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

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

[1] Pohang University of Science and Technology
[2] University of Oslo

Abstract. We present the HybridSynchAADL modeling language and formal analysis tool for virtually synchronous cyber-physical systems with complex control programs, continuous behaviors, and bounded clock skews, network delays, and execution times. We leverage the Hybrid PALS equivalence, so that it is sufficient to model and verify the simpler underlying synchronous designs. We define the HybridSynchAADL language as a sublanguage of the avionics modeling standard AADL for modeling such designs in AADL. HybridSynchAADL models are given a formal semantics and analyzed using Maude with SMT solving, which allows us to represent advanced control programs and communication features in Maude, while capturing timing uncertainties and continuous behaviors symbolically with SMT solving. We demonstrate the effectiveness of HybridSynchAADL on a number of applications, including autonomous drones that collaborate to achieve common goals.

1 Introduction

Many cyber-physical systems (CPSs) are *virtually synchronous*. They should logically behave as if they were synchronous—in each iteration of the system, all components, in lockstep, read inputs and perform transitions which generate outputs for the next iteration—but have to be realized in a distributed setting, with clock skews and message passing communication. Examples of virtually synchronous CPSs include avionics and automotive systems, networked medical devices, and distributed control systems such as the steam-boiler benchmark [2]. The underlying infrastructure of such critical systems often guarantees bounds on clock skews, network delays, and execution times.

Virtually synchronous CPSs are notoriously hard to design and to model check, because of the state space explosion caused by asynchronous communication. The *PALS* ("physically asynchronous, logically synchronous") formal pattern reduces the difficulty of modeling and verifying distributed *real-time* systems when the infrastructure provides bounds on network delays, clock skews, and execution times [5,34,35]: A synchronous design SD—where all components execute in lockstep and there is no asynchronous message passing, clock skews, or execution times—is stuttering bisimilar to, and therefore satisfies the same properties as, the corresponding asynchronous distributed "implementation" $PALS(SD)$.

PALS abstracts from the time when an event occurs, as long as it happens in a certain time interval. However, many virtually synchronous CPSs are networks of *hybrid* systems with continuous behaviors, where we cannot abstract from the time when a controller interacts with its continuous environment. *Hybrid PALS* [9] extends PALS to virtually synchronous distributed hybrid systems, taking into account sensing and actuating times that depend on imprecise local clocks. Although synchronous Hybrid PALS models can be encoded as SMT problems [9], it is difficult to encode complex virtually synchronous CPSs—with advanced control programs, data types, hierarchical structure, etc.—in SMT.

This paper therefore defines the HybridSynchAADL language for modeling virtually synchronous distributed hybrid systems using the avionics modeling standard AADL [24] (Section 3). Providing a formal semantics to such models—with control programs written in AADL's expressive Behavior Annex, having to cover all possible continuous behaviors based on imprecise clocks—is challenging. We use Maude [21] combined with SMT solving [12, 40] to *symbolically* encode all possible continuous behaviors depending on imprecise clocks, and provide in Section 6 a Maude-with-SMT semantics for HybridSynchAADL.

Section 4 presents the HybridSynchAADL tool supporting the modeling and verification of HybridSynchAADL models inside the OSATE tool environment for AADL. HybridSynchAADL invokes Maude combined with the SMT solver Yices2 [23] to provide symbolic reachability analysis and randomized simulation for verifying bounded reachability properties of HybridSynchAADL models with polynomial continuous dynamics. To make this analysis efficient, our tool implements several optimization techniques, including symbolic state merging, modular symbolic encoding, and multithreaded portfolio analysis.

We use our tool to model and verify a number of CPS applications, including distributed drones that communicate to reach the "same" location, or fly in formation, without crashing into each other (Section 5). We evaluate, and demonstrate, the effectiveness of the HybridSynchAADL tool by addressing the following questions (Section 7): (i) How effective is our tool compared to state-of-the-art CPS analysis tools? (ii) How effective is our method and tool in finding bugs? (iii) How effective is our new state merging technique?

HybridSynchAADL is one of few, if any, tools—certainly in an AADL context—that can formally analyze 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: (i) Hybrid PALS, which reduces the formal analysis of a virtually synchronous CPS to that of its synchronous design—albeit having to consider clock skews and sensing and actuation times; and (ii) the integration of Maude with SMT solving. Maude is suitable to analyze complex control programs, whereas SMT solving allows us to symbolically analyze continuous behaviors. The HybridSynchAADL tool, available at `https://hybridsynchaadl.github.io`, 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.

2 Preliminaries

PALS and Hybrid PALS. When the infrastructure guarantees bounds on clock skews, network delays, and execution times, the PALS pattern [5,34] reduces the problems of designing and verifying virtually synchronous distributed real-time systems to the much easier problems of designing and verifying their underlying synchronous designs. Formally, given a synchronous system design SD, bounds Γ on clock skews, network delays, and execution times, and a period p of each round, the PALS transformation gives us the asynchronous distributed real-time system $PALS(SD, \Gamma, p)$, which is stuttering bisimilar to SD.

The synchronous design SD is formalized as the synchronous composition of an *ensemble* of state machines with input and output ports [34]. 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 performs their transitions simultaneously, and outputs become inputs at the *next* iteration. PALS was extended to the multi-rate setting in [7], but, for simplicity of exposition, this paper focuses on the single-rate case.

Hybrid PALS [9] 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 [34]. In its ith iteration, a controller M samples the values of its environment at time $c_M(i \cdot p) + t_s$, where t_s is the *sampling time*, and then executes a transition. 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*.

AADL. The *Architecture Analysis & Design Language* (AADL) [24] is an industrial modeling standard used in avionics, aerospace, automotive, medical devices, and robotics to describe an embedded real-time system. In AADL, a component *type* specifies the component's *interface* (e.g., ports) and *properties* (e.g., periods), and a component *implementation* specifies its internal structure as a set of *subcomponents* and a set of *connections* linking their ports. An AADL construct may have *properties* describing its parameters, declared in *property sets*. The OSATE modeling environment provides a set of Eclipse plug-ins for AADL.

An AADL model describes a system of hardware and software components. This paper focuses on the software components, since we use AADL to specify *synchronous designs*. Software components include *threads* that model the application software to be executed 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. *Modes* represent the operational states of components. A component can have mode-specific property values, subcomponents, etc. Mode transitions are triggered by events.

Thread behavior is modeled as a guarded transition system with local variables using AADL's *Behavior Annex* [25]. When a thread is activated, transitions are applied until a *complete* state is reached. The *dispatch protocol* determines when a thread is executed. A *periodic* thread is activated at fixed time intervals.

Maude with SMT. Maude [21] is a language and tool for formally specifying and analyzing concurrent systems in rewriting logic. System states are specified as elements of algebraic data types, and transitions are specified using rewrite rules. A *rewrite theory* [33] is a triple $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory with Σ a signature (declaring sorts, subsorts, and function symbols) and E a set of equations; and R is a set of rewrite rules $l : \ t \longrightarrow t'$ **if** *cond*, where l is a label, t and t' are terms, and *cond* is a conjunction of equations and rewrites. A *rewrite* $t \longrightarrow^* t'$ holds if t' is reachable from t using the rewrite rules in $\mathcal{R}$.

A declaration `class C |` $\ att_1 : s_1, \ldots, att_n : s_n$ declares a class C with attributes $att_1, \ldots, att_n$ of sorts $s_1, \ldots, s_n$. An *object* o of class C is a term `<` $o : C$ `|` $\ att_1 : v_1, \ldots, att_n : v_n$ `>`, where v_i is the value of att_i. A `subclass` inherits the attributes and rewrite rules of its superclasses. A *configuration* is a multiset of objects and messages, with multiset union denoted by juxtaposition.

In rewriting modulo SMT [12, 40], (possibly infinite) sets of system states can be *symbolically* represented using *constrained terms*. A constrained term is a pair $\phi \parallel t$ of a constraint $\phi(x_1, \ldots, x_n)$ and a term $t(x_1, \ldots, x_n)$ over SMT variables $x_1, \ldots, x_n$, representing all instances of t such that ϕ holds: i.e., given the underlying SMT theory $\mathcal{T}$, $[\![\phi \parallel t]\!] = \{t(a_1, \ldots, a_n) \mid \mathcal{T} \models \phi(a_1, \ldots, a_n)\}$.

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

In addition to its explicit-state analysis methods for concrete states (ground terms), Maude provides SMT solving and *symbolic reachability analysis* for constrained terms, using connections to Yices2 [23] and CVC4 [16].

3 The HybridSynchAADL Modeling Language

This section presents the HybridSynchAADL language for modeling virtually synchronous CPSs in AADL. HybridSynchAADL can specify environments with continuous dynamics, synchronous designs of distributed controllers, and nontrivial interactions between controllers and environments with respect to imprecise local clocks and sampling and actuation times.

The HybridSynchAADL language is a subset of AADL extended with the following property set `Hybrid_SynchAADL`. We use a subset of AADL without changing the meaning of AADL constructs or adding new a annex, so that AADL modelers easily can develop and understand HybridSynchAADL models.

```
property set Hybrid_SynchAADL is
  Synchronous: inherit aadlboolean applies to (system);
  isEnvironment: inherit aadlboolean applies to (system);
```

```
  ContinuousDynamics: aadlstring applies to (system);
  Max_Clock_Deviation: inherit Time applies to (system);
  Sampling_Time: inherit Time_Range applies to (system);
  Response_Time: inherit Time_Range applies to (system);
end Hybrid_SynchAADL;
```

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

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

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

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

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

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

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

An environment component interacts with discrete controllers by sending its state values, and by receiving actuator commands that may update the values of state variables or trigger mode (and hence trajectory) changes. This behavior is specified in HybridSynchAADL using *connections between ports and data subcomponents*. A connection from a data subcomponent inside the environment to an output data port declares that the value of the data subcomponent is "sampled" by a controller. A connection from an environment's input port to a data subcomponent inside the environment declares that a controller command arrived at the input port and updates the value of the data subcomponent.

Controller Components. Discrete controllers are usual software components in the Synchronous AADL subset [8,10]. A controller component is specified using the behavioral and structural subset of AADL: hierarchical system, process, thread components, and thread behaviors defined by the Behavior Annex [25].

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.

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

The top-level system component declares the following properties to state that the entire model is a synchronous design with a period T:

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

Communication. In HybridSynchAADL, connections are constrained 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 beginning of the *next* iteration. As explained in [8, 10], *delayed connections between data ports* meet this requirement. Therefore, two controller components can be connected only by data ports with delayed connections: `Timing => Delayed`.

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

4 The HybridSynchAADL Tool

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

Property Specification Language. HybridSynchAADL's *property specification language* allows the user to specify invariant and reachability properties in an intuitive way as propositional formulas whose atomic propositions are AADL Boolean expressions. Since HybridSynchAADL models are infinite-state systems, we only consider properties over behaviors up to a given time bound.

A "named" atomic proposition can be declared with an identifier as follows:

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

Each identifier is fully qualified with its component path. A *scoped expression* of the form *path* | *exp* denotes that each component path in *exp* begins with *path*.

The following *named invariant property* holds if for every (initial) state satisfying the initial condition φ_{init}, all states reachable within the time bound τ_{bound} satisfy the invariant condition φ_{inv}.

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

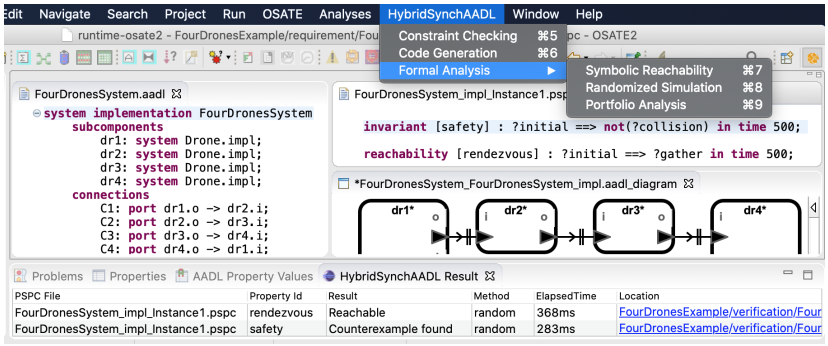

Fig. 1. Interface of the HybridSynchAADL tool.

A *reachability property* (the dual of an invariant) holds if a state satisfying φ_{goal} is reachable from some state satisfying φ_{init} within the time bound τ_{bound}.

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

Tool Interface. The tool first statically checks whether a given model is a valid model that satisfies the syntactic constraints of HybridSynchAADL. It uses OSATE's code generation capability to synthesize the corresponding Maude model. Finally, our tool invokes Maude and an SMT solver to check whether the model satisfies given invariant and reachability requirements.

HybridSynchAADL provides two formal analysis methods. *Symbolic reachability analysis* can verify that all possible behaviors—imposed by sensing and actuation times based on imprecise clocks—satisfy a given requirement;[3] if not, a counterexample is generated. *Randomized simulation* repeatedly executes the model (using Maude) until a counterexample is found, by randomly choosing concrete sampling and actuating times, nondeterministic transitions, etc.

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

Figure 1 shows the interface of our tool that is fully integrated into OSATE. The left editor shows the code of `FourDronesSystem` in Section 5, the bottom right editor shows its graphical representation, and the top right editor shows two properties in the property specification language. The HybridSynchAADL menu contains three items for constraint checking, code generation, and formal analysis. The `Portfolio Analysis` item has been already clicked, and the `Result` view at the bottom displays the analysis results in a readable format.

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

5 Case Study: Collaborating Autonomous Drones

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

We use distributed consensus algorithms [39] 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, where vectors $\vec{x}_i$ and $\vec{v}_i$, for $1 \leq i \leq N$, denote the position and velocity of the i-th drone, respectively. The continuous dynamics of the i-th drone is specified by the differential equation $\dot{\vec{x}}_i = \vec{v}_i$. Let A denote the adjacency matrix representing the underlying 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 position and velocity, and gives the new velocity value to the environment as a control command.

The goal of rendezvous [39] is for all drones to arrive near a common location simultaneously. Using a distributed consensus algorithm, the velocity $\vec{v}_i$ of the i-th drone is given by $\vec{v}_i = -\sum_{j=1}^{N} A_{ij}(\vec{x}_i - \vec{x}_j)$. 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. The location and time of the rendezvous are not given.

The HybridSynchAADL Model. Figure 2 shows a rendezvous model with four drones. Due to lack of space, we show a simplified model with one-dimensional dynamics. The top-level component includes four `Drone` components (Fig. 2a). Each drone sends its position through its output port `o`, and receives the position of the other drone through its input port `i`. The component (i.e., the entire system) is declared to be synchronous with period 100 ms.

A `Drone` component in Fig. 2b contains a controller `ctr` and an environment `env`. The controller `ctr` obtains the current position from `env` via input port `pos`, and sends a new velocity to `env` via output port `vel`, according to its sampling and actuating times declared in the implementation `FourDronesSystem.impl`.

Figure 2c shows a thread component for a drone controller. When the thread dispatches, the transition from `init` to `exec` is taken. If the distance to the connected drone is too close, the new velocity is set to 0 and a `halt` event is sent. Otherwise, the new velocity is set toward the connected drone according to a *discretized* version of the distributed consensus algorithm, and a `move` event is sent. Finally, the current position is assigned to the output port `o`.

```
system FourDronesSystem
end FourDronesSystem;

system implementation FourDronesSystem.impl
 subcomponents
   dr1: system Drone.impl;  dr2: system Drone.impl;
   dr3: system Drone.impl;  dr4: system Drone.impl;
 connections
   C1: port dr1.o -> dr2.i; C2: port dr2.o -> dr3.i;
   C3: port dr3.o -> dr4.i; C4: port dr4.o -> dr1.i;
 properties
   Hybrid_SynchAADL::Synchronous => true;
   Period => 100ms;
   Hybrid_SynchAADL::Max_Clock_Deviation => 10ms;
   Hybrid_SynchAADL::Sampling_Time => 2ms .. 4ms;
   Hybrid_SynchAADL::Response_Time => 6ms .. 9ms;
   Timing => Delayed applies to C1, C2, C3, C4;
end FourDronesSystem.impl;
```

(a) A top-level component.

```
system Drone
 features
   i: in data port Base_Types::Float;
   o: out data port Base_Types::Float;
end Drone;

system implementation Drone.impl
 subcomponents
   ctr: system DroneControl.impl;
   env: system Environment.impl;
 connections
   C1: port ctr.o -> o;
   C2: port i -> ctr.i;
   C3: port env.pos -> ctr.pos;
   C4: port ctr.vel -> env.vel;
   C5: port ctr.halt -> env.halt;
   C6: port ctr.move -> env.move;
end Drone.impl;
```

(b) A drone component.

```
thread DroneControlThread
 features
   i: in data port Base_Types::Float;
   o: out data port Base_Types::Float;
   pos: in data port Base_Types::Float;
   vel: out data port Base_Types::Float;
   halt: out event port;     move: out event port;
 properties
   Dispatch_Protocol => Periodic;
end DroneControlThread;

thread implementation DroneControlThread.impl
 annex behavior_specification {**
   variables
     n : Base_Types::Float;
   states
     init: initial complete state;
     exec, outp: state;
   transitions
     init -[on dispatch]-> exec;
     exec -[abs(pos - i) < 0.4]-> outp {
       vel := 0; halt! };
     exec -[otherwise]-> outp {
       n := -0.1 * (pos - i); move!;
       if (n > 0) vel := 1 else vel := -1 end if };
     outp -[ ]-> init { o := pos };  **};
end DroneControlThread.impl;
```

(c) A controller thread.

```
system Environment
  features
   pos: out data port Base_Types::Float;
   vel: in data port Base_Types::Float;
   halt: in event port;
   move: in event port;
  properties
   Hybrid_SynchAADL::isEnvironment => true;
end Environment;

system implementation Environment.impl
 subcomponents
   x: data Base_Types::Float;
   v: data Base_Types::Float;
 connections
   C1: port x -> pos;
   C2: port vel -> v;
 modes
   fly: initial mode;
   hover: mode;
   fly -[halt]-> hover;
   hover -[move]-> fly;
 properties
   Hybrid_SynchAADL::ContinuousDynamics =>
     "x(t) = x(0) + v * t;" in modes (fly),
     "x(t) = x(0);" in modes (hover);
end Environment.impl;
```

(d) An environment.

Fig. 2. A simplified rendezvous model in HybridSynchAADL.

Figure 2d shows an `Environment` component that specifies the physical model of the drone. Data subcomponents `x` and `v` represent the position and velocity of the drone. There are two modes `fly` and `hover` with their respective continuous dynamics, declared using the property `ContinuousDynamics`. The value of `x` is sent to the controller through the output port `pos`. When a controller sends an actuation command to ports `vel`, `halt`, or `move`, the value of `v` is updated by the value of `vel`, or the mode changes according to the mode transitions.

PSPC File	Property Id	Result	Method	ElapsedTime	Location
FourDronesSystem_impl_Instance2.pspc	rendezvous	Reachable	symbolic	72988ms	SimplifiedFourDrone
FourDronesSystem_impl_Instance2.pspc	safety	No counterexample found	symbolic	27305ms	SimplifiedFourDrone

Fig. 3. Results for the modified properties.

Verification. We consider two properties of the drone rendezvous model in Fig 2: (i) drones do not collide (`safety`), and (ii) all drones could eventually gather together (`rendezvous`). Because the drone model is a distributed hybrid system, these properties depend on the continuous behavior *perturbed by* sensing and actuating times based on imprecise local clocks. We analyze them up to bound 500 ms using HybridSynchAADL portfolio analysis.

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

We define three atomic propositions `collision`, `gather`, and `initial` for four drones `dr1`, `dr2`, `dr3`, and `dr4`. Two drones collide if the distance between them is less than 0.05. All nodes have gathered if the distance between each pair of nodes is less than 2. For example, `gather` and `initial` are defined as follows. There are infinitely many initial states satisfying the proposition `initial`.

```
proposition [gather]: abs(dr1.env.x - dr2.env.x) < 2
                      ...
                  and abs(dr2.env.x - dr4.env.x) < 2
                  and abs(dr3.env.x - dr4.env.x) < 2;
proposition [initial]: abs(dr1.env.x - 1) < 0.1 and abs(dr2.env.x - 2) < 0.1
                   and abs(dr3.env.x - 3) < 0.1 and abs(dr4.env.x - 4) < 0.1;
```

The result of the analysis is shown in Figure 1. A counterexample is found for `safety` and there is a witness for `rendezvous`, both obtained by randomized simulation. The invariant may have been violated because the drones can have any speed in the initial state, since `initial` has no velocity constraints.

We therefore modify `safety` and `rendezvous` below by adding such a velocity constraint `velconst` to the initial condition. As shown in Fig. 3, all possible behaviors up to the bound now satisfy `safety`; this time the results were obtained by symbolic reachability analysis.

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

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

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

6 Executable Formal Semantics of HybridSynchAADL

This section presents the Maude-with-SMT semantics of HybridSynchAADL that implements our tool's analysis commands. Since the HybridSynchAADL modeling language extends Synchronous AADL, the semantics of discrete controllers extends that of Synchronous AADL [8, 10], while the semantics of continuous environments (and interactions with them) is new. We only give an brief overview of the semantics, and refer to the longer report [1] for more details.

Representing HybridSynchAADL Models. Each component is represented as an object instance of a subclass of the base class `Component`. The attribute `features` denotes a set of `Port` objects, `subcomponents` denotes a set of `Component` objects, `connections` denotes its connections, and `properties` denotes its properties.

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

The type of each AADL component corresponds to a subclass of `Component`. The class `Thread` has attributes for thread behaviors, such as transitions. The class `Env` has attributes for continuous dynamics, sampling and actuating times, and mode transitions. Ports and data components are also modeled as objects.

We use a *constrained object* of the form $\phi \parallel obj$ to symbolically represent (possibly infinitely many) instances of object obj, with $\phi(x_1, \ldots, x_n)$ an SMT constraint and $obj(x_1, \ldots, x_n)$ a *pattern*. Object patterns can be hierarchical, because `subcomponents` may include object patterns. For example, the following shows a constrained object for an environment component `env`.

$x_v > 0 \ \wedge \ \neg b_h \ \wedge \ x_p + y_v * t_0 > x_v$ ||
```
< env : Env | features : < vel  : EnvInPort | content : y_v >
                         < pos  : EnvOutPort | content : y_p >
                         < halt : EnvInPort | content : b_h > ...,
    data : < v : Data | value : x_v, ... > < x : Data | value : x_p, ... >,
    jumps : fly -[halt]-> hover, hover -[move]-> fly,    currMode : fly,
    flows : hover [x(t)= x(0)] ; fly [x(t)= x(0)+ v * t], ... >
```

Semantic Operations. Our semantics defines various *semantic operations* that specify the behavior of components, threads, environments, communications, etc. These semantic operations are defined on constrained terms. In particular, the operation `executeStep` specifies a *symbolic rewrite* relation for one synchronous iteration of a single AADL component: `executeStep`$(\phi \parallel obj) \rightsquigarrow^* \phi' \parallel obj'$.

A symbolic synchronous step of the entire system is then formalized by the following rule `step` using `executeStep`. A symbolic rewrite to $\{\phi \wedge \phi' \parallel$ `SYSTEM`$\}$ holds if a symbolic rewrite `executeStep`$(\phi \parallel$ `< C : System | >`$) \rightsquigarrow^* \phi' \parallel$ `SYSTEM` holds and the accumulated constraint $\phi \wedge \phi'$ is satisfied.

```
[step]: {φ ∥ < C : System | features : none >} ⟶ {φ ∧ φ' ∥ SYSTEM}
if executeStep(φ ∥ < C : System | >) ⟶ φ' ∥ SYSTEM  ⋀ T ⊨ φ ∧ φ' .
```

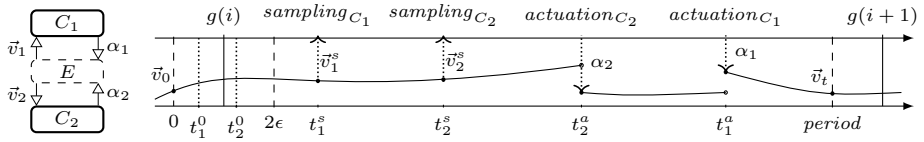

Fig. 4. Interactions between environment E and two controllers C_1 and C_2.

Modular Symbolic Encoding. Figure 4 depicts the behavior of an environment E that interacts with two controllers C_1 and C_2 *in a single iteration*. The time frame is "shifted" to the left from the global time frame $[g(i), g(i+1)]$ by a maximal clock skew ϵ. Because C_i runs according to its local clock, the period of C_i begins at any time $0 < t_i^0 < 2\epsilon$. E sends the state values $\vec{v}_i^s$ at sampling time t_i^s to C_i, and receives C_i's command α_i at actuating time t_i^a. The semantics of environments is formalized by rewrite rules using `executeStep`, which build the *symbolic constraints* to encode the environment behavior shown in Fig. 4.

We symbolically encode asynchronous environment interactions in a *modular* way. In Fig. 4, different orders of interactions can lead to different behaviors. Enumerating all possible interleavings of components can lead to state-space explosion. Instead, the values of input and output ports at different sampling and actuating times are encoded by symbolic variables, and each component performs `executeStep` *independently*. The correspondence between the input and output ports are then symbolically declared using equality constraints.

Merging Symbolic States. Even for one component, `executeStep` can produce many different execution results. In particular, for an environment interacting with n controllers, there are $O((2n)!/2^n)$ different symbolic execution results due to different orderings of sampling and actuating events. The modular encoding above can symbolically eliminate the interleavings of components, but cannot eliminate the nondeterminism in a single component.

To *symbolically* reduce the number of different execution results, we merge two terms that are syntactically identical except for SMT subterms. 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 t. An *abstraction of built-ins* for t, denoted by $abs(t)$, is a constrained term $(x_1 = u_1 \wedge \cdots \wedge x_n = u_n) \parallel t(x_1, \ldots, x_n)$, and it is semantically equivalent to t (i.e., $[\![abs(t)]\!] = [\![true \parallel t]\!]$) [40].

Two abstractions of built-ins $\phi_1 \parallel t_1$ and $\phi_2 \parallel t_2$ are *mergeable* iff there is a renaming substitution ρ with $t_1 = \rho t_2$ (i.e., t_1 and t_2 are equivalent up to renaming). The *merged term* is the constrained term $(\phi_1 \vee \rho\phi_2) \parallel t_1$. For example, $y = 2 + x \parallel f(y)$ and $z = 3 \parallel f(z)$ can be merged into $(y = 2 + x \vee y = 3) \parallel f(y)$. We have proved in the report [1] that $[\![(\phi_1 \vee \rho\phi_2) \parallel t_1]\!] = [\![\phi_1 \parallel t_1]\!] \cup [\![\phi_2 \parallel t_2]\!]$ holds, which ensures the soundness and completeness of our method.

We define a new "merge" operation that collects all the execution results by `executeStep` and merges all mergeable results. This operation always generates a single "merged" result by construction for our HybridSynchAADL semantics, and so the `step` rule for the entire system also does for one synchronous step.

7 Experimental Evaluation

This section evaluates the HybridSynchAADL tool by addressing the following questions: (1) How effective is our tool compared to state-of-the-art CPS analysis tools? (2) How effective is our portfolio analysis method for finding bugs? (3) How effective is our novel state merging technique for symbolic analysis?

To answer these questions, we have analyzed HybridSynchAADL models of networked thermostat and water tank systems (adapted from [6, 30, 38]), and rendezvous and formation control of distributed drones. Many variants of these models are considered: different numbers of components, different sampling and actuating times, single-integrator and double-integrator dynamics, etc.

We have run all experiments on a 16-core 32-thread Intel Xeon 2.8GHz with 256 GB memory. We use a specialized implementation of Maude connected with Yices 2.6, where MCSAT is enabled for nonlinear arithmetic. We have repeated each experiment 10 times (with different random seeds) and report the average results. The models and the experimental results are available in [1].

Comparison with CPS Analysis Tools. We compare HybridSynchAADL's symbolic analysis with three reachability analysis tools for hybrid automata: SpaceEx [26], Hylaa [14], and HyComp [20]. We have developed networks of hybrid automata for distributed thermostats, water tanks, and drone systems, and measure the execution times for analyzing all reachable states of each model up to given bounds using these tools, with timeout 20 minutes.

In the hybrid automata models, each component is modeled as an automaton with five modes: starting a round, sampling, performing a controller transition, actuation, and synchronous communication. For a fair comparison, the controller behavior of each component is encoded as a single mode transition. In addition, we use flat hybrid automata (obtained using HYST [13]) for Hylaa, which does not support networks of hybrid automata.

The experimental results are summarized in Fig. 5, as execution times (seconds, log scale) over time bounds (ms), with N the number of (thermostat, water tank, or drone) components. The results for double-integrator dynamics (where control input is given by acceleration instead of velocity) in the third row do not include HyComp, since it cannot deal with nonlinear polynomial dynamics.

As shown in Fig. 5, HybridSynchAADL outperforms the other tools in most cases. Consider, e.g., "Formation" with single-integrator dynamics for $N = 4$. HybridSynchAADL needs 59 seconds for bound 500, whereas SpaceEx needs 87 seconds and HyComp needs 68 seconds already for bound 100. Hylaa—currently the best performing tool for affine continuous dynamics [15]—did not perform well. This shows HybridSynchAADL's capability to analyze distributed hybrid systems with a large number (e.g., 5^4) of modes/discrete states.

Analyzing Invariant Properties. We evaluate the power of HybridSynchAADL for analyzing bounded invariants. We measure the time taken to find counterexamples, using HybridSynchAADL's three analysis functions, in "faulty" models obtained by modifying the sampling and actuating times. The following table

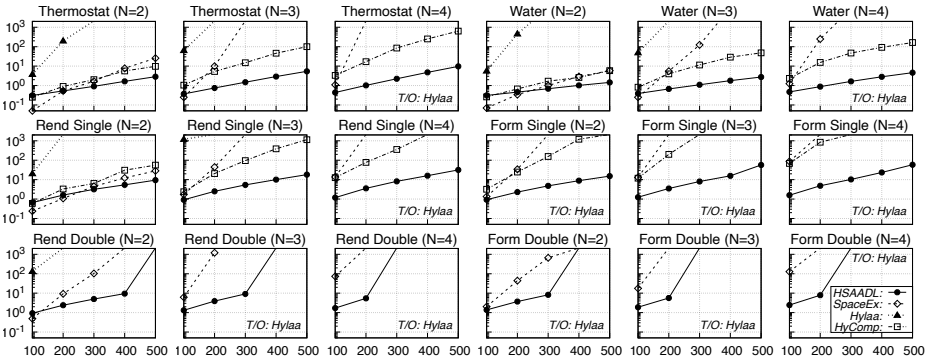

Fig. 5. HybridSynchAADL symbolic analysis vs. SpaceEx, Hylaa, and HyComp.

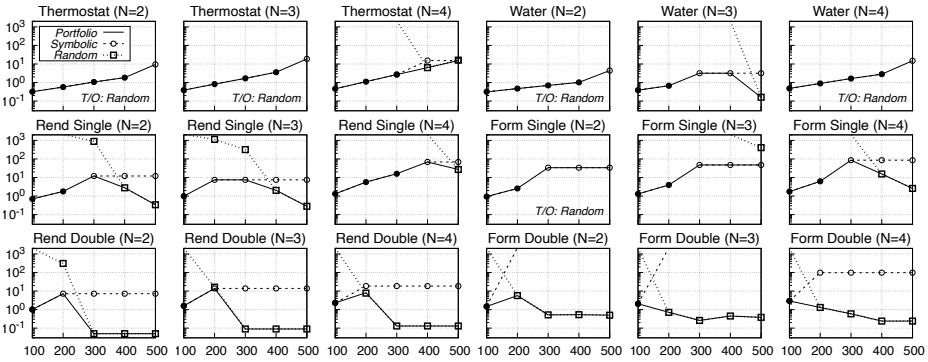

Fig. 6. HybridSynchAADL portfolio analysis.

summarizes the time parameters and invariant properties, with ϵ the maximal clock skew. All models have period 100 (milliseconds). The timeout is 20 minutes.

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

Figure 6 shows the experimental results, with the same x- and y-axis and N as in Fig. 5. Empty shapes indicate that a counterexample is found, and filled circles indicate that symbolic analysis terminates and reports no counterexample. Once a counterexample is found for one bound T by symbolic analysis, the results for larger bounds $T' > T$ are exactly the same, since symbolic analysis uses a breadth-first strategy. The execution time of portfolio analysis is the minimum execution time of symbolic analysis and randomized simulation.

As expected, symbolic analysis is effective to find subtle counterexamples, and randomized simulation is effective for finding obvious bugs. Since the injected

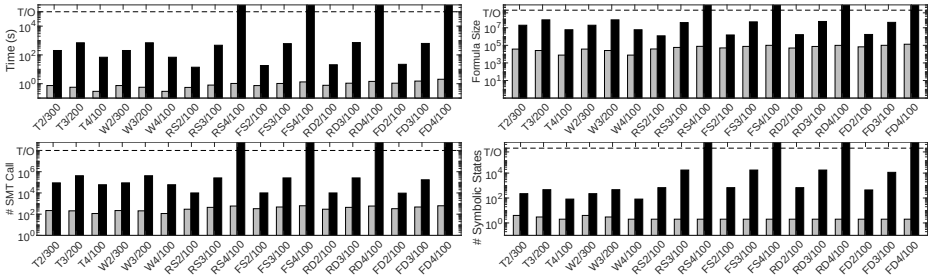

Fig. 7. Symbolic analysis with merging and without merging.

faults are caused by excessive sampling and actuating times, violations are easier to find with a larger bound. Consider, e.g., "Formation" with single-integrator dynamics for $N = 4$. Symbolic analysis found a counterexample for bound 300 in 86 seconds, while randomized simulation timed out. For bound 400, randomized simulation found a counterexample in less than 16 seconds. This demonstrates the usefulness of HybridSynchAADL's portfolio analysis.

Effect of State Merging. To evaluate the effect of state merging, we have performed symbolic reachability analysis to generate all reachable symbolic states up to given bounds, with and without state merging. We measure the execution time, the size of accumulated SMT formulas, the number of calls to the SMT solver, and the number of reachable symbolic states. The timeout is 180 minutes.

Figure 7 highlights some experimental results (see the longer report [1] for more details). Grey bars denote symbolic analysis with merging, and black bars denote symbolic analysis without merging. A horizontal label has the form *model* N/*bound*, where N denotes the number of components in the model, and `T`, `W`, `RS`, `FS`, `RD`, and `FD` denote thermostats, water tanks, rendezvous single, formation single, rendezvous double, and formation double, respectively.

The results show that state merging significantly improves the performance of symbolic analysis. The reason is that symbolic analysis with state merging involves a much smaller size of accumulated SMT constraints. For example, in `RS3-100`, with state merging, the size of the accumulated SMT constraints is around $57,800$, and the analysis took 0.6 seconds. Without merging, the size of the constraints is 39 million, and the analysis took 471 seconds.

8 Related Work

Our tool can perform model checking of virtually synchronous CPSs with both complex control programs and continuous behaviors (and clock skews, etc.), whereas most formal tools are strong at analyzing either discrete or continuous behaviors. The latter includes reachability analysis tools for hybrid automata [14, 20, 26], which do not deal well with the "discrete complexity" (e.g., complex

control programs) of CPSs. HybridSynchAADL can also easily specify and analyze continuous dynamics and imprecise local clocks at the same time.

The *Hybrid Annex* [4] allows specifying continuous behaviors in AADL, and provides some theorem proving [3]. Only a "synchronous" subset without message delays is considered, and clock skews, etc., are not taken into account. Controllers are defined in Hybrid CSP instead of in AADL's convenient and expressive Behavioral Annex. Another hybrid annex is proposed in [37], and an AADL sublanguage, called AADL+, is given in [32]. Both approaches come with some kind of operational semantics and simulation, but with no formal analysis.

Synchronous AADL [8,10,11] supports the modeling and analysis of virtually synchronous distributed real-time systems without continuous behaviors. The explicit-state model checker Maude is used to analyze these models. In contrast, we analyze continuous behaviors for all possible sampling/actuation times. This required us to leave the explicit-state world and use Maude with SMT solving.

The paper [9] shows how Hybrid PALS models with simple controllers—and their bounded reachability problem—can be encoded as logical formulas and analyzed by dReal [27]. However, there is no tool support in [9], and it is difficult to model complex CPSs in SMT. In contrast, this paper provides a tool for modeling Hybrid PALS models using a well-known modeling standard.

Our work is also related to a broader body of work on "almost-synchronous" systems, including quasi-synchrony [17, 18, 29, 31], GALS [28, 36], approximate synchrony [22], time-triggered architectures [41, 42], etc. Our method makes it possible to verify almost-synchronous systems *with continuous behaviors*, which are typically not considered in related work. We provide a convenient language and modeling environment for modeling almost-synchronous CPSs.

9 Concluding Remarks

We have presented the HybridSynchAADL modeling language and analysis tool for formally modeling and analyzing the *synchronous designs*—and, by the Hybrid PALS equivalence, therefore also of the corresponding *asynchronous distributed system* with bounded clock skews, asynchronous communication, network delays, and execution times—of virtually synchronous networks of hybrid systems with potentially complex control programs in the well-known modeling standard AADL. Our tool provides randomized simulation and symbolic reachability analysis (using Maude combined with SMT), and is fully integrated into the OSATE modeling environment for AADL. We have developed and implemented a number of optimization techniques to improve the performance of the analysis. We demonstrate the efficiency of our tool on a number of distributed hybrid systems, including collaborating drones, and show that in most cases our tool outperforms state-of-the-art hybrid systems reachability analysis tools.

Currently, HybridSynchAADL's symbolic analysis is restricted to systems with (nonlinear) polynomial continuous dynamics, because the underlying SMT solver, Yices2, cannot deal with general classes of ODEs. We should therefore integrate Maude with ODE solvers such as dReal [27] and Flow* [19].

References

1. Supplementary material: HybridSynchAADL technical report, semantics, benchmarks, and the tool, https://hybridsynchaadl.github.io/tacas21/
2. Abrial, J., Börger, E., Langmaack, H. (eds.): Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control, LNCS, vol. 1165. Springer (1996)
3. Ahmad, E., Dong, Y., Wang, S., Zhan, N., Zou, L.: Adding formal meanings to AADL with Hybrid Annex. In: Proc. FACS. LNCS, vol. 8997. Springer (2015)
4. Ahmad, E., Larson, B.R., Barrett, S.C., Zhan, N., Dong, Y.: Hybrid Annex: an AADL extension for continuous behavior and cyber-physical interaction modeling. In: Proc. ACM SIGAda annual conference on High integrity language technology (HILT'14). ACM (2014)
5. Al-Nayeem, A., Sun, M., Qiu, X., Sha, L., Miller, S.P., Cofer, D.D.: A formal architecture pattern for real-time distributed systems. In: Proc. RTSS. IEEE (2009)
6. Bae, K., Gao, S.: Modular SMT-based analysis of nonlinear hybrid systems. In: Proc. FMCAD. pp. 180–187. IEEE (2017)
7. Bae, K., Meseguer, J., Ölveczky, P.C.: Formal patterns for multirate distributed real-time systems. Science of Computer Programming **91**, 3–44 (2014)
8. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous AADL and its formal analysis in Real-Time Maude. In: Proc. ICFEM'11. LNCS, vol. 6991. Springer (2011)
9. Bae, K., Ölveczky, P.C., Kong, S., Gao, S., Clarke, E.M.: SMT-based analysis of virtually synchronous distributed hybrid systems. In: Proc. HSCC. ACM (2016)
10. Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of Multirate Synchronous AADL. In: Proc. FM'14. LNCS, vol. 8442. Springer (2014)
11. Bae, K., Ölveczky, P.C., Meseguer, J., Al-Nayeem, A.: The SynchAADL2Maude tool. In: Proc. FASE'12. LNCS, vol. 7212. Springer (2012)
12. Bae, K., Rocha, C.: Symbolic state space reduction with guarded terms for rewriting modulo SMT. Science of Computer Programming **178**, 20–42 (2019)
13. Bak, S., Bogomolov, S., Johnson, T.T.: Hyst: a source transformation and translation tool for hybrid automaton models. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control. pp. 128–133 (2015)
14. Bak, S., Duggirala, P.S.: Hylaa: A tool for computing simulation-equivalent reachability for linear systems. In: Proc. HSCC. pp. 173–178 (2017)
15. Bak, S., Tran, H.D., Johnson, T.T.: Numerical verification of affine systems with up to a billion dimensions. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. pp. 23–32 (2019)
16. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. pp. 171–177. Springer (2011)
17. Baudart, G., Bourke, T., Pouzet, M.: Soundness of the quasi-synchronous abstraction. In: Proc. FMCAD. pp. 9–16. IEEE (2016)
18. Caspi, P., Mazuet, C., Paligot, N.R.: About the design of distributed control systems: The quasi-synchronous approach. In: International Conference on Computer Safety, Reliability, and Security. Springer (2001)
19. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Proc. CAV. pp. 258–263. Springer (2013)
20. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: HyComp: An SMT-based model checker for hybrid systems. In: Proc. TACAS. LNCS, vol. 9035. Springer (2015)

21. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude – A High-Performance Logical Framework, Lecture Notes in Computer Science, vol. 4350. Springer (2007)
22. Desai, A., Seshia, S.A., Qadeer, S., Broman, D., Eidson, J.C.: Approximate synchrony: An abstraction for distributed almost-synchronous systems. In: Proc. CAV'15. LNCS, vol. 9207. Springer (2015)
23. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV. LNCS, vol. 8559, pp. 737–744. Springer (July 2014)
24. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language. Addison-Wesley (2012)
25. França, R., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL Behaviour Annex - experiments and roadmap. In: Proc. ICECCS'07. IEEE (2007)
26. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Proc. CAV. LNCS, vol. 6806. Springer (2011)
27. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: Proc. CADE. Lecture Notes in Computer Science, vol. 7898. Springer (2013)
28. Girault, A., Ménier, C.: Automatic production of globally asynchronous locally synchronous systems. In: International Workshop on Embedded Software. pp. 266–281. Springer (2002)
29. Halbwachs, N., Mandel, L.: Simulation and verification of asynchronous systems by means of a synchronous model. In: Sixth International Conference on Application of Concurrency to System Design (ACSD'06). pp. 3–14. IEEE (2006)
30. Henzinger, T.: The theory of hybrid automata. In: Verification of Digital and Hybrid Systems, NATO ASI Series, vol. 170, pp. 265–292. Springer (2000)
31. Larrieu, R., Shankar, N.: A framework for high-assurance quasi-synchronous systems. In: 2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE). pp. 72–83. IEEE (2014)
32. 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)
33. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)
34. 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)
35. Miller, S., Cofer, D., Sha, L., Meseguer, J., Al-Nayeem, A.: Implementing logical synchrony in integrated modular avionics. In: Proc. IEEE/AIAA 28th Digital Avionics Systems Conference. IEEE (2009)
36. Potop-Butucaru, D., Caillaud, B.: Correct-by-construction asynchronous implementation of modular synchronous specifications. Fundamenta Informaticae **78**(1), 131–159 (2007)
37. Qian, Y., Liu, J., Chen, X.: Hybrid AADL: a sublanguage extension to AADL. In: Proc. Internetware'13. ACM (2013)
38. Raisch, J., Klein, E., Meder, C., Itigin, A., O'Young, S.: Approximating automata and discrete control for continuous systems — two examples from process control. In: Hybrid systems V. pp. 279–303. Springer (1999)
39. Ren, W., Beard, R.W.: Distributed consensus in multi-vehicle cooperative control. Springer (2008)

40. 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)
41. Rushby, J.: Systematic formal verification for fault-tolerant time-triggered algorithms. IEEE Transactions on Software Engineering **25**(5), 651–660 (1999)
42. 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)