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

No Author Given

No Institute Given

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. 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 when the infrastructure provides bounds on network delays, clock skews, and execution times [5, 39, 40]: 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* [11] 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 [11], 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 conveniently modeling virtually synchronous distributed hybrid systems using the avionics modeling standard AADL [28] (Section 2). Providing a formal semantics to such models—with control programs written in AADL's expressive Behavior Annex, having to cover all possible continuous behaviors based on imprecise clocks—is challenging. We use Maude [25] combined with SMT solving [15, 46] to *symbolically* encode all possible continuous behaviors with all possible sensing and actuating times depending on imprecise clocks, and provide in Section **??** a Maude-with-SMT semantics for HybridSynchAADL.

Section 3 presents the HybridSynchAADL tool supporting the modeling and verification of HybridSynchAADL models inside the OSATE tool environment for AADL. Our tool provides an intuitive property specification language for specifying bounded reachability properties of HybridSynchAADL models. HybridSynchAADL invokes Maude combined with the SMT solver Yices2 [27] to provide symbolic reachability analysis and randomized simulation for verifying such bounded reachability properties of HybridSynchAADL models with polynomial continuous dynamics. 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 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 (Section **??**). We evaluate, and demonstrate, the effectiveness of the HybridSynchAADL tool by addressing the following questions (Section **??**):

1. How effective is our tool compared to state-of-the-art CPS analysis tools?
2. How effective is our method and tool in finding bugs?
3. How effective is our new state merging technique?
4. How effective is Hybrid PALS in verifying virtually synchronous CPSs with continuous behaviors?

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:

1. 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
2. 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.

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 is available at `https://hybridsynchaadl.github.io`.

The rest of this paper is organized as follows. Section **??** explains some background on Hybrid PALS, AADL, and Maude with SMT. Section 2 presents the HybridSynchAADL language. Section 3 presents the HybridSynchAADL tool. Section **??** presents case studies on virtually synchronous CPSs for controlling distributed drones. Section **??** presents its semantics in Maude with SMT. Section **??** shows the experimental results. Section **??** discusses the related work. Finally, Section 4 presents some concluding remarks.

2 The HybridSynchAADL Modeling Language

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

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

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

There are two kinds of components in HybridSynchAADL: *continuous* environments and *discrete* controllers. Environments are specified as system components whose continuous dynamics is specified using continuous functions or ordinary differential equations. Discrete controllers are usual AADL software

components in the Synchronous AADL subset [9, 12] of AADL.[1] The top-level system component declares the following properties to state that the model is a synchronous design and to declare the period of the system, respectively.

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

Example 1. We use a simple networked thermostat system as a running example. There are two thermostats that control the temperatures of two rooms located in different places. The goal is to maintain similar temperatures in both rooms. For this purpose, the controllers communicate with each other over a network, and turn the heaters on or off, based on the current temperature of the room and the temperature of the other room. Figure 1 shows the architecture of this networked thermostat system. For room i, for $i = 1, 2$, the controller `ctrl`i controls its environment `env`i (using "connections" explained below).

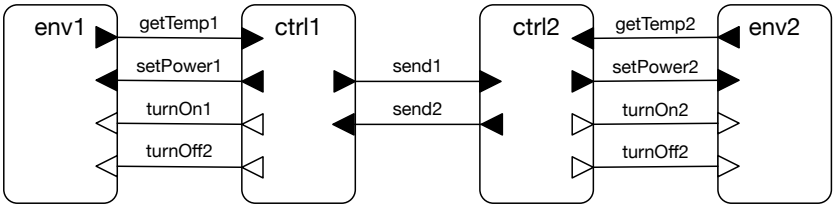

Fig. 1. A networked thermostat system.

2.1 Environment Components

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

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

```
Hybrid_SynchAADL::ContinuousDynamics =>
      "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:

[1] Just like Synchronous AADL can be extended to *multi-rate* controllers [12], it is possible to extend HybridSynchAADL to the multi-rate case in the same way, but for clarity and ease of exposition we focus on the single-rate case in this paper.

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

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

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

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

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

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

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

The value of `x` is sent to the controller through the output port `temp`, declared by the connection `port x -> temp`. When a discrete controller sends an actuation command through input ports `power`, `on_control`, and `off_control`, the mode changes according to the mode transitions, and the value of `p` can be updated by the value of input port `power`, declared by the connection `port x -> temp`.

2.2 Controller Components

Discrete controllers are usual AADL software components in the Synchronous AADL subset [10, 13]. A controller component is specified using the behavioral

and structural subset of AADL: hierarchical system, process, thread components, data subcomponents; ports and connections; and thread behaviors defined by the Behavior Annex [29]. The hardware and scheduling features of AADL, which are not relevant to synchronous *designs*, are not considered in HybridSynchAADL.

Dispatch. The execution of an AADL thread is specified by the *dispatch protocol*. A thread with an *event-triggered* dispatch (such as aperiodic, sporadic, timed, or hybrid dispatch protocols) is dispatched when it receives an event. Since all "controller" components are executed in lock-step in HybridSynchAADL, each thread must have *periodic* dispatch by which the thread is dispatched at

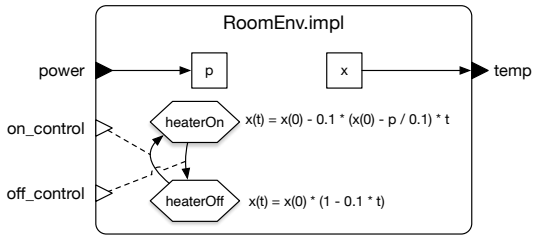

Fig. 2. An environment of the thermostat controller.

```
system RoomEnv
  features
    temp: out data port Base_Types::Float;
    power: in data port Base_Types::Float;
    on_control: in event port;          off_control: in event port;
  properties
    Hybrid_SynchAADL::isEnvironment => true;
end RoomEnv;

system implementation RoomEnv.impl
  subcomponents
    x: data Base_Types::Float {Data_Model::Initial_Value => ("15");};
    p: data Base_Types::Float {Data_Model::Initial_Value => ("5");};
  connections
    C: port x -> temp;                  R: port power -> p;
  modes
    heaterOff: initial mode;            heaterOn: mode;
    heaterOff -[on_control]-> heaterOn; heaterOn -[off_control]-> heaterOff;
  properties
    Hybrid_SynchAADL::ContinuousDynamics =>
      "x(t) = x(0) - 0.1 * (x(0) - p / 0.1) * t;" in modes (heaterOn),
      "x(t) = x(0) * (1 - 0.1 * t);" in modes (heaterOff);
end RoomEnv.impl;
```

Fig. 3. A `RoomEnv` component.

the beginning of each period. The periods of all the threads are identical to the period declared in the top-level component. In AADL, this behavior is declared by the thread component property:

```
Dispatch_Protocol => Periodic;
```

Timing Properties. A controller receives the state of the environment at some *sampling time*, and sends a controller command to the environment at some *actuation time*. Sampling and actuation take place according to the local clock of the controller, which may differ from the "ideal clock" by up to the maximal clock skew. These time values are declared by the component properties:

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

The upper sampling time bound must be strictly smaller than the upper bound of actuation time, and the lower bound of actuation time must be strictly greater than the lower bound of sampling time. Also, the upper bounds of both sampling and actuating times must be strictly smaller than the maximal execution time to meet the (Hybrid) PALS constraints [11].

Initial Values and Parameters. In AADL, *data* subcomponents represent data values, such as Booleans, integers, and floating-point numbers. The initial values of data subcomponents and output ports are specified using the property:

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

Sometimes initial values can be *parameters*, instead of concrete values. E.g., you can check whether a certain property holds from initial values satisfying a certain constraint for those parameters (see Section 3). In HybridSynchAADL, such unknown parameters can be declared using the following AADL property:

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

Example 3. Consider again our networked thermostat system. Figure 4 shows a thread component `ThermostatThread` that turns the heater on or off depending on the average value `avg` of the current temperatures of the two rooms. It has event output ports `on_control` and `off_control`, data input ports `curr` and `tin`, and data output ports `set_power` and `tout`. The ports `on_control`, `off_control`, `set_power`, and `curr` are connected to an environment, and `tin` and `tout` are connected to another controller component (see Fig. 5). The implementation has the data subcomponent `avg` whose initial value is declared as a parameter.

When the thread dispatches, the transition from state `init` to `exec` is taken, which updates `avg` using the values of the input ports `curr` and `tin`, and assigns to the output port `tout` the value of `curr`. Since `exec` is not a complete state, the thread continues executing by taking one of the other transitions, which

may send an event. For example, if the value of `avg` is smaller than 10, a control command that sets the heater's power to 5 is sent through the port `set_power`, and an event is sent through the port `off_control`. The resulting state `init` is a complete state, and the execution of the current dispatch ends.

2.3 Communication

There are three kinds of ports in AADL: *data* ports, *event* ports, and *event data* ports. In AADL, *event* and *event data* ports can trigger the execution of threads, whereas *data* ports cannot. In HybridSynchAADL, connections are constrained for synchronous behaviors: no connection is allowed between environments, or between environments and the enclosing system components.

Connections Between Discrete Controllers. All (non-actuator) output values of controller components generated in an iteration are available to the receiving *controller* components at the beginning of the *next* iteration. As explained in

```
thread ThermostatThread
  features
    on_control: out event port;         off_control: out event port;
    set_power: out data port Base_Types::Float;
    curr: in data port Base_Types::Float;
    tin: in data port Base_Types::Float;
    tout: out data port Base_Types::Float
            {Data_Model::Initial_Value => ("0");};
  properties
    Dispatch_Protocol => Periodic;
    Hybrid_SynchAADL::Max_Clock_Deviation => 0.3ms;
    Hybrid_SynchAADL::Sampling_Time => 1ms .. 5ms;
    Hybrid_SynchAADL::Response_Time => 7ms .. 9ms;
end ThermostatThread;

thread implementation ThermostatThread.impl
  subcomponents
    avg : data Base_Types::Float {Data_Model::Initial_Value => ("param");};
  annex behavior_specification{**
    states
      init : initial complete state;    exec : state;
    transitions
      init -[on dispatch]-> exec { avg := (tin + curr) / 2; tout := curr };
      exec -[avg > 25]-> init { off_control! };
      exec -[avg < 20 and avg >= 10]-> init { set_power := 5; on_control! };
      exec -[avg < 10]-> init { set_power := 10; on_control! };         **};
end ThermostatThread.impl;
```

Fig. 4. A simple thermostat controller.

[10,13], *delayed connections between data ports* meet this requirement. Therefore, two controller components can be connected only by data ports with delayed connections, declared by the connection property:

```
Timing => Delayed;
```

Connections Between Controllers and Environments. In HybridSynchAADL, interactions between a controller and an environment occur *instantaneously* at the sampling and actuating times of the controller.[2] Because an environment does not "actively" send data for sampling, every output port of an environment must be a *data* port, whereas its input ports could be of any kind.

On the other hand, any types of input ports, such as data, event, event data ports, are available for environment components. Specifically, a discrete controller can trigger a mode transition of an environment through event ports. Therefore, no extra requirement is needed for connections, besides the usual constraints for port to port connections in AADL.

Example 4. Figure 5 shows an implementation of a top-level system component `TwoThermostats` of our networked thermostat system, depicted in Figure 1. This component has no ports and contains two thermostats and their environments. The controller system component `Thermostat.impl` is implemented using the thread component `ThermostatThread.impl` in Fig. 4, and the environment component `RoomEnv.impl` is given in Fig. 3. Each discrete controller `ctrl`i, for $i = 1, 2$, is connected to its environment component `env`i using four connections `turnOn`i, `turnOff`i, `setPower`i, and `getTemp`i. The controllers `ctrl1` and `ctrl2` are connected with each other using delayed data connections `send1` and `send2`.

3 Property Specification Language

HybridSynchAADL's *property specification language* allows the user to easily specify invariant and reachability properties in an intuitive way, without having to understand Maude or the formal representation of the models. Such properties are given by propositional logic formulas whose atomic propositions are AADL Boolean expressions. Because HybridSynchAADL models are infinite-state systems, we only consider properties over behaviors up to a given time bound.

Atomic propositions are given by Boolean expressions in the AADL Behavior Annex syntax. Each identifier is fully qualified with its component path in the AADL syntax. A *scoped expression* of the form *path* | *exp* denotes that each component path of each identifier in the expression *exp* begins with *path*. A "named" atomic proposition can be declared with an identifier as follows:

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

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

```
system implementation TwoThermostats.impl
  subcomponents
    ctrl1: system Thermostat.impl;        ctrl2: system Thermostat.impl;
    env1: system RoomEnv.impl;            env2: system RoomEnv.impl;
  connections
    turnOn1:   port ctrl1.on_control  -> env1.on_control;
    turnOff1:  port ctrl1.off_control -> env1.off_control;
    setPower1: port ctrl1.set_power   -> env1.power;
    getTemp1:  port env1.temp         -> ctrl1.curr;
    send1:     port ctrl1.tout         > ctrl2.tin;
    turnOn2:   port ctrl2.on_control  -> env2.on_control;
    turnOff2:  port ctrl2.off_control -> env2.off_control;
    setPower2: port ctrl2.set_power   -> env2.power;
    getTemp2:  port env2.temp         -> ctrl2.curr;
    send2:     port ctrl2.tout        -> ctrl1.tin;
  properties
    Hybrid_SynchAADL::Synchronous => true;
    Period => 10 ms;
    Timing => Delayed applies to send1, send2;
end TwoThermostats.impl;
```

Fig. 5. A top level component with two thermostat controllers.

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

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

invariant [*name*]: φ_{init} ==> φ_{inv} **in time** τ_{bound}

A *reachability property* (the dual of an invariant) holds if a state satisfying φ_{goal} is reachable from some state satisfying the initial condition φ_{init} within the time bound τ_{bound}. It is worth noting that a reachability property can be expressed as an invariant property by negating the goal condition.

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

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

$$path \mid exp$$

denotes that the component path of each identifier in the expression *exp* begins with *path*. For example, c_1 . c_2 | ((x_1 > x_2) and (b_1 = b_2)) is equivalent to (c_1 . c_2 . x_1 > c_1 . c_2 . x_2) and (c_1 . c_2 . b_1 = c_1 . c_2 . b_2). These

scopes can be nested so that one scope may include another scope. For example, c_1 | ((c_2 | (x > c_3 . y)) = (c_4 | (c_5 | b))) is equivalent to the expression (c_1 . c_2 . x > c_1 . c_2 . c_3 . y) = c_1 . c_4 . c_5 . b.

Example 5. Consider the thermostat system in Section 2 that consists of two thermostat controllers `ctrl1` and `ctrl2` and their environments `env1` and `env2`, respectively. The following declares two propositions `inRan1` and `inRan2` using the property specification language. For example, `inRan1` holds if the value of `env1`'s data subcomponent `x` is between 10 and 25.

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

The following declares the invariant property `inv`. The initial condition states that the value of `env1`'s data subcomponent `x` satisfies $|x-15| < 3$ and the value of `env2`'s data subcomponent `x` satisfies $|x-7| < 1$. This property holds if for each initial state satisfying the initial condition, any reachable state within the time bound 30 satisfies the conditions `inRan1`, `inRan2`, and `env1.x > env2.x`.

```
invariant [inv]: abs(env1.x - 15) < 3 and abs(env2.x - 7) < 1
     ==> ?inRan1 and ?inRan2 and (env1.x > env2.x) in time 30
```

4 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 [31] and Flow* [23] to analyze systems whose continuous behaviors are given as (nonlinear) ODEs. Finally, we can also extend HybridSynchAADL from single-rate to multi-rate controllers, in a similar way as [8].

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. Kong, S., Gao, S., Chen, W., Clarke, E.M.: dReach: δ-reachability analysis for hybrid systems. In: Proc. TACAS. Lecture Notes in Computer Science, vol. 7898. Springer (2015)
7. Bae, K., Gao, S.: Modular SMT-based analysis of nonlinear hybrid systems. In: Proc. FMCAD. pp. 180–187. IEEE (2017)
8. Bae, K., Meseguer, J., Ölveczky, P.C.: Formal patterns for multirate distributed real-time systems. Science of Computer Programming **91**, 3–44 (2014)
9. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous aadl and its formal analysis in real-time maude. In: International Conference on Formal Engineering Methods. pp. 651–667. Springer (2011)
10. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous AADL and its formal analysis in Real-Time Maude. In: Proc. ICFEM'11. LNCS, vol. 6991. Springer (2011)
11. Bae, K., Ölveczky, P.C., Kong, S., Gao, S., Clarke, E.M.: SMT-based analysis of virtually synchronous distributed hybrid systems. In: Proc. HSCC. ACM (2016)
12. Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of multirate synchronous aadl. In: Proc. FM. Lecture Notes in Computer Science, vol. 8442, pp. 94–109. Springer (2014)
13. Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of Multirate Synchronous AADL. In: Proc. FM'14. LNCS, vol. 8442. Springer (2014)
14. Bae, K., Ölveczky, P.C., Meseguer, J., Al-Nayeem, A.: The SynchAADL2Maude tool. In: Proc. FASE'12. LNCS, vol. 7212. Springer (2012)
15. Bae, K., Rocha, C.: Symbolic state space reduction with guarded terms for rewriting modulo SMT. Science of Computer Programming **178**, 20–42 (2019)
16. Bak, S., Bogomolov, S., Johnson, T.T.: Hyst: a source transformation and translation tool for hybrid automaton models. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control. pp. 128–133 (2015)
17. Bak, S., Duggirala, P.S.: Hylaa: A tool for computing simulation-equivalent reachability for linear systems. In: Proc. HSCC. pp. 173–178 (2017)
18. Bak, S., Tran, H.D., Johnson, T.T.: Numerical verification of affine systems with up to a billion dimensions. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. pp. 23–32 (2019)
19. Bao, Y., Chen, M., Zhu, Q., Wei, T., Mallet, F., Zhou, T.: Quantitative performance evaluation of uncertainty-aware Hybrid AADL designs using statistical model checking. IEEE Transactions on CAD of Integrated Circuits and Systems **36**(12), 1989–2002 (2017)

20. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. pp. 171–177. Springer (2011)
21. Baudart, G., Bourke, T., Pouzet, M.: Soundness of the quasi-synchronous abstraction. In: Proc. FMCAD. pp. 9–16. IEEE (2016)
22. Caspi, P., Mazuet, C., Paligot, N.R.: About the design of distributed control systems: The quasi-synchronous approach. In: International Conference on Computer Safety, Reliability, and Security. Springer (2001)
23. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Proc. CAV. pp. 258–263. Springer (2013)
24. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: HyComp: An SMT-based model checker for hybrid systems. In: Proc. TACAS. LNCS, vol. 9035. Springer (2015)
25. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude – A High-Performance Logical Framework, Lecture Notes in Computer Science, vol. 4350. Springer (2007)
26. Desai, A., Seshia, S.A., Qadeer, S., Broman, D., Eidson, J.C.: Approximate synchrony: An abstraction for distributed almost-synchronous systems. In: Proc. CAV'15. LNCS, vol. 9207. Springer (2015)
27. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV. LNCS, vol. 8559, pp. 737–744. Springer (July 2014)
28. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language. Addison-Wesley (2012)
29. França, R., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL Behaviour Annex - experiments and roadmap. In: Proc. ICECCS'07. IEEE (2007)
30. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Proc. CAV. LNCS, vol. 6806. Springer (2011)
31. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: Proc. CADE. Lecture Notes in Computer Science, vol. 7898. Springer (2013)
32. Girault, A., Ménier, C.: Automatic production of globally asynchronous locally synchronous systems. In: International Workshop on Embedded Software. pp. 266–281. Springer (2002)
33. Halbwachs, N., Mandel, L.: Simulation and verification of asynchronous systems by means of a synchronous model. In: Sixth International Conference on Application of Concurrency to System Design (ACSD'06). pp. 3–14. IEEE (2006)
34. Henzinger, T.: The theory of hybrid automata. In: Verification of Digital and Hybrid Systems, NATO ASI Series, vol. 170, pp. 265–292. Springer (2000)
35. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. Acm Sigplan Notices **47**(6), 193–204 (2012)
36. Larrieu, R., Shankar, N.: A framework for high-assurance quasi-synchronous systems. In: 2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE). pp. 72–83. IEEE (2014)
37. Liu, J., Li, T., Ding, Z., Qian, Y., Sun, H., He, J.: AADL+: a simulation-based methodology for cyber-physical systems. Frontiers Comput. Sci. **13**(3), 516–538 (2019)
38. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)
39. Meseguer, J., Ölveczky, P.C.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. Theoretical Computer Science **451**, 1–37 (2012)

40. Miller, S., Cofer, D., Sha, L., Meseguer, J., Al-Nayeem, A.: Implementing logical synchrony in integrated modular avionics. In: Proc. IEEE/AIAA 28th Digital Avionics Systems Conference. IEEE (2009)
41. Ölveczky, P.C., Boronat, A., Meseguer, J.: Formal semantics and analysis of behavioral aadl models in real-time maude. In: Formal Techniques for Distributed Systems, pp. 47–62. Springer (2010)
42. Potop-Butucaru, D., Caillaud, B.: Correct-by-construction asynchronous implementation of modular synchronous specifications. Fundamenta Informaticae **78**(1), 131–159 (2007)
43. Qian, Y., Liu, J., Chen, X.: Hybrid AADL: a sublanguage extension to AADL. In: Proc. Internetware'13. ACM (2013)
44. Raisch, J., Klein, E., Meder, C., Itigin, A., O'Young, S.: Approximating automata and discrete control for continuous systems — two examples from process control. In: Hybrid systems V. pp. 279–303. Springer (1999)
45. Ren, W., Beard, R.W.: Distributed consensus in multi-vehicle cooperative control. Springer (2008)
46. Rocha, C., Meseguer, J., Muñoz, C.: Rewriting modulo SMT and open system analysis. Journal of Logical and Algebraic Methods in Programming **86**(1), 269–297 (2017)
47. Rushby, J.: Systematic formal verification for fault-tolerant time-triggered algorithms. IEEE Transactions on Software Engineering **25**(5), 651–660 (1999)
48. Tripakis, S., Pinello, C., Benveniste, A., Sangiovanni-Vincent, A., Caspi, P., Di Natale, M.: Implementing synchronous models on loosely time triggered architectures. IEEE Transactions on Computers **57**(10), 1300–1314 (2008)