_n)$ for $i = 1, \ldots, n$, is written as a semicolon-separated string of the following form, where $e_i$ denotes an expression over these $n$ variables and `d/dt(`$x_i$`)` denotes the derivative of variable $x_i$:

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

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

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

Environments can include constant state variables that do not change continuously but can be changed discretely through controller commands. The dynamics of a constant variable, say `x`, can be specified as `d/dt(x) = 0` or `x(`$t$`) = x(0)`, and can be omitted in HybridSynchAADL.

Each 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.*

Consider a data subcomponent $d$ inside an environment component $E$. A connection from $d$ to $E$'s output port $o$ declares that the value of $d$ is "sampled" by a controller through the output port $o$. A connection from $E$'s input port $i$ to $d$ declares that a controller command arrived at the input port $i$ updates the value of $d$. Some input ports of $E$ may receive no value; in this case, the data subcomponent of the environment is unchanged.

## 3.3 Communication

In HybridSynchAADL, connections are limited for synchronous behaviors: no connection is allowed between environments, or between environments and the enclosing system components.

All (non-actuator) outputs of controller components generated in an iteration are available to the receiving *controller* components at the start of

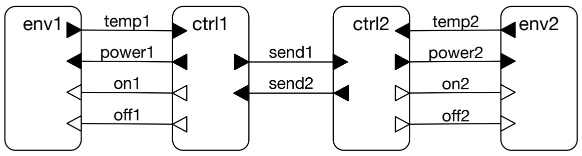


**Figure 2** A networked thermostat system; filled triangles denote data ports and hollow triangles denote event ports.

the *next* iteration. As explained in [32, 33], *delayed connections between data ports* meet this requirement. Controller components can be connected only by data ports with delayed connections, declared by the connection property:

```
Timing => Delayed;
```

Interactions between an environment and a controller occur *instantaneously* at the sampling and actuating times of the controller. Since an environment does not "actively" send data, each output port of an environment must be a *data* port, whereas its input ports could be of any kind.

## 3.4 An Example

Assume that two thermostats control the temperatures in two different rooms. The goal is to maintain similar temperatures in both rooms. For this purpose, as shown in Figure 2, 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 3 shows a top-level system component `TwoThermostats`. This component includes two thermostat controllers and their environments as subcomponents. Each discrete controller `ctrl`$i$ of room $i \in \{1, 2\}$ is connected to its environment `env`$i$ using four connections. The controllers `ctrl1` and `ctrl2` are connected to each other with delayed data connections `send1` and `send2`.

Figure 4 and Figure 5 show a controller component `Thermostat` that turns its heater on or off. It has event output ports `on_ctrl` and `off_ctrl`, data input ports `curr` and `tin`, and data output ports `set_power` and `tout`, where the initial value of `tout` is 0. The implementation has a data subcomponent `avg` whose initial value is declared to be a parameter (`"param"`).

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

```
system TwoThermostats
end TwoThermostats;

system implementation TwoThermostats.impl
  subcomponents
    ctrl1: system Thermostat.impl;
    ctrl2: system Thermostat.impl;
    env1: system RoomEnv.impl;
    env2: system RoomEnv.impl;
  connections
    temp1: port env1.temp -> ctrl1.curr;
    on1: port ctrl1.on_ctrl -> env1.on_ctrl;
    power1: port ctrl1.set_power -> env1.power;
    off1: port ctrl1.off_ctrl -> env1.off_ctrl;
    temp2: port env2.temp -> ctrl2.curr;
    on2: port ctrl2.on_ctrl -> env2.on_ctrl;
    power2: port ctrl2.set_power -> env2.power;
    off2: port ctrl2.off_ctrl -> env2.off_ctrl;
    send1: port ctrl1.tout -> ctrl2.tin;
    send2: port ctrl2.tout -> ctrl1.tin;
  properties
    Hybrid_SynchAADL::Synchronous => true;
    Period => 10 ms;
    Timing => Delayed applies to send1, send2;
end TwoThermostats.impl;
```

**Figure 3** A top level component.

`tin`, and assigns to the output port `tout` the value of `curr`. Because `exec` is not a complete state, the thread continues executing by taking one of the other transitions. E.g., if `avg` is smaller than 10, a command that sets the heater's power to 10 is sent through the port `set_power`, and an event is sent through the event output port `on_ctrl`. The resulting state `init` is a complete state, and the execution of the current dispatch ends.

Figure 6 shows an environment component. It has data output port `temp`, data input port `power`, and event input ports `on_ctrl` and `off_ctrl`. The implementation has two data subcomponents `x` and `p`, denoting the temperature of the room and the heater's power, respectively, to represent the state variables with the specified initial values.

There are two modes `heaterOn` and `heaterOff` with their respective continuous dynamics, using continuous functions over time parameter $t$, where `heaterOff` is the initial mode. The value of the variable `x` change continuously according to the mode and the continuous dynamics. On the other hand, `p` is a constant state variable, and its dynamics `d/dt(p) = 0` is omitted.

The value of `x` is sent to the controller through the output port `temp`, using the connection `x -> temp`. When a discrete controller sends

```
system Thermostat
  features
    on_ctrl: out event port;
    off_ctrl: 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"); };
  properties
    Hybrid_SynchAADL::Max_Clock_Deviation => 0.3ms;
    Hybrid_SynchAADL::Sampling_Time => 1ms .. 5ms;
    Hybrid_SynchAADL::Response_Time => 7ms .. 9ms;
end Thermostat;

system implementation Thermostat.impl
  subcomponents
    ctrlProc: process ThermostatProc.impl;
  connections
    O1: port ctrlProc.on_ctrl -> on_ctrl;
    O2: port ctrlProc.off_ctrl -> off_ctrl;
    O3: port ctrlProc.set_power -> set_power;
    O4: port ctrlProc.tout -> tout;
    I1: port tin -> ctrlProc.tin;
    I2: port curr -> ctrlProc.curr;
end Thermostat.impl;

process ThermostatProc
  features
    on_ctrl: out event port;
    off_ctrl: 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 ThermostatProc;

process implementation ThermostatProc.impl
  subcomponents
    ctrlThread: thread ThermostatThread.impl;
  connections
    O1: port ctrlThread.on_ctrl -> on_ctrl;
    O2: port ctrlThread.off_ctrl -> off_ctrl;
    O3: port ctrlThread.set_power -> set_power;
    O4: port ctrlThread.tout -> tout;
    I1: port tin -> ctrlThread.tin;
    I2: port curr -> ctrlThread.curr;
end ThermostatProc.impl;
```

**Figure 4** A `Thermostat` component.

an actuation command through the input ports `power`, `on_ctrl`, and `off_ctrl`, the mode changes according to the mode transitions, and the value of `p` can be updated with the value of the input port `power`, declared by `port power -> p`.

```
thread ThermostatThread
  features
    on_ctrl: out event port;
    off_ctrl: 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;
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_ctrl! };
      exec -[avg < 20 and avg >= 10]-> init
        { set_power := 5; on_ctrl! };
      exec -[avg < 10]-> init
        { set_power := 10; on_ctrl! };
  **};
end ThermostatThread.impl;
```

**Figure 5** A simple thermostat controller thread.

## 4 Maude Representation of HybridSynchAADL Models

Section 3 introduces the HybridSynchAADL modeling language and describes the meaning of this language informally using English prose. To formally analyze HybridSynchAADL models, such models must have a precisely defined mathematical meaning. This and the following two sections therefore define such a formal semantics for HybridSynchAADL using object-oriented rewriting modulo SMT.

In particular, this section explains how a HybridSynchAADL model, i.e., HybridSynchAADL components, are symbolically represented as Maude terms with SMT constraints. Each component in the HybridSynchAADL model is represented as an object; since HybridSynchAADL models have a hierarchical structure, our semantics is based on hierarchical objects, where an attribute of an object may contain other objects.

```
system RoomEnv
  features
    temp: out data port Base_Types::Float;
    power: in data port Base_Types::Float;
    on_ctrl: in event port;
    off_ctrl: 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_ctrl]-> heaterOn;
    heaterOn -[off_ctrl]-> 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;
```

**Figure 6** A RoomEnv component.

## 4.1 Components

Each HybridSynchAADL component is represented as an instance of a subclass of the following base class Component. The attribute features denotes a set of Port objects representing the ports of the component; subcomponents denotes a set of Component objects, representing the component's subcomponents; connections denotes its connections; and properties denotes its properties:

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

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

```
class Ensemble .
subclass Ensemble < Component .

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

Thread components are represented as object instances of the following class Thread, which adds attributes for representing the transition system that defines the thread's behaviors using AADL's Behavior Annex. The attribute transitions denotes the set of transitions; currState denotes the current state of the transition system; completeStates denotes the complete states; and variables denotes the local (temporary) variables and their types:

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

A thread transition is represented as a term $s$ -[*guard*]-> $s'$ {*actions*}, where $s$ is a source state, $s'$ is a destination state, *guard* is a guard condition, and {*actions*} is an action block:

```
sort Transition .
op _-[_]->__ : Location TransGuard Location
               ActionBlock -> Transition [ctor] .
```

Environment components are represented as object instances of the class Env. The attribute currMode denotes the current mode; flows represents the continuous dynamics in each mode, as explained in Section 3.2; jumps denotes the mode transitions; and sampling and response denote the sets of sampling and actuating times, respectively, for the controllers of the environment:

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

A mode transition is represented as a term $m$ -[*triggers*]-> $m'$ of sort EnvJump, where $m$ is a source mode, $m'$ is a target mode, and *triggers* is a set of port names:

```
sort EnvJump .
op _-[_]->_ : Location NeSet{FeatureRef} Location
              -> EnvJump [ctor] .
```

The continuous dynamics for mode $m$ is represented as a term $m$ [*dynamics*] of sort EnvJump, where *dynamics* is either ODEs or continuous functions as explained in Section 3:

```
sorts AADLDiffEq AADLEnvFunc .
op d/dt[_]=_ : ComponentRef AADLExp
                -> AADLDiffEq [ctor] .
op _(_)=_ : ComponentRef VarId AADLExp
             -> AADLEnvFunc [ctor] .

sorts EnvFlow FlowItem .
op __ : Location FlowItem -> EnvFlow [ctor] .
op [_] : NeSet{AADLDiffEq} -> FlowItem [ctor] .
op [_] : NeSet{AADLEnvFunc} -> FlowItem [ctor] .
```

A sampling/actuating time is represented as a term $o : (l, u)$ of sort `InterTiming`, where $o$ is an object identifier of a controller, $l$ is a lower time bound, and $u$ is an upper time bound:

```
sort InterTiming .
op _:(_,_) : ComponentId Rat Rat
             -> InterTiming [ctor] .
```

Data components, which specify the state variables of threads and environments, are represented as instances of the following class `Data`, where the attribute `value` denotes the current value (sort `DataContent` is explained in Section 4.3):

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

## 4.2 Ports and Connections

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

```
class Port | content : DataContent,
             properties : PropertyAssociation .

class InPort .
class OutPort .
subclass InPort OutPort < Port .
```

We distinguish between the ports of controllers and the ports of environments. As mentioned in Section 3.3, the communication between controllers is *delayed* (i.e., outputs produced in one iteration are available at the destination in the next iteration), whereas the communication between a controller and an environment happens in the same iteration according to the sampling and actuating times.

Controller ports are represented as instances of the `DataPort` class. An input data port contains the extra attribute `cache` to keep the previously received value; if an input port $p$ has received "no value" $\bot$ in the latest dispatch, the thread can use the value in the `cache`, while the behavior annex expression $p$`'fresh` becomes *false*:

```
class DataPort .
subclass DataPort < Port .

class DataInPort | cache : DataContent .
subclass DataInPort  < InPort  DataPort .

class DataOutPort .
subclass DataOutPort < OutPort DataPort .
```

Similarly, environment ports are represented as instances of the `EnvPort` class. An environment port keeps the identifier of the `target` controller. The attribute `envCache` contains the data content in the previous iteration to symbolically encode the immediate communication (see Section 6.1):

```
class EnvPort | target : CompRef,
                envCache : DataContent .
subclass EnvPort < Port .

class EnvInPort .
subclass EnvInPort  < EnvPort InPort .

class EnvOutPort .
subclass EnvOutPort < EnvPort OutPort .

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

Connections are represented as terms of the form $p_i$ `-->` $p_o$, with $p_i$ the source port name and $p_o$ the target port name. The name of a port $p$ in a subcomponent $c$ is represented as a term $c$ `..` $p$. A connection from $c_1$'s output port $p_1$ to $c_2$'s input port $p_2$ is represented as $c_1$ `..` $p_1$ `-->` $c_2$ `..` $p_2$. A level-up (resp., level-down) connection from $c$'s port $p$ to a port $p'$ of the "current" component is written as $c$ `..` $p$ `-->` $p'$ (resp., $p'$ `-->` $c$ `..` $p$):

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

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

We use slightly different representations for connections between data subcomponents and ports inside environments. A connection from a data subcomponent $d$ to an output port $p$ (for sampling) is represented as a term $d$ `==>` $p$, and a connection from an input port $p$ to a data subcomponent $d$ (for updating data) is written $p$ `=>>` $d$:

```
sort EnvConnection .
subsort EnvConnection < Connection .
op _==>_ : ComponentId FeatureId
        -> EnvConnection [ctor] .
op _=>>_ : FeatureId ComponentId
        -> EnvConnection [ctor] .
```

For example, an instance of `TwoThermostats` in Figure 3 is represented as the following object (where some parts are replaced by '...'):

```
< TwoThermostatsInstance : System |
  features : none,
  subcomponents : < ctrl1 : System | ... >
                  < ctrl2 : System | ... >
                  < env1 : Env | ... >
                  < env2 : Env | ... >,
  connections :
    ctrl1 .. onctrl --> env1 .. onctrl ;
    ctrl1 .. offctrl --> env1 .. offctrl ;
    ctrl1 .. setpower --> env1 .. power ;
    env1 .. temp --> ctrl1 .. curr ;
    ctrl1 .. tou --> ctrl2 .. tin ;
    ctrl2 .. onctrl --> env2 .. onctrl ;
    ctrl2 .. offctrl --> env2 .. offctrl ;
    ctrl2 .. setpower --> env2 .. power ;
    env2 .. temp --> ctrl2 .. curr ;
    ctrl2 .. tou --> ctrl1 .. tin,
  properties :
    TimingProperties::Period => 10 ;
    HybridSynchAADL::Synchronous => true >
```

## 4.3 Constrained Objects

*SMT expressions* are declared as Maude terms of sort `Exp`. They are constructed by constants, variables, and the usual SMT operators:

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

sort BoolExp .
subsorts BoolValue SMTBoolVar < BoolExp < Exp .
op not_ : BoolExp -> BoolExp .
op _and_ : BoolExp BoolExp -> BoolExp [assoc comm] .
op _or_ : BoolExp BoolExp -> BoolExp [assoc comm] .
...

sort RealExp.
subsorts RealValue SMTRealVar < RealExp < Exp .
op -_ : RealExp -> RealExp .
op _+_ : RealExp RealExp -> RealExp [assoc comm] .
op _*_ : RealExp RealExp -> RealExp [assoc comm] .
...

sort UnitExp .
subsorts UnitValue < UnitExp < Exp .
op _===_ : UnitExp UnitExp -> BoolExp .
...
```

Constants include Boolean values, rational numbers (of sort `RealValue`), and a unit value $*$ denoting the presence of an event:

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

Variable terms, of sort `SMTVar`, include Boolean variables of the form $b(id)$, and real variables of the form $r(id)$, where $id$ is a natural number:

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

To symbolically represent a (possibly infinite) set of objects, we use a *constrained object* of the form $\phi(x_1, \ldots, x_n) \parallel obj(x_1, \ldots, x_n)$, where $\phi$ is an SMT formula and $obj$ is an object "pattern" over SMT variables $x_1, \ldots, x_n$. Object patterns can be hierarchical, because `subcomponents` may include object patterns. Likewise, a *constrained configuration* is a term of the form $\phi \parallel conf$, where $conf$ is a multiset of object patterns. A multiset of constrained objects is equivalent to a constrained configuration where all object constraints are conjuncted (see Section 5.1):

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

subsorts ConstObject Configuration < ConstConfig .
op _||_ : BoolExp Configuration -> ConstConfig .
op __ : ConstConfig ConstConfig -> ConstConfig .
```

We symbolically represent data contents using SMT expressions. In HybridSynchAADL, a data content for a port or a data component can be either no value (i.e., some "don't care" value $\bot$) or a (Boolean or real) value. A data content is represented as a pair $e$ `#` $b$ of an SMT expression $e$ and a Boolean condition $b$. If $b$ is *false*, then the data content is $\bot$; if $b$ is *true*, then the data content is a value represented by the expression $e$:

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

### 4.4 An Example

The following shows a constrained object representing an instance of the thread component `ThermostatThread` in Figure 5, where `avg` has a symbolic data content `r(0) # b(0)`. This constrained object includes both cases where the content of `avg` is present (`b(0) = true`) and absent (`b(0) = false`). If the content of `avg` is present, its value `r(0)` must be greater than 0, by the constraint `not b(0) or r(0) > 0`.

```
not b(0) or r(0) > 0  ||
< ctrlThread : Thread |
  features :
    < onctrl : DataOutPort |
      content : * # false,  property : none >
    < offctrl : DataOutPort |
      content : * # false,  property : none >
    < setpower : DataOutPort |
      content : 0 # false,  property : none >
    < curr : DataInPort |
      content : 0 # false,  cache : 0 # false,
      property : none >
    < tin  : DataInPort |
      content : 0 # false,  cache : 0 # false,
      property : none >
    < tou : DataOutPort |
      content : 0 # true,   property : none >,
  subcomponents :
    < avg : Data | value : r(0) # b(0), ... >,
  connections : empty,
  properties :
    TimingProperties::Period => 10 ;
    HybridSynchAADL::SamplingTime => 1.0 .. 5.0 ;
    HybridSynchAADL::ResponseTime => 7.0 .. 9.0 ;
    HybridSynchAADL::Synchronous => true,
  variables : empty,
  transitions :
    init -[on dispatch]-> exec
      {avg := (tin + curr) / 2 ; tou := curr} ;
    exec -[avg > 25]-> init {offctrl !} ;
    exec -[(avg < 20) and (avg > 10)]-> init
      {setpower := 5 ; onctrl !} ;
    exec -[avg <= 10]-> init
      {setpower := 10 ; onctrl !} ;
    exec -[otherwise]-> init {skip},
  currState : init,    completeStates : init >
```

The following shows a constrained object for an instance of the environment component `RoomEnv` in Figure 6; the ports include symbolic variables, such as `r(0)`, `r(1)`, `b(0)`, `b(1)`, etc. When the `contents` of output port `temp` and input port `power` are present (`b(2) and b(3)`), the value of `temp` is greater than 0 but less than the value of `power` (`r(1) > 0 and r(1) < r(0)`). For input ports `offctrl` and `onctrl`, the `envCache` can be either present or absent, because `b(0)` and `b(1)` are unconstrained.

```
not(b(2) and b(3)) or (r(1) > 0 and r(1) < r(0)) ||
< env1 : Env |
  features :
    < temp : EnvOutPort |
      content : r(1) # b(3),  target : ctrl1,
      envCache : r(1) # b(3), property : none >
    < offctrl : EnvInPort |
      content : * # false,    target : ctrl1,
      envCache : * # b(0),    property : none >
    < onctrl : EnvInPort |
      content : * # false,    target : ctrl1,
      envCache : * # b(1),    property : none >
    < power : EnvInPort |
      content : r(0) # b(2),  target : ctrl1,
      envCache : 0 # false,   property : none >,
  subcomponents :
    < p : Data | value : 5 # true, ... >
    < x : Data | value : 15 # true, ... >,
  connections : x ==> temp ; power =>> p,
  properties :
    Hybrid_SynchAADL::isEnvironment => true;
    TimingProperties::Period => 10;
    HybridSynchAADL::Synchronous => true,
  currMode : toff,
  jumps : ton -[offctrl]-> toff ;
          toff -[onctrl]-> ton,
  flows : ton [x(t)= x - (0.1 *(x - p / 0.1)* t)] ;
          toff [x(t)= x * (1.0 - (0.1 * t))],
  sampling : ctrl1 : (1,5),
  response : ctrl1 : (7,9) >
```

## 5 Symbolic Semantics of Discrete Controllers

This section presents the formal semantics for the discrete subset of HybridSynchAADL.

Since the HybridSynchAADL modeling language extends Synchronous AADL, the semantics of discrete controllers extends that of Synchronous AADL [32, 33], formalized in Maude using "concrete" rewriting with ground terms. In contrast, our Maude-with-SMT semantics is formalized using symbolic rewriting with constrained terms.

### 5.1 Ensemble Behavior

Our semantics defines various *semantic operations* on constrained terms to specify the behavior of controllers, environments, communications, etc. In particular, the semantic operation `executeStep` defines a *symbolic rewrite* relation for a "big-step" synchronous iteration of a single component $obj$: i.e., `executeStep`$(\phi \parallel obj) \rightsquigarrow^{*} \phi' \parallel obj'$.

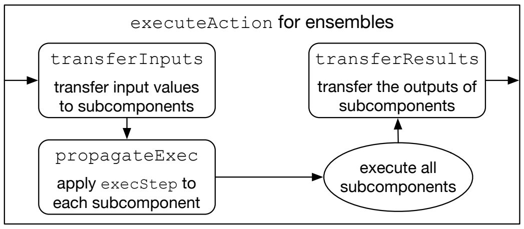

executeAction for ensembles
transferInputs
transfer input values to subcomponents
transferResults
transfer the outputs of subcomponents
propagateExec
apply execStep to each subcomponent
execute all subcomponents

**Figure 7** The structure of execAction for ensembles.

```
op executeStep : ConstObject ~> ConstObject .
```

Semantic operations, including executeStep, are declared to be partial functions (arrow ~>). Since a term containing partial operations does not have a sort, we can ensure that equations and rules for semantic operations are only applied to an object of sort Object in which all subcomponents have already finished their semantic operations.

A (symbolic) synchronous step of the entire system—a top-level system component with no ports—is then formalized by the following rule:

```
vars OBJ OBJ' : Object .  vars C C' : ComponentId .
vars COMPS COMPS' : Configuration .
vars PHI PHI' : BoolExp .

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

A symbolic rewrite $\{\phi \parallel obj\} \rightsquigarrow \{\phi' \parallel obj'\}$ therefore holds by the above rule if there is a symbolic rewrite executeStep($\phi \parallel obj$) $\rightsquigarrow^* \phi' \parallel obj'$, provided that *obj* has no ports.

Figure 7 visualizes the behavior of ensemble components, such as systems and processes, which is formalized using executeStep as follows:

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

In this rule, each input port of the subcomponents gets a value from its source by transferInputs. Then, the operation executeStep is applied to each subcomponent by propagateExec. Any term of sort Object obtained by rewriting propagateExec(PHI, OBJ) is nondeterministically assigned to OBJ' with a new constraint PHI'. The function check-sat invokes an SMT solver

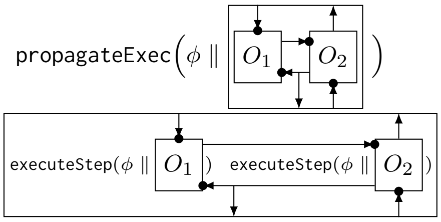


**Figure 8** The propagateExec function, where $O_1$ and $O_2$ indicate subcomponents, arrows indicate connections, and bullets indicate values of input/output ports.

to check the satisfiability of the constraint PHI'. Finally, the outputs of the subcomponents are transferred by transferResults.

For a constrained ensemble object $\phi \parallel ensem$, propagateExec applies the operation executeStep to each subcomponent *obj* of *ensem* constrained by $\phi$, as illustrated in Figure 8.

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

Each term executeStep($\phi \parallel obj$) can then be individually executed, and we use the following equations to obtain a constrained configuration where all resulting constraints are combined.

```
eq (PHI || COMPS) (PHI' || COMPS')
 = (PHI and PHI') || COMPS COMPS' .
eq (PHI || COMPS) OBJ = PHI || (COMPS OBJ) .
```

### 5.1.1 Transferring Data

We model transferring data using two types of messages: transIn messages, delivered to input ports of subcomponents, and transOut messages, delivered to output ports of an ensemble:

```
op transIn : DataContent FeatureRef -> Msg [ctor] .
op transOut : DataContent FeatureRef -> Msg [ctor] .
```

The following equations formalize the message passing behavior of these messages:

```
vars PN PN' : FeatureRef .  vars P P' : FeatureId .
vars D D' : DataContent . vars B B' B'' : BoolExp .
var CONXS : Set{Connection} .  vars E E' : Exp .
vars PORTS PORTS' : Configuration .

eq < C : Ensemble | features : PORTS transIn(D,PN),
```

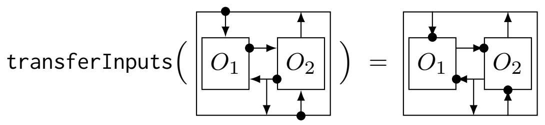


**Figure 9** The `transferInputs` function.

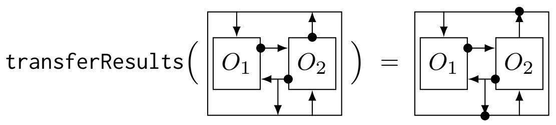


**Figure 10** The `transferResults` function.

```
                  subcomponents : COMPS >
 = < C : Ensemble | features : PORTS,
          subcomponents : transIn(D,PN) PORTS > .

eq transIn(D, C .. P)
   < C : Component | features :
             < P : InPort | content : D' > PORTS >
 = < C : Component | features :
             < P : InPort | content : D > PORTS > .

eq < C : Ensemble | features : PORTS,
          subcomponents : transOut(D,PN) COMPS >
= < C : Ensemble | features : PORTS transOut(D,PN),
          subcomponents : COMPS > .

eq transOut(D, P) < P : OutPort | content : D' >
 = < P : OutPort | content :  D > .
```

The function `transferInputs` moves data from the input ports of an ensemble or the output ports of its subcomponents into their connected input ports of the subcomponents, as depicted in Figure 9. This function generates `transIn` messages using two functions `transEnsIn` and `transFBOut`:

```
eq transferInputs(< C : Ensemble |
        features : PORTS, connections : CONXS,
        subcomponents : COMPS >)
= < C : Ensemble |
       features : transEnsIn(CONXS, PORTS),
       subcomponents : transFBOut(CONXS, COMPS) > .
```

For each input port `P` connected to an input port of a subcomponent, `transEnsIn` creates a `transIn` message with its current data content. The function `trInAux` is defined to deal with "fan-out" connections in which a single source port is connected to several target ports:

```
eq transEnsIn((P --> C'..P') ; CONXS,
     < P : InPort | content : E # B > PORTS)
 = trInAux(E # B, P, (P --> C'..P') ; CONXS)
   transEnsIn(remove(P, CONXS),
     < P : InPort | content : E # false > PORTS ) .
eq transEnsIn(CONXS, PORTS) = PORTS [owise] .

eq trInAux(D, PN, (PN --> C'..P') ; CONXS)
 = transIn(D, C'..P') trInAux(D, PN, CONXS) .
eq trInAux(D, PN, CONXS) = none [owise] .

eq remove(PN, (PN --> PN') ; CONXS)
 = remove(PN, CONXS) .
eq remove(PN, CONXS) = CONXS [owise] .
```

For each output port `P` of a subcomponent connected to an input port of another subcomponent, `transFBOut` produces a `transIn` message:

```
eq transFBOut((C..P --> C'..P') ; CONXS,
     < C : Component | features : < P : OutPort |
                   content : E # B > PORTS > COMPS)
 = trInAux(E # B, C..P, C..P --> C'..P' ; CONXS)
   transFBOut(remove(C..P, CONXS),
     < C : Component | features : < P : OutPort |
              content : E # false > PORTS > COMPS) .
eq transFBOut(CONXS, COMPS) = COMPS [owise] .
```

The function `transferResults` transfers data from the output ports of the subcomponents to their connected output ports of *obj*; if such an output port is also connected to another subcomponent, it keeps the data for the output in the next step, as illustrated in Figure 10. As in the `transferInputs` case, this function generates `transOut` messages as follows:

```
eq transferResults(< C : Ensemble |
      subcomponents : COMPS, connections : CONXS >)
 = < C : Ensemble |
      subcomponents : transEnsOut(CONXS, COMPS) > .

eq transEnsOut((C..P --> P') ; CONXS,
     < C : Component | features : < P : OutPort |
                   content : D > PORTS > COMPS)
 = trOutAux(D, C..P,  (C..P --> P') ; CONXS)
    transEnsOut(remove(C..P, CONXS), COMPS
      < C : Component | features :
          < P : OutPort | content :
              fbdata(D, C..P, CONXS) > PORTS >) .
eq transEnsOut(CONXS, COMPS) = COMPS [owise] .

eq trOutAux(D, C..P, (C..P --> P') ; CONXS)
 = transOut(D, P')  trOutAux(D, C..P, CONXS) .
eq trOutAux(D, C..P, CONXS) = none [owise] .

eq fbdata(E # B, C..P, C..P --> C'..P' ; CONXS)
 = E # B .
eq fbdata(E # B, C..P, CONXS) = E # false [owise] .
```

## 5.2 Thread Behavior

Figure 11 visualizes the behavior of thread components, which is defined by the following rule:

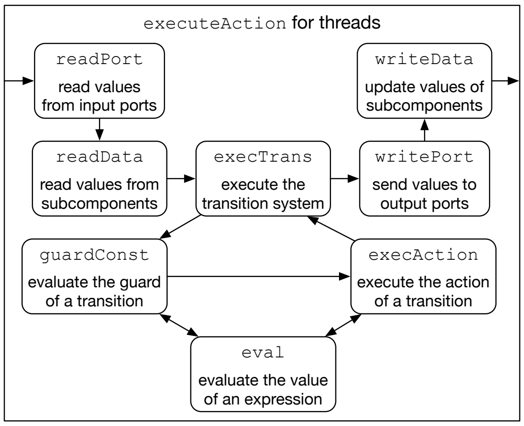

executeAction for threads
readPort
read values from input ports
writeData
update values of subcomponents
readData
read values from subcomponents
execTrans
execute the transition system
writePort
send values to output ports
guardConst
evaluate the guard of a transition
execAction
execute the action of a transition
eval
evaluate the value of an expression

**Figure 11** The structure of execAction for threads.

```
var LS : Set{Location} .    vars L L' : Location .
var VIS : Map{VarId,DataType} .
vars PRS PRS' : PropertyAssociation .
vars TRS TRS' : Set{Transition} .
vars FMAP FMAP' : FeatureMap .
vars DATA DATA' : DataValuation .

crl executeStep(
    PHI  || < C : Thread |      features : PORTS,
        subcomponents : COMPS,  properties : PRS,
        transitions : TRS,      currState : L,
        completeStates : LS,    variables : VIS >)
 =>
    PHI' || < C : Thread |
        features : writePort(FMAP',PORTS'),
        subcomponents : writeData(DATA',COMPS),
        currState : L' >
if {PORTS',FMAP} := readPort(PORTS)
/\ DATA := readData(COMPS)
/\ execTrans(emptyVal(VIS), FMAP, DATA, PRS, TRS,
       L, LS, PHI) => L' | FMAP' | DATA' | PHI' .
```

The function readPort returns a map from each input port to its content; readData returns a map from each data subcomponent to its value; execTrans obtains a computation result of the transition system; writePort updates the content of each output port; and writeData updates the value of each data subcomponent.

### 5.2.1 Port and Data Operations

Given a set of controller ports, readPort builds a map from port identifiers to their data contents, removes the content from each input port, and returns the resulting ports and the feature map. Since a Behavior Annex expression $p$'fresh returns true if the content of input port $p$ is "fresh", a feature map item for $p$ is a pair $d : f$ of content $d$ and freshness flag $f$, where $d$ is also a pair $e \# b$ with $b$ indicating the presence of the content.

The following equations define the function readPort. First, each output port P is related to E # false (which indicates $\bot$):

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

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

Consider an input port P with content E # B and cached content E' # B'. If the content is present (B is *true*), then P is related to the feature map item (E # B) : true; otherwise, P is related to (E' # B') : false using the cache value. This can be written compactly using the conditional operator ?. The content of P is set to absent, and the cache attribute is updated with D:

```
ceq readPort(< P : DataInPort | content : E # B,
        cache : E' # B' > PORTS, PORTS', FMAP)
 = readPort(PORTS, < P : DataInPort |
        content : E # false, cache : D > PORTS',
      insert(P, D : B, FMAP))
if D := (B ? E : E') # (B or B') .
```

The function writePort replaces the content of each output port by the content in the map FMAP:

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

For data components, the function readData builds a map from each identifier to its value:

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

Similarly, writeData updates the values of the data subcomponents using a given map:

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

```
eq writeData(DATA, COMPS, COMPS')
 = COMPS COMPS' [owise] .
```

### 5.2.2 Executing Transitions

The following rule trans defines the behavior of the operation execTrans. A transition from the current state L is nondeterministically chosen from the set TRS of transitions. The operation execAction executes the actions ACT of the chosen transition, provided that the guard condition GC evaluates to *true* (the constraint B) and the new constraint PHI' is satisfiable. If the next state L' is a complete state (L' in LS), the operation ends; otherwise, execTrans is applied again:

```
vars VAL VAL' : VarValuation . var GC : TransGuard .
var ACT : ActionBlock .    vars EXP EXP' : AADLExp .

crl [trans]: execTrans(VAL, FMAP, DATA, PRS,
                       TRS, L, LS,  PHI)
=> if L' in LS then
      L' | FMAP' | DATA' | PHI'
   else execTrans(VAL, FMAP', DATA',
                  PRS, TRS, L', LS,  PHI') fi
if (L -[GC]-> L' ACT) ; TRS' := TRS
/\ B := guardConst(GC,L,TRS',VAL,FMAP,DATA,PRS)
/\ execAction(ACT, VAL, FMAP, DATA, PRS, PHI and B)
   => VAL' | FMAP' | DATA' | PHI'
/\ check-sat(PHI') .
```

The function guardConst is defined as follows. An on dispatch guard always evaluates to *true*. A constraint for a Boolean guard EXP is obtained using the eval operation. An otherwise guard evaluates to *true* if for all transitions from the current state L (obtained by outTrs), their guard conditions evaluate to *false*, where there is no other otherwise guard (by noOwise):

```
eq guardConst(on dispatch, L, TRS, VAL, FMAP, DATA,
   PRS) = true .
eq guardConst(EXP, L, TRS, VAL, FMAP, DATA, PRS)
 = eval(EXP, VAL, FMAP, DATA, PRS) .
ceq guardConst(otherwise, L, TRS, VAL, FMAP, DATA,
  PRS) = allGCfalse(TRS', VAL, FMAP, DATA, PRS)
 if TRS' := outTrs(L, TRS) /\ noOwise(TRS') .

eq allGCfalse((L -[EXP]-> L' ACT) ; TRS, VAL, FMAP,
  DATA, PRS) = not(eval(EXP, VAL, FMAP, DATA, PRS))
  and allGCfalse(TRS, VAL, FMAP, DATA, PRS) .
eq allGCfalse(empty, VAL, FMAP, DATA, PRS) = true .
```

### 5.2.3 Evaluating Expressions

The function eval evaluates an AADL behavior expression and returns the resulting data content, given local variable values VAL, port values FMAP, state variable values DATA, and property values PRS. When eval returns a data content $e$ # $b$, the second item $b$ indicates that every identifier in the input expression has a value (i.e., not $\bot$).

The case of AADL values V are defined by the following equation, where the second item is always true. We only consider Boolean values, integers, and floating point numbers:

```
var VALUE : AADLValue .   var VI : VarId .
var PR : PropertyId .

eq eval(VALUE, VAL, FMAP, DATA, PRS)
 = VALUE # true .
```

The following equations define the cases for identifiers: local variable VI, port identifier P, state variable C, and property name PR:

```
eq eval(VI, VAL, FMAP, DATA, PRS) = VAL[VI] .
eq eval(P, VAL, FMAP, DATA, PRS) = data(FMAP[P]) .
eq eval(C,  VAL, FMAP, DATA, PRS) = DATA[C] .
eq eval(PR, VAL, FMAP, DATA, PRS) = value(PRS[PR]) .
```

The following equation defines the case of a fresh expression. As mentioned, a feature map item has the form $(e \# b)$ : $f$, where $b$ denotes the presence of the content and $f$ denotes the freshness. Thus, eval returns $f \# b$ in this case:

```
ceq eval(fresh(P), VAL, FMAP, DATA, PRS)
  = B # B'  if E # B' : B := FMAP[P] .
```

The cases for the other expressions are defined by propagating eval to the subexpressions. For example, the semantics of an addition expression is defined as follows (the second equation indicates that the value of the entire expression is $\bot$ if the value of any subexpression is $\bot$):

```
eq eval(EXP + EXP', VAL, FMAP, DATA, PRS)
 = eval(EXP, VAL, FMAP, DATA, PRS) +
   eval(EXP', VAL, FMAP, DATA, PRS) .

eq (E # B) + (E' # B') = (E + E') # (B and B') .
```

### 5.2.4 Executing Actions

The operation execAction computes a behavior action, and returns a "behavior configuration" for local variables, ports, and state variables. These configurations contain symbolic expressions and cumulative constraints, and represent (possibly infinite) sets of concrete configurations.

The following equations define the semantics of an assignment action *id* `:=` *exp*, assigning to *id* the evaluated value of *exp*, where the presence of *exp* is given as a constraint (`PHI and B`):

```
ceq execAction(VI := EXP, VAL, FMAP, DATA, PRS, PHI)
  = insert(VI, E # true, VAL) | FMAP | DATA | PRS |
    PHI and B
 if E # B := eval(EXP, VAL, FMAP, DATA, PRS) .

ceq execAction(P := EXP, VAL, FMAP, DATA, PRS, PHI)
 = VAL | insert(P, E # true, FMAP) | DATA | PRS |
   PHI and B
 if E # B := eval(EXP, VAL, FMAP, DATA, PRS) .

ceq execAction(C := EXP, VAL, FMAP, DATA, PRS, PHI)
  = VAL | FMAP | insert(C, E # true, DATA) | PRS |
    PHI and B
 if E # B := eval(EXP, VAL, FMAP, DATA, PRS) .
```

For a conditional statement, given a "concrete" configuration, the branch condition can evaluate to either *true* or *false*. Thus, `execAction` produces both cases with different constraints (e.g., `EXP` and `not(EXP)` in the following equations):

```
vars AS AS' : ActionSequence .   var A : Action .

crl execAction(if (EXP) AS else AS' end if, FMAP,
   DATA, PRS, PHI)  =>  execAction(AS, VAL, FMAP,
   DATA, PRS, PHI and E and B)
 if E # B := eval(EXP, VAL, FMAP, DATA, PRS) .

crl execAction(if (EXP) AS else AS' end if, FMAP,
   DATA, PRS, PHI)  =>  execAction(AS', VAL, FMAP,
   DATA, PRS, PHI and E and B)
 if E # B := eval(not(EXP), VAL, FMAP, DATA, PRS) .
```

Similarly, `execAction` produces both *true* and *false* cases for the branch condition of a loop:

```
crl execAction(while (EXP) {AS}, VAL, FMAP, DATA,
 PRS, PHI)  =>  execAction({AS ; while (EXP) {AS}},
 VAL, FMAP, DATA, PRS, PHI and E and B)
 if E # B := eval(EXP, VAL, FMAP, DATA, PRS) .

crl execAction(while (EXP) {AS}, VAL, FMAP, DATA,
 PRS, PHI)  =>  VAL | FMAP | DATA | PRS | (PHI and
 E and B)
 if E # B := eval(not(EXP), VAL, FMAP, DATA, PRS) .
```

For a sequence of actions $\{Act_1\,;\,\cdots;\,Act_n\}$, each action in the sequence is executed based on the execution results of the previous actions:

```
ceq execAction({A ; AS}, VAL, FMAP, DATA, PRS, PHI)
 = execAction({AS}, VAL', FMAP', DATA', PRS, PHI')
 if VAL' | FMAP' | DATA' | PHI' :=
    execAction(A, VAL, FMAP, DATA, PRS, PHI) .

eq execAction({A}, VAL, FMAP, DATA, PRS, PHI)
 = execAction(A, VAL, FMAP, DATA, PRS, PHI) .
```

## 6 Symbolic Semantics of Continuous Environments

This section presents the semantics of continuous environments in HybridSynchAADL. An environment component can *continuously* change its state variables according to the continuous dynamics, while *discretely* interacting with its controllers according to the sampling and actuating times. In our Maude-with-SMT semantics, nontrivial interactions between environments and controllers are specified using symbolic rewriting with constrained terms, where continuous dynamics—with all possible sampling and actuating times based on imprecise clocks—is encoded in SMT.

### 6.1 Environment Behavior

Figure 12 depicts the behavior of an environment $E$ that interacts with two controllers $C_n$, $n = 1, 2$, *in a single iteration*. Let $g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ denote the *global time* $g(i)$ at the beginning of the $i$-th period, where $g(i+1) - g(i) = \mathit{period}$. The time frame is "shifted" to the left from the global time frame $[g(i), g(i+1)]$ by a maximal clock skew $\epsilon > 0$.

1. The state variables of $E$ have initial values $\vec{v}_0$, and change continuously over time according to $E$'s continuous dynamics;
2. The period of each controller $C_n$ begins at any time $0 < t_n^0 < 2\epsilon$, because $C_n$ runs according to its local clock;
3. $E$ sends the state values $\vec{v}_n^s$ at time $t_n^s$ to each controller $C_n$, where $t_n^s - t_n^0$ denote the sampling time declared by $C_n$; and
4. $E$ receives $C_n$'s command $\alpha_n$ at time $t_n^a$ (and may change its continuous dynamics), where $t_n^a - t_n^0$ denote the actuating time.

Unlike controller components, the behavior of environment components cannot be directly specified as synchronous composition. Indeed, the environment behavior is *asynchronous*, because different orders of "interaction events" can lead to different behaviors (e.g., $\mathit{sampling}_{C_1}$, $\mathit{sampling}_{C_2}$,

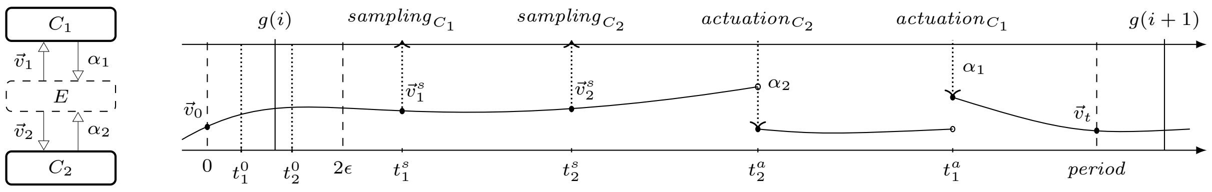


**Figure 12** Interactions between environment $E$ and two controllers $C_1$ and $C_2$.

$actuation_{C_1}$, and $actuation_{C_2}$ in Figure 12). Moreover, interactions between environments and controllers are *immediate*, whereas synchronous composition requires that interactions between components must be *delayed*. This implies that any *concrete semantics* of HybridSynchAADL cannot be easily defined as synchronous composition.

Previously, there were two approaches to deal with asynchronous interactions in AADL. One way is to explicitly enumerate all possible interleavings of components [34], which quickly leads to state space explosion. In Hybrid PALS [10], a controller and an environment are combined into an *environment-restricted* machine, where a controller is given as a "flat" state machine. However, this technique is not applicable to HybridSynchAADL, since a controller component may include arbitrarily complex (hierarchical) subcomponents.

This paper presents an alternative approach to symbolically encode asynchronous interactions in a *modular* way. We encode the values of input and output ports at different sampling and actuating times as symbolic variables, and run `executeStep` on each component *independently*, without considering interleavings of components. We then declare the correspondence between the input and output ports using equality constraints. This relies on the fact that an environment interacts *only once* with each of its controllers in a single iteration.

Figure 13 illustrates a modular encoding of the interactions in Figure 12. A term $o(\vec{i} \mid \vec{o})$ represents an object instance with input ports values $\vec{i}$ and output ports values $\vec{o}$. We perform `executeStep` on each component without interleavings, where the input port values are replaced by SMT variables. For the enclosing ensemble, `executeStep` includes "connection" constraints that declare equalities for the SMT variables in the input ports and the symbolic values in the output ports.

$$\frac{\begin{array}{c}\texttt{executeStep}(true \parallel o_{C_1}(x_1 \mid \cdot)) \rightsquigarrow^* \psi_{C_1} \parallel o'_{C_1}(\cdot \mid \alpha_1)\\ \texttt{executeStep}(true \parallel o_{C_2}(x_2 \mid \cdot)) \rightsquigarrow^* \psi_{C_2} \parallel o'_{C_2}(\cdot \mid \alpha_2)\\ \texttt{executeStep}(true \parallel o_E(y_1, y_2 \mid \cdot)) \rightsquigarrow^* \psi_E \parallel o'_E(\cdot \mid v_1, v_2)\\ \phi' = \psi_{C_1} \wedge \psi_{C_2} \wedge \psi_E\\ \psi_{conn} = (x_1 = v_1 \ \wedge \ x_2 = v_2 \ \wedge \ y_1 = \alpha_1 \ \wedge \ y_2 = \alpha_2)\\ \mathsf{check\text{-}sat}(\phi \wedge \phi' \wedge \psi_{conn})\end{array}}{\begin{array}{c}\texttt{executeStep}(\phi \parallel < \ldots \mid \texttt{subcomponents} : o_{C_1}\ o_{C_2}\ o_E >)\\ \rightsquigarrow^*\\ \phi \wedge \phi' \wedge \psi_{conn} \parallel < \ldots \mid \texttt{subcomponents} : o'_{C_1}\ o'_{C_2}\ o'_E >\end{array}}$$

**Figure 13** Modular encoding of asynchronous interactions.

The semantics of environments is formalized by rewrite rules using `executeStep`, which build the *constrained objects* with SMT constraints to encode the environment behavior shown in Figure 12. All information for interaction with controllers—including the values of input and output ports at different sampling/actuating times—is encoded as SMT variables. The immediate communication between environments and controllers is encoded as symbolic constraints. As a result, the semantics of ensembles with environment subcomponents can be almost identical to the one in Section 5.1.

The behavior of environment components is defined by the following rule. We add the attribute `varGen` to the class `Env` to maintain counters for generating fresh SMT variables:

```
vars PRTS1 PRTS2 : Configuration .
vars STS RTS : Set{InterTiming} .
vars VG VG' VG1 VG2 VG3 : VarGen .
vars JUMPS JUMPS' : Set{EnvJump} .
vars FLOWS FLOWS' : Set{EnvFlow} .
vars IPH OPH : BoolExp .

crl executeStep(
    PHI  ||  < C : Env |   features : PORTS,
        properties : PRS,  subcomponents : COMPS
        currMode : L,      connections : CONXS,
        jumps : JUMPS,     flows : FLOWS,
        sampling : STS,    response : RTS,
        varGen : VG >)
 =>
```

| # Iteration | | 1 | 2 | 3 | ... |
|---|---|---|---|---|---|
| Controller | | $d_1$ | $d_2$ | $d_3$ | ... |
| Input port | content | · | $d_1$ | $d_2$ | ... |
| | envCache | · | $x_1$ | $x_2$ | ... |
| readEnvPort | FMAP | $x_1$ | $x_2$ | $x_3$ ... | |
| | constraint | *true* | $x_1 = d_1$ | $x_2 = d_2$ | ... |

**Figure 14** The behavior of readEnvPort.

| # Iteration | | 0 | 1 | 2 | ... |
|---|---|---|---|---|---|
| Environment | FMAP | | $d_1$ | $d_2$ | ... |
| | envCache | | $x_0$ | $x_1$ | ... |
| writeEnvPort | content | $x_0$ | $x_1$ | $x_2$ | ... |
| | constraint | | $x_0 = d_1$ | $x_1 = d_2$ | ... |
| Controller | | | $x_0$ | $x_1$ | ... |

**Figure 15** The behavior of writeEnvPort.

```
    PHI' and OPH ||  < C : Env | features : PRTS2,
          currState : L',        varGen : VG3,
          subcomponents : writeData(DATA',COMPS) >
if {PRTS1,FMAP,IPH,VG1} := readEnvPort(PORTS,VG)
/\ DATA := readData(COMPS)
/\ execEnv(0, FMAP, DATA, PRS, L, JUMPS, FLOWS,
          STS, RTS, CONXS, PORTS, PHI and IPH, VG1)
    => L' | FMAP' | DATA' | PHI' | VG2
/\ {PRTS2,OPH,VG3} := writeEnvPort(FMAP',PRTS1,VG2)
/\ check-sat(PHI' and OPH) .
```

The function readEnvPort returns a map from each input port to its content, and writeEnvPort updates the content of each output port; these functions also return extra constraints to encode the environment communication. The operation execEnv builds symbolic constraints to encode the behavior of the environment in one-step iteration. The function readData returns a map from each data subcomponent to its value, and writeData updates the value of each data subcomponent.

## 6.2 Environment Port Operations

Given environment ports, readEnvPort removes the contents $\vec{d}$ from the input ports, and builds a map from port identifiers to fresh SMT variables $\vec{x}_i$ denoting the contents *sent by the controllers in the same iteration*. Since the current contents $\vec{d}$ are sent by the controllers in the previous iteration, readEnvPort builds equality constraints to relate the *current* contents $\vec{d}$ and the "cached" SMT variables $\vec{x}_{i-1}$ generated in the previous iteration. In this way, we symbolically encode immediate communication using delayed communication.

As described in Figure 14, we use the attribute envCache to keep the symbolic content generated in the previous round. Suppose that a controller sends a data content $d_i$ to an input port $p$ in the $i$-th iteration. Because of delayed communication, the content of $p$ is then $d_{i-1}$ in the $i$-th iteration. As mentioned above, readEnvPort relates the port identifier $p$ to a fresh variable $x_i$, and generates the equality constraint $x_{i-1} = d_{i-1}$.

The following equations define the function readEnvPort using an auxiliary function with an extra argument to carry intermediate results:

```
eq readEnvPort(PORTS, VG)
 = readEnvPort(PORTS, none, empty, true, VG) .
eq readEnvPort(none, PORTS, FMAP, PHI, VG)
 = {PORTS, FMAP, PHI, VG} [owise] .
```

Each input port P is related to a symbolic content V # BV with fresh variables V and BV, and the current content E # B and envCach E' # B' are declared to be identical as a constraint:

```
var V : SMTVar .     var BV : SMTBoolVar .

ceq readEnvPort(< P : EnvInPort | content : E # B,
       envCache : E' # B' > PORTS, PORTS',
       FMAP, PHI, VG)
 = readEnvPort(PORTS, PORTS' < P : EnvInPort |
       content  : E # false, envCache : V # BV >,
       insert(P, V # BV : true, FMAP),
       PHI and E === E' and B === B', VG2)
if {V, VG1} := freshVar(VG, type(E))
/\ {BV,VG2} := freshVar(VG1,Boolean) .
```

Each output port P is related to E # false, indicating ⊥ with the second item false:

```
eq readEnvPort(< P : EnvOutPort |
    content : E # B > PORTS, PORTS', FMAP, PHI, VG)
= readEnvPort(PORTS, PORTS' < P : EnvOutPort | >,
    insert(P, E # false, FMAP), PHI, VG) .
```

Likewise, the function writeEnvPort updates the contents of the output ports with fresh SMT variables, and builds equality constraints to relate the *current* contents in the feature map FMAP and the "cached" variables generated in the previous iteration. Thus, the input ports of the controllers receive the contents sent from the environment in the same iteration. We use again envCache to keep the symbolic content sent in the previous round, to implement this behavior, as described in Figure 15, where $x_0$ denotes the initial content:

```
eq writeEnvPort(FMAP, PORTS, VG)
 = writeEnvPort(PORTS, none, FMAP, true, VG) .
```

```
ceq writeEnvPort(< P : EnvOutPort | content : D,
        envCache : E # B > PORTS, PORTS',
    FMAP, PHI, VG)
 = writeEnvPort(PORTS, PORTS' < P : EnvOutPort |
        content  : V # BV, envCache : V # BV >,
    FMAP, PHI and E === E' and B === B', VG2)
  if E' # B' := FMAP[P]
  /\ {V, VG1} := freshVar(VG, type(E))
  /\ {BV,VG2} := freshVar(VG1,Boolean) .

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

The constraints for environment inputs in one iteration are generated by `readEnvPort` in the *next* iteration. Therefore, `executeStep` for ensembles is slightly modified to include such constraints (by the function `finalConst`) as follows:

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

The function `finalConst` simply collects the environment input constraints by `readEnvPort`:

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

## 6.3 Executing Environments

We build the SMT constraints for the environment behavior, shown in Figure 12, using two operations: `execEnv` for the continuous behavior, and `envStep` for the discrete behavior. We define three rewrite rules to specify the semantics of environments: `env-cont` for continuous state changes, `env-samp` for sampling, and `env-resp` for actuation.

### 6.3.1 Continuous Transitions

The following rule `env-cont` defines a "continuous transition" from a state at time `T` to a state at time `T'`, where `T'` is as a fresh SMT variable and the new environment state `DATA'` by the continuous dynamics is obtained using `execFlow`:

```
vars T T' PER SK : RealExp .     vars LT UT : Rat .
var PNS : Set{FeatureRef} .  var CI : ComponentId .
var FLOW : FlowItem . var FUNCS : Set{AADLEnvFunc} .

crl [env-cont]: execEnv(T, FMAP, DATA, PRS, L,
    JUMPS, FLOWS, STS, RTS, CONXS, PORTS, PHI, VG)
=>
    envStep(T', FMAP, DATA', PRS, L, JUMPS, FLOWS,
       STS, RTS, CONXS, PORTS, PHI', VG')
if {T',VG'} := freshVar(VG,Real)
/\ flows((L [FLOW]) ; FLOWS') := FLOWS
/\ DATA' := execFlow(FLOW, T'-T, FMAP, DATA, PRS)
/\ PHI' := PHI and T <= T' and B .
```

In HybridSynchAADL, continuous dynamics is specified using either ODEs or continuous real functions. For example, consider a continuous real function `C(VI) = EXP` over an input argument `VI`:[7] the function `execFlow` computes its value using `eval`, where a given duration `T` is assigned to the local variable identifier `VI` as follows:

```
ceq execFlow((C(VI) = EXP ; FUNCS),T,FMAP,DATA,PRS)
 = execFlow(FUNCS, T, FMAP, insert(C,D,DATA), PRS)
 if D := eval(EXP,(VI |-> T # true),FMAP,DATA,PRS) .
eq execFlow(empty, T, FMAP, DATA, PRS) = DATA .
```

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

```
ceq envStep(T, FMAP, DATA, PRS, L, JUMPS, FLOWS,
        empty, empty, CONXS, PORTS, PHI, VG)
 = L | FMAP | DATA | PHI and B and T === PER | VG
if PER # B := eval(TimingProperties::Period,
               FMAP, DATA, PRS) .
```

### 6.3.2 Environment Sampling

The following rule `env-samp` specifies the behavior of sampling operations. A sampling time bound $(lt, ut)$ of a controller `C` is nondeterministically chosen in the left-hand side of the rule:

```
crl [env-samp]:
  envStep(T, FMAP, DATA, PRS, L, JUMPS, FLOWS,
    (C :(LT,UT), STS), RTS, CONXS, PORTS, PHI, VG)
```

[7] There exist specialized solvers to support SMT solving with ODEs [35]. Because they have not been integrated with Maude, the current version only supports continuous functions.

```
=>
  execEnv(T, FMAP', DATA, PRS, L, JUMPS, FLOWS,
          STS, RTS, CONXS, PORTS, PHI and B, VG)
if B := timeConst(T, LT, UT, FMAP, DATA, PRS)
/\ FMAP' := smpPort(C, FMAP, DATA, CONXS, PORTS) .
```

The function timeConst gives the constraint for the sampling time T with respect to the clock skew and the sampling time bound. As explained, because the period of controller C happens at any time between 0 and $2\epsilon$, the sampling happens between $lt$ and $ut + 2\epsilon$ as follows:

```
ceq timeConst(T, LT, UT, FMAP, DATA, PRS) =
 real(LT) <= T and T <= real(UT) + 2 * SK and B  if
 SK # B := eval(HybridSynchAADL::MaxClockDeviation,
        FMAP, DATA, PRS) .
```

The function smpPort updates the feature map item for each output port P connected to C with the corresponding state variable CI, provided that P is connected to the controller C:

```
ceq smpPort(C, FMAP, DATA, CI ==> P ; CONXS, PORTS)
 = smpPort(C, insert(P,DATA[CI],FMAP), DATA, CONXS,
          PORTS)  if validTarget(P, C, PORTS) .
eq smpPort(C, FMAP, DATA, CONXS, PORTS)
 = FMAP [owise] .

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

### 6.3.3 Environment Actuation

The following rules env-resp specify the behavior of actuation operations. In the first rule, there is a mode transition from the current mode L, where a trigger port P, connected with the controller C, has received a content (isPortPresent). In the second rule, all trigger input ports from L, connected with C, have received no content (allTrigAbsent). The actuating time T is constrained by the clock skew and the actuating time bound (timeConst), and all state variables "connected" to the controller C are updated with the port values (actData):

```
crl [env-resp]:
   envStep(T, FMAP, DATA, PRS, L, JUMPS, FLOWS,
     STS, (C :(LT,UT), RTS), CONXS, PORTS, PHI, VG)
=>
   execEnv(T, FMAP, DATA', PRS, L', JUMPS, FLOWS,
     STS, RTS, CONXS, PORTS, PHI and B and B', VG)
if L -[P, PNS]-> L' ; JUMPS' := JUMPS
/\ validTarget(P, C, PORTS)
/\ B := timeConst(T, LT, UT, FMAP, DATA, PRS)
/\ B' := isPortPresent(P, FMAP)
/\ DATA' := actData(C, FMAP, CONXS, PORTS, DATA) .

crl [env-resp]:
   envStep(T, FMAP, DATA, PRS, L, JUMPS, FLOWS,
     STS, (C :(LT,UT), RTS), CONXS, PORTS, PHI, VG)
=>
   envStep(T, FMAP, DATA', PRS, L, JUMPS, FLOWS,
      STS, RTS, CONXS, PORTS, PHI and B and B', VG)
if B := timeConst(T, LT, UT, FMAP, DATA, PRS)
/\ B' := allTrigAbsent(L, C, JUMPS, FMAP, PORTS)
/\ DATA' := actData(C, FMAP, CONXS, PORTS, DATA) .
```

The function isPortPresent returns a constraint that a given input port P has received a value. The function allTrigAbsent returns a constraint that all trigger input ports of each mode transition from a given mode L are not present if they are connected to a given controller C:

```
ceq isPortPresent(P, FMAP) = B and B'
 if E # B : B' := FMAP[P] .

eq allTrigAbsent(L, C, (L -[PNS]-> L') ; JUMPS,
  FMAP, PORTS) = allTrigAbsent(L, C, JUMPS, FMAP,
  PORTS) and allTrigAbsent(PNS, C, FMAP, PORTS) .
eq allTrigAbsent(L, C, JUMPS, FMAP, PORTS)
 = true [owise] .

ceq allTrigAbsent((P, PNS), C, FMAP, PORTS)
 = not isPortPresent(P,FMAP) and allTrigAbsent(PNS,
   C, FMAP, PORTS)  if validTarget(P, C, PORTS) .
eq allTrigAbsent(PNS, C, FMAP, PORTS)
 = true [owise] .
```

The function actData updates the content of each state variable CI by the content of an input port P with a connection P =>> CI, provided P has received a value from a given controller C. If P is not present, CI is not updated; this is encoded using the conditional operator _?_:_:

```
ceq actData(C, FMAP, P =>> CI ; CONXS, PORTS, DATA)
  = actData(C, FMAP, CONXS, PORTS,
       insert(CI, (B ? E : E') # (B or B'), DATA))
 if validTarget(P, C, PORTS)
 /\ E # B : B'' := FMAP[P] /\ E' # B' := DATA[CI] .

eq actData(C, FMAP, CONXS, PORTS, DATA)
 = DATA [owise] .
```

## 7 Formal Analysis using the HybridSynchAADL Tool

To support the convenient modeling and formal analysis of HybridSynchAADL models within the OSATE tool environment, we have developed the HybridSynchAADL OSATE plugin that:

1. provides an intuitive language to specify properties/requirements of models,
2. automatically generates a Maude-with-SMT model from a HybridSynchAADL model, and
3. performs various formal analyses using Maude with SMT solving as a back-end.

Our tool implements a state merging technique, adapted from [15], to significantly improve analysis performance, and is available at https://hybridsynchaadl.github.io.

Section 7.1 presents our tool's property specification language and its Maude semantics; i.e., how properties are analyzed in Maude with SMT. Section 7.2 explains our state merging techniques for optimizing the performance of such Maude-with-SMT analysis. Section 7.3 introduces the tool and its user interface. Finally, Section 7.4 explains our approaches for gaining confidence in the correctness of the tool implementation.

## 7.1 Specifying Properties

Our tool's *property specification language* allows the user to easily specify invariant and reachability properties—without having to understand Maude or SMT—as propositional formulas whose atomic propositions are AADL Boolean expressions. Since HybridSynchAADL models typically are infinite-state systems, we only consider properties over behaviors up to a given time bound.

Atomic propositions are Boolean expressions in the AADL Behavior Annex syntax. A *named proposition* can be declared in HybridSynchAADL as follows, where each identifier is fully qualified with its component path in the AADL syntax:[8]

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

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

We can simplify component paths that appear repeatedly using component scopes. A *scoped expression* of the form *path* | *exp* denotes that each component path of each identifier in *exp* begins with *path*. For example, $c_1 . c_2$ | $(x_1 > x_2)$ is equivalent to $(c_1 . c_2 . x_1 > c_1 . c_2 . x_2)$.

The following *named invariant property* holds if for every state satisfying the initial condition $\varphi_{init}$, all states reachable within the time bound $\tau_{bound}$ satisfy the property $\varphi_{inv}$.

```
invariant [name]: φ_init ==> φ_inv in time τ_bound
```

A *named reachability property* (the dual of an invariant) holds if a state satisfying $\varphi_{goal}$ is reachable from some state satisfying the initial condition $\varphi_{init}$ within the time bound $\tau_{bound}$.

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

### 7.1.1 Semantics

The formal semantics of the property specification language is defined in Maude. Given a Boolean expression `COND` and a top-level component `OBJ`, its value, written $[\![$`COND`$]\!]$ `OBJ`, is *true* if it evaluates to a data content $b$ `#` $b'$, and both $b$ and $b'$ are true. The following equation defines $[\![$`COND`$]\!]$ `OBJ`, where `nil` denotes the empty path:

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

Given a scoped expression, `normal` returns a *normalized* expression without component scopes. This function is defined inductively as follows, where `VAR` denotes a variable identifier, and `PH` and `PH'` denote component paths:

```
eq normal(PH, VALUE) = VALUE .
eq normal(PH, VAR) = PH . VAR .
eq normal(PH, PH' | EXP) = normal(PH . PH', EXP) .
eq normal(PH, EXP and EXP')
 = normal(PH, EXP) and normal(PH, EXP') .
eq normal(PH, EXP or EXP')
 = normal(PH, EXP) or normal(PH, EXP') .
...
```

Given a set of components, the function `evalPS` computes the content $e$ `#` $b$ of a normalized expression using the function `eval` (defined in Section 5.2.3) as follows, where `AtomicComponent` is a superclass of the "atomic" classes `Thread` and `Env`:

```
eq evalPS(VALUE, COMPS) = VALUE .
eq evalPS(C . PH . VAR,  COMPS < C : Ensemble |
        subcomponents : COMPS' >)
 = evalPS(PH . VAR, COMPS') .
eq evalPS(C . VAR,  COMPS < C : AtomicComponent |
        subcomponents : DATA, properties : PRS >)
 = eval(VAR, none, DATA, PRS) .
eq evalPS(EXP and EXP', COMPS)
 = evalPS(EXP, COMPS) and evalPS(EXP', COMPS) .
eq evalPS(EXP or  EXP', COMPS)
```

[8]A component path is given by a period-separated path of component identifiers in AADL; for example, $c_1 . c_2 . b$.

```
 = evalPS(EXP, COMPS) or  evalPS(EXP', COMPS) .
...
```

A reachability property $\varphi_{init}$ `==>` $\varphi_{goal}$ `in time` $\tau_{bound}$ then corresponds to the following Maude search command to find its witness, where $N$ is the quotient of $\tau_{bound}$ by the period of the model and `initState` is the term representation of the entire model:

```
search [1,N]
       {(([[φ_init]] initState) || initState}
          =>* {PHI || OBJ}
       such that
          check-sat(PHI and
             finalConst(OBJ) and ([[φ_goal]] OBJ)) .
```

Similarly, an invariant property of the form $\varphi_{init}$ `==>` $\varphi_{inv}$ `in time` $\tau_{bound}$ corresponds to the following Maude search command to find its counterexample (i.e., the property holds if the search command *cannot* find a solution):

```
search [1,N]
       {(([[φ_init]] initState) || initState}
          =>* {PHI || OBJ}
       such that
          check-sat(PHI and
             finalConst(OBJ) and not([[φ_inv]] OBJ)) .
```

### 7.1.2 Example

Consider the thermostat system in Section 3 with two thermostat controllers `ctrl1` and `ctrl2` and their environments `env1` and `env2`. The following declares two propositions `inRan1` and `inRan2`. For example, `inRan1` holds if the state variable `x` of the environment `env1` is between 10 and 25.

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

The following declares the invariant property `inv`, which holds iff for each initial state satisfying $|$`env1.x`$- 15| < 3$ and $|$`env2.x`$- 7| < 1$, any state reachable within time bound 30 satisfies `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
```

The property `inv` corresponds to the following Maude search command. The constant `initState` is replaced by the term representation of the entire model, and `inRan1` and `inRan2` are replaced by the related Boolean expressions.[9]

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

## 7.2 Merging Symbolic States

Nontrivial control programs with many branches and guarded transitions typically involve a large number of symbolic states. Furthermore, for an environment interacting with $n$ controllers, there are $O((2n)!/2^n)$ symbolic execution results due to different orders of sampling and actuating events. Thus, `executeStep` can generate many different execution results for one environment component.

We *symbolically* reduce the number of different execution results by merging two terms that are syntactically identical except for SMT subterms into one constrained term. Consider a term $t(u_1, \ldots, u_n)$ with SMT subterms $u_1, \ldots, u_n$. Let $x_1, \ldots, x_n$ be fresh SMT variables that do not appear in the term $t$. An *abstraction of built-ins* for $t$, denoted by $abs(t)$, is a constrained term $(x_1 = u_1 \wedge \cdots \wedge x_n = u_n) \parallel t(x_1, \ldots, x_n)$, which is semantically equivalent to the original term $t$ (i.e., $[\![abs(t)]\!] = [\![true \parallel t]\!]$) [14].

Abstractions of built-ins $\phi_1 \parallel t_1$ and $\phi_2 \parallel t_2$ are *mergeable* iff $t_1 = \rho t_2$ for a renaming substitution $\rho$ (i.e., $t_1$ and $t_2$ are equivalent up to renaming). The *merged term* is then the constrained term

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

E.g., $y > x \parallel f(y)$ and $z = 3 \parallel f(z)$ can be merged into $(y > x \vee y = 3) \parallel f(y)$. The following lemma ensures the correctness of our method.

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

*Proof* By definition (Section 2.3), $u \in [\![(\phi_1 \vee \rho\phi_2) \parallel t_1]\!]$ iff there is a substitution $\theta$ such that $u = \theta t_1$ and $\mathcal{T} \models \theta(\phi_1 \vee \rho\phi_2)$. Since $t_1 = \rho t_2$, $u = \theta\rho t_2$ and either $\mathcal{T} \models \theta\phi_1$ or $\mathcal{T} \models \theta\rho\phi_2$ holds. That is, one of the

[9]Since the period of the system is 10 (ms), we search for states reachable within $30/10 = 3$ iterations/steps of the system.

following must hold: (i) $u = \theta t_1$ and $\mathcal{T} \models \theta\phi_1$, or (ii) $u = \theta\rho t_2$ and $\mathcal{T} \models \theta\rho\phi_2$. By definition, we have $u \in [\![\phi_1 \parallel t_1]\!]$ or $u \in [\![\rho(\phi_2 \parallel t_2)]\!]$. Since $\rho$ is a renaming substitution, $u \in [\![\rho(\phi_2 \parallel t_2)]\!]$ iff $u \in [\![\phi_2 \parallel t_2]\!]$. As a result, $[\![(\phi_1 \vee \rho\phi_2) \parallel t_1]\!] = [\![\phi_1 \parallel t_1]\!] \cup [\![\phi_2 \parallel t_2]\!]$. □

**Algorithm 1** `executeStep` with State Merging

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

```
1 $S \leftarrow \{(\phi' \parallel obj') \mid \texttt{executeStep}(\phi \parallel obj) \rightsquigarrow^* \phi' \parallel obj'\}$
2 $\widehat{S} \leftarrow \{(\phi' \wedge \psi \parallel t) \mid (\phi' \parallel obj') \in S,\ (\psi \parallel t) = abs(obj')\}$
3 while $\exists(\varphi_1 \parallel t_1), (\varphi_2 \parallel t_2) \in \widehat{S}.\ mergeable(t_1, t_2)$ do
4     $\varphi \parallel u \leftarrow$ a merged term of $\varphi_1 \parallel t_1$ and $\varphi_2 \parallel t_2$
5     $\widehat{S} \leftarrow (\widehat{S} \cup \{\varphi \parallel u\}) \setminus \{\varphi_1 \parallel t_1,\ \varphi_2 \parallel t_2\}$
6 return $\widehat{S}$
```

Algorithm 1 summarizes the new "merging" operation. It collects all the execution results by `executeStep` and merges all mergeable results. Algorithm 1 always generates a single "merged" result, since any constrained objects obtained by `executeStep` from the same object are mergeable in our HybridSynchAADL semantics. Hence, the `step` rule for the entire system will yield a single symbolic state for one synchronous step.

More precisely, we define a function `symAbs` to obtain abstractions of built-ins. For two terms $t_1$ and $t_2$, `symAbs` returns a triple $(u, \phi_1, \phi_2)$, where $\phi_1 \parallel u$ and $\phi_2 \parallel u$ are, abstractions of $t_1$ and $t_2$, respectively, with the same SMT variables. E.g., `symAbs` for two SMT expressions $e_1$ and $e_2$ is a triple $(x,\ x = e_1,\ x = e_2)$ with a fresh variable $x$ (`VG` is a counter for generating fresh variables):

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

We define `symAbs` for each "pattern" of terms, such as data contents and data valuations, that can appear in the execution results of `execTrans` and `execEnv`. To illustrate, consider data contents of the form `e # b` in our semantics. Using `symAbs` for SMT expressions described above, `symAbs` for data contents is defined as follows:

```
ceq symAbs(E # B, E' # B', VG)
  = {ME # MB, CS1 and CS2, CS1' and CS2', VG2}
 if {ME, CS1, CS1', VG1} := symAbs(E, E', VG)
 /\ {MB, CS2, CS2', VG2} := symAbs(B, B', VG1) .
```

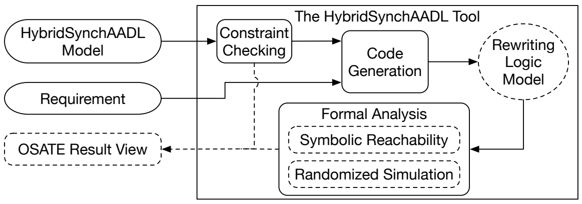


**Figure 16** Architecture of the HybridSynchAADL tool.

The function `symMerge` syntactically merges those terms into a single constrained term using `symAbs` for each pattern, where `max` returns the maximum of two fresh variable counters. We do not need to perform extra renaming, because the same set of fresh variables are used for `symAbs`:

```
ceq symMerge(L   | FMAP  | DATA  | PHI  | VG,
             L'  | FMAP' | DATA' | PHI' | VG')
  = ML | MFMAP | MDATA | MPHI | VG3
if VG0 := max(VG,VG')
/\ {ML,PHI1,PHI1',VG1} := symAbs(L,L',VG0)
/\ {MFMAP,PHI2,PHI2',VG2} := symAbs(FMAP,FMAP',VG1)
/\ {MDATA,PHI3,PHI3',VG3} := symAbs(DATA,DATA',VG2)
/\ MPHI := (PHI and PHI1 and PHI2 and PHI3) or
           (PHI' and PHI1' and PHI2' and PHI3') .
```

Our technique can be considered an instance of a symbolic state-space reduction approach [15] that merges symbolic states using disjunction and generalization. Algorithm 1 is adapted from [15] to deal with arbitrary constrained objects in HybridSynchAADL. As shown in Section 9.3, this state merging dramatically improves the performance of symbolic analysis and makes the formal analysis feasible for such distributed hybrid systems.

## 7.3 The HybridSynchAADL Tool

Figure 16 shows the architecture of the HybridSynchAADL tool. The tool first statically checks whether a given model satisfies the syntactic constraints of HybridSynchAADL. It uses OSATE's code generation facilities to synthesize a Maude-with-SMT model from the HybridSynchAADL model. It invokes Maude and an SMT solver to check whether the model satisfies given invariant and reachability requirements. Our tool is implemented in around 6,200 lines of Maude code and around 8,600 lines of Java and Xtend code.

Syntactic validation of a HybridSynchAADL model ensures that the Maude-with-SMT model is symbolically executable. E.g., an environment component must declare the component property `Hybrid_SynchAADL::ContinuousDynamics` over its data subcomponents of type `Base_Types::Float`. The tool checks other "trivial" constraints that are assumed in the formal semantics; 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 satisfy a given requirement; if not, a counterexample is generated. *Randomized simulation* repeatedly executes the model until a counterexample is found, by randomly choosing concrete sampling and actuating times, nondeterministic transitions, etc. A counterexample (or witness) shows the data subcomponent values of all AADL components in the system in each synchronous step.

Our tool also provides *portfolio analysis* that combines symbolic reachability analysis and randomized simulation. Symbolic reachability analysis can guarantee the absence of a counterexample, and can find subtle counterexamples, whereas randomized simulation is effective for finding "obvious" bugs. The tool runs both analysis methods in parallel using multithreading, and displays the result of the analysis that terminates first.

Figure 17 shows the interface of our tool that is fully integrated into OSATE. The left editor shows the code of `FourDronesSystem` in Section 8 below, 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 clicked, and the `Result` view at the bottom displays the results in a readable format.

### 7.4 Testing the Implementation

We have not undertaken a formal proof that our implementation is correct. Instead, we have thoroughly tested the formal semantics and the generated models. Since our Maude-with-SMT semantics is executable, it is straightforward to write test cases and check that the results are as expected. We have developed a test suite for unit testing of the equations and rules in the formal semantics, and a test suite for system testing of the code generation and state merging.

Our test suite includes 850 test cases for unit testing and 98 test cases for system testing. There are 782 equations and rewrite rules in our formal semantics, including state merging, and we have achieved 100% rule/equation coverage. To cover various language features of HybridSynchAADL, we use more than 10 models, including those in Section 9, for system testing. Our test suite also includes differential tests to check whether state merging gives the same result.

Finally, our confidence in our implementation is also strengthened by the fact that the results of analyzing the tool-generated HybridSynchAADL models are consistent with the analysis performed using other formal tools on corresponding "hand-coded" models in Section 9.

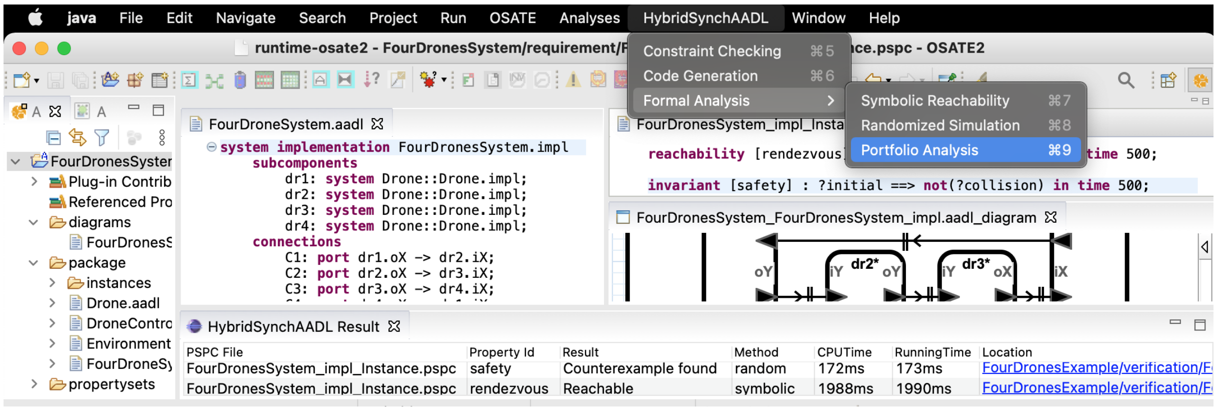
## 8 Case Study: Collaborating Autonomous Drones

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*. The controllers are physically distributed, because each controller is included in the drone hardware. Our models take into account network delays, asynchronous communication, continuous dynamics, clock skews, execution times, sampling and actuating times, etc.—for some of these features indirectly, via the Hybrid PALS equivalence.

### 8.1 Distributed Consensus of Drones

We use distributed consensus algorithms [36] to synchronize the drone movements. Each drone has an *information state* that represents the drone's local view of the coordination task, such as the rendezvous position, the center of a formation, etc. There is no centralized controller with a "global" view. Each drone periodically exchanges the information state with neighboring drones, and eventually the information states of all drones should converge to a common value.

Consider $N$ drones with double-integrator dynamics, where $\vec{x}_i$, $\vec{v}_i$, and $\vec{a}_i$ denote the position, velocity and, acceleration of the $i$-th drone, $1 \leq i \leq N$, respectively. The continuous dynamics of the $i$-th drone is then specified by the differential equations $\dot{\vec{x}}_i = \vec{v}_i$ and $\dot{\vec{v}}_i = \vec{a}_i$. Let $A$ denote the adjacency matrix representing the underlying



```
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 dr2.oX -> dr3.iX;
        C3: port dr3.oX -> dr4.iX;
```

```
reachability [rendezvous]                    time 500;

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

| PSPC File | Property Id | Result | Method | CPUTime | RunningTime | Location |
|---|---|---|---|---|---|---|
| FourDronesSystem_impl_Instance.pspc | safety | Counterexample found | random | 172ms | 173ms | FourDronesExample/verification/F |
| FourDronesSystem_impl_Instance.pspc | rendezvous | Reachable | symbolic | 1988ms | 1990ms | FourDronesExample/verification/F |

**Figure 17** Interface of the HybridSynchAADL tool.

communication network. If $A_{ij}$ is 0, then the $i$-th drone cannot receive information from the $j$-th drone. The controller samples the drone's state, and gives the new acceleration to the environment as a control command.

The goal of rendezvous is for all distributed drones to arrive at a common location at the same time (without crashing into each other). According to the distributed consensus algorithm [36], the acceleration $\vec{a}_i$ of the $i$-th drone is given by:

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

where $\gamma > 0$ is a coupling strength parameter for velocities. In each iteration, the information state $\vec{x}_i$ of the $i$-th drone is directed toward the information states of its neighbors in $A$, and eventually converges to a consensus value.

In formation control, 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. The acceleration $\vec{a}_i$ of the $i$-th drone, $1 \leq i < N$, is given by the following equation [36]:

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

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

In both cases, assuming that acceleration is negligible, we also consider a a simplified model for drones with *single-integrator* dynamics. The acceleration is then 0, and a controller gives a velocity as an actuation command. For rendezvous and formation control, the velocity $\vec{v}_i$ is given by the following equations, respectively [36]:

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

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

## 8.2 The HybridSynchAADL Model

Figure 18 illustrates a rendezvous model of four drones with single-integrator dynamics.[10] Each drone is connected to two other drones. A drone component consists of an environment and its controller: an environment defines the physical model of the drone, such as position and velocity, and a controller interacts with the environment according to the sampling and actuating times.

In each round, a controller determines a new velocity. The controller obtains the position $\vec{x}$ from its environment at its sampling time. The position of the connected drone was sent in the previous round. The environment of the drone changes its position according to the velocity $\vec{v}$ indicated by its controller, where the new velocity $\vec{v}$ becomes effective at its actuation time.

Figures 19–24 define the drone system in HybridSynchAADL when the drones move in a

[10]We have developed a variety of HybridSynchAADL models for rendezvous and formation control of different numbers of drones with single-integrator and double-integrator dynamics. All of them are available at https://hybridsynchaadl.github.io.

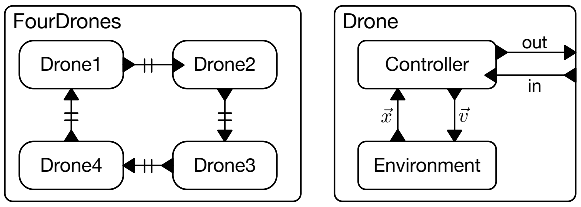


**Figure 18** The AADL architecture of four drones (left), and a drone component (right), where arrows with double bars denote delayed connections.

```
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;
```

**Figure 19** A top-level system component.

two-dimensional space and use single-integrator dynamics.

The top-level system component is declared to be synchronous with period 100 ms, and includes four `Drone` subcomponents (Figure 19). 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 connections between drone components are delayed, the initial values of the source output ports are 0.0, and the maximal clock skew is defined to be 10 ms.

A drone component has two input ports `iX` and `iY` and two output ports `oX` and `oY` (Figure 20). Its

```
system Drone
  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;
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 => 30ms .. 33ms;
    Hybrid_SynchAADL::Response_Time => 60ms .. 63ms;
end Drone.impl;
```

**Figure 20** A drone component.

implementation contains a controller `ctrl` and an environment `env`. The controller `ctrl` receives the current position from the environment `env` via the input ports `cX` and `cY`, and sends a new velocity to `env` via the output ports `vX` and `vY`, according to its sampling and actuating times. The controller `ctrl` also communicates with outside components using `Drone`'s input and output ports.

A controller system component has 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 (Figure 21). Its implementation includes the process component `ctrlProc`, which again includes the thread component `cThread`, as shown in Figure 22. 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 23 shows a thread component for a drone controller. When the thread dispatches, the transition from state `init` to `exec` is taken. If the distance to the connected drone is too close, then the new velocity is set to 0, and the `cls` flag is set to true. Otherwise, the new velocity is set toward the connected drone by a *discretized* version of the distributed consensus algorithm with a predefined

```
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;
```

**Figure 21** 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;
```

**Figure 22** 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
    cls: 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) < 0.1 and
            abs(cY - iY) < 0.1]-> output {
       vX := 0; vY := 0; cls := 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 (cls) 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 (cls) vY := 1.5 else vY := 0.0 end if
       else vY := -2.5 end if; cls := false };
     output -[]-> init { oX := cX; oY := cY }; **};
end DroneControlThread.impl;
```

**Figure 23** Controller thread.

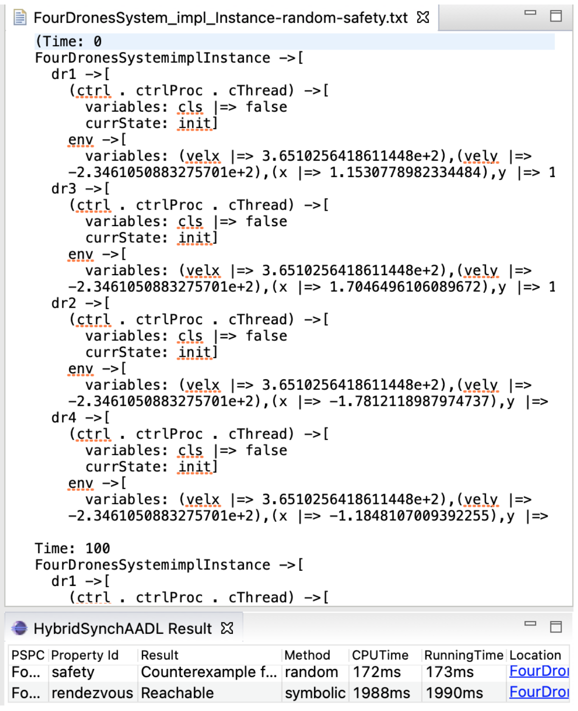
set of velocities. Finally, the current position is assigned to the output ports oX and oY.

Figure 24 shows an environment component. Data components 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 the controller sends an actuation command, the values of velx and vely are updated by the values of the input ports vX and vY. The continuous dynamics of $(x, y)$ is given as ordinary differential equations $\dot{x} = vel_x$ and $\dot{y} = vel_y$.

```
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 =>
        "d/dt(x) = velx; d/dt(y) = vely;";
    Data_Model::Initial_Value => ("param")
        applies to x, y, velx, vely;
end Environment.impl;
```

**Figure 24** An environment component.


FourDronesSystem_impl_Instance-random-safety.txt

```
(Time: 0
FourDronesSystemimplInstance ->[
  dr1 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: cls |=> false
      currState: init]
    env ->[
      variables: (velx |=> 3.6510256418611448e+2),(vely |=>
    -2.3461050883275701e+2),(x |=> 1.1530778982334484),y |=> 1
  dr3 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: cls |=> false
      currState: init]
    env ->[
      variables: (velx |=> 3.6510256418611448e+2),(vely |=>
    -2.3461050883275701e+2),(x |=> 1.7046496106089672),y |=> 1
  dr2 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: cls |=> false
      currState: init]
    env ->[
      variables: (velx |=> 3.6510256418611448e+2),(vely |=>
    -2.3461050883275701e+2),(x |=> -1.7812118987974737),y |=>
  dr4 ->[
    (ctrl . ctrlProc . cThread) ->[
      variables: cls |=> false
      currState: init]
    env ->[
      variables: (velx |=> 3.6510256418611448e+2),(vely |=>
    -2.3461050883275701e+2),(x |=> -1.1848107009392255),y |=>

Time: 100
FourDronesSystemimplInstance ->[
  dr1 ->[
    (ctrl . ctrlProc . cThread) ->[
```

HybridSynchAADL Result

| PSPC | Property Id | Result | Method | CPUTime | RunningTime | Location |
|---|---|---|---|---|---|---|
| Fo... | safety | Counterexample f... | random | 172ms | 173ms | FourDro |
| Fo... | rendezvous | Reachable | symbolic | 1988ms | 1990ms | FourDro |

**Figure 25** Results for the two properties.

## 8.3 Formal Analysis

We consider two properties of the drone model: drones do not collide (`safety`), and 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 portfolio analysis.

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

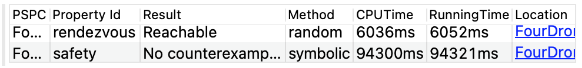
We define three atomic propositions for four drones `dr1`, `dr2`, `dr3`, and `dr4` as follows. Two drones collide if the (horizontal and vertical) distance between them is less than 0.1. All nodes have *gathered* if each pairwise distance is less than 2.0. There are infinitely many initial states satisfying the proposition `initial`.

```
proposition [collision]:
   (abs(dr1.env.x - dr2.env.x) < 0.1 and
      abs(dr1.env.y - dr2.env.y) < 0.1) or
   (abs(dr1.env.x - dr3.env.x) < 0.1 and
      abs(dr1.env.y - dr3.env.y) < 0.1) or
   ...
   (abs(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) < 2.0 and
   abs(dr1.env.y - dr2.env.y) < 2.0 and
   abs(dr1.env.x - dr3.env.x) < 2.0 and
   abs(dr1.env.y - dr3.env.y) < 2.0 and
   ...
   abs(dr3.env.x - dr4.env.x) < 2.0 and
   abs(dr3.env.y - dr4.env.y) < 2.0;

proposition [initial]:
   abs(dr1.env.x - 1.2) < 0.1 and
   abs(dr1.env.y - 1.7) < 0.1 and
   abs(dr2.env.x + 1.7) < 0.1 and
   abs(dr2.env.y + 1.2) < 0.1 and
   abs(dr3.env.x - 1.7) < 0.1 and
   abs(dr3.env.y - 1.2) < 0.1 and
   abs(dr4.env.x + 1.2) < 0.1 and
   abs(dr4.env.y + 1.7) < 0.1;
```

The result of the analysis is shown in Figure 25. A witness for `rendezvous` is found by symbolic analysis in 2.0 seconds, and a counterexample is found for `safety` by randomized simulation in 0.2 seconds. A counterexample of `safety` is shown in the editor as a sequence of states. The invariant may have been violated because the drones can have any speed in the initial state.



| PSPC | Property Id | Result | Method | CPUTime | RunningTime | Location |
|---|---|---|---|---|---|---|
| Fo... | rendezvous | Reachable | random | 6036ms | 6052ms | FourDro |
| Fo... | safety | No counterexamp... | symbolic | 94300ms | 94321ms | FourDro |

**Figure 26** Results for the modified properties.

We modify `safety` and `rendezvous` by adding a constraint `velconst` to the initial condition. As shown in Figure 26, symbolic reachability analysis verifies that all possible behaviors up to the bound satisfy this *modified* `safety` property.

```
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) <= 1 and abs(dr1.env.vy) <= 1 and
  abs(dr2.env.vx) <= 1 and abs(dr2.env.vy) <= 1 and
  abs(dr3.env.vx) <= 1 and abs(dr3.env.vy) <= 1 and
  abs(dr4.env.vx) <= 1 and abs(dr4.env.vy) <= 1;
```

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

## 9 Experimental Evaluation

This section evaluates the HybridSynchAADL tool by addressing the following questions mentioned in the introduction:

1. How effective is our symbolic analysis method compared to other state-of-the-art formal analysis tools for CPSs?
2. How effective is our portfolio analysis method for finding bugs?
3. How effective is our state merging technique?
4. How effective is Hybrid PALS for reducing the complexity of model checking virtually synchronous CPSs?

To answer these questions, we have analyzed HybridSynchAADL models of networked water tank and thermostat controllers (adapted from [37–39]), and rendezvous and formation control for distributed drones. Many variants of these models are considered: different numbers of components, single-integrator and double-integrator dynamics, different sampling and actuating times, etc.

We have run all experiments on Intel Xeon 2.8GHz with 256 GB memory. For our tool, we use a specialized version of Maude with Yices 2.6 for nonlinear polynomial arithmetic [40]. We have repeated each experiment 10 times (with different random seeds) and report the average results. The models and the experimental results are available at https://hybridsynchaadl.github.io/sttt.

### 9.1 Comparing with Other Tools

We compare our symbolic reachability analysis method with four reachability analysis tools for hybrid automata: HyComp [41], SpaceEx [42], Flow* [43], and dReach [44]. For these tools, we have "encoded" the *synchronous designs* of the HybridSynchAADL models as networks of hybrid automata. Each (drone, water tank, and thermostat) component is modeled as a hybrid automaton with three modes, where the behavior of a controller is encoded as a single transition.

Figure 27 shows a hybrid automaton for a single component of drone rendezvous with single-integrator dynamics. It specifies the behavior of an environment component (see Section 6), where sampling occurs by `sampling` transitions, and controller transition/actuation occurs by `actuation` transitions.[11] All components are synchronized by transition `synch`, which assigns the output $(x_{out}, y_{out})$ to the input $(x_{in}, y_{in})$ of the connected component. For Flow* and dReach, which do not support networks of hybrid automata, we build flat hybrid automata using HYST [45].

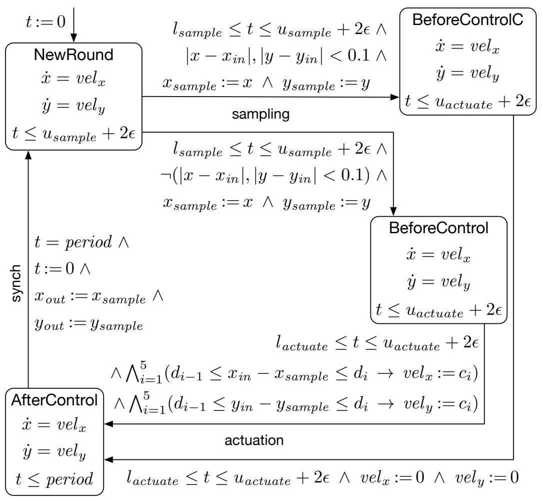
We measure the execution times for analyzing invariant properties *up to* bound $1,000$ ms, with a timeout of 120 minutes. We consider two invariant properties for each model: $Inv_\top$, which holds, and $Inv_\bot$, which does not hold. For SpaceEx, we use PHAVer for linear dynamics, and STC for nonlinear polynomial dynamics. For Flow*, we use adaptive steps, and TM orders 1 (for single) and 2 (for double). We use the default precision for dReach, and BMC for HyComp.

The experimental results are summarized in Table 1, as execution times (seconds) over time bounds $(B \cdot 100\,\text{ms})$, where $N$ denotes the number of components. The results for double-integrator

[11] We use equivalent control logic for both HybridSynchAADL models and hybrid automata models. For the drone rendezvous models, the control logics used in this experiment are simplified from one presented in Section 8.

**Table 1** HybridSynchAADL vs. HyComp, SpaceEx, dReach, and Flow* (`T/O` denotes a timeout).

| | | $Inv_\top$ | | | | | | | | $Inv_\bot$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Model | Tool | $N=2$ | | $N=3$ | | $N=4$ | | $N=5$ | | $N=2$ | | $N=3$ | | $N=4$ | | $N=5$ | |
| | | Time | $B$ | Time | $B$ | Time | $B$ | Time | $B$ | Time | $B$ | Time | $B$ | Time | $B$ | Time | $B$ |
| Thermostat | `HSAADL` | 297.1 | 10 | 410.5 | 8 | 186.8 | 7 | 260.5 | 7 | 31.4 | 7 | 28.7 | 6 | 6.3 | 4 | 8.6 | 4 |
| | `HyComp` | 278.9 | 10 | 1,529.3 | 10 | 1,388.6 | 4 | 3,020.5 | 4 | 8.7 | 7 | 29.7 | 6 | 34.0 | 4 | 57.7 | 4 |
| | `SpaceEx` | 5,150.3 | 8 | 5,279.1 | 2 | 9.4 | 1 | 1,329.7 | 1 | 8.2 | 7 | `T/O` | - | `T/O` | - | `T/O` | - |
| | `dReach` | 532.9 | 3 | 126.4 | 1 | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - |
| | `Flow*` | 590.2 | 3 | 95.1 | 1 | 2,827.5 | 1 | `T/O` | - | 56.2 | 5 | `T/O` | - | `T/O` | - | `T/O` | - |
| Water Tank | `HSAADL` | 9.1 | 10 | 20.1 | 10 | 40.0 | 10 | 67.9 | 10 | 4.8 | 8 | 3.5 | 5 | 9.4 | 6 | 9.3 | 5 |
| | `HyComp` | 8.1 | 10 | 27.9 | 10 | 129.1 | 10 | 270.4 | 10 | 9.7 | 8 | 22.9 | 5 | 122.6 | 6 | 280.4 | 5 |
| | `SpaceEx` | 36.2 | 10 | 6,970.1 | 4 | 4,839.2 | 1 | `T/O` | - | 2.0 | 8 | `T/O` | - | `T/O` | - | `T/O` | - |
| | `dReach` | 340.1 | 2 | 101.7 | 1 | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - |
| | `Flow*` | 92.0 | 2 | 286.7 | 1 | `T/O` | - | `T/O` | - | 30.1 | 5 | 1,520.9 | 2 | `T/O` | - | `T/O` | - |
| Rend (single) | `HSADDL` | 32.9 | 8 | 40.2 | 6 | 24.6 | 5 | 37.3 | 5 | 12.1 | 4 | 6.1 | 3 | 21.5 | 4 | 40.5 | 4 |
| | `HyComp` | 3,584.2 | 8 | 2801.9 | 6 | 3320.6 | 5 | 225.1 | 3 | 12.9 | 4 | 25.7 | 3 | `T/O` | - | `T/O` | - |
| | `SpaceEx` | 11.1 | 1 | 18.5 | 1 | 401.5 | 1 | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - |
| | `dReach` | 185.9 | 1 | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - |
| | `Flow*` | 2271.9 | 1 | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | 824.1 | 2 | `T/O` | - | `T/O` | - |
| Form (single) | `HSAADL` | 155.9 | 9 | 73.8 | 7 | 1,623.2 | 8 | 304.1 | 7 | 1.7 | 2 | 2.8 | 2 | 59.8 | 3 | 23.2 | 3 |
| | `HyComp` | 528.3 | 8 | 1,538.9 | 5 | 3,267.3 | 5 | 1,248.0 | 3 | 10.9 | 2 | 29.4 | 2 | `T/O` | - | `T/O` | - |
| | `SpaceEx` | 820.1 | 1 | `T/O` | - | `T/O` | - | `T/O` | - | 27.1 | 1 | `T/O` | - | `T/O` | - | `T/O` | - |
| | `dReach` | 197.9 | 1 | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - |
| | `Flow*` | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - |
| Rend (double) | `HSAADL` | 4.5 | 3 | 7.9 | 3 | 5.0 | 2 | 6.8 | 2 | 6.5 | 2 | 3.7 | 2 | 5.5 | 2 | 7.6 | 2 |
| | `SpaceEx` | 290.1 | 1 | `T/O` | - | `T/O` | - | `T/O` | - | 23.1 | 1 | `T/O` | - | `T/O` | - | `T/O` | - |
| | `dReach` | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - |
| | `Flow*` | 700.2 | 1 | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - |
| Form (double) | `HSAADL` | 8.2 | 3 | 5.4 | 2 | 7.3 | 2 | 9.5 | 2 | 6.8 | 2 | 5.4 | 2 | 14.1 | 2 | 10.1 | 2 |
| | `SpaceEx` | 6,237.3 | 1 | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - |
| | `dReach` | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - |
| | `Flow*` | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - | `T/O` | - |



$t := 0$
NewRound
$\dot{x} = vel_x$
$\dot{y} = vel_y$
$t \leq u_{sample} + 2\epsilon$
$l_{sample} \leq t \leq u_{sample} + 2\epsilon \wedge$
$|x - x_{in}|, |y - y_{in}| < 0.1 \wedge$
$x_{sample} := x \wedge y_{sample} := y$
sampling
BeforeControlC
$\dot{x} = vel_x$
$\dot{y} = vel_y$
$t \leq u_{actuate} + 2\epsilon$
$l_{sample} \leq t \leq u_{sample} + 2\epsilon \wedge$
$\neg(|x - x_{in}|, |y - y_{in}| < 0.1) \wedge$
$x_{sample} := x \wedge y_{sample} := y$
BeforeControl
$\dot{x} = vel_x$
$\dot{y} = vel_y$
$t \leq u_{actuate} + 2\epsilon$
$t = period \wedge$
$t := 0 \wedge$
$x_{out} := x_{sample} \wedge$
$y_{out} := y_{sample}$
synch
$l_{actuate} \leq t \leq u_{actuate} + 2\epsilon$
$\wedge \bigwedge_{i=1}^{5}(d_{i-1} \leq x_{in} - x_{sample} \leq d_i \rightarrow vel_x := c_i)$
$\wedge \bigwedge_{i=1}^{5}(d_{i-1} \leq y_{in} - y_{sample} \leq d_i \rightarrow vel_y := c_i)$
AfterControl
$\dot{x} = vel_x$
$\dot{y} = vel_y$
$t \leq period$
actuation
$l_{actuate} \leq t \leq u_{actuate} + 2\epsilon \wedge vel_x := 0 \wedge vel_y := 0$

**Figure 27** A hybrid automaton component.

dynamics do not include HyComp, which does not support nonlinear polynomial dynamics. For $Inv_\top$, Table 1 shows the largest time bound for which the tool could prove the absence of counterexamples. Often, *tools timed out when trying to verify that $Inv_\top$ holds up to time bound 500.* For $Inv_\bot$, the table shows the smallest bound for which the tool found counterexamples.

As shown in Table 1, HybridSynchAADL outperforms the other tools in most cases, in particular for complex models with larger $N$. Consider "Rend (single)" (rendezvous with single-integrator dynamics) with $N = 4$. For $Inv_\top$, HybridSynchAADL needs 24.6 seconds for $B = 5$, whereas SpaceEx needs 401.5 seconds for $B = 1$ and timed out for $B > 1$. For $Inv_\bot$, HybridSynchAADL found a counterexample at $B = 4$ in 21.5 seconds, whereas all other tools timed out. It is worth noting that Flow* occasionally found (spurious) counterexamples at smaller bounds, because of over-approximation by the Taylor model flowpipe construction.

### 9.1.1 Fairness of the Comparison

How trustworthy and fair are our comparisons with the other tools? The "hand-coded" models for the other tools used in the experiments may be overly complex or behave differently from the original HybridSynchAADL models. To achieve a fair comparison, we have manually "optimized" the hand-coded models to minimize the number of modes and transitions while maintaining the same synchronous behavior as the corresponding HybridSynchAADL models.

For example, we have also defined a hybrid automaton model shown in Figure 28 for drone rendezvous with single integrator dynamics, which includes a "direct" representation of the behavior actions in the original model. This model has more modes and transitions than the hand-optimized model in Figure 27 used in the experiments, which results in worse performance, e.g., as shown in Table 2 for analyzing $Inv_\top$ using SpaceEx.[12]

We have also closely inspected both the models and the analysis results to ensure that such hand-optimized models behave as the corresponding HybridSynchAADL models. As discussed in Section 7.4, the results of analyzing the HybridSynchAADL models are consistent with the results of analyzing the hand-coded models using the other tools; in particular, a counterexample in one model is also a counterexample in the corresponding model.

## 9.2 Effect of Portfolio Analysis

We evaluate the power of HybridSynchAADL for analyzing invariant properties. We measure the time taken to find counterexamples in "faulty" models obtained by modifying the sampling and actuating times using HybridSynchAADL's three analysis functions. We use different time bounds for observing analysis results with varying time bounds. Table 3 summarizes the time parameters

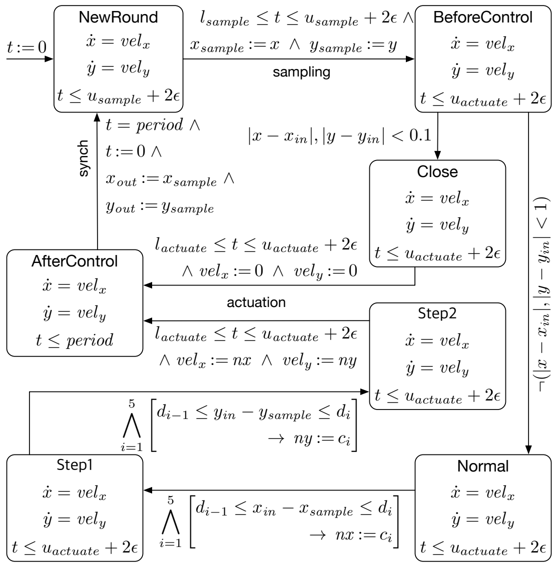

NewRound
$\dot{x} = vel_x$
$\dot{y} = vel_y$
$t \leq u_{sample} + 2\epsilon$
$t := 0$
$l_{sample} \leq t \leq u_{sample} + 2\epsilon \wedge$
$x_{sample} := x \wedge y_{sample} := y$
sampling
BeforeControl
$\dot{x} = vel_x$
$\dot{y} = vel_y$
$t \leq u_{actuate} + 2\epsilon$
$t = period \wedge$
$t := 0 \wedge$
$x_{out} := x_{sample} \wedge$
$y_{out} := y_{sample}$
synch
$|x - x_{in}|, |y - y_{in}| < 0.1$
Close
$\dot{x} = vel_x$
$\dot{y} = vel_y$
$t \leq u_{actuate} + 2\epsilon$
$\neg(|x - x_{in}|, |y - y_{in}| < 1)$
AfterControl
$\dot{x} = vel_x$
$\dot{y} = vel_y$
$t \leq period$
$l_{actuate} \leq t \leq u_{actuate} + 2\epsilon$
$\wedge\ vel_x := 0 \wedge vel_y := 0$
actuation
$l_{actuate} \leq t \leq u_{actuate} + 2\epsilon$
$\wedge\ vel_x := nx \wedge vel_y := ny$
Step2
$\dot{x} = vel_x$
$\dot{y} = vel_y$
$t \leq u_{actuate} + 2\epsilon$
$\bigwedge_{i=1}^{5} [d_{i-1} \leq y_{in} - y_{sample} \leq d_i \rightarrow ny := c_i]$
Step1
$\dot{x} = vel_x$
$\dot{y} = vel_y$
$t \leq u_{actuate} + 2\epsilon$
$\bigwedge_{i=1}^{5} [d_{i-1} \leq x_{in} - x_{sample} \leq d_i \rightarrow nx := c_i]$
Normal
$\dot{x} = vel_x$
$\dot{y} = vel_y$
$t \leq u_{actuate} + 2\epsilon$

**Figure 28** A complex hybrid automaton component.

**Table 2** Analyzing $Inv_\top$ using SpaceEx

| Model | | Figure 27 | | Figure 28 | |
|---|---|---|---|---|---|
| | N | Time | $B$ | Time | $B$ |
| Thermostat | 5 | 79.5 | 1 | 4,910.8 | 1 |
| WaterTank | 5 | 2,748.9 | 1 | T/O | - |
| Rend (single) | 4 | 505.5 | 1 | T/O | - |
| Form (single) | 3 | 507.9 | 1 | 730.4 | 1 |
| Rend (double) | 2 | 253.6 | 1 | T/O | - |
| Form (double) | 2 | 6,027.6 | 1 | T/O | - |

and invariant properties, with $\epsilon$ the maximal clock skew. All models have period 100 (milliseconds). The timeout is 20 minutes.

Table 4 shows the experimental results. For symbolic analysis (symbolic), once a counterexample is found (FO) for one bound $T$, the results for larger bounds $T' \geq T$ are exactly the same as for $T$, since symbolic analysis uses a breadth-first strategy. For randomized simulation (random), a timeout means that a counterexample is not found (NF) by repeated simulations until timed out. The results for portfolio analysis, which are the minimum values of randomized simulation and symbolic analysis results, are shown in yellow.

As expected, symbolic analysis is effective for finding subtle counterexamples, and randomized simulation is effective for finding obvious bugs. Since the injected faults are caused by excessive sampling and actuating times, errors are easier to find with a larger bound. As an example, see

[12]To reduce time-outs when analyzing the complex models, we use narrower initial conditions than those used for Table 1.

**Table 3** Timing parameters and properties.

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

"Form (single)" with $N = 4$. For $B = 3$, symbolic analysis found a counterexample in less than 24 seconds while randomized simulation timed out. However, for $B = 4$, randomized simulation found a counterexample in less than 15 seconds. Portfolio analysis is effective in both cases.

## 9.3 Effect of State Merging

We have performed symbolic analysis to generate all reachable symbolic states up to given bounds, with and without state merging. We measure the execution time (seconds), the size of accumulated SMT formulas (thousands), the number of calls to the SMT solver, and the number of reachable symbolic states, with a timeout of 3 hours. The results are summarized in Table 5.

As shown, state merging significantly improves the performance of symbolic analysis. Symbolic analysis with state merging always generates a single symbolic state for each synchronous step, whereas analysis without merging may generate many symbolic states. As a result, state merging involves much smaller SMT constraints and fewer SMT calls than without state merging.

Consider, e.g., "Thermostat" with $N = 3$ for $B = 2$. With state merging, the analysis took 0.6 seconds, where the size of the constraints is around 31,200 and the number of calls is 218. Without merging, the analysis took 4,054.8 seconds, where the size of the constraints is around 396 million, and the number of calls is around 2 million.

## 9.4 Effect of Hybrid PALS

To evaluate the complexity reduction provided by Hybrid PALS, we compare model checking of synchronous designs with model checking of the corresponding asynchronous models. We have therefore also developed an asynchronous semantics for HybridSynchAADL models (adapted from the semantics of AADL in [34]). In the experiments, we consider *highly simplified* distributed models, where:

- all clocks are perfectly synchronized;
- controller execution takes zero time;
- there is no network delay; and
- sampling/actuating times are chosen from predefined values nondeterministically.

We measure the execution times for generating all reachable *concrete* states up to given bounds in synchronous designs and asynchronous models, with a timeout of 6 hours, To generate concrete states for synchronous designs, we use concrete sampling and actuating times in a way similar to randomized simulation, but we choose them nondeterministically from predefined values.

The results are shown in Table 6, with |Sample| the number of predefined sampling/actuating times. As seen, the number of reachable states can be very large, even for very simple distributed models. For "Form (single)" with $N = 2$, $B = 1$, and $|\text{Sample}| = 1$, the number of reachable states is more than 3.8 million. It took more than 2.2 hours to generate these states, whereas model checking of the synchronous model needed less than 1 second for the same case.

# 10 Related Work

***Reachability Analysis Tools for CPSs***

One distinguishing feature of our work is that we perform model checking verification of virtually synchronous CPSs with typical CPS features such as advanced control programs, continuous behaviors, communication delays, execution times, and clock skews, whereas most formal frameworks are strong at analyzing either discrete or continuous behaviors. The latter class includes reachability analysis tools for (networks of "finite-location") hybrid automata such as SpaceEx [42], HyComp [41], and dReach [44], which do not deal well with the "discrete complexity" (e.g., complex control programs) of CPSs. In addition, HybridSynchAADL can easily specify and analyze both continuous dynamics and imprecise local clocks.

**Table 4** HybridSynchAADL portfolio analysis (T/O denotes a timeout).

| Model | N | Method | Time (s) / Result | | | | |
|---|---|---|---|---|---|---|---|
| | | | $B = 1$ | $B = 2$ | $B = 3$ | $B = 4$ | $B = 5$ |
| Thermostat | 2 | random | T/O / - | T/O / - | 19.1 / FO | 0.04 / FO | 0.03 / FO |
| | | symbolic | 0.2 / NF | 0.5 / NF | 1.0 / FO | 1.0 / FO | 1.0 / FO |
| | 3 | random | T/O / - | T/O / - | T/O / - | 5.4 / FO | 0.5 / FO |
| | | symbolic | 0.3 / NF | 0.7 / NF | 1.6 / NF | 3.3 / FO | 3.3 / FO |
| | 4 | random | T/O / - | T/O / - | T/O / - | 2.8 / FO | 1.6 / FO |
| | | symbolic | 0.4 / NF | 1.0 / NF | 2.4 / NF | 5.7 / FO | 5.7 / FO |
| Water Tank | 2 | random | T/O / - | T/O / - | 73.0 / FO | 0.04 / FO | 0.03 / FO |
| | | symbolic | 0.2 / NF | 0.4 / NF | 0.7 / FO | 0.7 / FO | 0.7 / FO |
| | 3 | random | T/O / - | T/O / - | 96.9 / FO | 0.1 / FO | 0.04 / FO |
| | | symbolic | 0.3 / NF | 0.6 / NF | 1.2 / FO | 1.2 / FO | 1.2 / FO |
| | 4 | random | T/O / - | 0.2 / FO | 0.1 / FO | 0.1 / FO | 0.1 / FO |
| | | symbolic | 0.4 / NF | 0.9 / FO | 0.9 / FO | 0.9 / FO | 0.9 / FO |
| Rend (single) | 2 | random | T/O / - | T/O / - | T/O / - | 11.2 / FO | 0.3 / FO |
| | | symbolic | 0.6 / NF | 2.1 / NF | 4.3 / FO | 4.3 / FO | 4.3 / FO |
| | 3 | random | T/O / - | T/O / - | 9.7 / FO | 0.4 / FO | 0.1 / FO |
| | | symbolic | 0.8 / NF | 2.8 / NF | 6.1 / FO | 6.1 / FO | 6.1 / FO |
| | 4 | random | T/O / - | 118.8 / FO | 0.4 / FO | 0.1 / FO | 0.1 / FO |
| | | symbolic | 1.1 / NF | 3.9 / FO | 3.9 / FO | 3.9 / FO | 3.9 / FO |
| Form (single) | 2 | random | T/O / - | T/O / - | 657.4 / FO | 2.4 / FO | 1.3 / FO |
| | | symbolic | 0.8 / NF | 2.2 / NF | 8.0 / FO | 8.0 / FO | 8.0 / FO |
| | 3 | random | T/O / - | 9.8 / FO | 2.7 / FO | 1.4 / FO | 0.6 / FO |
| | | symbolic | 1.0 / NF | 3.4 / FO | 3.4 / FO | 3.4 / FO | 3.4 / FO |
| | 4 | random | T/O / - | T/O / - | T/O / - | 14.4 / FO | 2.6 / FO |
| | | symbolic | 1.5 / NF | 5.6 / NF | 23.4 / FO | 23.4 / FO | 23.4 / FO |
| Rend (double) | 2 | random | T/O / - | 112.6 / FO | 0.1 / FO | 0.1 / FO | 0.1 / FO |
| | | symbolic | 0.8 / NF | T/O / - | T/O / - | T/O / - | T/O / - |
| | 3 | random | T/O / - | 118.5 / FO | 0.1 / FO | 0.1 / FO | 0.1 / FO |
| | | symbolic | 1.3 / NF | 4.0 / FO | 4.0 / FO | 4.0 / FO | 4.0 / FO |
| | 4 | random | T/O / - | 43.5 / FO | 0.1 / FO | 0.1 / FO | 0.1 / FO |
| | | symbolic | 1.8 / NF | 5.7 / FO | 5.7 / FO | 5.7 / FO | 5.7 / FO |
| Form (double) | 2 | random | T/O / - | 33.1 / - | 0.8 / FO | 0.7 / FO | 0.7 / FO |
| | | symbolic | 1.2 / NF | T/O / - | T/O / - | T/O / - | T/O / - |
| | 3 | random | T/O / - | 81.4 / FO | 1.4 / FO | 0.3 / FO | 0.2 / FO |
| | | symbolic | 1.6 / NF | 5.7 / FO | 5.7 / FO | 5.7 / FO | 5.7 / FO |
| | 4 | random | T/O / - | 2.0 / FO | 0.2 / FO | 0.2 / FO | 0.2 / FO |
| | | symbolic | 2.3 / NF | 10.7 / FO | 10.7 / FO | 10.6 / FO | 10.6 / FO |

**Table 5** Symbolic analysis with merging and without merging (Time in seconds, and |Const| in thousands).

| Model | N | B | Symbolic with merging | | | | Symbolic without merging | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Time | \|Const\| | #Call | #State | Time | \|Const\| | #Call | #State |
| Thermostat | 2 | 1 | 0.2 | 4.3 | 63 | 2 | 0.6 | 53.3 | 739 | 10 |
| | | 2 | 0.4 | 18.3 | 147 | 3 | 26.5 | 3,145.6 | 20,757 | 86 |
| | | 3 | 0.8 | 44.4 | 231 | 4 | 552.9 | 54,858.7 | 225,308 | 331 |
| | | 4 | 1.7 | 82.7 | 525 | 5 | 3,839.6 | 325,659.1 | 983,691 | 1,035 |
| | 3 | 1 | 0.2 | 7.2 | 93 | 2 | 14.6 | 13,333.1 | 15,331 | 64 |
| | | 2 | 0.6 | 31.2 | 218 | 3 | 4,054.8 | 396,093.6 | 1,987,483 | 1,828 |
| | | 3 | 1.4 | 79.1 | 343 | 4 | T/O | - | - | - |
| | 4 | 1 | 0.3 | 10.1 | 123 | 2 | 382.9 | 32,706.3 | 321,595 | 442 |
| | | 2 | 0.9 | 46.2 | 289 | 3 | T/O | - | - | - |
| Water Tank | 2 | 1 | 0.2 | 4.5 | 55 | 2 | 2.7 | 246.7 | 2,754 | 37 |
| | | 2 | 0.4 | 17.8 | 132 | 3 | 145.7 | 18,272.3 | 101,898 | 158 |
| | | 3 | 0.7 | 40.9 | 209 | 4 | 836.9 | 111,454.8 | 435,132 | 588 |
| | | 4 | 1.1 | 74.1 | 286 | 5 | 4,239.9 | 567,051.3 | 1,619,352 | 2,006 |
| | 3 | 1 | 0.2 | 6.9 | 82 | 2 | 65.9 | 5,727.3 | 51,534 | 217 |
| | | 2 | 0.6 | 30.6 | 197 | 3 | T/O | - | - | - |
| | 4 | 1 | 0.3 | 9.2 | 109 | 2 | 1,563.3 | 123,763.2 | 939,078 | 1,297 |
| | | 2 | 0.9 | 45.8 | 262 | 3 | T/O | - | - | - |
| Rend (single) | 2 | 1 | 0.5 | 38.1 | 293 | 2 | 14.0 | 1,205.5 | 9,993 | 677 |
| | | 2 | 1.5 | 133.8 | 585 | 3 | 4,867.7 | 370,978.8 | 1,795,183 | 15,095 |
| | | 3 | 2.9 | 299.6 | 877 | 4 | T/O | - | - | - |
| | 3 | 1 | 0.7 | 57.5 | 438 | 2 | 481.5 | 38,865.8 | 260,113 | 17,577 |
| | | 2 | 2.3 | 227.2 | 875 | 3 | T/O | - | - | - |
| | 4 | 1 | 1.0 | 76.2 | 583 | 2 | T/O | - | - | - |
| Form (single) | 2 | 1 | 0.7 | 50.7 | 328 | 2 | 18.1 | 1,519.9 | 10,028 | 677 |
| | | 2 | 2.1 | 184.6 | 655 | 3 | T/O | - | - | - |
| | 3 | 1 | 1.0 | 75.6 | 473 | 2 | 617.2 | 47,029.3 | 260,148 | 17,577 |
| | | 2 | 3.1 | 302.9 | 945 | 3 | T/O | - | - | - |
| | 4 | 1 | 1.3 | 100.4 | 618 | 2 | T/O | - | - | - |
| Rend (double) | 2 | 1 | 0.7 | 49.7 | 293 | 2 | 21.1 | 1,646.5 | 9,993 | 677 |
| | | 2 | 2.2 | 177.8 | 585 | 3 | T/O | - | - | - |
| | 3 | 1 | 1.1 | 74.5 | 438 | 2 | 731.5 | 52,137.0 | 260,113 | 17,577 |
| | | 2 | 3.5 | 301.6 | 875 | 3 | T/O | - | - | - |
| | 4 | 1 | 1.4 | 99.3 | 583 | 2 | T/O | - | - | - |
| Form (double) | 2 | 1 | 1.0 | 67.5 | 328 | 2 | 22.0 | 1,734.3 | 9,560 | 443 |
| | | 2 | 3.3 | 250.1 | 655 | 3 | T/O | - | - | - |
| | 3 | 1 | 1.5 | 100.7 | 473 | 2 | 634.2 | 41,306.8 | 173,100 | 11,493 |
| | | 2 | 5.2 | 408.8 | 945 | 3 | T/O | - | - | - |
| | 4 | 1 | 2.0 | 133.8 | 618 | 2 | T/O | - | - | - |

**Table 6** Analyzing distributed asynchronous models (Time in seconds, and #State in thousands).

| | | | Synchronous Models | | | | | | Asynchronous Models | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Model | N | B | \|Sample\| = 1 | | \|Sample\| = 2 | | \|Sample\| = 3 | | \|Sample\| = 1 | | \|Sample\| = 2 | | \|Sample\| = 3 | |
| | | | Time | #State | Time | #State | Time | #State | Time | #State | Time | #State | Time | #State |
| Thermostat | 2 | 1 | 0.01 | 0.02 | 0.02 | 0.1 | 0.1 | 0.2 | 5.3 | 10.0 | 55.4 | 104.9 | 343.5 | 564.2 |
| | | 2 | 0.01 | 0.03 | 0.1 | 0.3 | 0.9 | 2.6 | 6.6 | 13.0 | 217.3 | 471.0 | T/O | - |
| | | 3 | 0.01 | 0.05 | 0.7 | 2.1 | 30.1 | 78.0 | 9.4 | 19.2 | 1,368.9 | 1,799.2 | T/O | - |
| | 3 | 1 | 0.01 | 0.03 | 0.1 | 0.2 | 0.3 | 0.7 | 4,424.2 | 1,812.7 | T/O | - | T/O | - |
| Water Tank | 2 | 1 | 0.005 | 0.01 | 0.02 | 0.1 | 0.1 | 0.2 | 5.4 | 9.9 | 55.5 | 101.8 | 368.7 | 619.6 |
| | | 2 | 0.01 | 0.03 | 0.4 | 0.8 | 5.5 | 9.6 | 6.4 | 12.2 | 113.6 | 226.0 | 8,975.7 | 4,854.8 |
| | | 3 | 0.02 | 0.04 | 1.5 | 2.9 | 109.2 | 167.9 | 7.5 | 14.5 | 220.1 | 417.3 | T/O | - |
| | 3 | 1 | 0.01 | 0.02 | 0.1 | 0.2 | 0.7 | 1.5 | 4,624.3 | 1,796.9 | T/O | - | T/O | - |
| Rend (single) | 2 | 1 | 0.01 | 0.03 | 0.02 | 0.1 | 0.1 | 0.2 | 6.0 | 10.7 | 53.5 | 90.5 | 251.4 | 393.0 |
| | | 2 | 0.01 | 0.05 | 0.2 | 0.6 | 1.2 | 4.1 | 9.7 | 19.4 | 73.1 | 135.2 | 317.5 | 528.8 |
| | | 3 | 0.02 | 0.07 | 2.6 | 8.9 | 100.7 | 297.9 | 15.0 | 31.1 | 107.0 | 208.1 | 447.7 | 769.5 |
| | 3 | 1 | 0.01 | 0.04 | 0.1 | 0.2 | 0.2 | 0.5 | 970.4 | 939.7 | T/O | - | T/O | - |
| Form (single) | 2 | 1 | 0.01 | 0.03 | 0.05 | 0.1 | 0.2 | 0.3 | 7,937.9 | 3,888.4 | T/O | - | T/O | - |
| | 3 | 1 | 0.02 | 0.05 | 0.1 | 0.3 | 0.5 | 0.9 | T/O | - | T/O | - | T/O | - |
| Rend (double) | 2 | 1 | 0.01 | 0.02 | 0.03 | 0.1 | 0.1 | 0.1 | 145.1 | 188.2 | 1,557.6 | 1,500.6 | 15,348.1 | 6,339.2 |
| | | 2 | 0.02 | 0.04 | 0.1 | 0.2 | 0.6 | 8.6 | 826.3 | 1,121.8 | 10,200.0 | 5,495.6 | T/O | - |
| | | 3 | 0.03 | 0.07 | 0.9 | 1.7 | 12.9 | 24.8 | 2,773.4 | 2,764.0 | T/O | - | T/O | - |
| | 3 | 1 | 0.01 | 0.03 | 0.1 | 0.1 | 0.2 | 0.3 | T/O | - | T/O | - | T/O | - |
| Form (double) | 2 | 1 | 0.02 | 0.03 | 0.1 | 0.1 | 0.1 | 0.1 | T/O | - | T/O | - | T/O | - |
| | 3 | 1 | 0.03 | 0.04 | 0.1 | 0.2 | 0.4 | 0.4 | T/O | - | T/O | - | T/O | - |

### *Hybrid Systems in AADL*

The *Hybrid Annex* for AADL [30] allows modeling continuous behaviors in AADL. Its developers also provide theorem proving support for proving properties in Hoare Logic combined with Duration Calculus [46]. Controller behaviors are defined in Hybrid CSP. Only a "synchronous" subset of AADL is considered: clock skews, message delays, and execution times are not taken into account. In contrast, we use AADL's expressive Behavior Annex to specify controller behaviors. HybridSynchAADL provides automatic model checking analysis instead of interactive theorem proving. Moreover, we consider (virtually synchronous) CPSs—with clock skews, network delays, etc.—using the Hybrid PALS equivalence.

In [47], an *Uncertainty Annex* is added to the Hybrid Annex. Uncertain Hybrid AADL models can be transformed into networks of priced timed automata that can then be subjected to statistical model checking using Uppaal-SMC to evaluate the *performance* of the models. Another hybrid annex is proposed in [31], and an AADL sublanguage, called AADL+, where continuous behaviors can be defined using stochastic differential equations, is given in [48]. Both approaches come with some kind of operational semantics and simulation, but with no formal analysis support.

### *PALS and AADL*

*Synchronous AADL* [33, 49] and its multi-rate extension [32] support the modeling and analysis of synchronous PALS models of virtually synchronous distributed real-time systems without continuous behaviors in AADL. Since the time when an event takes place can be abstracted away, there is no need to consider clock skews, and any (sufficiently expressive) explicit-state model checker, such as Maude, can be used.

In contrast, HybridSynchAADL must model continuous behaviors and clock skews, and must analyze all possible behaviors based on when the continuous components are sampled and actuated, which depend on the imprecise local clocks. This required us to leave the explicit-state world and use Maude with SMT solving. In this way, we can cover all possible behaviors, but are currently restricted to reachability analysis.

### *Formal Analysis of Hybrid PALS Models*

The paper [10] shows how synchronous Hybrid PALS models with simple finite-state machine controllers—and their reachability problems—can be encoded as logical formulas and analyzed by the dReal solver for nonlinear theories over the real numbers. However, there is no tool support in [10], and it is difficult to model complex CPSs in SMT. In contrast, our approach provides a simple and intuitive way of modeling synchronous Hybrid PALS models using a well-known modeling standard. In addition, since we use Maude with SMT solving instead of just SMT solving, we can also analyze cyber-physical systems involving complex control programs and data types.

### *Almost-Synchronous Systems*

Our work is also related to a broader body of work on modeling and analyzing "almost-synchronous" systems, including quasi-synchrony [21, 50–52], GALS [53, 54], virtual synchrony [8, 10, 20], time-triggered architectures [19, 22], approximate synchrony [55], etc. A common theme of these approaches is to simplify the design and verification of distributed real-time systems using various synchronization methods. Our method makes it possible to model and verify almost-synchronous systems *with continuous behaviors*, including of *continuous behaviors perturbed by clock skews*, and sampling/actuation times, which are typically not considered in related work. We also provide a convenient language and modeling environment for modeling almost-synchronous CPSs.

## 11 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 define the formal semantics of the HybridSynchAADL modeling language, and of our tool's analysis commands, using Maude combined with SMT solving. We have developed and implemented a number of optimization techniques to improve the performance of the analysis. We demonstrate the efficiency of our tool on a number of distributed hybrid systems, including collaborating drones, and show that in most cases our tool outperforms state-of-the-art hybrid systems reachability analysis tools.

### *Limitations and Future Work*

HybridSynchAADL's symbolic analysis is currently restricted to systems with (nonlinear) polynomial continuous dynamics, because the underlying SMT solver, Yices2, cannot deal with general classes of ODEs. We should therefore integrate Maude with ODE solvers such as dReal [35] and Flow* [43] to analyze systems whose continuous behaviors are given as (nonlinear) ODEs.

Since a number of CPSs, including the control system for turning airplanes considered in [3], are virtually synchronous *multi-rate* hybrid systems, we should also extend HybridSynchAADL to the multi-rate setting, just as PALS and Synchronous AADL have been extended to the multi-rate case for systems *without* continuous behaviors [23, 32].

HybridSynchAADL supports a fairly powerful subset of AADL and its Behavior Annex for specifying discrete controllers, and could easily specify the applications in this paper. It might nevertheless be useful to support additional features of AADL and its Behavior Annex—such as subprograms, composite data types, arrays of components, and so on—to be able to conveniently specify even more complex systems. Extending HybridSynchAADL should be motivated by larger applications, such as, e.g., the steam-boiler control benchmark [6], which should be seen as a distributed cyber-physical system.

Although, as shown in this paper, our tool's performance is on par with state-of-the-art formal tools for hybrid systems, we should continue our work on improving the efficiency and scalability of our tool; again, this should be triggered by advanced applications. In particular, our tool does not work efficiently for CPSs with nonlinear continuous dynamics, such as the double-integrator models in Section 9, since they lead to higher-order nonlinear constraints that state-of-the-art SMT solvers, such as Yices2, cannot effectively deal with.

Like most formal analysis tools for hybrid systems with non-trivial continuous dynamics [42–44, 56], HybridSynchAADL supports *reachability analysis* (as well as randomized simulations), which is a hard problem for such systems. Nevertheless, it would be desirable to support more expressive formalisms, e.g., based on temporal logic, for defining system requirements. *Signal temporal logic* (STL) [57] is a popular temporal logic for cyber-physical systems. Supporting STL model checking of HybridSynchAADL models is therefore a longterm goal. Some of us have taken steps towards reaching this goal by developing efficient STL model checking algorithms that reduce an STL bounded model checking problem to a problem which can be solved using state-of-the-art SMT solvers [58, 59]. However, that work targets hybrid automata, and should therefore be extended/adapted to the expressive HybridSynch-AADL framework, which also takes clock skews into account.

Another promising avenue towards reaching the goal of analyzing temporal logic properties of HybridSynchAADL models is *statistical model checking* (SMC) [60]. SMC verifies a (probabilistic/stochastic) temporal logic property—or estimates the expected value of a performance measure—*up to the desired level of statistical confidence*. SMC is based on analyzing a number of paths obtained by randomized simulations of purely probabilistic models (that is, models with no unquantified nondeterminism). The key point is that HybridSynchAADL already provides randomized simulations of HybridSynchAADL models (although these simulation models may not be purely probabilistic). Furthermore, the parallelized statistical model checker PVeStA [61] can be directly used to analyze Maude models, and supports the expressive property specification formalism QuaTEx [62] which extends various temporal logics. Therefore, integrating SMC analysis of HybridSynchAADL models using PVeStA (or other SMC tools) seems to be a promising route towards analyzing HybridSynchAADL models w.r.t. complex temporal logic requirements. This would require resolving (probabilistically) the nondeterminism in the control programs.

**Acknowledgments.** We are grateful to John Hatcliff and the anonymous reviewers for very helpful comments on an earlier version of this paper.

## References

[1] Steiner, W., Bauer, G., Hall, B., Paulitsch, M., Varadarajan, S.: TTEthernet dataflow concept. In: 2009 Eighth IEEE International Symposium on Network Computing and Applications, pp. 319–322 (2009). IEEE

[2] Leen, G., Heffernan, D., Dunne, A.: Digital networks in the automotive vehicle. Computing & Control Engineering Journal **10**(6), 257–266 (1999)

[3] Bae, K., Krisiloff, J., Meseguer, J., Ölveczky, P.C.: Designing and verifying distributed cyber-physical systems using Multirate PALS: An airplane turning control system case study. Science of Computer Programming **103**, 13–50 (2015)

[4] Arney, D., Jetley, R., Jones, P., Lee, I., Sokolsky, O.: Formal methods based development of a PCA infusion pump reference model: Generic infusion pump (GIP) project. In: HCMDSS-MDPnP, pp. 23–33 (2007). IEEE

[5] Kim, C., Sun, M., Mohan, S., Yun, H., Sha, L., Abdelzaher, T.F.: A framework for the safe interoperability of medical devices in the presence of network failures. In: ICCPS, pp. 149–158 (2010)

[6] Abrial, J., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control. LNCS, vol. 1165. Springer, Berlin, Heidelberg (1996)

[7] 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, pp. 161–170. IEEE, USA (2009)

[8] 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, USA (2009)

[9] 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)

[10] Bae, K., Ölveczky, P.C., Kong, S., Gao, S., Clarke, E.M.: SMT-based analysis of virtually synchronous distributed hybrid systems. In: Proc. HSCC, pp. 145–154. ACM, New York, NY, USA (2016)

[11] Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language. Addison-Wesley, USA (2012)

[12] França, R.B., Bodeveix, J.-P., Filali, M., Rolland, J.-F., Chemouil, D., Thomas, D.: The AADL Behaviour Annex - experiments and roadmap. In: Proc. ICECCS'07. IEEE, USA (2007)

[13] 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, Berlin, Heidelberg (2007)

[14] 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)

[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] Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 1–39 (2018)

[17] Dutertre, B.: Yices 2.2. In: Proc. CAV. LNCS, vol. 8559, pp. 737–744. Springer, Berlin, Heidelberg (2014)

[18] Lee, J., Kim, S., Bae, K., Ölveczky, P.C.: HybridSynchAADL: Modeling and formal analysis of virtually synchronous CPSs in AADL. In: Proc. CAV'21. LNCS, vol. 12759, pp. 491–504. Springer, Berlin, Heidelberg (2021)

[19] Rushby, J.: Systematic formal verification for fault-tolerant time-triggered algorithms. IEEE Transactions on Software Engineering **25**(5), 651–660 (1999)

[20] Bae, K., Ölveczky, P.C.: MSYNC: A generalized formal design pattern for virtually synchronous multirate cyber-physical systems. ACM Trans. Embedd. Comput. Syst. (Proc. EMSOFT'21) **20**(5s,Article 105) (2021)

[21] 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 (2001). Springer

[22] 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)

[23] Bae, K., Meseguer, J., Ölveczky, P.C.: Formal patterns for multirate distributed real-time systems. Science of Computer Programming **91**, 3–44 (2014)

[24] Steiner, W., Rushby, J.: TTA and PALS: Formally verified design patterns for distributed cyber-physical systems. In: 2011 IEEE/AIAA

30th Digital Avionics Systems Conference, pp. 7–51 (2011). IEEE

[25] Skeirik, S., Stefanescu, A., Meseguer, J.: A constructor-based reachability logic for rewrite theories. Fundamenta Informaticae **173**(4), 315–382 (2020)

[26] Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.: Maude manual (version 3.1). Technical report, SRI International, Menlo Park (2020). http://maude.cs.illinois.edu/w/index.php/Maude_Manual_and_Examples

[27] Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)

[28] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV, pp. 171–177 (2011). Springer

[29] Barrett, C., Stump, A., Tinelli, C., *et al.*: The SMT-LIB standard: Version 2.0. In: SMT, vol. 13, p. 14 (2010)

[30] 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, New York, NY, USA (2014)

[31] Qian, Y., Liu, J., Chen, X.: Hybrid AADL: a sublanguage extension to AADL. In: Proc. Internetware'13. ACM, New York, NY, USA (2013)

[32] Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of Multirate Synchronous AADL. In: Proc. FM'14. LNCS, vol. 8442. Springer, Berlin, Heidelberg (2014)

[33] 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, Berlin, Heidelberg (2011)

[34] 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, Berlin, Heidelberg (2010)

[35] 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, pp. 208–214. Springer, Berlin, Heidelberg (2013)

[36] Ren, W., Beard, R.W.: Distributed Consensus in Multi-vehicle Cooperative Control. Springer, Berlin, Heidelberg (2008)

[37] Henzinger, T.: The theory of hybrid automata. In: Verification of Digital and Hybrid Systems. NATO ASI Series, vol. 170, pp. 265–292. Springer, Berlin, Heidelberg (2000)

[38] Bae, K., Gao, S.: Modular SMT-based analysis of nonlinear hybrid systems. In: Proc. FMCAD, pp. 180–187. IEEE, USA (2017)

[39] 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, Berlin, Heidelberg (1999)

[40] Yu, G., Bae, K.: Maude-SE: a tight integration of Maude and SMT solvers. Proc. International Workshop on Rewriting Logic and its Applications (2020)

[41] Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: HyComp: An SMT-based model checker for hybrid systems. In: Proc. TACAS. LNCS, vol. 9035. Springer, Berlin, Heidelberg (2015)

[42] 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, Berlin, Heidelberg (2011)

[43] Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Proc. CAV, pp. 258–263 (2013).

Springer

[44] Kong, S., Gao, S., Chen, W., Clarke, E.M.: dReach: $\delta$-reachability analysis for hybrid systems. In: Proc. TACAS. Lecture Notes in Computer Science, vol. 7898, pp. 200–205. Springer, Berlin, Heidelberg (2015)

[45] Bak, S., Bogomolov, S., Johnson, T.T.: HYST: a source transformation and translation tool for hybrid automaton models. In: Proc. HSCC'15, pp. 128–133 (2015)

[46] 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, Berlin, Heidelberg (2015)

[47] 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)

[48] 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)

[49] Bae, K., Ölveczky, P.C., Meseguer, J., Al-Nayeem, A.: The SynchAADL2Maude tool. In: Proc. FASE'12. LNCS, vol. 7212. Springer, Berlin, Heidelberg (2012)

[50] Baudart, G., Bourke, T., Pouzet, M.: Soundness of the quasi-synchronous abstraction. In: Proc. FMCAD, pp. 9–16 (2016). IEEE

[51] 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 (2014). IEEE

[52] 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 (2006). IEEE

[53] Girault, A., Ménier, C.: Automatic production of globally asynchronous locally synchronous systems. In: International Workshop on Embedded Software, pp. 266–281 (2002). Springer

[54] Potop-Butucaru, D., Caillaud, B.: Correct-by-construction asynchronous implementation of modular synchronous specifications. Fundamenta Informaticae **78**(1), 131–159 (2007)

[55] 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, pp. 429–448. Springer, Berlin, Heidelberg (2015)

[56] Bak, S., Duggirala, P.S.: Hylaa: A tool for computing simulation-equivalent reachability for linear systems. In: Proc. HSCC, pp. 173–178 (2017)

[57] Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems. Lecture Notes in Computer Science, vol. 3253, pp. 152–166. Springer, Berlin, Heidelberg (2004)

[58] Bae, K., Lee, J.: Bounded model checking of signal temporal logic properties using syntactic separation. In: Proc. POPL 2019 (2019)

[59] Lee, J., Yu, G., Bae, K.: Efficient SMT-based model checking for signal temporal logic. In: Proc. 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21), pp. 343–354 (2021). IEEE

[60] Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1), 6–1639 (2018)

[61] AlTurki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: Proc. CALCO 2011. Lecture Notes in Computer Science, vol. 6859, pp. 386–392. Springer, Berlin, Heidelberg (2011)

[62] Agha, G.A., Meseguer, J., Sen, K.: PMaude:

Rewrite-based specification language for probabilistic object systems. Electronic Notes in Theoretical Computer Science **153**(2), 213–239 (2006)