Formal Model Engineering of Distributed CPSs using AADL: From Behavioral AADL Models to Multirate Hybrid Synchronous AADL

Kyungmin Bae[1] and Peter Csaba Ölveczky[2]

[1] Pohang University of Science and Technology, South Korea
[2] University of Oslo, Norway

Abstract. A promising way of integrating formal methods into industrial system design is to endow industrial modeling tools with automatic formal analyses. In this paper we identify some challenges for providing such formal methods "backends" for cyber-physical systems (CPSs), and argue that Maude could meet these challenges. We then give an overview of our research on integrating Maude analysis into the OSATE tool environment for the industrial CPS modeling standard AADL.
Since many critical distributed CPSs are "logically synchronous," a key feature making automatic formal analysis practical is the use of *synchronizers* for CPSs. We identify a sublanguage of AADL to describe *synchronous* CPS designs. We can then use Maude to effectively verify such synchronous designs, which under certain conditions also verifies the corresponding asynchronous *distributed* systems, *with* clock skews and communication delays. We then explain how we have extended our methods to *multirate* systems and to CPSs with continuous behaviors.
We illustrate the effectiveness of Maude-based formal model engineering of industrial CPSs on avionics control systems and collections of drones. Finally, we identify future directions in this line of research.

1 Introduction and Overview

Modern cyber-physical systems (CPSs) are complex and safety-critical systems. Formal methods should therefore be an integral part of their design process. However, despite the availability of powerful formal tools, there still seems to be a barrier in industry to using formal methods. This could have many reasons, including lack of available formal methods experts, perceived need to be an expert to use formal methods, and lack of user-friendly and robust formal tools.

Formal Model Engineering. Maybe the most promising way of integrating formal methods into industrial model development processes is to provide automatic "push-button" formal analysis as an "under-the-hood backend" which is integrated into the designer's model development environment. The modeler can then continue to use her favorite modeling language and IDE without worrying about formal methods, and still get powerful formal analyses *for free*. Edward Lee calls such model development *formal model engineering* in [8].

Enabling such formal model engineering requires:

– Targeting a modeling language widely used in industrial model development.
– Identifying a useful significant subset of the industrial modeling language.
– A formal semantics for the (subset of the) industrial modeling language.
– Defining an intuitive property specification language in which the developer can easily express properties about her model, without knowing anything about formal methods or the formal representation of her model, and without having to master the syntax of the formal method.
– Integrating *automatic* formal analyses of the user's model into *her* tool.
– Presenting the result of the formal analysis in an intuitive way.

Challenges. Endowing industrial modeling environments for CPSs, which are distributed (real-time) embedded systems, with effective under-the-hood formal analysis is very challenging, due to factors that include:

1. Industrial modeling languages by their very nature tend to be large and expressive. The target formalism must therefore be very expressive to provide a formal semantics for the source language.
2. However, in formal methods usually only less expressive, often decidable, formalisms (such as timed automata for real-time systems) provide *automatic* analyses, whereas more expressive formalisms (such as differential dynamic logic [61]) typically only support interactive theorem proving analysis, which is not desired for formal model engineering.
3. Automatic model checking verification of industrial distributed CPSs quickly becomes infeasible due to the many interleavings caused by distribution.
4. Industrial CPSs may have both *continuous* behaviors and complex *discrete* control programs, in addition to clock skews, network delays, and so on.
5. The need to provide a query language in which requirements can be intuitively expressed without knowing the formal representation of the model.
6. Verification typically applies to a specific deployment scenario, and small changes (such as network delays) can invalidate the verification result.

Maude as a Formal Model Engineering Backend for CPSs. We argue that Maude [27], with its Real-Time Maude [58, 59] and Maude-SMT [72] extensions, is a suitable formal framework for formal model engineering of industrial CPSs.

Maude is formal modeling language and high-performance formal analysis tool based on rewriting logic [48, 50]. Rewriting logic is an expressive logic for concurrency where the static parts of the system (data types, etc.) are defined by algebraic equational specifications, and where dynamic behavior is defined by rewrite rules. Maude is particularly suitable to model distributed systems in an object-oriented style. Maude provides a range of explicit-state formal analysis methods, including rewriting for simulation, reachability analysis, and linear temporal logic (LTL) model checking. Real-time systems can also be specified and analyzed in Real-Time Maude, which provides time-bounded analysis commands and timed CTL model checking [44]. Maude has recently integrated SMT solving, which enables *symbolic* reasoning that manipulates *symbolic* states; i.e., state patterns with variables that represent possibly infinitely many concrete states.

Maude addresses the challenges 1, 2, 4, and 5 above as follows:

1. Rewriting logic is well known to be expressive (see, e.g., [50] for a dated overview of some applications), and can capture complex control program languages—evidenced, e.g., by rewriting logic providing semantics to programming languages like C, Java, LLVM, EVM, Scheme, and so on [53]—and a wide range of communication forms, including sophisticated communication models for wireless sensor networks [60] and mobile ad hoc networks [46].
2. Maude and Real-Time Maude combine this expressive and general modeling formalism with providing a range of *automatic* analysis methods, including reachability analysis and LTL and timed CTL model checking.
4. The recent integration of SMT solving into Maude allows us to analyze systems with *continuous* behaviors (and clock skews) symbolically in Maude.
5. In contrast to many formal tools, Maude supports *parametric* atomic propositions, which are needed to define intuitive property specification languages (which are parametric in the input model), as illustrated below.

(Logically) Synchronous CPSs. We target CPSs, which, by definition, are *networks* of embedded systems that typically interact using message passing. Many industrial (distributed) CPSs are *logically synchronous*: at the end of each period all components read messages, perform transitions, and generate output to be read at the end of the next period. Examples of such logically synchronous systems include avionics and automotive systems [9,43,67], networked medical devices [7,36], and other distributed control systems such as the steam-boiler benchmark [1]. However, they have to be realized in a distributed setting, with imprecise local clocks, messaging delays, execution times, and so on.

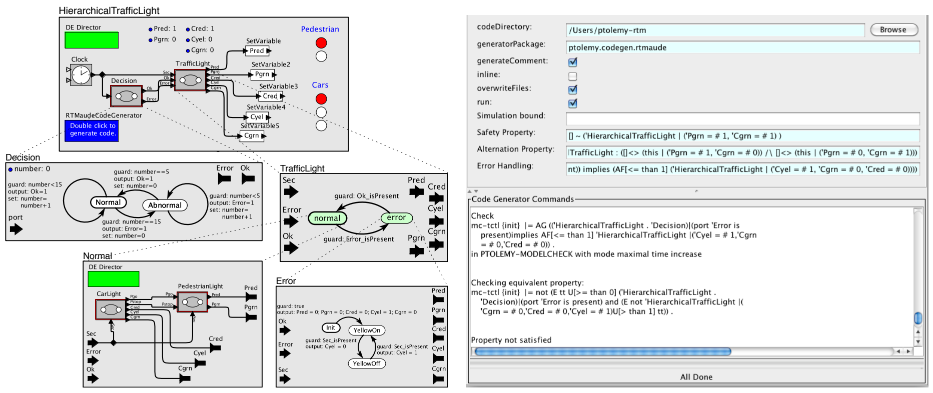
Note that clock synchronization is well understood (e.g., IEEE 1588 [30]); we can therefore often guarantee a bound on the drift of the local clocks. Furthermore, the infrastructure of many critical CPSs—such as cars, airplanes, manufacturing plants, robots, etc.—guarantee bounds on the communication delays.

Synchronizers for CPSs: Taming the Design and Model Checking Complexity. Challenge (3) still remains: model checking distributed CPSs quickly becomes infeasible due to the many different behaviors caused by interleavings.

One way of making model checking of logically synchronous distributed CPSs feasible is to apply *synchronizers for CPSs*, such as the *time-triggered architecture* (TTA) [38,65], PALS ("physically asynchronous, logically synchronous") [3,52], and their generalization MSYNC [13]. The idea is that an idealized *fully synchronous design SD* (*without* distribution, message delays, clock skews, execution times, etc.) under some assumptions Γ about the underlying infrastructure is "logically equivalent" to the desired asynchronous distributed system $async(SD, \Gamma)$.

Synchronizers for CPSs also address Challenge (6): By verifying the synchronous design, we also prove the correctness of the corresponding "implementations" for *all* deployments satisfying the TTA/PALS/MSYNC constraint.

PALS was motivated by an avionics system developed by Rockwell Collins, where model checking even very simplified versions of the (asynchronous) CPS design took more than 30 hours, whereas model checking the logically equivalent synchronous design takes less than 0.1 seconds.



codeDirectory: /Users/ptolemy-rtm
generatorPackage: ptolemy.codegen.rtmaude
generateComment: ✓
inline:
overwriteFiles: ✓
run: ✓
Simulation bound:
Safety Property: [] ~ ('HierarchicalTrafficLight | ('Pgrn = # 1, 'Cgrn = # 1))
Alternation Property: TrafficLight : ([]<> (this | ('Pgrn = # 1, 'Cgrn = # 0)) /\ []<> (this | ('Pgrn = # 0, 'Cgrn = # 1)))
Error Handling: nt)) implies (AF[<= than 1] ('HierarchicalTrafficLight | ('Cyel = # 1, 'Cgrn = # 0, 'Cred = # 0))))

Code Generator Commands

```
Check
mc-tctl {init}  |= AG (('HierarchicalTrafficLight . 'Decision)|(port 'Error is
   present)implies AF[<= than 1] 'HierarchicalTrafficLight |('Cyel = # 1,'Cgrn
   = # 0,'Cred = # 0)) .
in PTOLEMY-MODELCHECK with mode maximal time increase


Checking equivalent property:
mc-tctl {init}  |= not (E tt U[>= than 0] ('HierarchicalTrafficLight .
   'Decision)|(port 'Error is present) and (E not 'HierarchicalTrafficLight |(
   'Cgrn = # 0,'Cred = # 0,'Cyel = # 1)U[> than 1] tt)) .


Property not satisfied
```

All Done

Fig. 1. A Ptolemy II DE model of a traffic light system (left), and its Real-Time Maude timed CTL model checking inside Ptolemy II (right).

AADL. In [15] we provided a Real-Time Maude formal analysis backend for *discrete-event* (DE) models of UC Berkeley's Ptolemy II tool [62], which has industrial users (e.g., Bosch). As shown in Fig. 1 (left), the user can develop her model using the excellent graphical interface of Ptolemy II. She can then click on the blue button to generate a Real-Time Maude model and can enter her Real-Time Maude query, all while staying inside Ptolemy II. Such Real-Time Maude timed CTL model checking revealed a previously unknown flaw in a model of a traffic light system in Ptolemy II's model repository [44] (Fig. 1 (right)).

We wanted to target industrial tools widely used by designers of safety-critical CPSs. In particular, due to our collaboration with Rockwell Collins on avionics applications, we targeted a modeling standard for CPSs used in avionics. The *Architecture Analysis and Design Language* (AADL) [31] is an industrial modeling standard for avionics, automotive, and cyber-physical systems developed and used by companies and organizations such as Carnegie Mellon University, US Army, Honeywell, Rockwell Collins, Lockheed Martin, General Dynamics, Airbus, the European Space Agency, Dassault, EADS, Ford, and Toyota. Model development in AADL is supported by the Open Source AADL Tool Environment (OSATE).

Overview. This invited paper on "the evolution of our software-component-based" research first provides brief preliminaries to Maude and its extensions, and to AADL. Section 3 introduces two avionics applications that motivated this work.

Section 4 summarizes the first step towards providing Maude-based formal analysis of CPSs via AADL: a Real-Time Maude semantics of a "behavioral subset" of AADL [55]. This can be used to verify AADL models of distributed systems. However, explicit-state model checking of distributed CPSs quickly becomes infeasible: The above-mentioned *active standby* avionics system has only three components, and 10 boolean-valued messages are sent/received in each round. There are therefore 10! different orders in which messages can be received in each round, and hence a similar number of different states, so that model checking the AADL model of active standby was not feasible [56].

This inspired work by us and colleagues at UIUC and Rockwell Collins to develop complexity-reducing *formal design and verification patterns*—the topic of an invited FACS 2011 talk [49, 51]—for logically synchronous CPSs that have to be realized in a distributed setting while meeting critical timing constraints. In Section 5 we summarize the PALS formal pattern/synchronizer for CPSs. PALS allows us to design and verify the much simpler underlying synchronous design *SD*—without asynchrony, message delays, clock skews, etc.—so that *SD* satisfies a property ϕ if and only if the distributed realization $\text{PALS}(\mathit{SD}, p, \Gamma)$ does so. We also mention the huge performance gains obtained by using PALS.

Section 6 presents the *Synchronous AADL* modeling language and the *SynchAADL2Maude* tool. *Synchronous AADL* allows designers to model synchronous designs of CPSs in general, and synchronous PALS and TTA models in particular, in AADL. We also present an intuitive property specification language for such AADL models, define the Real-Time Maude semantics of Synchronous AADL, and integrate Synchronous AADL modeling and Real-Time Maude analysis of such models into the OSATE tool environment for AADL.

PALS and much work in the field assume synchronous systems where all components act in synchrony, and therefore operate at the same frequency. However, some CPSs are composed of different kinds of "off-the-shelf" components which operate at *different* frequencies. One prototypical example is an aircraft: The aileron controllers of commercial aircrafts typically operate at frequency 30–100 Hz, whereas the rudder controller operates at 30–50 Hz [2], yet they (and other controllers) need to synchronize to make a safe turn. With José Meseguer we therefore extended PALS (in a FACS 2012 paper [11]), Synchronous AADL, and the SynchAADL2Maude tool to the *multi-rate* setting. In one application, we use Euler approximations of continuous dynamics to analyze a textbook algorithm for turning an airplane. This work is summarized in Section 7.

The airplane turning example emphasizes that many CPSs have *continuous* environments. Defining synchronizers in this case is tricky, since we can no longer abstract from the exact time when a continuous environment is sampled and actuated; times which depend on the imprecise local clocks. To capture all possible behaviors depending on imprecise local clocks, and continuous environments, we combine Maude analysis with SMT solving. We summarize this *Hybrid PALS* synchronizer and the Maude+SMT-based formal analysis of logically synchronous *hybrid* CPSs that we have integrated into OSATE in Section 8.

Finally, we discuss future directions in this line of research in Section 9, and give some concluding remarks in Section 10.

2 Preliminaries

AADL. The *Architecture Analysis & Design Language* (AADL) [31] is an industrial modeling standard used in avionics, aerospace, automotive, medical devices, and robotics to describe an embedded real-time system as an assembly of software components mapped onto a hardware platform. 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.

Software components include *threads* that model the application software and *data* components representing data types. *System* components are the top-level components. A port is a *data* port, an *event* port, or an *event data* port. A component can have different *modes* and 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* [32]. A *periodic* thread is activated at fixed time intervals, and an *aperiodic* thread is activated when it receives an event. When a thread is activated, transitions are applied until a *complete* state is reached (or the thread suspends). The actions performed when a transition is applied may update local variables, generate outputs, and/or suspend the thread. Actions are built from basic actions using sequencing, conditionals, and finite loops.

Maude and Real-Time Maude. Maude [27] is a an executable formal specification language and high-performance analysis tool for distributed systems. A Maude module specifies a *rewrite theory* [48] $\mathcal{R} = (\Sigma, E, L, R)$, where:

- Σ is an algebraic *signature*, i.e., a set of *sorts*, *subsorts*, and *function symbols*.
- (Σ, E) is an *order-sorted equational logic theory* [35] specifying the system's data types, with E a set of (possibly conditional) equations and axioms.
- L is a set of rule *labels*.
- R is a collection of *labeled conditional rewrite rules* **crl** [l] : t => t' **if** *cond*, with t, t' Σ-terms and $l \in L$, that specify the system's local transitions.

A one-step rewrite $t \longrightarrow_{\mathcal{R}} t'$ [48] holds iff t can be rewritten to t' by a rewrite rule in $\mathcal{R}$, and $\longrightarrow^{*}_{\mathcal{R}}$ denotes the reflexive-transitive closure of $\longrightarrow_{\mathcal{R}}$.

A declaration **class** C | att_1 : s_1 , . . . , att_n : s_n declares an *object class* C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C is a term < o : C | att_1 : $val_1, \ldots, att_n$: val_n >, where o (of sort `Oid`) is the object's *identifier*, and val_1 to val_n are the current values of the attributes att_1 to att_n. *Messages* are terms of sort `Msg`. A system state is modeled as a term of sort `Configuration`, and consists of a *multiset* of objects and messages. The system's transitions are specified using rewrite rules.

The `rewrite` command simulates one behavior of the system. The command `search [`n`]` t_0 `=>*` *pattern* `such that` *cond* searches for at most n states reachable from state t_0 that match *pattern* and satisfy the condition *cond*. Maude's linear temporal logic (LTL) model checker checks whether all paths from the initial state satisfy an LTL formula. Such formulas are constructed by (possibly parametric) state propositions of sort `Prop`, and the usual LTL operators `True`, `~` (negation), `\/`, `/\`, `->`, `[]` ("always"), `<>` ("eventually"), `U` ("until"), and `O` ("next").

To specify real-time systems, Real-Time Maude [57,59] adds *tick* rewrite rules `crl [`l`] : {`t_1`} => {`t_2`} in time `τ` if `*cond* to model *time elapse*, where the whole state has the form `{`t`}`. This rule specifies that it takes time $\tau\sigma$ for the state `{`$t_1\sigma$`}` to evolve to the state `{`$t_2\sigma$`}`, provided that the condition *cond* holds for the matching substitution σ. Since the whole state has the form `{`t`}`, and `{_}` cannot appear in a subterm of t, the form of the tick rewrite rules ensures that time elapses in the *entire* state when a tick rule is applied. Other (non-tick) rewrite rules are considered to take zero time. Real-Time Maude provides a range of time-bounded and unbounded search, LTL, and timed CTL model checking commands [44,57]; the time-bounded LTL model checking command is written `(mc `t_0` |=t `*formula*` in time <= `τ`)`.

Maude+SMT. Constrained terms [20,63] symbolically represent (possibly infinite) sets of system states. 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 $x_1, \ldots, x_n$. It represents the set $[\![\phi \parallel t]\!]$ of all instances of the pattern t such that ϕ holds.

A one-step *symbolic rewrite* $\phi_t \parallel t \rightsquigarrow_{\mathcal{R}} \phi_u \parallel u$ on constrained terms [63] symbolically represents a (possibly infinite) set of system transitions. We denote by $\rightsquigarrow^*_{\mathcal{R}}$ the reflexive-transitive closure of $\rightsquigarrow_{\mathcal{R}}$. For a symbolic rewrite $\phi_t \parallel t \rightsquigarrow^*_{\mathcal{R}} \phi_u \parallel u$, there exists a "concrete" rewrite $t' \longrightarrow^*_{\mathcal{R}} u'$ with $t' \in [\![\phi_t \parallel t]\!]$ and $u' \in [\![\phi_u \parallel u]\!]$. Conversely, for any concrete rewrite $t' \longrightarrow^*_{\mathcal{R}} u'$ with $t' \in [\![\phi_t \parallel t]\!]$, there exists a symbolic rewrite $\phi_t \parallel t \rightsquigarrow^*_{\mathcal{R}} \phi_u \parallel u$ with $u' \in [\![\phi_u \parallel u]\!]$.

Maude provides SMT solving and *symbolic reachability analysis* for constrained terms, using connections to Yices2 [28] and CVC4 [21]. Maude supports SMT theories for Booleans, integers, and reals in the SMT-LIB standard [22].

3 Two Motivating Applications

Even though the methods summarized in this paper are applicable to many CPSs, much of the work was motivated by two (classes of) avionics applications:

1. An avionics control system developed at Rockwell Collins whose modeling and, in particular, NuSMV model checking was much harder than expected.
2. Together with a student, Joshua Krisiloff, at the UIUC aeronautics department we wanted to investigate to what degree formal methods could be used to analyze control algorithms for airplanes, for example for turning an airplane.

The Active Standby System. There are multiple physically separated *cabinets* ("main computer systems") on an aircraft, so that physical damage does not take out the computer system. The *active standby* system [54] developed at Rockwell Collins focuses on the logic for deciding which of *two* cabinets is *active*.

The two sides/cabinets receive inputs through communication channels. Each side can fail, and recover after failure. If one side fails, the non-failed side should be the active side. The pilot can also toggle the active status of these sides. The *full* functionality of each side is dependent on the two sides' perception

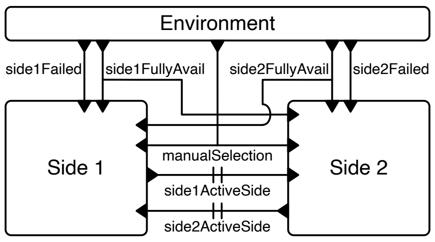


Fig. 2. The active standby system.

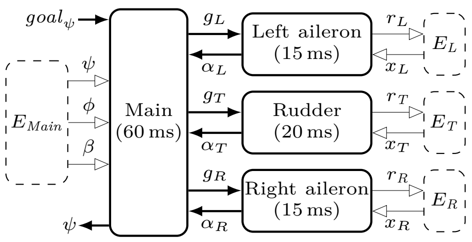


Fig. 3. Turning an airplane.

of the availability of other system components. The architecture of the system is shown in Figure 2. The system consists of three components: Side 1, Side 2, and Environment. The Environment component is an abstract representation of other components that interact with Side 1 and Side 2. The components have the same period and dispatch at the same time. Each time Environment dispatches, it sends five Boolean (nondeterministically generated) values, denoting, respectively, whether side i is failed or not fully available, and whether the pilot wants to change the active side ("manualSelection"). The connections between the two sides are "delayed;" a message sent in one round is read in the next round.

There are five important properties that active standby should satisfy [52,54]. One of them is: *A side that is not fully available should not be the active side if the other side is fully available (provided neither side has failed, the availability of a side has not changed, and the pilot has not made a manual selection).*

Turning an Airplane. An airplane turns by *rolling* in the direction of the turn by moving its two *ailerons* (flaps attached to each wing), and reduces the *adverse yaw* caused by the rolling by moving its *rudder*. To achieve a smooth turn these devices should synchronize, even though their controllers have different periods.

A controller orchestrates these devices to turn an aircraft [9,16]. As shown in Figure 3, there are four components. The main controller has period 60 ms, the aileron subcontrollers have period 15 ms, and the rudder subcontroller has period 20 ms. Each component interacts with its *physical* environment.

Each subcontroller $M \in \{L, T, R\}$ determines the moving rate r_M to move its surface towards the goal angle g_M provided by the main controller. It also sends the "sampled" angle α_M to the main controller. Its physical environment E_M specifies the continuous behavior of the surface angle x_M by the control command r_M. The angle x_M gradually changes according to the ODE: $\dot{x}_M = r_M$.

The main controller determines the goal angles for the subcontrollers to make a coordinated turn, given a goal direction $goal_\psi$ and the sampled surface angles $(\alpha_L, \alpha_T, \alpha_R)$ from the subcontrollers. The physical environment E_{Main} specifies the current direction ψ, the roll angle ϕ, and the yaw angle β. The lateral dynamics of an aircraft can be modeled as the following ODEs [69]:

$$\begin{aligned}
&\dot{\psi} = (g/V)\tan\phi, && \dot{\beta} = Y(\beta,\vec{x})/mV - r + (g/V)\cos\beta\sin\phi,\\
&\dot{\phi} = p && \dot{p} = (c_1 r + c_2 p)\cdot r\tan\phi + c_3 L(\beta,\vec{x}) + c_4 N(\beta,\vec{x}),\\
& && \dot{r} = (c_8 p - c_2 r)\cdot r\tan\phi + c_4 L(\beta,\vec{x}) + c_9 N(\beta,\vec{x}),
\end{aligned}$$

where g is the gravitational constant, m is the mass of the aircraft, V is the velocity of the aircraft, p is the rolling moment, r is the yawing moment, and Y, L, and N are linear functions of β and the surface angles $\vec{x} = (x_L, x_V, x_R)$.

The yaw angle should be close to 0 during a turn, and the airplane should reach the goal direction with both roll angle and yaw angle close to 0 [9].

4 Formal Semantics and Analysis for "Behavioral AADL"

Although AADL is an industrial (SAE, Society of Automotive Engineers) standard modeling language for safety-critical embedded systems, it lacks a formal semantics. In [55], with colleagues at UIUC and Leicester, we therefore defined, for the first time (see [55] for a discussion on related work) an executable formal semantics for what we call a "behavioral subset" of AADL, in Real-Time Maude.

AADL is a huge and complex standard. Formal approaches therefore target limited fragments of AADL. We targeted a behavioral subset of AADL suitable to define distributed software designs—with all software structuring mechanisms of AADL (system, process, and thread components); (event, event data, and data) ports and their connections; mode-specific properties; mode transitions; and both periodic, aperiodic, sporadic, and background thread dispatch; and so on. We did not target the "hardware platform" part of AADL. Thread behaviors are modeled using AADL's Behavior Annex standard [32].

In [55] we defined the Real-Time Maude semantics for this subset of AADL. AADL components are *hierarchical*; the formal model should reflect the hierarchical structure, e.g., for understanding or "mapping back" analysis results.

Some key Real-Time Maude features that are crucial to define a decently effective semantics for this fragment of AADL include:

- An expressive formalism is needed to capture this rich subset of AADL, including a Turing-complete programming language used to define transitions.
- *Hierarchical objects* (an object attribute may have sort `Configuration` and therefore contain subconfigurations) allow us to define the semantics in an object-based style, yet preserve the hierarchical structure of AADL models.
- Rewriting logic's division into equations and rewrite rules—where only the latter contribute to the state space—is crucial to provide an efficient semantics. For example, "executing" the program associated with each AADL transition can be done equationally (i.e., in one atomic step).

Real-Time Maude Semantics. We can only give a very brief sample of our semantics, and refer to [56] for details. The key observation is that the semantics of a component-based language naturally can be defined in an object-oriented style, where each component instance is modeled as an object. As mentioned, the hierarchical structure of AADL components is reflected in the nested structure of objects. Any AADL component instance is represented as an object instance of a subclass of the following class `Component`, which contains the attributes common to all kinds of components (systems, processes, threads, etc.):

```
class Component | features : Configuration,   subcomponents : Configuration,
                  properties : Properties,    connections : ConnectionSet,
                  modes : Modes,              inModes : ModeNameSet .
```

The attribute `features` denotes the features of a component (i.e., its ports); `subcomponents` denotes the subcomponents of the object; `properties` denotes its *properties*, such as the dispatch protocol for threads; `connections` denotes the set of port connections of the object; `modes` contains the object's mode transition system; and `inModes` gives the set of modes (of the immediate supercomponent) in which the component is available. The `Thread` class is declared as follows:

```
class Thread | behavior : ThreadBehavior,    status : ThreadStatus,
               deactivated : Bool .
subclass Thread < Component .
```

The `behavior` attribute denotes the transition system associated with the thread. The `status` indicates the current status of the thread (`active`, `completed`, `suspended`, etc.). The attribute `deactivated` indicates whether the thread is deactivated because it is not in the current "active" modes of the system.

The following rewrite rule specifies the execution of an *active* thread. If the thread is in state `L1`, a transition from `L1` whose guard evaluates to `true` is executed. The resulting `status` is `sleeping(...)` if the statement list `SL` contains `delay` statements; otherwise, the thread is `completed` or `inactive` if the resulting state `L2` is a complete state, and remains `active` if not:

```
crl [apply-transition] :
< O : Thread | status : active,  deactivated : B,  features : PORTS,
               behavior : states current: L1 complete: LS1 others: LS2
                          state variables VAL
                          transitions (L1 -[GUARD]-> L2 {SL}) ; TRANSITIONS >
=>
< O : Thread | status : (if SLEEP-TIME > 0 then sleeping(SLEEP-TIME) else
                            (if (not L2 in LS1) then active else
                             (if B then inactive else completed fi) fi) fi),
               features : NEW-PORTS,
               behavior : states current: L2 complete: LS1 others: LS2
                          state variables NEW-VAL
                          transitions (L1 -[GUARD]-> L2 {SL}) ; TRANSITIONS >
 if evalGuard(GUARD, PORTS, VAL)
 /\ transResult(NEW-PORTS, NEW-VAL, SLEEP-TIME) :=
            executeTransition(L1 -[GUARD]-> L2 {SL}, PORTS, VAL) .
```

The (equationally defined) function `executeTransition` executes a transition with `PORTS` the states of the ports and `VAL` the values of the state variables. The function returns a triple `transResult(`p, σ, t`)`, where p is the state of the ports after the execution, σ is the resulting values of the state variables, and t is the sum of the `delays` in the transition actions. The transitions are modeled as a multiset; therefore, *any* enabled transition can be applied in the rule.

Tool Support. This first work did not fully integrate formal analysis into the OSATE tool environment for AADL: The *ADDL2Maude* OSATE plugin used OSATE's code generation facilities to automatically generate Real-Time Maude models from AADL specifications. However, the analysis required running commands in Real-Time Maude. Nevertheless, the user did not have to know the Real-Time Maude representation of her AADL model, since we provided convenient syntax for defining state patterns. The term

```
value of v in component fullComponentName in globalComponent
```

gives the value of the state variable v in the thread identified by the full component name *fullComponentName* in the system in state *globalComponent*, and

```
location of component fullComponentName in globalComponent
```

gives the current location/state of the transition system in the given thread. For LTL model checking, we pre-defined parametric atomic propositions such as *full thread name* `@` *location*, which holds when the thread is in state *location*.

Applications. We used AADL2Maude and Real-Time Maude analysis of the generated model to analyze an AADL model developed at UIUC of a network of medical devices consisting of a *controller*, a *ventilator machine* that assists a patient's breathing during surgery, and an *X-ray* device. When a button is pushed to take an X-ray, and the ventilator machine has *not* paused in the past 10 minutes, the ventilator machine pauses for two seconds, starting one second after the button is pushed, and the X-ray should be taken after two seconds.

To execute the system, we added a *test activator* that pushes the button every second. We then used reachability analysis to check whether an *undesired* state, where the X-ray thread is in state `xray` (X-ray being taken) and the ventilator is *not* in state `paused`, can be reached (which leads to blurry pictures):

```
Maude> (search [1]  initialize({MAIN system Wholesys . impl}) =>* {CONF}
        such that  ((location of component (MAIN -> Xray -> xmPr -> xmTh)
                    in CONF) == xray  and  (location of component (MAIN ->
                    Ventilator -> vmPr -> vmTh) in CONF) =/= paused) .)

Solution 1        CONF --> ...
```

(The unexpected result showed that such a bad state can indeed be reached.) We used time-bounded LTL model checking to verify that an X-ray must be taken within three seconds of the start of the system (surprisingly, this command returned a counterexample revealing a subtle and previously unknown error):

```
Maude> (mc initialize({MAIN system Wholesys . impl}) |=t
          <> ((MAIN -> Xray -> xmPr -> xmTh) @ xray) in time <= 3 .)
```

We also wanted to verify an AADL model of the *active standby* system. However, no serious analysis finished within reasonable time, as explained in [56].

Nevertheless, this work defines a formal semantics of a useful subset of AADL, and could be used to find subtle flaws in existing AADL models.

5 The PALS Synchronizer for CPSs

Model checking even simplified versions (e.g., without clock drifts and communication delays) of the *active standby* system turned out to be unfeasible. Even *designing* the system, to ensure that messages are read in the correct round in the presence of fast and slow local clocks and message delays, required modeling buffering of both incoming and outgoing messages and subtle use of (local-clock-based) timers to read and transmit messages at the correct times.

The active standby system shares many characteristics with (distributed) CPSs in fields such as avionics, cars, robotics, and other control systems:

– The “underlying logic” is *synchronous*: At the beginning/end of each “period” all components should in lockstep read incoming messages and perform local transitions, which change the state of each component and generate messages for the *next* iteration. However, the system has to be realized in a distributed setting, with message delays, imprecise local clocks, execution times, etc.
– Cars, airplanes, factories, robots, etc., typically use dedicated local networks where network delays are bounded.
– Clock synchronization is well understood; e.g., the Precision Time Protocol (IEEE 1588) achieves sub-microsecond clock accuracy on local area networks [29]. We can therefore often give a bound ϵ on the skew of the clocks.

All of this inspired us and colleagues at UIUC and Rockwell Collins to define the PALS (“physically asynchronous, logically synchronous”) *formal pattern* [51] to greatly simplify both the design and the verification of these “logically synchronous” (distributed) CPSs when the infrastructure guarantees bounds μ, α, and ϵ on the communication delays, transition computation times, and clock drifts, respectively [3, 52, 66]. For such bounds $\Gamma = (\mu, \alpha, \epsilon)$, PALS is a mapping

$$(SD, p, \Gamma) \;\mapsto\; \mathrm{PALS}(SD, p, \Gamma)$$

where SD is the underlying synchronous design, p is the period of the components, and $\mathrm{PALS}(SD, p, \Gamma)$ is the corresponding distributed system model. SD is formalized as the synchronous composition of a collection of state (“Mealy”) machines, each with typed input and output ports, whose ports are connected by a *wiring diagram*. Each machine M has a transition relation $\delta_M \subseteq (\mathit{inputs} \times$

State) $\times$ (*State*, *outputs*) and performs one transition in each iteration of the system. The distributed real-time system PALS(SD, p, Γ) is formalized in [52] using Real-Time Maude. It is proved in [52] that as long as the *PALS constraint* $p \geq \mu + 2\epsilon + \max(2\epsilon, \alpha)$ is satisfied (the difference between two clocks is always less than 2ϵ), then *SD* and PALS(SD, p, Γ) satisfy the same CTL* formulas. (The constraint in [52] also takes into account the minimum network delay.)

It is therefore sufficient to specify and verify the much simpler underlying synchronous model. PALS then provides the corresponding distributed model, which satisfies the same properties as the synchronous design.

Synchronizers (like GALS and LTTA) relating synchronous and asynchronous systems are well known. However, very few, such as the *time-triggered architecture* (TTA) [38, 65], target CPSs with bounded network delays and clock skews, and therefore do not guarantee timeliness of the resulting asynchronous system (see [52] for details). TTA has similar assumptions as PALS, but has a significantly longer optimal period p than PALS, and some other differences [13, 52, 68].

Effectiveness and Application. In [52] we modeled the synchronous PALS design of active standby in Maude, as well as a much simplified asynchronous model, with perfect clocks and no execution times: The synchronous model has **185** reachable states and could be model checked in less than a second. The asynchronous model has **3,047,832** reachable states when message delays are 0; a simple reachability command explores these states in 2,000 seconds. When the message delay could be 0 or 1, attempts at exploring the state space aborted.

With PALS we could easily analyze the requirements of active standby using LTL model checking: most did not hold. Inspecting the counterexamples, we came up with, and verified, modified properties, which were the properties ones found by the Rockwell Collins team using NuSMV and the PALS methodology.

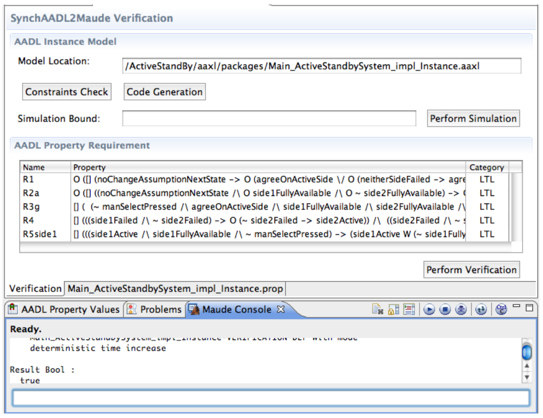
6 Synchronous AADL

Synchronous AADL [14] is a subset of AADL in which synchronous designs in general, as well as synchronous PALS models, can be specified. Synchronous AADL is defined as a behavioral subset of AADL explained in Section 4, together with syntactic constraints to identify AADL models that can be considered synchronous: e.g., threads have periodic dispatch; components have only data ports; and connections between threads are delayed. Each AADL construct in the subset has the same meaning in AADL and Synchronous AADL, and properties specific to Synchronous AADL are declared using the property set `SynchAADL`.

The formal semantics of Synchronous AADL is defined in Real-Time Maude, but now specifies the synchronous composition of AADL components.

Real-Time Maude analysis of Synchronous AADL models is integrated into OSATE using the SynchAADL2Maude plugin [18]. Given a Synchronous AADL model *SD*, the tool checks whether *SD* is a valid Synchronous AADL model, generates the corresponding Real-Time Maude model, and invokes Real-Time Maude to analyze whether *SD* satisfies given LTL properties. SynchAADL2Maude

```
<verification>
  ...
  <command>
    <name>R4</name>
    <value type = "ltl">
     [](  ((side1Failed /\ ˜ side2Failed) ->
            O (˜ side2Failed -> side2Active))
        /\ ((side2Failed /\ ˜ side1Failed) ->
            O (˜ side1Failed -> side1Active)))
    </value>
  </command>
  <definition>
    <name>side1Failed</name>
    <value>
      value of s1Failed in component
       (MAIN -> s1 -> sideProcess ->
        sideThread) is true
    </value>
  </definition>
  ...
</verification>
```


SynchAADL2Maude Verification

AADL Instance Model

Model Location: /ActiveStandBy/aaxl/packages/Main_ActiveStandbySystem_impl_Instance.aaxl

Simulation Bound:

AADL Property Requirement

Name	Property	Category
R1	O ([] (noChangeAssumptionNextState -> O (agreeOnActiveSide \/ O (neitherSideFailed -> agre	LTL
R2a	O ([] ((noChangeAssumptionNextState /\ O side1FullyAvailable /\ O ~ side2FullyAvailable) -> (	LTL
R3g	[] ((~ manSelectPressed /\ agreeOnActiveSide /\ side1FullyAvailable /\ side2FullyAvailable /\	LTL
R4	[] (((side1Failed /\ ~ side2Failed) -> O (~ side2Failed -> side2Active)) /\ ((side2Failed /\ ~ s	LTL
R5side1	[] (((side1Active /\ side1FullyAvailable /\ ~ manSelectPressed) -> (side1Active W (~ side1Fully	LTL

```
Ready.
    deterministic time increase

Result Bool :
  true
```

Fig. 4. Properties in the XML format (left) and SynchAADL2Maude window (right).

provides predefined atomic propositions to easily specify system properties, and LTL properties to be analyzed are managed by an XML file. Figure 4 shows an example of such properties and the SynchAADL2Maude window for the active standby system, where the analysis results are shown in the Maude Console.

7 Multirate PALS and MR-SynchAADL

Whereas PALS, TTA, and synchronous systems in general require that all components have the same period, different ("off-the-shelf") controllers often operate at different frequencies, yet need to synchronize, as in the airplane turning system in Section 3. We have therefore extended PALS, the Synchronous AADL modeling language, and the SynchAADL2Maude tool to the multirate setting [11, 12, 17].

Composing components with different periods is tricky, since a controller with period 30ms receives and sends messages every 30ms, whereas one with period with 50ms only does so every 50ms. We address this as follows:

– A component may only communicate with components whose period is a multiple of its own period (possibly through a hierarchy); or vice versa.
– User-defined *input adaptors* turn one output value from a slow component to k input values for a connecting component which is k times faster. Likewise, the slow component's input adaptor turns k outputs from the fast component into a single input to the slow component.

Since a synchronous system can be represented as a single machine [12, 52], we can have hierarchical models. Figure 5 shows an example of multirate systems, and Figure 6 shows its corresponding hierarchical synchronous design, where each machine and its local environment are annotated with its period.

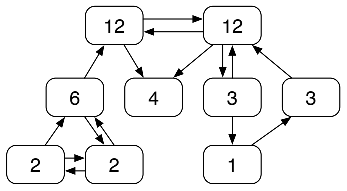


Fig. 5. A hierarchical multirate system.

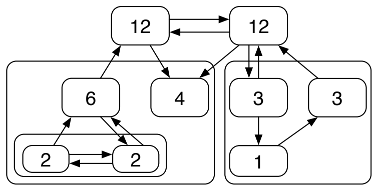


Fig. 6. A multirate synchronous design.

Multirate PALS. We "slow down" the faster components, so that all components in a multirate synchronous design operate in lock-step as in the single-rate case. A fast component slowed down by a factor k performs k *internal transitions* during one (slow) period. It consumes k inputs and produces k outputs at each port in each slow step. A slow controller, which should only read *one* input (in each input port) during such a slow step, therefore uses an *input adaptor* to transform such a k-tuple output into a single input value; and vice versa for slow-to-fast connections.

For a multirate synchronous design *SD*, bounds Γ, and a global period p, Multirate PALS gives the distributed real-time system $\mathcal{M}\mathrm{PALS}(\mathit{SD}, p, \Gamma)$ where each component operates according to its own period [11, 12]. However, a fast machine may not be able to finish all of its k internal transitions in a slow period p before the output messages must be sent. If only $k_f < k$ outputs can be sent before the deadline, the input adaptor must ignore the last $k - k_f$ values in a k-tuple input. $\mathcal{M}\mathrm{PALS}(\mathit{SD}, p, \Gamma)$ and *SD* satisfy the same properties when all input adaptors satisfy this condition and the PALS constraint is satisfied [12].

Multirate Synchronous AADL. To specify hierarchical multirate synchronous designs in AADL, we defined the Multirate Synchronous AADL language [17] as a sublanguage of AADL with a property set `MR_SynchAADL`. The MR-SynchAADL plugin supports modeling and formal analysis of Multirate Synchronous AADL models within OSATE, with a dedicated property specification language fully integrated with the OSATE editor (e.g., supporting syntax highlighting).

Multirate Synchronous AADL extends Synchronous AADL by allowing different components to have different periods (satisfying the above conditions), and by allowing us to associate an *input adaptor* α to an input port using the property `MR_SynchAADL::InputAdaptor =>` α. 1-to-k input adaptors map a single value to a k-vector of values, and k-to-1 input adaptors map a k-vector of values to a single value. Multirate Synchronous AADL provides a collection of predefined input adaptors, including `repeat_input`—mapping v to $(v, v, \ldots, v)$—, and `last`—mapping $(v_1, \ldots, v_k)$ to v_k—, and so on. The formal semantics of Multirate Synchronous AADL is also defined in Real-TIme Maude.

Application. In [9, 12, 17], we modeled and analyzed the Multirate PALS synchronous model of the airplane example in Section 3 using Real-Time Maude.

The continuous behavior was numerically approximated using Euler's method. The formal analysis revealed a flaw that caused an unsafe turn. This led to a redesign of the system, which was verified using model checking. Again, (bounded) model checking of a highly simplified *asynchronous* model was unfeasible due to the many interleavings caused. The airplane example was also modeled in Multirate Synchronous AADL and analyzed using MR-SynchAADL in [17].

8 Hybrid PALS and HybridSynchAADL

CPS controllers may interact with *physical* environments, whose *continuous* dynamics can be modeled as ordinary differential equations (ODEs). To precisely analyze such CPSs with the PALS methodology, we have developed the Hybrid PALS synchronizer [16] and the HybridSynchAADL language and tool [40, 41].

Hybrid PALS extends Multirate PALS to CPSs with continuous dynamics. In contrast to (Multirate) PALS, we cannot abstract away the times at which physical states are sampled and actuated. We must also to take into account *imprecise local clocks*, since such *sampling and actuating times* depend on them.

A formal analysis of such CPSs involves an infinite number of continuous trajectories depending on imprecise local clocks. We therefore use Maude combined with SMT solving to *symbolically* encode all possible continuous behaviors, and define a symbolic (Maude with SMT) semantics for HybridSynchAADL language.

Hybrid PALS. We consider multirate systems consisting of discrete controllers and physical environments, as shown in Figure 7. A controller M is a nondeterministic machine parameterized by any behavior of its physical environment E_M. Environments can also be physically correlated (dashed lines in Figure 7), meaning that changes in one environment immediately affect another.

A state of a physical environment E_M is a valuation of real-valued parameters $\vec{x} = (x_1, \dots, x_l)$. The continuous behavior of E_M is modeled by systems of ODEs that specify different trajectories of $\vec{x}$ over time. A control command from its controller M defines *which* trajectories E_M follows. In Figure 8, E_M follows trajectory τ_1 from state v_1 for duration $t_2 - t_1$ when command a_1 is received.

Let $c_M(i)$ denote the *global time* when the i-th period of a controller M begins according to its local clock. M samples the state of E_M at time $c_M(i) + t_s$, where t_s is a value in its sampling time interval. M then performs a transition and determines a new control command. E_M receives the new command at time $c_M(i) + t_a$, where t_a is a value in its actuating (or response) time interval.

A hybrid synchronous design $\mathit{SD} \upharpoonright E$ is composed of a multirate synchronous design SD and a collection E of physical environments. The "discrete" behavior is given by the synchronous design SD *restricted by* the behavior of E, and the "continuous" behavior is given by a set of trajectories of E *realizable by* control commands from SD.

For a hybrid synchronous design $\mathit{SD} \upharpoonright E$, a global period p, and bounds $\mathit{\Gamma}$, Hybrid PALS produces the distributed *hybrid* system $\mathcal{M}\mathrm{PALS}(\mathit{SD}, p, \mathit{\Gamma}) \upharpoonright E$, where each controller interacts with its physical environment in E. Hybrid PALS

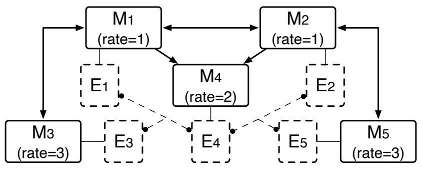


Fig. 7. A hybrid multirate system.

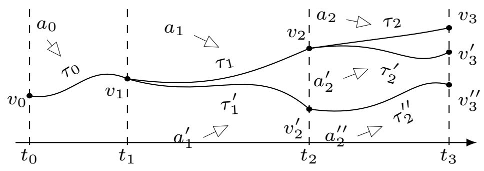


Fig. 8. A controlled physical environment.

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

```
system implementation RoomEnv.impl
 subcomponents x: data Base_Types::Float; p: data Base_Types::Float;
 connections   C: port x -> temp;          R: port power -> p;
 modes         hOff: initial mode;         hOn: mode;
               hOff -[on_ctrl]-> hOn;      hOn -[off_ctrl]-> hOff;
 properties    Hybrid_SynchAADL::ContinuousDynamics =>
                 "x(t)= x(0)- 0.1*(x(0)- p/0.1)* t;"  in modes (hOn),
                 "x(t)= x(0)*(1 - 0.1 * t);"  in modes (hOff);
end RoomEnv.impl;
```

Fig. 9. An environment component.

ensures that both the synchronous design $SD \upharpoonright E$ and the distributed hybrid system $\mathcal{M}\text{PALS}(SD, p, \Gamma) \upharpoonright E$ exhibit exactly the same set of trajectories of E.

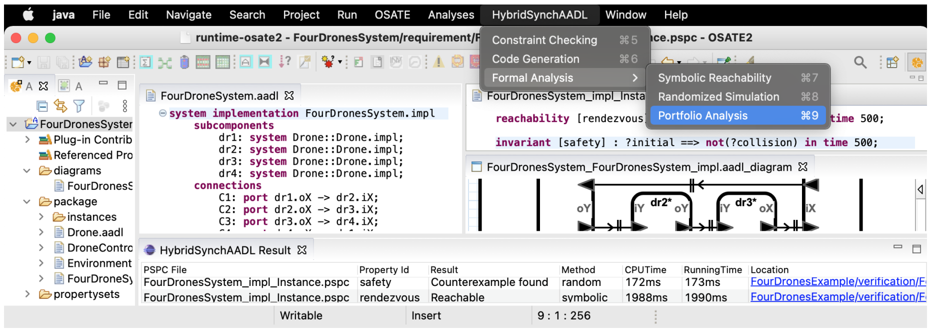
HybridSynchAADL. The HybridSynchAADL language and tool [39–41] extends (Multirate) Synchronous AADL and MR-SynchAADL to hybrid synchronous designs. A physical environment is modeled as an environment component in HybridSynchAADL, and can have different *modes* to specify different trajectories. The continuous dynamics in is specified using the property `Hybrid_SynchAADL::ContinuousDynamics`. Figure 9 shows an example of an environment components with two modes. A controller is an ordinary software component. It declares the three properties `Hybrid_SynchAADL::Max_Clock_Deviation`, `Hybrid_SynchAADL::Sampling_Time`, and `Hybrid_SynchAADL::Response_Time` to specify the maximal clock skew, and sampling and actuating time intervals.

We defined the Maude+SMT semantics of HybridSynchAADL for *single-rate* designs. A *constrained object* of the form ϕ `||` *obj* symbolically represents (infinitely many) instances of *obj* satisfying the constraint ϕ. The behavior of individual components for one synchronous iteration is specified by the operation `executeStep`, defined by rewrite rules on constrained terms. The synchronous step of the entire system is then formalized by the following rule, which captures all possible behaviors from any instance of $\{\phi$ `||` *obj*$\}$:

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

Tool Support. HybridSynchAADL supports Maude+SMT reachability analysis of single-rate hybrid synchronous designs within OSATE. The property specification language allows the user to specify reachability properties of the form

$$\texttt{reachability } \varphi_{init} \texttt{ ==> } \varphi_{goal} \texttt{ in time } \tau,$$



```
system implementation FourDronesSystem.impl
    subcomponents
        dr1: system Drone::Drone.impl;
        dr2: system Drone::Drone.impl;
        dr3: system Drone::Drone.impl;
        dr4: system Drone::Drone.impl;
    connections
        C1: port dr1.oX -> dr2.iX;
        C2: port dr2.oX -> dr3.iX;
        C3: port dr3.oX -> dr4.iX;
```

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

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

Fig. 10. HybridSynchAADL window in OSATE.

which holds if a state satisfying φ_{goal} is reachable from a state satisfying φ_{init} within time τ. The tool provides three analysis methods: symbolic reachability analysis using Maude+SMT; randomized simulation, which repeatedly runs the model by randomly choosing concrete data values, sampling and actuating times, etc.; and portfolio analysis, which invokes randomized simulation and symbolic reachability analysis in parallel using multithreading.

Applications. We used HybridSynchAADL to model and analyze distributed drones that collaborate to perform rendezvous, formation control, and packet delivery [39–41]. For rendezvous, we symbolically analyzed two time-bounded properties: drones do not collide within time 500 (`[safety]`), and all drones gather together within time 500 (`[rendezvous]`). These properties use three user-defined propositions `init`, `collide`, and `gather`, and are defined as follows in our property specification language:

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

Figure 10 shows the tool interface that is fully integrated into OSATE, where the analysis results are shown in the `Result` view. We compare the performance of HybridSynchAADL's symbolic analysis with four hybrid systems reachability analysis tools, such as [25, 26, 33, 37]. The experiments showed that (in most cases) HybridSynchAADL outperforms these state-of-the-art tools.

9 Future Research Directions

We briefly mention some future research directions on three key components of our formal model engineering approach to developing reliable distributed CPSs: (1) *synchronizers* to reduce the design and analysis complexity; (2) "AADL-based" *modeling languages* with formal semantics; and (3) *formal analysis techniques*.

Synchronizers. The MSYNC synchronizer [13] generalizes Multirate PALS and TTA, and should be extended to CPSs with continuous dynamics. We should also extend other synchronizers, such as loosely time-triggered architectures [23, 70], to the multirate setting with continuous dynamics.

AADL-based Modeling Languages with Formal Semantics. We should extend the Maude-SMT semantics of Hybrid Synchronous AADL to multirate systems. The Maude-SMT analysis in HybridSynchAADL can only deal with polynomial continuous dynamics. To support general classes of ODEs, we should integrate Maude with ODE solvers, such as dReal [34]. The airplane turning algorithm in Section 3 could then be symbolically analyzed using HybridSynchAADL. We should also support more AADL language constructs and the modeling of (Hybrid) MSYNC models. Language extensions should be guided by more applications, such as the steam-boiler controller [1] and aerospace systems [24, 47].

Formal Analysis. We should support formal analysis methods beyond explicit-state LTL/TCTL model checking and symbolic reachability analysis, e.g., STL model checking [10, 42, 73] and statistical model checking [4, 45, 64].

HybridSynchAADL employs state merging [19,20] to improve the performance of Maude+SMT symbolic analysis. We can further improve the performance by applying incremental rewriting modulo SMT [71]. Abstracting away imprecise local clocks in Hybrid PALS, e.g., by over-approximation, should also significantly improve the scalability of the analysis.

Maude+SMT analysis opens up the possibility of having *parametric* initial states, and automatically *synthesizing* parameter values that make the system satisfy desired properties, as we have done for timed automata and Petri nets [5,6]. We should explore such parameter synthesis for logically synchronous CPSs.

10 Concluding Remarks

Equipped with the expressive Maude and Real-Time Maude formalisms—which should be able to capture the semantics of industrial modeling languages—whose tools nevertheless provide powerful *automatic* formal analyses, our goal is to integrate formal analysis of CPSs into modeling tools used in industry, so that a designer can formally analyze her models without knowing formal methods.

Motivated by projects at UIUC, and having access to AADL models of a network of medical devices and of a Rockwell Collins-developed avionics system, we decided to target the component-based industrial modeling standard AADL. In this paper we give, for the first time, an overview of this effort.

We first gave a Real-Time Maude semantics to a significant subset of the "software" parts of AADL, and used Real-Time Maude analysis to find subtle flaws in the AADL model of the medical system. However, we could not analyze the avionics system, whose NuSMV model checking also caused major problems.

Observing that the active standby system and many other CPSs have "logically synchronous" designs but have to be realized as distributed systems on local area networks, we developed the PALS *formal pattern* as one of the first "synchronizers for CPSs."[3] It is therefore sufficient to model and verify the much simpler underlying synchronous designs. In particular, PALS reduced the intractable task of model checking active standby to one that could be done in less than a second.

To make PALS-based formal analysis of "logically synchronous" distributed systems on local area networks available to AADL modelers, we identified an annotated sublanguage of AADL, called Synchronous AADL, that can be used by AADL modelers not only to model synchronous PALS models, but also synchronous systems in general, and integrated Real-Time Maude analysis of Synchronous AADL models into the OSATE tool environment for AADL.

Since controllers may operate at different frequencies, yet need to synchronize, we extended PALS and Synchronous AADL to the multirate setting. Multirate PALS, introduced in a FACS 2012 paper, was, to the best of our knowledge, the first synchronizer for multirate CPSs. We applied our methodology to an airplane turning control algorithm and found that it did not provide a safe yaw.

The airplane turning example emphasized that many CPSs interact with *continuous* environments. We therefore extended our methods to hybrid systems. Hybrid PALS cannot abstract from the times when sensing and actuating such environments happen, which depend on local clocks with bounded but unknown skews. The recent integration of SMT solving with Maude allows us to capture all the possible continuous behaviors symbolically. We defined the HybridSynchAADL language and integrated Maude+SMT-based simulation and reachability analysis into OSATE. Experiments on collaborating UAVs showed that HybridSynchAADL analysis in many cases outperforms state-of-the-art hybrid systems reachability tools such as HyComp, Flow*, SpaceEx, and dReach [41].

Finally, we outlined some future directions in this line of work.

Acknowledgments We thank Olga Kouchnarenko and the organizers of FACS 2023 for inviting us to present this work summarizing some of our "software component-based" research to celebrate the 20th anniversary of the FACS conference. In this paper we report on research initiated by some of us and Darren Cofer and Steven Miller at Rockwell Collins and José Meseguer and Lui Sha at University of Illinois at Urbana-Champaign; we sincerely thank them all, as well as the coauthors of all the papers summarized in this paper. Bae was supported by the NRF grants funded by the Korea government (No. 2021R1A5A1021944 and No. RS-2023-00251577).

References

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

[3] We do not have space to discuss related work in this overview paper, but our papers have extensive discussions of related work that justify our claims.

2. Al-Nayeem, A., Sha, L., Cofer, D.D., Miller, S.M.: Pattern-based composition and analysis of virtually synchronized real-time distributed systems. In: 2012 IEEE/ACM Third International Conference on Cyber-Physical Systems. pp. 65–74. IEEE (2012)
3. Al-Nayeem, A., Sun, M., Qiu, X., Sha, L., Miller, S.P., Cofer, D.D.: A formal architecture pattern for real-time distributed systems. In: Proc. RTSS. pp. 161–170. IEEE, USA (2009)
4. AlTurki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: International Conference on Algebra and Coalgebra in Computer Science (CALCO'11). LNCS, vol. 6859, pp. 386–392. Springer (2011)
5. Arias, J., Bae, K., Olarte, C., Ölveczky, P.C., Petrucci, L., Rømming, F.: Rewriting logic semantics and symbolic analysis for parametric timed automata. In: 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS'22). pp. 3–15. ACM (2022)
6. Arias, J., Bae, K., Olarte, C., Ölveczky, P.C., Petrucci, L., Rømming, F.: Symbolic analysis and parameter synthesis for time Petri nets using Maude and SMT solving. In: International Conference on Applications and Theory of Petri Nets and Concurrency (PETRI NETS'23). LNCS, vol. 13929, pp. 369–392. Springer (2023)
7. Arney, D., Jetley, R., Jones, P., Lee, I., Sokolsky, O.: Formal methods based development of a PCA infusion pump reference model: Generic infusion pump (GIP) project. In: HCMDSS-MDPnP. pp. 23–33. IEEE (2007)
8. Bae, K., Ölveczky, P.C., Feng, T.H., Tripakis, S.: Verifying Ptolemy II discrete-event models using Real-Time Maude. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM'09. Lecture Notes in Computer Science, vol. 5885, pp. 717–736. Springer Verlag (2009)
9. Bae, K., Krisiloff, J., Meseguer, J., Ölveczky, P.C.: Designing and verifying distributed cyber-physical systems using Multirate PALS: An airplane turning control system case study. Science of Computer Programming **103**, 13–50 (2015)
10. Bae, K., Lee, J.: Bounded model checking of signal temporal logic properties using syntactic separation. Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
11. Bae, K., Meseguer, J., Ölveczky, P.C.: Formal patterns for multi-rate distributed real-time systems. In: Formal Aspects of Component Software. LNCS, vol. 7684, pp. 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
12. Bae, K., Meseguer, J., Ölveczky, P.C.: Formal patterns for multirate distributed real-time systems. Science of Computer Programming **91**, 3–44 (2014)
13. Bae, K., Ölveczky, P.C.: MSYNC: a generalized formal design pattern for virtually synchronous multirate cyber-physical systems. ACM Transactions on Embedded Computing Systems (TECS) **20**(5s), 1–26 (2021)
14. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous AADL and its formal analysis in Real-Time Maude. In: Proc. ICFEM'11. LNCS, vol. 6991. Springer, Berlin, Heidelberg (2011)
15. Bae, K., Ölveczky, P.C., Feng, T.H., Lee, E.A., Tripakis, S.: Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude. Science of Computer Programming **77**(12), 1235–1271 (2012)
16. Bae, K., Ölveczky, P.C., Kong, S., Gao, S., Clarke, E.M.: SMT-based analysis of virtually synchronous distributed hybrid systems. In: Proc. HSCC. pp. 145–154. ACM, New York, NY, USA (2016)
17. Bae, K., Ölveczky, P.C., Meseguer, J.: Definition, semantics, and analysis of Multirate Synchronous AADL. In: Proc. FM'14. LNCS, vol. 8442. Springer (2014)
18. Bae, K., Ölveczky, P.C., Meseguer, J., Al-Nayeem, A.: The SynchAADL2Maude tool. In: Proc. FASE'12. LNCS, vol. 7212. Springer, Berlin, Heidelberg (2012)

19. Bae, K., Rocha, C.: Guarded terms for rewriting modulo SMT. In: Formal Aspects of Component Software (FACS'17). LNCS, vol. 10487, pp. 78–97. Springer (2017)
20. Bae, K., Rocha, C.: Symbolic state space reduction with guarded terms for rewriting modulo SMT. Science of Computer Programming **178**, 20–42 (2019)
21. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. LNCS, vol. 6806, pp. 171–177. Springer (2011)
22. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: Version 2.0. In: SMT. vol. 13, p. 14 (2010)
23. Baudart, G., Benveniste, A., Bourke, T.: Loosely time-triggered architectures: Improvements and comparisons. ACM Transactions on Embedded Computing Systems (TECS) **15**(4), 1–26 (2016)
24. Bozzano, M., Bruintjes, H., Cimatti, A., Katoen, J.P., Noll, T., Tonetta, S.: Formal methods for aerospace systems: Achievements and challenges. In: Cyber-Physical System Design From an Architecture Analysis Viewpoint: Communications of NII Shonan Meetings. pp. 133–159. Springer (2017)
25. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Computer Aided Verification (CAV'13). LNCS, vol. 8044, pp. 258–263. Springer (2013)
26. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: HyComp: An SMT-based model checker for hybrid systems. In: Proc. TACAS. LNCS, vol. 9035. Springer, Berlin, Heidelberg (2015)
27. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude – A High-Performance Logical Framework, Lecture Notes in Computer Science, vol. 4350. Springer, Berlin, Heidelberg (2007)
28. Dutertre, B.: Yices 2.2. In: Proc. CAV. LNCS, vol. 8559, pp. 737–744. Springer, Berlin, Heidelberg (July 2014)
29. Eidson, J.: `https://www.nist.gov/document/tutorial-basicpdf/` (2005), accessed July 16, 2023
30. Eidson, J.C., Fischer, M., White, J.: IEEE-1588™ standard for a precision clock synchronization protocol for networked measurement and control systems. In: Proceedings of the 34th Annual Precise Time and Time Interval Systems and Applications Meeting. pp. 243–254 (2002)
31. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language. Addison-Wesley, USA (2012)
32. 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, USA (2007)
33. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Proc. CAV. LNCS, vol. 6806. Springer, Berlin, Heidelberg (2011)
34. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: Proc. CADE. Lecture Notes in Computer Science, vol. 7898, pp. 208–214. Springer, Berlin, Heidelberg (2013)
35. Goguen, J., Meseguer, J.: Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theoretical Computer Science **105**, 217–273 (1992)
36. Kim, C., Sun, M., Mohan, S., Yun, H., Sha, L., Abdelzaher, T.F.: A framework for the safe interoperability of medical devices in the presence of network failures. In: ICCPS. pp. 149–158 (2010)

37. 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, pp. 200–205. Springer, Berlin, Heidelberg (2015)
38. Kopetz, H., Bauer, G.: The time-triggered architecture. Proceedings of the IEEE **91**(1), 112–126 (2003)
39. Lee, J., Bae, K., Ölveczky, P.C.: An extension of HybridSynchAADL and its application to collaborating autonomous UAVs. In: International Symposium on Leveraging Applications of Formal Methods. LNCS, vol. 13703, pp. 47–64. Springer (2022)
40. Lee, J., Bae, K., Ölveczky, P.C., Kim, S., Kang, M.: Modeling and formal analysis of virtually synchronous cyber-physical systems in AADL. International Journal on Software Tools for Technology Transfer **24**(6), 911–948 (2022)
41. Lee, J., Kim, S., Bae, K., Ölveczky, P.C.: HybridSynchAADL: Modeling and formal analysis of virtually synchronous CPSs in AADL. In: Proc. CAV'21. LNCS, vol. 12759, pp. 491–504. Springer, Berlin, Heidelberg (2021)
42. Lee, J., Yu, G., Bae, K.: Efficient SMT-based model checking for signal temporal logic. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 343–354. IEEE (2021)
43. Leen, G., Heffernan, D., Dunne, A.: Digital networks in the automotive vehicle. Computing & Control Engineering Journal **10**(6), 257–266 (1999)
44. Lepri, D., Ábrahám, E., Ölveczky, P.C.: Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories. Science of Computer Programming **99**, 128–192 (2015)
45. Liu, S., Meseguer, J., Ölveczky, P.C., Zhang, M., Basin, D.: Bridging the semantic gap between qualitative and quantitative models of distributed systems. Proceedings of the ACM on Programming Languages **6**(OOPSLA2), 315–344 (2022)
46. Liu, S., Ölveczky, P.C., Meseguer, J.: Modeling and analyzing mobile ad hoc networks in Real-Time Maude. J. Log. Algebraic Methods Program. **85**(1), 34–66 (2016). https://doi.org/10.1016/j.jlamp.2015.05.002
47. Mavridou, A., Stachtiari, E., Bliudze, S., Ivanov, A., Katsaros, P., Sifakis, J.: Architecture-based design: A satellite on-board software case study. In: Formal Aspects of Component Software (FACS'16). LNCS, vol. 10231, pp. 260–279. Springer (2017)
48. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73–155 (1992)
49. Meseguer, J.: Taming distributed system complexity through formal patterns. In: Arbab, F., Ölveczky, P.C. (eds.) Formal Aspects of Component Software - 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7253, pp. 1–2. Springer (2011). https://doi.org/10.1007/978-3-642-35743-5_1, https://doi.org/10.1007/978-3-642-35743-5_1
50. Meseguer, J.: Twenty years of rewriting logic. Journal of Logic and Algebraic Programming **81**(7), 721–781 (2012)
51. Meseguer, J.: Taming distributed system complexity through formal patterns. Science of Computer Programming **83**, 3–34 (2014)
52. 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)
53. Meseguer, J., Roşu, G.: The rewriting logic semantics project: A progress report. Information and Computation **231**, 38–69 (2013)

54. Miller, S., Cofer, D., Sha, L., Meseguer, J., Al-Nayeem, A.: Implementing logical synchrony in integrated modular avionics. In: Proc. IEEE/AIAA 28th Digital Avionics Systems Conference. IEEE, USA (2009)
55. Ölveczky, P.C., Boronat, A., Meseguer, J.: Formal semantics and analysis of behavioral AADL models in Real-Time Maude. In: Proc. FMOODS/FORTE'10. LNCS, vol. 6117, pp. 47–62. Springer (2010)
56. Ölveczky, P.C., Boronat, A., Meseguer, J., Pek, E.: Formal semantics and analysis of behavioral AADL models in Real-Time Maude. `https://olveczky.se/RealTimeMaude/AADL/webTechRep.pdf` (2010)
57. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation **20**(1–2), 161–196 (2007)
58. Ölveczky, P.C., Meseguer, J.: The Real-Time Maude tool. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08). LNCS, vol. 4963, pp. 332–336. Springer (2008)
59. Ölveczky, P.C.: Real-Time Maude and its applications. In: Proc. WRLA'14. LNCS, vol. 8663. Springer (2014)
60. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theor. Comput. Sci. **410**(2-3), 254–280 (2009). https://doi.org/10.1016/j.tcs.2008.09.022
61. Platzer, A.: Differential dynamic logic for hybrid systems. Journal of Automated Reasoning **41**(2), 143–189 (2008)
62. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014), `http://ptolemy.org/books/Systems`
63. 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)
64. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: QMaude: Quantitative specification and verification in rewriting logic. In: Formal Methods (FM'23). LNCS, vol. 14000, pp. 240–259. Springer (2023)
65. Rushby, J.: Systematic formal verification for fault-tolerant time-triggered algorithms. IEEE Transactions on Software Engineering **25**(5), 651–660 (1999)
66. Sha, L., Al-Nayeem, A., Sun, M., Meseguer, J., Ölveczky, P.C.: PALS: Physically asynchronous logically synchronous systems. Tech. rep., Department of Computer Science, University of Illinois at Urbana-Champaign (2009), `http://hdl.handle.net/2142/11897`
67. Steiner, W., Bauer, G., Hall, B., Paulitsch, M., Varadarajan, S.: TTEthernet dataflow concept. In: 2009 Eighth IEEE International Symposium on Network Computing and Applications. pp. 319–322. IEEE (2009)
68. Steiner, W., Rushby, J.: TTA and PALS: Formally verified design patterns for distributed cyber-physical systems. In: 2011 IEEE/AIAA 30th Digital Avionics Systems Conference. pp. 7B5–1. IEEE (2011)
69. Stevens, B.L., Lewis, F.L.: Aircraft control and simulation. John Wiley & Sons (2003)
70. 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)
71. Whitters, G., Nigam, V., Talcott, C.L.: Incremental rewriting modulo SMT. In: Automated Deduction (CADE'23). LNCS, vol. 14132, pp. 560–576. Springer (2023)
72. Yu, G., Bae, K.: Maude-SE: a tight integration of Maude and SMT solvers. Proc. International Workshop on Rewriting Logic and its Applications (2020)

73. Yu, G., Lee, J., Bae, K.: STLmc: Robust STL model checking of hybrid systems using SMT. In: International Conference on Computer Aided Verification (CAV'22). LNCS, vol. 13371, pp. 524–537. Springer (2022)