Designing and Verifying Distributed Cyber-Physical Systems using Multirate PALS: An Airplane Turning Control System Case Study

Kyungmin Bae[a,*], Joshua Krisiloff[a,b], José Meseguer[a], Peter Csaba Ölveczky[c]

[a] *Department of Computer Science, University of Illinois at Urbana-Champaign*
[b] *Department of Aerospace Engineering, University of Illinois at Urbana-Champaign*
[c] *Department of Informatics, University of Oslo*

Abstract

Distributed cyber-physical systems (DCPS), such as aeronautics and ground transportation systems, are very hard to design and verify, because of asynchronous communication, network delays, and clock skews. Their model checking verification typically becomes unfeasible due to the huge state space explosion caused by the system's concurrency. The Multirate PALS ("physically asynchronous, logically synchronous") methodology has been proposed to reduce the design and verification of a DCPS to the much simpler task of designing and verifying its underlying synchronous version, where components may operate with different periods. This paper presents a methodology for formally modeling and verifying multirate DCPSs using Multirate PALS. In particular, this methodology explains how to deal with the system's physical environment in Multirate PALS. We illustrate our methodology with a multirate DCPS consisting of an airplane maneuvered by a pilot, who turns the airplane to a specified angle through a distributed control system. Our formal analysis using Real-Time Maude revealed that the original design did not achieve a smooth turning maneuver, and led to a redesign of the system. We then use model checking and Multirate PALS to prove that the redesigned system satisfies the desired correctness properties, whereas model checking the corresponding *asynchronous* model is unfeasible. This shows that Multirate PALS is not only effective for formal DCPS verification, but can also be used effectively in the DCPS *design* process.

Keywords: Multirate PALS, Cyber-physical systems, Rewriting logic, Real-Time Maude, Model checking, Hybrid systems

1. Introduction

Distributed cyber-physical systems (DCPS) are pervasive in areas such as avionics, aeronautics, ground transportation systems, robotics, manufacturing, medical devices, and so on. DCPS design and verification is quite challenging, because to the usual complexity of a non-distributed CPS one has to add the additional complexities of asynchronous communication, network delays, and clock skews, which can easily lead a DCPS into inconsistent states. In particular, any hopes of applying model checking verification techniques in a direct manner to a DCPS look rather dim, due to the typically huge state space explosion caused by the system's concurrency. For these reasons, we and other colleagues at UIUC and Rockwell-Collins have been developing the *Physically Asynchronous but Logically Synchronous* (PALS) methodology [25, 27], which can drastically reduce the system complexity of a DCPS so as to make it amenable to model checking verification. The PALS methodology applies to the frequently occurring case of a DCPS whose implementation must be asynchronous due to physical constraints and for fault-tolerance reasons, but whose *logical* design requires that the system components should act together in a virtually synchronous way. For example, distributed control systems are typically of this nature.

*Corresponding author

The key idea of PALS is that if the underlying infrastructure provides performance bounds Γ on the computation times, networks delays, and imprecisions of the local clocks, then the task of designing and verifying a virtually synchronous DCPS can be reduced to the much simpler task of designing and verifying the idealized synchronous system.[1] This is achieved by a *model transformation*

$$\mathcal{E} \;\mapsto\; \mathcal{A}(\mathcal{E}, \Gamma)$$

that maps a synchronous design $\mathcal{E}$ and performance bounds Γ to a distributed implementation $\mathcal{A}(\mathcal{E}, \Gamma)$, which is *correct-by construction* and that, as shown in [25], is *bisimilar* to $\mathcal{E}$. This bisimilarity is the essential feature allowing the drastic reduction in system complexity and making model checking verification feasible: since bisimilar systems satisfy the same temporal logic properties, we can verify that the asynchronous system $\mathcal{A}(\mathcal{E}, \Gamma)$ satisfies a temporal logic property φ—which would typically be impossible to model check directly on $\mathcal{A}(\mathcal{E}, \Gamma)$—by verifying the same property φ on the vastly simpler synchronous system $\mathcal{E}$.

The original PALS methodology presented in [25, 27] assumes a *single logical period*, during which all components of the DCPS must communicate with each other and transition to their next states. However, a DCPS such as a distributed control system may have components that must operate with different periods for physical reasons, even though those periods may all divide an overall longer period. That is, many such systems, although still having to be virtually synchronous for their correct behavior, are in fact *multirate* systems, with some components running at a faster rate than others. Three of us have therefore defined in [8, 9] a mathematical model of a multirate synchronous system $\mathcal{E}$, and have formally defined a model transformation $\mathcal{E} \;\mapsto\; \mathcal{MA}(\mathcal{E}, T, \Gamma)$, with underlying performance bounds Γ and global period T, that generalizes the original single-rate PALS transformation to multirate systems. As before, we have proved that $\mathcal{MA}(\mathcal{E}, T, \Gamma)$ is a correct-by-construction implementation of $\mathcal{E}$, and that $\mathcal{E}$ and $\mathcal{MA}(\mathcal{E}, T, \Gamma)$ are *bisimilar*, making it possible to verify temporal logic properties of $\mathcal{MA}(\mathcal{E}, T, \Gamma)$ on the much simpler system $\mathcal{E}$.

Many distributed cyber-physical systems, such as automotive and avionics systems, control *physical entities* and are in fact *distributed hybrid systems*: a collection of digital components that communicate asynchronously with each other and that interact with their environment, whose continuous dynamics is typically governed by differential equations. Until now, both PALS and Multirate PALS have focused on distributed *real-time* systems without continuous behaviors; all case studies have been such real-time systems. This paper investigates the suitability of Multirate PALS to design and verify nontrivial virtually synchronous distributed *hybrid* systems. We present a general method for modeling distributed hybrid systems using Multirate PALS. We then define a general framework for formally specifying such Multirate PALS designs $\mathfrak{E}$ in Real-Time Maude [29]. Given a specification of a multirate ensemble $\mathfrak{E}$ in Real-Time Maude, our framework defines an executable semantics for the synchronous composition of $\mathfrak{E}$ that can be used to simulate and model check this multirate synchronous composition in Real-Time Maude.

We use our methodology and the Real-Time Maude framework to formally specify in detail a multirate distributed hybrid system consisting of an airplane maneuvered by a pilot, who wants to turn the airplane to a desired direction through a distributed control system with effectors located in the airplane's wings and rudder. Our formal analysis revealed that the original design did not achieve a smooth turn. This led to a redesign of the system with new control laws which, as verified in Real-Time Maude by model checking, satisfies the desired correctness properties. This shows that the Multirate PALS methodology is not only effective for formal DCPS verification, but, when used together with a tool like Real-Time Maude, can also be used effectively in the DCPS *design* process, even before properties are verified.

To illustrate the performance benefits obtained by using Multirate PALS for system verification, we have also defined a Real-Time Maude model of the corresponding distributed *asynchronous* airplane control system under highly simplified assumptions (perfect and perfectly synchronized local clocks, zero execution time, and no network delays). Model checking even this simplified asynchronous model shows that the state space quickly explodes due to the interleavings caused by the asynchrony, rendering model checking verification of the asynchronous system unfeasible for nontrivial properties.

[1] For a simple avionics case study in [25], the number of system states for their simplest possible distributed version with perfect clocks and no network delays was 3,047,832, but the PALS pattern reduced the number of states to a mere 185.

To the best of our knowledge, this is the first time that the Multirate PALS methodology has been applied to the model checking verification of a DCPS, and the first time that PALS has been applied to a distributed hybrid system. In this sense, this paper complements our companion papers [8, 9], where the mathematical foundations of Multirate PALS are developed in detail, but where only a brief summary of some of the results presented here was given.

This paper is a substantial extension of our workshop paper [7]. In addition to a discussion of related work, the main new contributions include:

- More details about the case study and its formal analysis.
- A general methodology explaining how distributed hybrid systems can be modeled in Multirate PALS.
- Our previous Real-Time Maude modeling and execution framework is generalized to (both deterministic and) *nondeterministic* components.
- An asynchronous model of the airplane control system, which not only shows how the distributed hybrid systems behaves, but, in particular, allows us to compare the model checking performance of both the synchronous design and the asynchronous real-time "realization" of the system.

The rest of this paper is organized as follows. Section 2 gives some background on Multirate PALS and Real-Time Maude. Section 3 explains how multirate distributed hybrid systems can be modeled as multirate ensembles in Multirate PALS. Section 4 presents a modeling and execution framework for multirate ensembles in Real-Time Maude. Section 5 describes a simple model of an airplane turning control system whose continuous dynamics is governed by differential equations, and Section 6 formally specifies the airplane turning control system using the ensemble framework. Section 7 explains how the synchronous version of the airplane system was analyzed, redesigned, and formally verified in Real-Time Maude. Section 8 defines and model checks the simplified asynchronous version of the airplane control system and compares the model checking performance with that of the synchronous design. Finally, Section 9 discusses related work and Section 10 presents some concluding remarks.

2. Preliminaries on Multirate PALS and Real-Time Maude

2.1. Multirate PALS

In many distributed real-time systems, such as automotive and avionics systems, the system design is essentially a *synchronous design* that must be realized in a distributed setting. Both design and verification of such *virtually synchronous* distributed real-time systems is very hard because of asynchronous communication, network delays, clock skews, and because the state space explosion caused by the system's concurrency can make it unfeasible to apply model checking to verify required properties. The (single-rate) PALS ("physically asynchronous, logically synchronous") formal design pattern [25, 27] reduces the design and verification of a virtually synchronous distributed real-time system to the much simpler task of designing and verifying its synchronous version, provided that the network infrastructure can guarantee bounds on the messaging delays and the skews of the local clocks.

However, it is a fact of life that different components operate with different frequencies. We have therefore recently developed Multirate PALS [8, 9], which extends PALS to hierarchical *multirate* systems in which controllers with the same rate communicate with each other and with a number of faster subsystems or components. As is common for hierarchical control systems [1], we assume that the period of the higher-level controllers is a multiple of the period of each fast component. Given a multirate synchronous design $\mathfrak{E}$, a global period T, and *performance bounds* $\Gamma = (\varepsilon, \alpha_{\min}, \alpha_{\max}, \mu_{\min}, \mu_{\max})$ of the underlying network infrastructure, where:

- $\mathfrak{E}$ is a synchronous ensemble of state machines with local clocks and different periods connected by a wiring diagram (formally defined below),
- T is the period of the *slowest* component (the "top-level" component) in $\mathfrak{E}$;

- ε is the maximum *clock skew* of the local clocks for each component in $\mathfrak{E}$; that is, the difference between a local clock and the "precise" global system time is always less than ε;
- $0 \leq \alpha_{\min} \leq \alpha_{\max}$ are the minimum and maximum durations for any component to process inputs, make a transition, and produce outputs; and
- $0 \leq \mu_{\min} \leq \mu_{\max}$ are the minimum and maximum message transmission delays for communication between any two components.

Multirate PALS then defines a model transformation $\mathfrak{E} \mapsto \mathcal{MA}(\mathfrak{E}, T, \Gamma)$ from the multirate synchronous design $\mathfrak{E}$ to the corresponding multirate distributed real-time system $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$, where $\mathfrak{E}$ and $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ satisfy the same temporal logic properties.

2.1.1. Multirate Synchronous Models

In Multirate PALS, the synchronous design is formalized as the synchronous composition of a collection of *typed machines*, an *environment*, and a *wiring diagram* that connects the machines [25].

Definition 1. A *typed machine* is a tuple $M = (\mathcal{D}_i, S, \mathcal{D}_o, \delta_M)$ with $\mathcal{D}_i = D_{i_1} \times \cdots \times D_{i_n}$ an *input set*, S a set of *states*, $\mathcal{D}_o = D_{o_1} \times \cdots \times D_{o_m}$ an *output set*, and $\delta_M \subseteq (D_i \times S) \times (S \times D_o)$ a total *transition relation*.

That is, a typed machine M has n input ports and m output ports; an input to port k is an element of D_{i_k} and an output from port j is one of D_{o_j}.

To compose a collection of typed machines with different rates into a synchronous system in which all components perform one transition in lock-step in each iteration of the system, we "slow down" the faster machines so that all machines run at the slow rate in the synchronous composition. The *k-step deceleration* $M^{\times k}$ of a typed machine M *performs k internal transitions* in one synchronous step.

Definition 2. Given $M = (\mathcal{D}_i, S, \mathcal{D}_o, \delta_M)$ with $\mathcal{D}_i = D_{i_1} \times \cdots \times D_{i_n}$ and $\mathcal{D}_o = D_{o_1} \times \cdots \times D_{o_m}$, for $k \in \mathbb{N}_+$, the k-step *deceleration* is a machine $M^{\times k} = ((D_{i_1})^k \times \cdots \times (D_{i_n})^k,\ S,\ (D_{o_1})^k \times \cdots \times (D_{o_m})^k,\ \delta_{M^{\times k}})$ where:

$$\Big(\big(((d_{i_{1_1}}, \ldots, d_{i_{1_k}}), \ldots, (d_{i_{n_1}}, \ldots, d_{i_{n_k}})), s\big), \big(s', ((d_{o_{1_1}}, \ldots, d_{o_{1_k}}), \ldots, (d_{o_{m_1}}, \ldots, d_{o_{m_k}}))\big)\Big) \in \delta_{M^{\times k}}$$

iff there exist states $s_1, \ldots, s_{k-1} \in S$ such that:

$$\begin{aligned}
(((d_{i_{1_1}}, \ldots, d_{i_{n_1}}), s),\ (s_1, (d_{o_{1_1}}, \ldots, d_{o_{m_1}}))) &\in \delta_M \\
(((d_{i_{1_2}}, \ldots, d_{i_{n_2}}), s_1),\ (s_2, (d_{o_{1_2}}, \ldots, d_{o_{m_2}}))) &\in \delta_M \\
(((d_{i_{1_3}}, \ldots, d_{i_{n_3}}), s_2),\ (s_3, (d_{o_{1_3}}, \ldots, d_{o_{m_3}}))) &\in \delta_M \\
\vdots \qquad\qquad\qquad\qquad & \qquad\qquad \vdots \\
(((d_{i_{1_k}}, \ldots, d_{i_{n_k}}), s_{k-1}),\ (s', (d_{o_{1_k}}, \ldots, d_{o_{m_k}}))) &\in \delta_M.
\end{aligned}$$

Since the fast machine M consumes an input and produces an output in each of these internal steps, the decelerated machine $M^{\times k}$ consumes (resp. produces) k-tuples of inputs (resp. outputs) in each synchronous step. A k-tuple output from $M^{\times k}$ must be *adapted* so that it can be read by the slow machine. That is, the k-tuple must be transformed to a single value (e.g., the average of the k values, the last value, or any other function of the k values). Likewise, the single output from a slow component must be transformed to a k-tuple of inputs to the fast machine, e.g., a k-tuple $(d, \perp, \ldots, \perp)$ for some "don't care" value $\perp$. Such a transformation is formalized as an *input adaptor* $\alpha = \{\alpha_j : D'_j \to D_{i_j}\}_{j \in \{1,\ldots,n\}}$ for a machine M, each of which determines a desired input value $\alpha_k(d_k) \in D_{i_k}$ from an output $d_k \in D'_k$ of another typed machine, where $\alpha(d_1, \ldots, d_n) = (\alpha_1(d_1), \ldots, \alpha_n(d_n))$. Then, the *adaptor closure* M_α is the typed machine in which each input port j is enclosed by each input adaptor function α_j as shown in Figure 1.

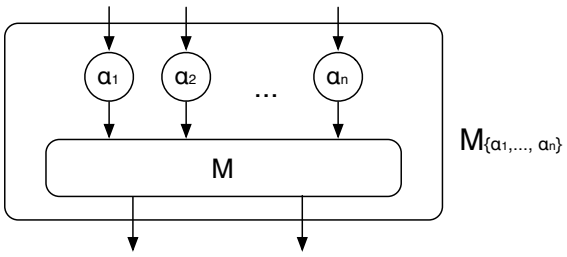

Figure 1: The adaptor closure $M_{\{\alpha_1,\ldots,\alpha_n\}}$ of a typed machine M.

Definition 3. The *adaptor closure* of $M = (\mathcal{D}_i, S, \mathcal{D}_o, \delta_M)$ with an adaptor $\alpha = \{\alpha_k : D'_k \to D_{i_k}\}_{k\in\{1,\ldots,n\}}$ is the typed machine $M_\alpha = ((D'_1 \times \cdots \times D'_n), S, \mathcal{D}_o, \delta_{M_\alpha})$ such that:

$$((\mathbf{d}_i, s), (s', \mathbf{d}_o)) \in \delta_{M_\alpha} \iff ((\alpha(\mathbf{d}_i), s), (s', \mathbf{d}_o)) \in \delta_M.$$

Typed machines can be "wired together" into *multirate machine ensembles* as shown in Figure 2, where "local" fast environments are integrated with their corresponding fast machines.

Definition 4. A multirate machine ensemble is a tuple

$$\mathfrak{E} = (J_S,\ J_F,\ e,\ \{M_j\}_{j\in J_s\cup J_F},\ E,\ src,\ rate,\ adap)$$

where:

- J_S is a finite set of ("controller component" or "slow machine") indices and J_F is a finite set of ("controlled component" or "fast machine") indices such that $J_S \cap J_F = \emptyset$;
- $e \notin J_S \cup J_F$ is the *environment index*;
- $\{M_j\}_{j\in J_S\cup J_F}$ is a family of typed machines;
- $E = (\mathcal{D}^e_i, \mathcal{D}^e_o)$ is the *environment* with $\mathcal{D}^e_i$ the environment's *input set* and $\mathcal{D}^e_o$ its *output set*;
- *src* is a surjective function that assigns to each input port (j, n) (input port n of machine j) the corresponding output port (or "source" for that input);
- $rate : J_F \to \mathbb{N}_+$ is a function assigning to each fast machine a value denoting how many times faster the machine runs compared to the controller machines; and
- *adap* is a function that assigns an input adaptor to each typed machine in $\{M_j\}_{j\in J_S\cup J_F}$.

We assume that: (i) an output domain is a subset of the corresponding input domain; i.e., $src(l, q) = (k, p)$ implies $D^k_{o_p} \subseteq D^l_{i_q}$, where $D^l_{i_q}$ denotes the domain of the qth input port of machine M_l (resp. qth input port of the environment if $l = e$), and same with output ports, and (ii) there are no connections between fast machines, or between the environment and fast machines; i.e., if $src(l, q) = (k, p)$, then $l \in J_S$ or $k \in J_S$.

The transitions of all machines in a multirate ensemble $\mathfrak{E}$ are performed synchronously with input adaptors, where each fast machine with rate k performs k "internal transitions" in one synchronous step, and whenever a machine has a feedback wire to itself and/or to another machine, then the output becomes an input at the *next* instant. The *synchronous composition* of a multirate ensemble $\mathfrak{E}$ is therefore equivalent to a *single machine* $M_\mathfrak{E}$, where each state of $M_\mathfrak{E}$ consists of the states of its subcomponents and the contents in the feedback outputs [9, 25]. For example, the synchronous composition of the multirate ensemble in Figure 2 is the machine depicted in the outer box.

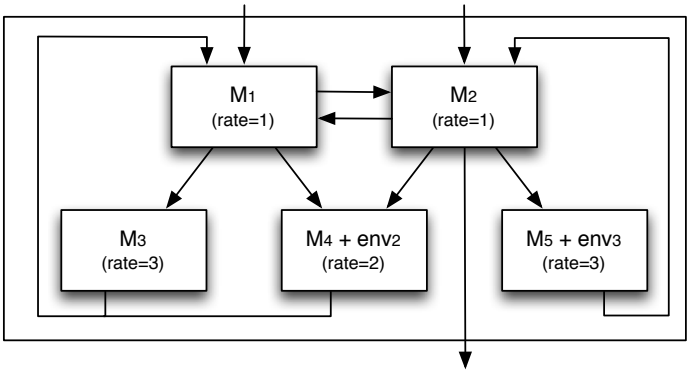

Figure 2: A multirate machine ensemble. M_1 and M_2 are controller machines, and env_2 and env_3 are local environments with faster rates, hidden from high-level controllers.

Definition 5. Given a multirate ensemble $\mathfrak{E} = (J_S, J_F, e, \{M_j\}_{j \in J_S \cup J_F}, E, src, rate, adap)$, its *synchronous composition* is the typed machine $M_{\mathfrak{E}} = (D_i^{\mathfrak{E}}, S^{\mathfrak{E}}, D_o^{\mathfrak{E}}, \delta_{\mathfrak{E}})$ where:

- $D_i^{\mathfrak{E}} = D_o^e$ and $D_o^{\mathfrak{E}} = D_i^e$.
- $S^{\mathfrak{E}} = (\Pi_{j \in J} S_j) \times (\Pi_{j \in J} D_{OF}^j)$, where, intuitively, D_{OF}^j stores the "feedback outputs" of transformed machine $(M_j^{\times rate(j)})_{adap(j)}$ for $j \in J_F$ or $(M_j)_{adap(j)}$ for $j \in J_S$, used as input in the next iteration.
- $\delta_{\mathfrak{E}} \subseteq (D_i^{\mathfrak{E}} \times S^{\mathfrak{E}}) \times (S^{\mathfrak{E}} \times D_o^{\mathfrak{E}})$ "combines" the transitions of the transformed machines $(M_f^{\times rate(f)})_{adap(f)}$ for $f \in J_F$ and $(M_s)_{adap(s)}$ for $s \in J_S$ into a synchronous step as further explained in [25].

Notice that a multirate ensemble $\mathfrak{E}$ is essentially a "single-rate" ensemble of the typed machines transformed by k-step decelerations and adaptor closures:

$$\{(M_f^{\times rate(f)})_{adap(f)} \mid f \in J_F\} \cup \{(M_s)_{adap(s)} \mid s \in J_S\}.$$

The full formal definition for the synchronous composition of a single-rate ensemble is given in the paper [25]. Since $M_{\mathfrak{E}}$ is itself a typed machine which can appear as a component in another multirate ensemble, we can easily define hierarchical multirate systems.

An environment e of a multirate ensemble $\mathfrak{E}$ can typically be restricted by *environment constraints*, e.g., given by a boolean predicate $c_e : D_o^e \to Bool$ so that $c_e(d_1^e, \ldots, d_{o_{m_e}}^e)$ is *true* if and only if the environment can generate output $(d_1^e, \ldots, d_{o_{m_e}}^e)$. We can associate a transition system defining the behaviors of a machine ensemble that operates in an environment as follows.

Definition 6. The transition system for a multirate ensemble $\mathfrak{E}$ is a pair $ts(\mathfrak{E}) = (S^{\mathfrak{E}} \times \mathcal{D}_i^{\mathfrak{E}}, \longrightarrow_{\mathfrak{E}})$, where $(\vec{s}_1, \vec{i}_1) \longrightarrow_{\mathfrak{E}} (\vec{s}_2, \vec{i}_2)$ iff an ensemble in state $\vec{s}_1$ and with input $\vec{i}_2$ from the environment has a transition to state $\vec{s}_2$ and the environment can generate output $\vec{i}_2$ in the next step:

$$(\vec{s}_1, \vec{i}_1) \longrightarrow_{\mathfrak{E}} (\vec{s}_2, \vec{i}_2) \iff \exists \vec{o}\, ((\vec{i}_1, \vec{s}_1), (\vec{s}_2, \vec{o})) \in \delta_{\mathfrak{E}} \wedge c_e(\vec{i}_2).$$

2.1.2. Multirate Asynchronous Models

The asynchronous model $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ adds "wrappers" around each machine in $\mathfrak{E}$ as shown in Figure 3, where machines and wrappers in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ perform *at their own rate*. These wrappers have an input buffer, an output buffer, and a local clock that deviates by less than ε from a global perfect clock. Since Multirate PALS is based on a number of patterns, we use multiple wrappers to define $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$, where each wrapper realizes the corresponding pattern in the distributed setting. The outermost wrapper is the standard "PALS wrapper" that encloses an input adaptor wrapper, which encloses either a (slow) typed machine or a k-slowed machine wrapper, which in turn encloses an ordinary (fast) typed machine.

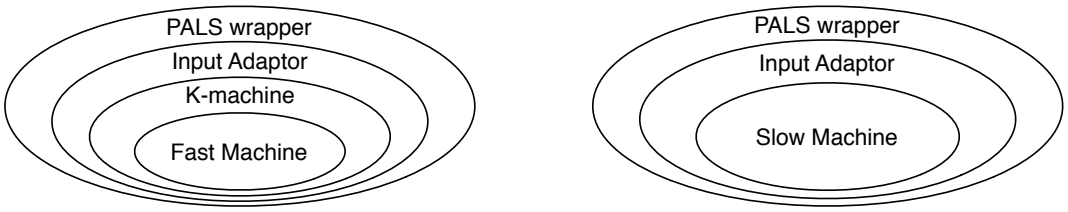

Figure 3: The wrapper hierarchies for fast components (left) and slow components (right).

However, a fast machine in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ may *not* be able to finish all of its internal transitions *before* the messages must be sent to the slow machine to ensure that they arrive before the beginning of the next round, due to execution times and network delays [9]. If there is not enough time for a fast machine to execute all of its k internal transitions in a global round T, but can only send *at most* $k' < k$ outputs, then the slow machine should only consider these k' values. The number of transitions a fast machine can perform in a global round T before its output must be sent is given by

$$k' = 1 + \lfloor (T \ \texttt{monus} \ \ (2\varepsilon + \mu_{\max} + \alpha_{\max_f})) \cdot (k/T) \rfloor,$$

where ε is the maximum clock skew, $\mu_{\max}$ is the maximal message delay, $\alpha_{\max_f}$ is the maximal execution time for the fast machine, and $x \ \texttt{monus} \ y = \max(x - y, 0)$. Therefore, if the source of the i-th input port of a slow machine M_j is a fast machine whose k' is less than its rate k, then the input adaptor function $adap(j)_i$ must be $(k'+1)$-*oblivious* [9], that is, for all values v_l and v'_l,

$$adap(j)_i(v_1, \ldots, v_{k'}, v_{k'+1}, \ldots, v_k) = adap(j)_i(v_1, \ldots, v_{k'}, v'_{k'+1}, \ldots, v'_k).$$

The behavior of those asynchronous components can be summarized as follows. The formal semantics of the asynchronous model $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ is specified in [9] by a rewrite theory in Real-Time Maude [29].

- All PALS wrappers have the same *slow period* T, and communicate with the other components by sending messages to, and receiving messages from, the other nodes in the network.
- When a new *slow* round begins according to its local clock, each input adaptor wrapper reads the inputs from the PALS wrapper and applies an appropriate input adaptor function.
- For a fast machine, the k-machine wrapper extracts each value from the k-tuple input and sends them to the enclosed typed machine at the beginning of each fast period T/k.
- The innermost typed machine runs at its given rate; at the beginning of its periods, it reads its input, performs a transition, and sends the outputs when the execution of the transition is finished.
- For a fast machine, the k-machine wrapper stores the outputs, and sends out the resulting k-*tuples* of outputs to its outer layer when it has received all k sets of outputs or its timer expires. If its timer expires, such a k-tuple output is padded with "don't care" values $\perp$ after the first k' values.
- Finally, the PALS wrapper sends out into the network the outputs from the layer below, after its output backoff timer has expired, which ensures that messages are not sent out too early.

The global states of $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ have the form $\{C; t\}$, where C is the configuration consisting of distributed components and messages traveling between the different components, and t is the global time. A global transition $\{C_1; t_1\} \longrightarrow \{C_2; t_2\}$ between global states is then specified by means of rewrite rules.

2.1.3. Relating the Synchronous and Asynchronous Models

For a multirate ensemble $\mathfrak{E}$, the asynchronous system $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ has in general more states than the synchronous composition $M_{\mathfrak{E}}$, since there are states in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ where different objects are in different stages in a round. However, we are interested in the *stable* states in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$, where all the input buffers of the PALS wrappers are full and all other input and output buffers are empty [25]. There exists an obvious function $sync : Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)) \to S^{\mathcal{E}} \times D_i^{\mathcal{E}}$ assigning to each stable state the corresponding synchronous state of $M_{\mathfrak{E}}$ [9, 25]. Further, we can define the "big-step" transition system $(Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)), \longrightarrow_{st})$, where $Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma))$ is a set of such stable states [9].

Assuming $(k' + 1)$-obliviousness of the input adaptors for inputs from fast machines, we can relate two stable states by $t_1 \sim_{obi} t_2$ iff: (i) their corresponding states are the same, and (ii) their corresponding input buffers *cannot* be distinguished by any input adaptors in $\mathfrak{E}$ [9].

Theorem 1. *[9] Given a multirate ensemble $\mathfrak{E}$, the relation $(\sim_{obi}; sync)$ is a total bisimulation between the stable transition system $(Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)), \longrightarrow_{st})$ and the synchronous transition system $ts(\mathfrak{E})$.*

Furthermore, if a state labeling function $\mathcal{L} : S^{\mathfrak{E}} \times \mathcal{D}_i^{\mathfrak{E}} \to \mathcal{P}(AP)$ is defined for $\mathfrak{E}$ and it cannot distinguish between $\sim_{obi}$-equivalent states, then two Kripke structures for $Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma))$ and $M_{\mathfrak{E}}$ are bisimilar to each other [9], so that they satisfy exactly the same CTL^* formulas.

2.2. Real-Time Maude

Real-Time Maude [29] is a language and tool that extends Maude [13] to support the formal specification and analysis of real-time systems. The specification formalism is based on *real-time rewrite theories* [28]—an extension of *rewriting logic* [11, 23]—and emphasizes *ease* and *generality* of specification. Real-Time Maude specifications are executable under reasonable assumptions, and the tool provides a variety of formal analysis methods, including simulation, reachability analysis, and LTL and timed CTL model checking.

2.2.1. Rewriting Logic Specification in Maude

A *membership equational logic* (MEL) [24] *signature* is a triple $\Sigma = (K, \sigma, S)$ with K a set of *kinds*, $\sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ a many-kinded signature, and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. The kind of a sort s is denoted by $[s]$. A Σ-*algebra* A consists of a set A_k for each kind k, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \to A_k$ for each operator $f \in \Sigma_{k_1 \cdots k_n, k}$, and a subset inclusion $A_s \subseteq A_k$ for each sort $s \in S_k$. The set $T_{\Sigma,k}$ denotes the set of ground Σ-terms with kind k, and $T_\Sigma(X)_k$ denotes the set of Σ-terms with kind k over the set X of kinded variables.

A MEL *theory* is a pair (Σ, E) with Σ a MEL-signature and E a finite set of MEL sentences, either conditional equations of the form $(\forall X)\ t = t'$ **if** $\bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j$ or conditional memberships of the form $(\forall X)\ t : s$ **if** $\bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j$, where $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$ for some kind $k \in \Sigma$, the latter stating that t is a term of sort s, provided the condition holds. Order-sorted notation $s_1 < s_2$ abbreviates the conditional membership $(\forall x : [s_1])\ x : s_2$ **if** $x : s_1$. Similarly, an operator declaration $f : s_1 \times \cdots \times s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom $(\forall\, x_1 : k_1, \ldots, x_n : k_n)\ f(x_1, \ldots, x_n) : s$ **if** $\bigwedge_{1\le i\le n} x_i : s_i$.

A Maude module specifies a *rewrite theory* [11, 23] of the form $(\Sigma, E \cup A, R)$, where: (i) $(\Sigma, E \cup A)$ is a membership equational logic theory specifying the system's state space as an algebraic data type with A a set of equational axioms (such as a combination of associativity, commutativity, and identity), to perform equational deduction *modulo* the axioms A, and (ii) R is a set of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form:

$$l : q \longrightarrow r \ \textbf{if} \ \bigwedge_i p_i = q_i \ \wedge \ \bigwedge_j w_j : s_j \ \wedge \ \bigwedge_m t_m \longrightarrow t'_m,$$

where l is a *label*, and q, r are Σ-terms of the same kind. Intuitively, such a rule specifies a *one-step transition* from a substitution instance of q to the corresponding substitution instance of r, *provided* the condition holds; that is, the substitution instance of each condition in the rule follows from $\mathcal{R}$.

We briefly summarize the syntax of Maude (see [13] for more details). Operators are introduced with the `op` keyword: `op` f `:` $s_1 \ldots s_n$ `->` s. Operators can have user-definable syntax, with underbars '`_`' marking each of the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. An operator can also be declared to be a *constructor* (`ctor`) that defines the carrier of a sort. The `frozen` attribute declares which argument positions are *frozen*; arguments in frozen positions cannot be rewritten by rewrite rules [13].

There are three kinds of logical statements in the Maude language, *equations*, *memberships* (declaring that a term has a certain sort), and *rewrite rules*, introduced with the following syntax:

- equations: `eq` $u = v$ or `ceq` $u = v$ `if` *condition*;
- memberships: `mb` $u : s$ or `cmb` $u : s$ `if` *condition*;
- rewrite rules: `rl [`l`]:` u `=>` v or `crl [`l`]:` u `=>` v `if` *condition*.

An equation $f(t_1, \ldots, t_n) = t$ with the `owise` (for "otherwise") attribute can be applied to a term $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied.[2] The mathematical variables in such statements are either explicitly declared with the keywords `var` and `vars`, or can be introduced on the fly in a statement without being declared previously, in which case they must be have the form $var : sort$. Finally, a comment is preceded by '`***`' or '`---`' and lasts till the end of the line.

2.2.2. Object-Oriented Specification in Real-Time Maude

A Real-Time Maude *timed module* specifies a *real-time rewrite theory* $\mathcal{R} = (\Sigma, E \cup A, R)$ [28], where:

- $(\Sigma, E \cup A)$ contains an equational subtheory $(\Sigma_{TIME}, E_{TIME}) \subseteq (\Sigma, E \cup A)$, satisfying the *TIME* axioms that specifies sort `Time` as the time domain (which can be discrete or dense). The supersort `TimeInf` extends the sort `Time` with an "infinity" value `INF`.
- The rules in R are decomposed into: (i) "ordinary" rewrite rules specifying the system's *instantaneous* (i.e., zero-time) local transitions, and (ii) *tick (rewrite) rules* that model the elapse of time in a system, having the form $[l] : \texttt{\{}t\texttt{\}} \xrightarrow{u} \texttt{\{}t'\texttt{\}}$ **if** *condition*, where t and t' are of sort `System`, u is of sort `Time` denoting the *duration* of the rewrite, and `{_}` is a built-in constructor of sort `GlobalSystem`. In Real-Time Maude, tick rules, together with their durations, are specified with the syntax:

$$\texttt{crl [}l\texttt{]: \{}t\texttt{\} => \{}t'\texttt{\} in time }u\texttt{ if }\textit{condition}.$$

The initial state must be reducible to a term `{`t_0`}`, for t_0 a ground term of sort `System`, using the equations in the specification. The form of the tick rules then ensures uniform time elapse in all parts of a system.

Real-Time Maude is particularly suitable to formally model distributed real-time systems in an object-oriented style. Each term t in a global system state `{`t`}` is a term of sort `Configuration` (which is a subsort of `System`), and has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative and having the `none` multiset as its identity element, so that rewriting is *multiset rewriting* supported directly in Maude.

In Real-Time Maude timed modules one can declare *classes*, *subclasses*, and *messages*. A *class* declaration `class` C `|` $att_1 : s_1, \ldots, att_n : s_n$ declares a class C with attributes $att_1, \ldots, att_n$ of sorts $s_1, \ldots, s_n$. An *object* of class C is represented as a term of sort `Object` and has the form:

$$\texttt{< O : C |}\ att_1 : val_1, \ldots, att_n : val_n\ \texttt{>},$$

[2] A specification with `owise` equations can be transformed to an equivalent system without such equations [13].

where O is the object's identifier, and $val_1, \ldots, val_n$ are its attribute values of sort $s_1, \ldots, s_n$. A *subclass*, introduced with the keyword `subclass`, inherits all the attributes, equations, and rules of its superclasses. A *message* is a term of sort `Msg`, where the declaration `msg` m : s_1 ... s_n `-> Msg` defines the syntax of the message (m) and the sorts (s_1 ... s_n) of its parameters.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rewrite rule

```
rl [l]: m(O,w)  < O : C | a1 : x,     a2 : O', a3 : z >
     =>         < O : C | a1 : x + w, a2 : O', a3 : z >  m'(O',x) .
```

defines a parametrized family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`, the attribute `a1` of the object `O` is changed to `x + w`, and a new message `m'(O',x)` is generated. The message `m(O,w)` is *removed* from the state by the rule, since it does *not* occur in the right-hand side of the rule. Likewise, the message `m'(O',x)` is *generated* by the rule, since it *only* occurs in the right-hand side of the rule. By convention, attributes whose values do not change and do not affect the next state of other attributes or messages, such as `a3` in our example, need not be mentioned in a rule. Similarly, attributes whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, such as `a2`, can be omitted from right-hand sides of rules.

2.2.3. Formal Analysis in Real-Time Maude

We summarize below the Real-Time Maude analysis commands used in our case study. Real-Time Maude's *timed fair rewrite* command

`(tfrew` t `in time <=` τ `.)`

simulates *one behavior* of the system within time τ from the initial state t. The *timed search* command

`(tsearch [`n`]` t `=>*` *pattern* `such that` *condition* `in time <=` τ `.)`

analyzes *all possible behaviors* by using a breadth-first strategy to search for n states that are reachable from the initial state t within time τ, match the search *pattern*, and satisfy the search *condition*. The *untimed* search command `utsearch` is similar, but without the time bound.

The Real-Time Maude's *linear temporal logic (LTL) model checker* checks whether each behavior from an initial state, possibly up to a time bound, satisfies an LTL formula. *State propositions* are declared as operators of sort `Prop`, and their semantics are given by equations of the forms

`eq {`*statePattern*`} |=` *prop* `=` b and `ceq {`*statePattern*`} |=` *prop* `=` b `if` *condition*

for b a term of sort `Bool`, which defines the state proposition *prop* to hold in all states `{`t`}` where `{`t`} |=` *prop* evaluates to `true`. An LTL formula [12] is constructed by state propositions and temporal logic operators such as `True`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), `U` ("until"), and `O` ("next"). Then, the *untimed* (resp., *timed*) model checking commands

`(mc` t `|=u` φ `.)` and `(mc` t `|=t` φ `in time <=` τ `.)`

check whether the formula φ holds in all behaviors from the initial state t (resp., *within time* τ).

3. Modeling Distributed Hybrid Systems Using Multirate PALS

As mentioned in the introduction, many cyber-physical systems, including the airplane control system in Section 5, interact with *physical* entities exhibiting continuous dynamics, and are therefore *distributed hybrid systems*. In Multirate PALS, any "local" environment of the "faster" components is assumed to have been incorporated into the corresponding typed machine itself [8, 9], but our previous work did not explain *how* such local physical environments can be combined with their controllers. This section proposes a methodology for integrating physical environments into their controlling typed machines within the Multirate PALS framework. Our methodology can be summarized as follows:

1. A physical environment E_M defines the continuous behaviors of the environment, and also defines how the environment reacts to actuator commands from the controllers.

2. The discrete component M is represented by a *nondeterministic typed machine* whose behavior is defined for *any possible* environment behavior.

3. The *environment restriction* of M by E_M defines an ordinary typed machine $M \restriction E_M$ whose transitions are constrained by the behavior of E_M.

This model is suitable under the reasonable assumption that the controller component M is tightly integrated with its physical environment, and "contains" the sensors and actuators. Even if the sensor and actuators are *remote* instead of *included* in the controller M and therefore there exists also a network delay between sensor/actuator and controller, the system can still be modeled by having another typed machine, containing the sensors and actuators, that interacts with the controller like any other machine. Our methodology can easily be combined with the Real-Time Maude modeling framework for Multirate PALS in Section 4.

3.1. The Methodology

The state of a physical environment at a certain point can be represented as a tuple of the values $(v_1, v_2, \ldots, v_l) \in \mathbb{R}^l$ of the physical parameters $x_1, \ldots, x_l$ of interest. For example, the Newtonian dynamics of a single object involves the four parameters $(x, y, z, t) \in \mathbb{R}^4$ for a position (x, y, z) of the object at time t. A physical environment is called *l-dimensional* iff its state space is a subset of $\mathbb{R}^l$. As usual in physical dynamics, the behavior of a physical environment can typically be expressed by differential equations and continuous functions involving its parameters $x_1, \ldots, x_l$, which specify *trajectories* of the parameters $x_1, \ldots, x_l$.

Definition 7. A *trajectory* of duration $T \in \mathbb{R}$ is a function $\tau : [0, T] \to \mathbb{R}$. The set $\mathcal{T}_T$ denotes the set of all trajectories of duration $T \in \mathbb{R}$. For an l-tuple of trajectories $\vec{\tau} = (\tau_1, \ldots, \tau_l) \in \mathcal{T}_T^l$, let

$$\vec{\tau}(x) = (\tau_1(x), \ldots, \tau_l(x)).$$

That is, a trajectory defines the continuous behavior of a physical parameter in a period of duration T. Such a trajectory is normally a continuous function, but can also be more general. We refer to [22] for more details about general trajectories for hybrid systems.

A digital controller M collects the state $(v_1, \ldots, v_l)$ of its physical environment E_M using its sensors, and affects the physical environment E_M through its actuators. Since we consider "periodic" digital components with period T, a physical environment E_M can be modeled as a *periodic dynamic system* that specifies any possible trajectories of its physical state during its period T.

Definition 8. An l-dimensional *periodic dynamic system* is a tuple $E_M = (C, P, T, \Lambda)$ where:

- C is a set of *control commands*, representing "actuator outputs" from the controller;
- $P \subseteq \mathbb{R}^l$ is a set of all possible values of the "physical parameters" $x_1, \ldots, x_l$ of E_M;
- $T \in \mathbb{R}_{>0}$ is the *period* of E_M;
- $\Lambda \subseteq (C \times P) \times \mathcal{T}_T^l$ is a total *physical transition relation* that defines possible trajectories $\vec{\tau} \in \mathcal{T}_T^l$ of duration T for each control command $c \in C$ beginning at each physical state $\vec{v} \in P$, satisfying:

$$\big((c, \vec{v}), \vec{\tau}\big) \in \Lambda \implies \vec{\tau}(0) = \vec{v} \wedge (\forall t \in [0, T])\ \vec{\tau}(t) \in P$$

That is, if the current physical state of E_M is $\vec{v}$ at the beginning of a period and its controller M gives an actuator output c to E_M, then the physical state of E_M follows the continuous trajectory $\vec{\tau}$ during period T, as illustrated in Figure 4.

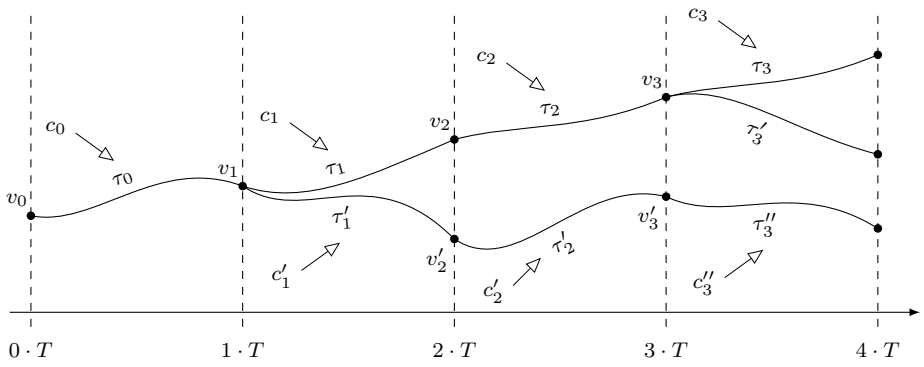

Figure 4: A periodic dynamic system in which its physical state follows the continuous trajectory τ_i from v_i to v_{i+1} according to the control command c_i during each period; e.g., $((c_0, v_0), \tau_0) \in \Lambda$, $((c_1, v_1), \tau_1) \in \Lambda$, and $((c_1', v_1), \tau_1') \in \Lambda$.

A machine M with environment E_M is a formalized as an ordinary typed machine with period T. In particular, its local state also contains the state of the continuous environment. However, since such a machine performs only *one* transition in each period, it can only change its local state once. Intuitive, such a controller M with environment E_M operates as follows:

1. At the "start of a transition," the state of the typed machine M contains the *current* (estimated) physical state of the environment.

2. A transition of a typed machine M does whatever it is supposed to do, including changing the controller state and sending outputs. The transition updates the part of the local state that represents the state of the environment to the state of the environment *at the end of the current period.*

3. An actuator command is "sent" at the beginning of the current round, and is assumed to take effect immediately. This is reflected in the fact that the new state of the controller includes the estimated state of the environment at the end of the round, when the actuator command has taken effect in the beginning of the "current" round.

It is natural to assume that it takes some time for the controller to compute the next actuator command. Therefore, the new actuator command, taking effect at the beginning of the current period, does *not* depend on the current input, but is determined by *only the current state*, which was computed during the previous period. Instead, the current input can be used to determine the *next* state of M when its transition is taken. In other words, the new actuator command is based on the current state of the physical environment and the inputs received at the beginning of the *previous* round.

More precisely, a generic discrete controller M for the periodic dynamic system $E_M = (C, P, T, \Lambda)$ is specified as an ordinary *nondeterministic* typed machine $M = (\mathcal{D}_i, S, \mathcal{D}_o, \delta_M)$, where the physical parameters of E_M are also included in M's state, and the control commands to E_M (i.e., actuator outputs) depend (only) on the current state of M. That is, M is defined as a highly nondeterministic machine which takes *all possible* environment behaviors into account. Then, the environment restriction $M \restriction E_M$ (formally defined below) restricts the behaviors of M to those that "fit" with the environment E_M. Such a relationship between a controller M and a physical environment E_M is captured by two projection functions:

Definition 9. Given a machine $M = (\mathcal{D}_i, S, \mathcal{D}_o, \delta_M)$ and a periodic dynamic system $E_M = (C, P, T, \Lambda)$:

- A *command function* $\pi_C : S \to C$ determines the control command to the physical environment E_M.
- A *environment state function* $\pi_P : S \to P$ represents the *observed* environment state.

As mentioned, the same discrete controller can be placed in many different physical environments that share the same physical parameters but show different behaviors. Hence, the behavior of the controller machine $M = (\mathcal{D}_i, S, \mathcal{D}_o, \delta_M)$ is *nondeterministically* defined for any possible behaviors of the physical parameters, and can be summarized more formally than above as follows:

- At the start of each period, M is in state $s \in S$, which determines the control command $\pi_C(s)$ to its physical environment and the physical state $\pi_P(s)$ at the beginning of the current round.[3]
- The actuator output $\pi_C(s)$ takes effect on its physical environment from the beginning of the current round and lasts until the end of the round.
- Based on the current state $s \in S$ and the input $\vec{d_i} \in \mathcal{D}_i$, the transition relation δ_M *nondeterministically* decides the output $\vec{d_o} \in \mathcal{D}_o$ and the next state $s' \in S$.
- The next state s' also determines the physical state $\pi_P(s')$ and the control command $\pi_C(s')$ for the beginning of the *next* period. That is, a transition of M also updates the values of the physical parameters to their (expected) values at the end of the current period/beginning of the next period.
- $\pi_P(s')$ represents the values of the physical parameters at the beginning of the next round, while the physical values (continuously) change from the current physical state $\pi_P(s)$, according the control command $\pi_C(s)$ until the beginning of the next round.
- However, since the behavior of those physical parameters has not been specified *yet*, the transition relation δ_M assumes that the values of the physical parameters in the current state s can be changed to any real numbers in the next state s', and assigns such *nondeterministically chosen* values to the physical parameters in the next state s'.

When the behavior of those physical parameters in machine M is specified by a periodic dynamic system $E_M = (C, P, T, \Lambda)$, the transition relation $\delta_{M \restriction E_M}$ of the restricted machine $M \restriction E_M$ by E_M is then a *subset* of δ_M that also follows the *physical constraints* Λ given by E_M. That is, $M \restriction E_M$ captures each *observable* moment of the E_M's continuous behavior for the corresponding typed machine M.

Definition 10. Given a typed machine $M = (\mathcal{D}_i, S, \mathcal{D}_o, \delta_M)$, a physical environment $E_M = (C, P, T, \Lambda)$, a control command function $\pi_C : S \to C$, and an environment state function $\pi_P : S \to P$, the *environment restriction* of M is the typed machine $M \restriction E_M = (\mathcal{D}_i, S, \mathcal{D}_o, \delta_{M \restriction E_M})$ such that $\big((\vec{d_i}, s), (s', \vec{d_o})\big) \in \delta_{M \restriction E_M}$ iff $\big((\vec{d_i}, s), (s', \vec{d_o})\big) \in \delta_M$ holds and there exists a trajectory $\vec{\tau} \in \mathcal{T}_T^l$ with

$$\big((\pi_C(s), \pi_P(s)), \vec{\tau}\big) \in \Lambda \ \wedge \ \vec{\tau}(0) = \pi_P(s) \ \wedge \ \vec{\tau}(T) = \pi_P(s').$$

That is, for the control command $\pi_C(s)$, the next physical state $\pi_P(s')$ must be *reachable* from the current physical state $\pi_P(s)$ by a *valid trajectory* $\vec{\tau}$ of the physical environment E_M.

Notice that M can be considered as a typed machine *parameterized* by different behaviors of physical parameters, and its *environment restriction* $M \restriction E_M$ defines the actual behavior of M when it is placed in the particular physical environment specified by the periodic dynamic system E_M.

Our definition does not require a specific formalism to model continuous trajectories for E_M, and our methodology does not specify *how* to "compute" the values of the continuous parameters at the beginning of each period. In the case study in Section 5, we use differential equations and continuous functions to specify continuous trajectories, and compute continuous values by solving differential equations and some numerical approximations in a way similar to the Euler method [5].

[3] It is the job of the machine M how to collect such physical values at the beginning of each period, since the sensors and actuators are already incorporated in the logic of M. The controller machine M may have immediate access to the parameters of its environment, or require some time to process those values; but as usual in PALS, the execution time of the machine, including environment input/output processing, must be less than the maximum execution time $\alpha_{\max}$ [25].

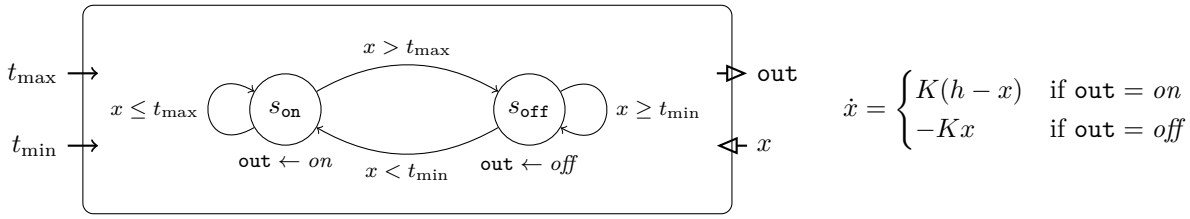

Figure 5: A digital thermostat controller modeled as a typed machine $M = (\mathbb{R}^2, \{s_{\texttt{on}}, s_{\texttt{off}}\} \times \mathbb{R}, \{*\}, \delta_M)$ with two input ports $\mathbb{R}^2$ (for the desired maximum temperature $t_{\max}$ and the desired minimum temperature $t_{\min}$) and no output port $\{*\}$. Its physical environment is represented by one environment parameter (the current temperature x), and one control command (the switch of the heater `out`).

Since we model a distributed cyber-physical system as a multirate machine ensemble $\mathfrak{E}$ in which each machine can be considered as the environment restriction of a controller typed machine M by its physical environment E_M, the Multirate PALS transformation can in principle generate a correct-by-construction distributed asynchronous model $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ from the synchronous model $\mathfrak{E}$ [8, 25]. However, there are two "hybrid systems" problems that Multirate PALS cannot solve at the moment:

- The physical environments of different distributed components can be *physically correlated* to each other. For example, if we consider two adjacent rooms, the temperature of one room will immediately affect the temperature of the other room unless they are perfectly insulated.
- In the distributed asynchronous model $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ different components read their sensor values at slightly different times, due to the clock skews. The "continuous behavior" of $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ can therefore be slightly different from the synchronous model $\mathfrak{E}$.

Providing general solutions for those challenges is beyond the scope of this paper. We therefore assume that any "physical correspondences" between different physical environments are faithfully captured by the ensemble connections in $\mathfrak{E}$. Also, we assume the the system is *stable* in the sense that small differences in the "sensor/effector timing" caused by the clock skews do not affect the correctness of the system.

3.2. Example: Digital Thermostat

Figure 5 shows a digital thermostat controller, operating at a certain frequency to control the temperature of a room, specified by the typed machine

$$M = (\mathbb{R}^2,\ \{s_{\texttt{on}}, s_{\texttt{off}}\} \times \mathbb{R},\ \{*\},\ \delta_M)$$

where each state $(s, x) \in \{s_{\texttt{on}}, s_{\texttt{off}}\} \times \mathbb{R}$ contains the "current" temperature x of the room (that is, the temperature at the beginning or at the end of the current period), and $\{*\}$ is a singleton set to denote that there is no output port. At each step, M receives two input values

$$(t_{\max}, t_{\min}) \in \mathbb{R}^2$$

from other typed machines, where $t_{\max}$ is the desired maximum temperature and $t_{\min}$ is the desired minimum temperature. Based on these inputs and its current state (s, x), M makes a transition to the next state (s', x'), where $x' \in \mathbb{R}$ can be any value since the behavior of the physical parameter x has not been specified *yet*. During its period until the next step, M gives a command `out` to turn the heater *on*/*off*, according to its state s. The behavior can be specified by the following transition relation δ_M:

$$\big((t_{\max}, t_{\min}), (s_{\texttt{on}}, x)\big), \big((\textbf{if } x \le t_{\max} \textbf{ then } s_{\texttt{on}} \textbf{ else } s_{\texttt{off}} \textbf{ fi},\ x'), *\big)\big) \in \delta_M \iff x' \in \mathbb{R}$$
$$\big((t_{\max}, t_{\min}), (s_{\texttt{off}}, x)\big), \big((\textbf{if } x < t_{\min} \textbf{ then } s_{\texttt{on}} \textbf{ else } s_{\texttt{off}} \textbf{ fi},\ x'), *\big)\big) \in \delta_M \iff x' \in \mathbb{R}.$$

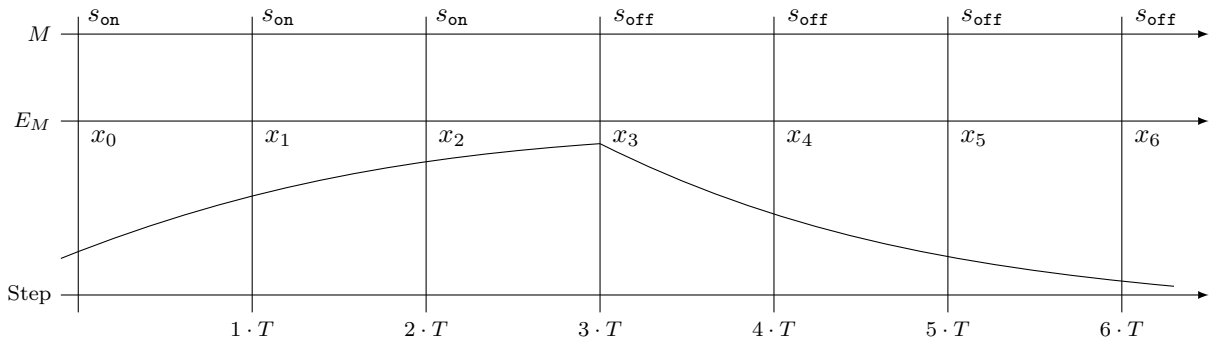

Figure 6: The behavior of the digital thermostat controller M with period T integrated with its physical environment E_M, where x_{i+1} is the next temperature after period T elapsed from x_i according to the given differential equation.

The physical environment E_M of the *nondeterministic* digital thermostat controller M is specified by the 1-dimensional periodic dynamic system with period T:

$$E_M = (\{on, off\},\ \mathbb{R},\ T,\ \Lambda)$$

with a set of control commands $\{on, off\}$, the physical state $\mathbb{R}$ for the current temperature x of the room, and the physical transition relation $\Lambda \subseteq (\{on, off\} \times \mathbb{R}) \times \mathcal{T}_T$ given by:

$$\big((\mathtt{out}, v), x)\big) \in \Lambda \iff \exists x : [0, T] \to \mathbb{R}.\ (x(0) = v)\ \wedge\ \dot{x} = \begin{cases} K(h - x) & \text{if } \mathtt{out} = on \\ -Kx & \text{if } \mathtt{out} = off, \end{cases}$$

where $K, h \in \mathbb{R}$ are constants depending on the size of the room and the power of the heater, respectively. That is, if the heater is on, the temperature x rises according to the differential equation $\dot{x} = K(h - x)$, and if the heater is off, the temperature x falls according to the differential equation $\dot{x} = -Kx$.

The physical environment E_M conforms to the controller machine M with the environment projection functions $\pi_C : (\{s_{\mathtt{on}}, s_{\mathtt{off}}\} \times \mathbb{R}) \to \{on, off\}$ and $\pi_P : (\{s_{\mathtt{on}}, s_{\mathtt{off}}\} \times \mathbb{R}) \to \mathbb{R}$, where

$$\pi_C(s_{\mathtt{on}}, x) = on \qquad \pi_C(s_{\mathtt{off}}, x) = off \qquad \pi_P(s, x) = x.$$

The environment restriction is the typed machine $M \restriction E_M = (\mathbb{R}^2, \{s_{\mathtt{on}}, s_{\mathtt{off}}\} \times \mathbb{R}, \{*\}, \delta_{M \restriction E_M})$, where:

$$\begin{aligned} &\big((t_{\max}, t_{\min}), (s, x)), ((s', x'), *)\big) \in \delta_{M \restriction E_M} \\ \iff &\big((t_{\max}, t_{\min}), (s, x)), ((s', x'), *)\big) \in \delta_M\ \wedge\ \big(\exists \tau \in \mathcal{T}_T.\ ((\pi_C(s), x), \tau) \in \Lambda\ \wedge\ \tau(0) = x\ \wedge\ \tau(T) = x'\big). \end{aligned}$$

The combined thermostat behavior of the environment restriction $M \restriction E_M$ is illustrated in Figure 6.

4. Multirate Synchronous Ensembles in Real-Time Maude

This section presents a generic framework for specifying and executing a *hierarchical* multirate machine ensemble in Real-Time Maude. Given a specification of typed machines, their periods, input adaptors, and a wiring diagram defining the connections between output ports and input ports, our framework produces an executable Real-Time Maude model of the synchronous composition of the multirate ensemble which can be used to simulate and formally verify this synchronous composition. If the system has "local" physical environments, then they should be integrated with the controller machines as explained in Section 3. Our framework extends the one in [7] by supporting also *nondeterministic* typed machines. The entire executable Real-Time Maude semantics is available at `http://formal.cs.illinois.edu/kbae/airplane`.

4.1. Representing Multirate Ensembles in Real-Time Maude

A multirate ensemble can naturally be specified in an object-oriented style, where the machines and the (sub)ensembles are modeled as objects. That is, the global state has the form `{`*Ensemble*`}` where *Ensemble* is an object representing the entire multirate machine ensemble.

4.1.1. Typed Machines

A typed machine is represented as an object instance of a subclass of the class `Component`, which has the attributes `period` and `ports`. The attribute `period` denotes the period of the typed machine, and `ports` contains the input and output "ports" of the typed machine, represented as a multiset of `Port` objects.

```
class Component | period : Time,
                  ports : Configuration .
```

We assume that any (input and output) data are terms of a supersort `Data`, which also contains the constant `bot`, denoting the "don't care" value $\perp$ used in the input adaptor functions.

```
sort Data .
op bot : -> Data [ctor] .
```

A port is represented as an object instance of the class `Port`, whose attribute `content` contains the data content as a list of values. The subclasses `InPort` and `OutPort` denote input and output ports, respectively.

```
class Port | content : List{Data} .
class InPort .
class OutPort .
subclass InPort OutPort < Port .
```

Component and port identifiers are terms of the subsorts `ComponentId` and `PortId`, respectively, of the sort `Oid` of *object identifiers*.

```
sorts ComponentId PortId .
subsorts ComponentId PortId < Oid .
```

To define the transition relation, the user must define the following built-in operator `delta`, for each single (atomic) typed machine, by means of equations (for deterministic transitions) or rewrite rules (for nondeterministic transitions).

```
op delta : [Object] -> [Object] .
```

The operator `delta` is declared to be a *partial function* using the *kind* `[Object]`. That is, a term containing the `delta` operator will only have a certain kind, but *not* a sort. This is used to ensure that a transition equation/rule in a *composition* is only applied when the transitions have been performed in all subcomponents (see Section 4.2).

Example 1. The environment restriction of the digital thermostat controller in Section 3.2 is specified in our Real-Time Maude framework by an object instance of the class `Thermostat`, where the `switch` attribute denotes whether the heater is on/off, the `period` attribute denotes the period of the controller, and the `temperature` attribute denotes the current temperature of the room.

```
class Thermostat | switch : Switch,
                   period : Float,
                   temperature : Float .
subclass Thermostat < Component .
sorts Switch .
ops on off : -> Switch [ctor] .
ops tmax tmin : -> PortId [ctor] .  --- port names
```

Since the transition relation of the thermostat controller is deterministic, we can define the transition operator `delta` by the following conditional *equation*, where the functions `rise` and `fall` computes the solutions of the given differential equations:

```
var C : ComponentId .    vars TMX TMN X PERIOD : Float .    vars CURR NEXT : Switch .
subsort Float < Data .                  --- floating point numbers are data
ops rise fall : Float Float -> Float .  --- functions for the temperature trajectories

ceq delta(
   < C : Thermostat | ports : < tmax : InPort | content : TMX >  < tmin : InPort | content : TMN >,
                      period : PERIOD,
                      temperature : X,
                      switch : CURR >)
 =
   < C : Thermostat | ports : < tmax : InPort | content : nil >  < tmin : InPort | content : nil >,
                      temperature : if (CURR == on) then rise(X, PERIOD) else fall(X, PERIOD) fi,
                      switch : NEXT >
 if NEXT := if (CURR == on) then (if X <= TMX then on else off fi)
                            else (if X <  TMN then on else off fi) fi .
```

That is, if the current switch is `on`, then the next switch is `on` if $x \leq t_{\max}$, and `off` otherwise. If the current switch is `off`, then the next switch is `on` if $x < t_{\min}$, and `off` otherwise. The new temperature in the next step after `PERIOD` elapsed is computed by the continuous functions depending on the current state.

4.1.2. Multirate Machine Ensembles

A multirate machine ensemble is modeled as an object instance of the class `Ensemble`, whose attribute `connections` denotes the wiring diagram and whose attribute `subcomponents` denotes the typed machines in the ensemble. We support the specification of *hierarchical* multirate ensembles by declaring `Ensemble` to be a subclass of `Component`, whose attribute `ports` denotes the ports to its external "environment."

```
class Ensemble | subcomponents : Configuration,
                 connections : Set{Connection} .
subclass Ensemble < Component .
```

For each component in the `subcomponents` attribute, its rate in the ensemble is implicitly given by the period of the component. The period of the entire ensemble (in its `period` attribute) must be equal to the period of the slowest components in the ensemble. To define the input adaptor for each input port of a single component, the user must declare the following built-in function `adaptor` by means of equations, where the sort `NeList{Data}` denotes a non-empty list of data:

```
op adaptor : ComponentId PortId NeList{Data} -> NeList{Data} .
```

A wiring diagram of an ensemble is modeled as a semicolon-separated set of connections. A connection is a term p_i `-->` p_o of sort `Connection`, where p_i and p_o are the source and target *port names*, respectively:

```
sorts Connection PortName .
subsort PortId < PortName .
op _._ : ComponentId PortId -> PortName [ctor] .
op _-->_ : PortName PortName -> Connection [ctor] .
```

A term C_1 . P_1 `-->` C_2 . P_2 represents a connection from an output port P_1 of a component C_1 to an input port P_2 of a component C_2. A connection between an environment port P and a port P_2 of a subcomponent C_2 is represented as a term P `-->` C_2`.`P_2 (for an environment input) or a term C_2`.`P_2 `->` P (for an environment output), where both sides are input ports or output ports. Section 6.5 illustrates how our airplane control system is represented as an ensemble object.

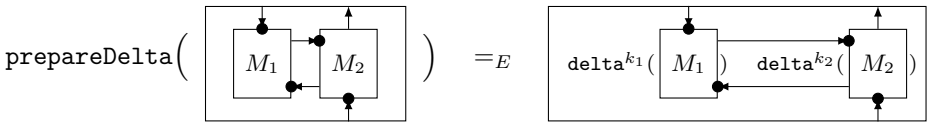

Figure 7: The `prepareDelta` function; `delta` is distributed to each subcomponent $i \in \{1, 2\}$ as many times as its rate k_i.

4.2. Defining Synchronous Compositions of Multirate Ensembles

Since an ensemble object of class `Ensemble` is also an instance of class `Component`, the transitions of the synchronous composition of the ensemble can be defined using the same operator `delta`. The following rewrite rule defines the transition relation of the synchronous composition specified in Definition 5:

```
var C : ComponentId .    var OBJ : Object .    var KOBJ : [Object] .

ops transferInputs transferResults : Object -> Object .
op prepareDelta : [Object] -> [Object] .

crl [sync]: delta(< C : Ensemble | >)
         =>
            transferResults(OBJ)
  if KOBJ := prepareDelta(transferInputs(< C : Ensemble | >))
  /\ KOBJ => OBJ .
```

using the three auxiliary functions `transferInputs`, `prepareDelta`, and `transferResults`. The meaning of this *conditional* rewrite rule is summarized as follows:

1. Each input port in the ensemble `C` receives a value from its source output port (`transferInputs`).
2. Appropriate input adaptors are applied to each input port, and then the operator `delta` of each subcomponent is applied multiple times according to its rate (`prepareDelta`) as illustrated in Figure 7; the resulting term is assigned to the variable `KOBJ` of *kind* `[Object]`.
3. Any term of sort `Object` resulting from rewriting the term given in `KOBJ` in zero or more steps can be *nondeterministically* assigned to the variable `OBJ`; since the operator `delta` does not yield terms of this sort, these resulting objects are "quiescent" objects where all the operations in a round have been performed. That is, the variable `OBJ` will only capture a term containing *no* `delta` in which all the transition relations for the subcomponents in the ensemble `C` are completely evaluated.
4. The new outputs in the subcomponents are transferred to the environment ports (`transferResults`).

When the system is specified by one top-level component for a multirate ensemble of (nondeterministic) typed machines, the dynamics of the system is specified by the following *conditional* tick rewrite rule that models an iteration of the synchronous composition of the ensemble:

```
var C : ComponentId .    vars T : Time .    vars OBJ1 OBJ2 : Object .

crl [step]: {< C : Ensemble | period : T >}
         =>
            {OBJ2}  in time T
  if OBJ1 := clearOutput(< C : Ensemble | >
  /\ delta(OBJ1) => OBJ2 .
```

For the top-level ensemble `C`, the outputs generated in the previous round is cleared by the `clearOutputs` function and the resulting term is assigned to the variable `OBJ1` of sort `Object`. Then, in a similar way to the `sync` rule above, any possible term of sort `Object` resulting from rewriting `delta(OBJ1)` in zero or more steps can be nondeterministically assigned to the variable `OBJ2` where `delta` is completely evaluated by rewrite rules. The function `clearOutputs` is declared as follows:

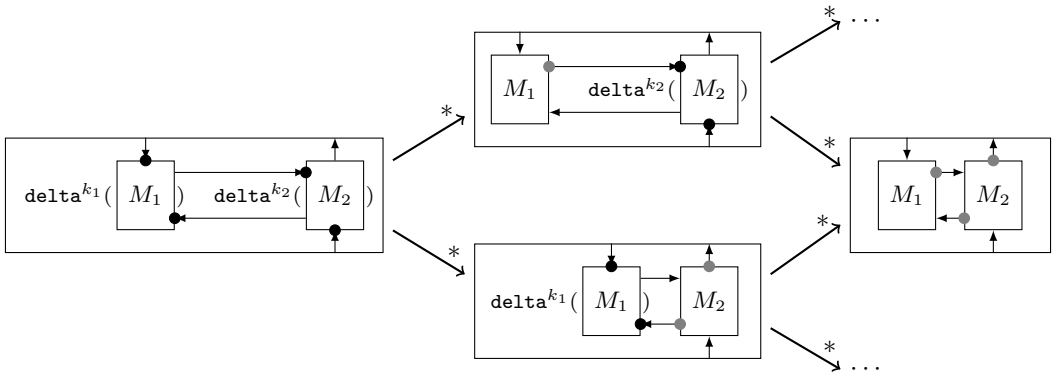

Figure 8: The two different execution orders that give the same result.

```
var C : ComponentId .  vars PORTS COMPS : Configuration .  var DL : List{Data} .

op clearOutput : Configuration -> Configuration .

eq clearOutput(< C : Component | ports : PORTS > COMPS)
 = < C : Component | ports : clearOutput(PORTS) >  clearOutput(COMPS) .
eq clearOutput(< P : OutPort | content : DL > PORTS)
 = < P : OutPort | content : nil >  clearOutput(PORTS) .
eq clearOutput(PORTS) = PORTS [owise] .
```

4.3. Executing Subcomponents using Partial Order Reductions

The transition of the synchronous ensemble in Section 4.2 is performed by *syntactically* distributing the operator `delta` to all the subcomponents, and then executing the transitions *concurrently* in the different typed machines. However, this straight-forward solution is computationally expensive, since the Maude engine computes all possible interleavings caused by applying the rewrite rules for `delta` in different orders. It is totally unnecessary to explore all these interleavings caused by executing the transitions concurrently in the different components during one period, since the PALS synchronous semantics ensures that the behaviors of the subcomponents are *independent* of each other during the execution of the transitions in a single round, as illustrated in Figure 8. Therefore, we can equally well "schedule" the execution of `delta` to completely avoid these interleavings, which in essence amounts to *partial order reduction* [12].

4.3.1. Scheduling Queue

For a multirate ensemble, we define a scheduling queue that gives a certain execution order to compute the transitions of the subcomponents. It is basically a list of component objects in which only the first component can execute its (nondeterministic) `delta` rewrite rules. We can specify such evaluation strategies for component lists by using the `frozen` attribute of the list cons operator `_::_` as follows:

```
sort ObjectQueue .
op nil : -> ObjectQueue [ctor] .
op _::_ : Object ObjectQueue -> ObjectQueue [ctor frozen(2)] .
```

The point is that we can put the objects in some list $object_1$ `::` ($object_2$ `::` (... `::` $object_n$)...), which means that the rewrite rules are first applied to $object_1$, and only when no rewrite rule can be applied to $object_1$ do we start applying rewrite rules to $object_2$, and so on. In particular, since the second argument of the operator `_::_` is declared `frozen`, for any list constructed by the operator, it allows rule rewriting only inside its first item. If some component is deterministic so that its `delta` is defined using only equations, then those equations can still be applied even though the component object is in the middle of the list.

When the first item in the scheduling queue is fully evaluated, which is a term of sort `Object`, it is removed from the queue so that the next item can be rewritten by rewrite rules, where `DetConfiguration` is a supersort of `Configuration` to denote a pair of *scheduling queues* and ordinary configurations:

```
var OBJ : Object .    vars COMPS : Configuration .    var QUEUE: [ObjectQueue] .
var N : Nat .         var P : PortId .                var NDL : NeList{Data} .

sort DetConfiguration .
subsort Configuration < DetConfiguration .
op _|_ : ObjectQueue Configuration -> DetConfiguration [ctor] .

ceq (OBJ :: QUEUE) | COMPS  =  QUEUE | (COMPS OBJ) .
eq  nil | COMPS  =  COMPS .
```

4.3.2. Definition of the `prepareDelta` Function

This section provides the formal definition of the `prepareDelta` function used in the `sync` rule that defines the transitions of the synchronous composition of an ensemble. The other functions `transferInput` and `transferResult` are explained in Appendix A. Rather than just distributing the `delta` operator over each subcomponent, the `prepareDelta` function constructs the scheduling queue of the subcomponents, where, for each subcomponent, its input adaptor function and the `delta` operator are applied:

```
op prepareDelta : [Object] -> [Object] .
op prepareDelta : Time [Configuration] [ObjectQueue] -> [DetConfiguration] .
op prepareDelta : Time Oid [Configuration] [ObjectQueue] -> [DetConfiguration] .

eq prepareDelta(< C : Ensemble | period : T, machines : COMPS >)
 = < C : Ensemble | machines : prepareDelta(T, COMPS, nil) > .
eq prepareDelta(T, COMPS, QUEUE)
 = if COMPS == none then (QUEUE | none) else prepareDelta(T, minCid(COMPS), COMPS, QUEUE) fi .
eq prepareDelta(T, C, < C : Component | period : T' > COMPS, QUEUE)
 = prepareDelta(T, COMPS, k-delta(quo(T,T'), applyAdaptors(< C : Component | >)) :: QUEUE) .
```

The function `quo(T,T')` returns the integer quotient of T and T', and the function `minCid(COMPS)` returns the smallest component identifier within `COMPS` in alphabetical order, assuming that there exist *no* duplicate component identifiers in an ensemble at the same level. For an ensemble with n subcomponents there exist in general $n!$ different scheduling queues that can be generated by matching in the `prepareDelta` equations; which one we choose does not matter.

The `k-delta` function applies the operator `delta` for the component (in the second argument) as many times as its rate (in the first argument). For example, `k-delta(3,OBJ)` is reduced to the term `delta(delta(delta(OBJ)))` by this function.

```
op k-delta : Nat [Object] -> [Object] .
eq k-delta(s N, < C : Component | >) = k-delta(N, delta(< C : Component | >)) .
eq k-delta(  0, < C : Component | >) = < C : Component | > .
```

The `applyAdaptors` function distributes the `adaptor` operator for input adaptors (which take a component identifier, a port identifier, and a non-empty list of data) over the input ports of the component:

```
op applyAdaptors : Object -> Object .
op applyAdaptors : ComponentId Configuration -> Configuration .

eq applyAdaptors(< C : Component | ports : PORTS >)
 = < C : Component | ports : adaptorAux(C,PORTS) > .
eq applyAdaptors(C, < P : InPort | content : NDL > PORTS)
 = < P : InPort | content : adaptor(C,P,NDL) >  applyAdaptors(C,PORTS) .
eq applyAdaptors(C, PORTS) = PORTS [owise] .
```

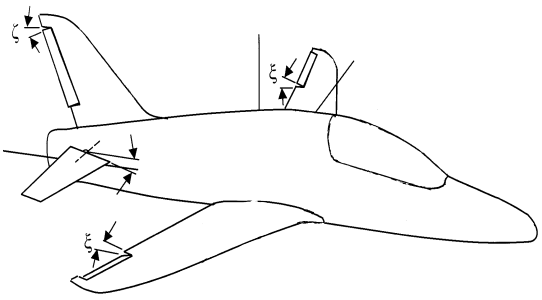

Figure 9: The ailerons and the rudder of an aircraft.

5. A Control System for Turning an Airplane

To smoothly turn an airplane, the airplane's *ailerons* and its *rudder* need to move in a synchronized way. (An aileron is a hinged surface attached to the end of the left or the right wing, and a rudder is a hinged surface attached to the vertical tail, as shown in Figure 9.) However, the controllers for the ailerons and the rudder typically operate at different frequencies. A *turning algorithm* should give instructions to those device controllers in order to achieve a smooth turn of the airplane.

This section explains the basic aeronautics theory behind a banked turn of an airplane, and then presents the architecture of a Multirate PALS model of a simple turning algorithm. This model is then formally defined in Section 6 and analyzed in Section 7.

5.1. The Aerodynamics Model

When an aircraft makes a turn, the aircraft rolls towards the desired direction of the turn, so that the lift force caused by the two wings acts as the centripetal force and the aircraft moves in a circular motion. The turning rate $\mathrm{d}\psi$ can be given by a function of the aircraft's roll angle ϕ:

$$\mathrm{d}\psi = (g/v) * \tan\phi \tag{1}$$

where ψ is the direction of the aircraft, g is the gravity constant, and v is the velocity of the aircraft [14]. The ailerons are used to control the rolling angle ϕ of the aircraft by generating different amounts of lift force in the left and the right wings. Figure 10 describes such a banked turn using the ailerons (the aircraft figures in this section are borrowed from [14]).

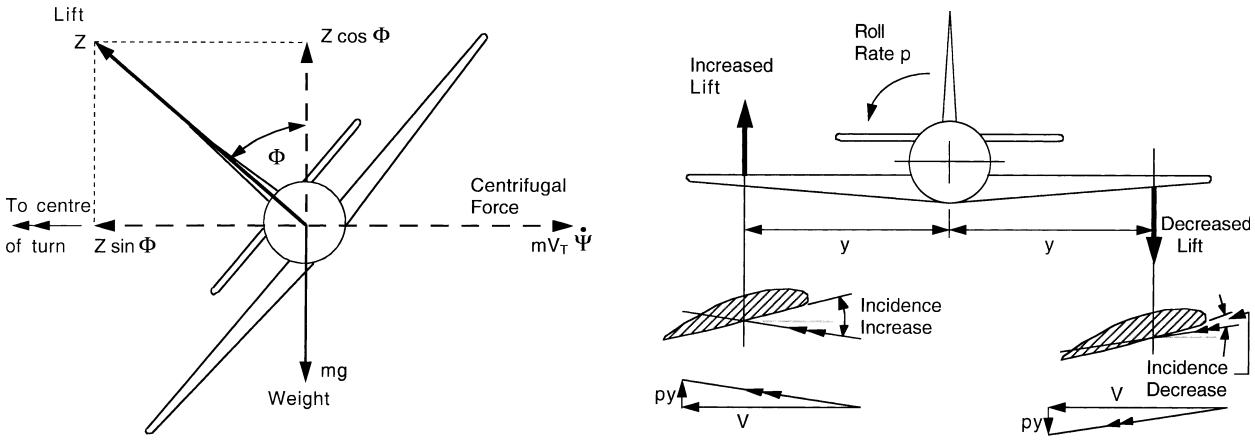

Figure 10: Forces acting in a turn of an aircraft with ϕ a roll angle and ρ a roll rate.

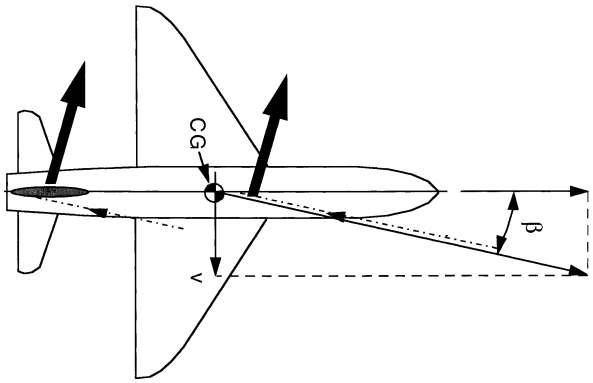

Figure 11: Adverse yaw.

However, the rolling of the aircraft causes a difference in drag on the left and the right wings, which produces a yawing moment in the opposite direction to the roll, called *adverse yaw*. This adverse yaw makes the aircraft sideslip in a wrong direction with the amount of the yaw angle β, as described in Figure 11. This undesirable side effect is countered by using the aircraft's rudder, which generates the side lift force on the vertical tail that opposes the adverse yaw. To turn an aircraft safely and effectively, the roll angle ϕ of the aircraft should be increased for the desired direction while the yaw angle β stays at 0.

The aerodynamics of turning an aircraft involves many factors, such as shape, pitch, speed, acceleration, temperature, air pressure, and so on. Therefore, we make the following assumptions to simplify the model:

- The wings have no dihedral angle (i.e., they are flat with respect to the horizontal axis of the aircraft).
- The altitude of the aircraft does not change, which can be separately controlled by using the aircraft's elevator (a flap attached to the horizontal tail of the aircraft).
- The aircraft maintains a constant speed by separately controlling the thrust power of the aircraft.
- There are no external influences such as wind, turbulence, or varying temperature.

If a differential equation $\mathrm{d}y^2 = f(x)$ is interpreted as follows:

$$\mathrm{d}y = \begin{cases} \sqrt{f(x)} & \text{if } f(x) \geq 0 \\ \sqrt{-f(x)} & \text{if } f(x) < 0, \end{cases}$$

then the roll angle ϕ and the yaw angle β can be modeled by the following differential equations [4]:

$$\mathrm{d}\phi^2 = (\mathit{Lift\ Right} - \mathit{Lift\ Left}) \,/\, (\mathit{Weight} * \mathit{Length\ of\ Wing}) \tag{2}$$

$$\mathrm{d}\beta^2 = \mathit{Drag\ Ratio} * (\mathit{Lift\ Right} - \mathit{Lift\ Left}) \,/\, (\mathit{Weight} * \mathit{Length\ of\ Wing}) + \mathit{Lift\ Vertical} \,/\, (\mathit{Weight} * \mathit{Length\ of\ Aircraft}). \tag{3}$$

The lift force from the left, the right, or the vertical tail wing is given by the following linear equation:

$$\mathit{Lift} = \mathit{Lift\ constant} * \mathit{Angle} \tag{4}$$

where, for *Lift Right* and *Lift Left*, *Angle* is the angle of the aileron, and for *Lift Vertical*, *Angle* is the angle of the rudder. The lift constant depends on the geometry of the corresponding wing, and the drag ratio is given by the size and the shape of the entire aircraft.

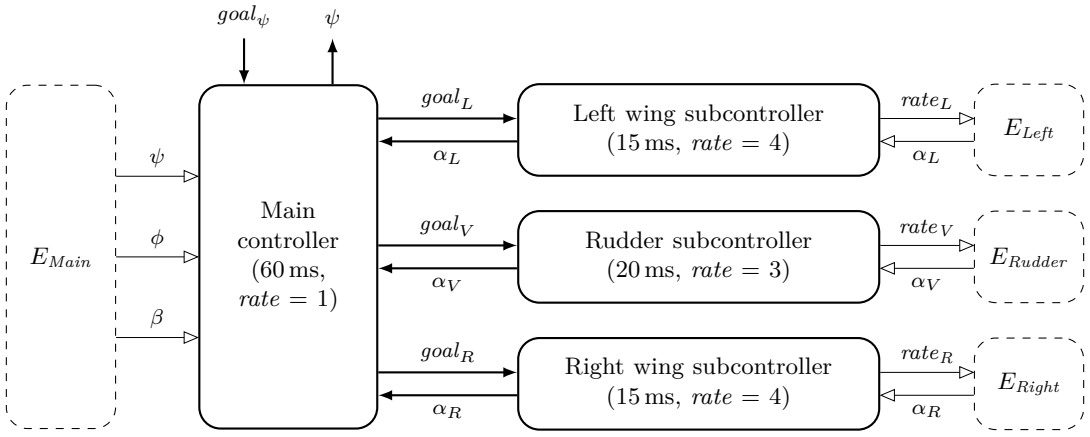

Figure 12: The distributed controllers for the airplane turning control system.

5.2. Architecture of the Distributed Airplane Controllers

In this paper we consider an airplane control system with the four distributed controllers operating at different frequencies: the main controller, the left wing subcontroller, the right wing subcontroller, and the rudder subcontroller. The main controller has a 60 ms period, the left and the right wing controllers have a 15 ms period (rate 4), and the rudder controller has a 20 ms period (rate 3). The distributed controllers, their physical environments, and the connections between the components are illustrated in Figure 12.

Each subcontroller moves the surface of the wing towards the goal angle specified by the main controller, and sends back the current angle of the wing, while the angle of the wing is modeled by its physical environment. For example, the left wing subcontroller receives the goal angle $goal_L$ from the main controller, sends the current angle α_L to the main controller, and determines the moving rate $rate_L$ of the wing for its physical environment E_{Left}. The angle of the left wing changes according to the differential equation $\dot{\alpha_L} = rate_L$ during its 15 ms period. The right wing and the rudder subcontrollers are similar.

Given a desired direction $goal_\psi$ specified by the pilot, the main controller should determine the angles of the (devices of the) subcontrollers needed to make a smooth turn. The physical environment E_{Main} of the main controller maintains the position of the aircraft (ψ, ϕ, β), where ψ is the current direction, ϕ is the roll angle, and β is the yaw angle. The continuous dynamics of those angles are specified by the aeronautics equations (1–3) above, which are parameterized by the current wing angles $(\alpha_L, \alpha_V, \alpha_R)$. For each step of the main controller (period 60 ms), it receives the goal direction $goal_\psi$, the wing angles $(\alpha_L, \alpha_V, \alpha_R)$, and the position (ψ, ϕ, β), and sends the new goal angles $(goal_L, goal_V, goal_R)$ to the subcontrollers.

The behavior of such a physical environment can be specified by a periodic dynamic system. These physical dynamic systems can define the physical behavior of the corresponding controllers as explained in Section 3. For example, the physical environment E_{Left} of the left wing subcontroller can be modeled as the 1-dimensional periodic dynamic system $E_{Left} = (\mathbb{R},\ (-180°, 180°],\ 15\,\text{ms},\ \Lambda_{E_{Left}})$, where:

- the control command set is $\mathbb{R}$ for the moving rate $rate_L$ from the left wing subcontroller;
- the state set is $(-180°, 180°]$ to maintain the angle of the left wing;
- the physical transition relation is $\Lambda_{E_{Left}} \subseteq (\mathbb{R} \times (-180°, 180°]) \times \mathcal{T}_{15}$ given by:

$$\big((rate_L, \alpha_{L_0}), \alpha_L\big) \in \Lambda_{E_{Left}} \iff (\exists\, \alpha_L \in \mathcal{T}_{15})\ \ \dot{\alpha_L} = rate_L \ \wedge\ \alpha_L(0) = \alpha_{L_0}.$$

  That is, if the current wing angle is α_{L_0} at the beginning of the round and the moving rate is $rate_L$, then the trajectory α_L of the wing angle changes according to the differential equation $\dot{\alpha_L} = rate_L$ during its 15 ms period, where the initial value $\alpha_L(0)$ of α_L is the current wing angle α_{L_0}.

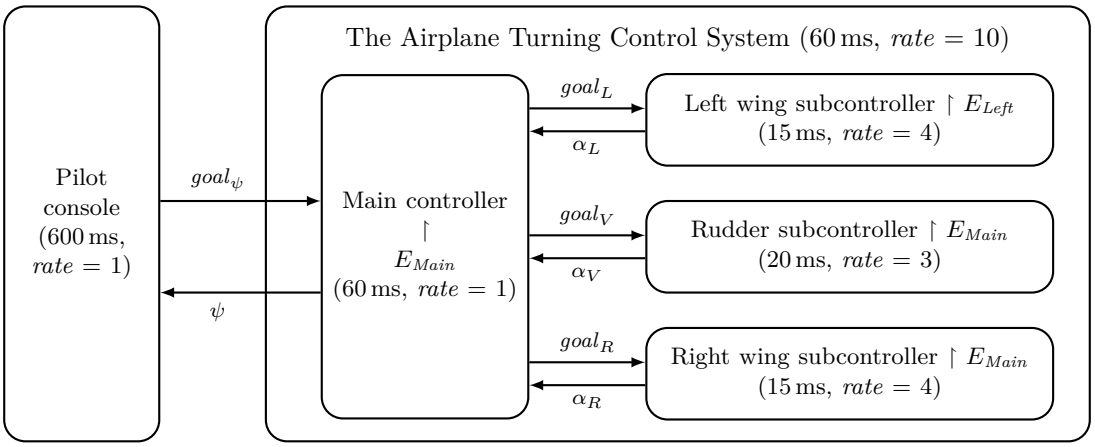

Figure 13: The architecture of our airplane turning control system.

Similarly, the physical environment E_{Main} for the main controller is specified by the 6-dimensional periodic dynamic system $E_{Main} = (\{*\},\ (-180°, 180°]^6,\ 60\,\text{ms},\ \Lambda_{E_{Main}})$, where:

- the control command set is the singleton set $\{*\}$, since there is no control command;
- the state set is $(-180°, 180°]^6$ to denote the direction angle ψ, the roll angle ϕ, the yaw angle β, the left wing angle α_L, the right wing angle α_R, and the rudder angle α_V;
- the physical transition relation is $\Lambda_{E_{Main}} \subseteq (\{*\} \times (-180°, 180°]^6) \times \mathcal{T}_{60}^6$, where

  $$\big((*, (\psi_0, \phi_0, \beta_0, \alpha_{L0}, \alpha_{R0}, \alpha_{V0})), (\psi, \phi, \beta, \alpha_L, \alpha_R, \alpha_V)\big) \in \delta_{E_{Main}}$$

  iff there exist trajectories $\psi, \phi, \beta, \alpha_L, \alpha_R, \alpha_V \in \mathcal{T}_{60}$ such that:

  $$\begin{aligned} & \dot{\psi} = (g/v) * \tan\phi \\ \wedge\ & (\dot{\phi})^2 = (C_l \cdot \alpha_R - C_l \cdot \alpha_L)/(W \cdot L_{Wing}) \\ \wedge\ & (\dot{\beta})^2 = C_d \cdot (C_l \cdot \alpha_R - C_l \cdot \alpha_L)/(W \cdot L_{Wing}) + C_l \cdot \alpha_V/(W \cdot L_{Aircraft}) \\ \wedge\ & (\psi_0, \phi_0, \beta_0, \alpha_{L0}, \alpha_{R0}, \alpha_{V0}) = (\psi(0), \phi(0), \beta(0), \alpha_L(0), \alpha_R(0), \alpha_V(0)), \end{aligned}$$

  where the first three lines denote the aeronautics differential equations (1–3). That is, given the current physical values $\psi_0, \phi_0, \beta_0, \alpha_{L0}, \alpha_{R0}, \alpha_{V0}$ at the beginning of the round, the trajectories ψ, ϕ, and β of the position angles change according the aeronautics equations during its 60 ms period, *provided that* the trajectories α_L, α_R, and α_V of the wing angles are given, where the initial values of the trajectories are the current values at the beginning of the round.

The behavior of E_{Main} is *nondeterministic*, since the wing angles α_L, α_R, and α_V are not specified by E_{Main}, but by the different physical environments E_{Left}, E_{Right}, and E_{Rudder}. We apply some approximation method to deal with this problem below.

We therefore specify the airplane turning control system as a multirate ensemble $\mathfrak{E}$ with four typed machines, as illustrated in Figure 13. Each typed machine in the ensemble $\mathfrak{E}$ incorporates its physical environment, where the behaviors of the physical parameters are specified by the corresponding periodic dynamic systems. For example, the main controller component now also maintains the position angles (ψ, ϕ, β) of the aircraft, and updates the angles using the aeronautics equations for each 60 ms period, according to the received wing angles $(\alpha_L, \alpha_V, \alpha_R)$ in its input port. The "external" environment for the entire airplane turning control system is the pilot console, which is modeled by another typed machine. The pilot console component is connected to the main controller on the outside of the control system.

Our synchronous model in Figure 13 is a *discrete approximation* of the original model in Figure 12. The physical environment E_{Main} of the main controller is *physically related* to the physical environments of the subcontrollers E_{Left}, E_{Rudder}, and E_{Right}, since the aeronautic equations depend on the continuous behavior of the wing angles. However, such "continuous trajectories" of values cannot be captured by typical ensemble connections, since any communication through ensemble connections is one-step delayed but those continuous values should be *immediately* delivered. Therefore, in order to compute the airplane position (ψ, ϕ, β) in E_{Main}, we use the "fixed" angles $(\alpha_L, \alpha_V, \alpha_R)$ received in the input ports of the main controller, instead of using continuous trajectories of the wing angles during a 60 ms period.[4] This also resolves the "nondeterministic" behavior problem of E_{Main} shown above, since the wing angles are provided by the main controller. It is important that the system is stable, so that the small differences caused by the approximation do not significantly affect the behavior of the system.

Using the framework introduced in Section 4 to specify and execute multirate synchronous ensembles in Real-Time Maude, we specify in Section 6 (and redefine in Section 7) the multirate ensemble $\mathfrak{E}$ defining the airplane control system. Our specification of the airplane control system is indeed *stable*, since we only use continuous control functions. In Section 7 we then exploit the bisimulation

$$\mathfrak{E} \simeq \mathcal{MA}(\mathfrak{E}, T, \Gamma)$$

to verify properties about the asynchronous realization $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ by model checking them on the much simpler system $\mathfrak{E}$ (see [8]); as shown in Section 8, it is unfeasible to directly model check the asynchronous model $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ due to the state space explosion caused by the interleavings.

6. Modeling the Airplane Turning Control System

This section formally specifies the airplane turning control system of Section 5 using the multirate ensemble framework in Real-Time Maude described in Section 4. The entire specification is available at http://formal.cs.illinois.edu/kbae/airplane.

6.1. Parameters and Data Types

The following parameters are chosen to be representative of a small general aviation aircraft. The speed of the aircraft is assumed to be $50\,m/s$ (that is, 180 km/h) and the gravity constant is $g = 9.80555\,m/s^2$.

```
eq planeSize     = 4.0 .     eq weight        = 1000.0 .     eq wingSize  = 2.0 .
eq vertLiftConst = 0.6 .     eq horzLiftConst = 0.4 .        eq dragRatio = 0.05 .
```

An angle of a wing is given by a floating-point number of sort `Float`. The function `angle` returns the value between $-180°$ and $180°$:

```
subsort Float < Data .

op angle : Float -> Float .
ceq angle(F:Float) = angle(F:Float - 360.0)  if F:Float > 180.0 .
ceq angle(F:Float) = angle(F:Float + 360.0)  if F:Float <= -180.0 .
eq  angle(F:Float) = F:Float [owise] .
```

[4] As mentioned in Section 3, it is beyond the scope of this paper to precisely model physical correspondences between different physical environments, since it would then be necessary to extend Multirate PALS.

6.2. Subcontrollers

The subcontrollers for the ailerons and the rudder are modeled as object instances of the following class `SubController`. The behavior of each subcontroller is straight-forward: in each iteration, it reads its input from the main controller—denoting the (updated) desired goal angle or $\perp$—, moves the corresponding aileron/rudder towards the goal angle, and outputs its current angle to the main controller.

```
class SubController | curr-angle : Float,
                      goal-angle : Float,
                      diff-angle : Float .
subclass SubController < Component .
```

A subcontroller increases/decreases the `curr-angle` toward the `goal-angle`, but the difference in a single (fast) round should be less than or equal to the maximal angle `diff-angle`. The transition function `delta` of a subcontroller is therefore defined by the following equation:

```
vars CA GA DA CA' GA' : Float .       vars LI LO : List{Data} .
ops input output : -> PortId [ctor] .

ceq delta(< C : SubController | ports : < input : InPort | content : D LI >
                                        < output : OutPort | content : LO >,
                                curr-angle : CA, goal-angle : GA, diff-angle : DA >)
  =
      < C : SubController | ports : < input : InPort | content : LI >
                                    < output : OutPort | content : LO CA' >,
                            curr-angle : CA', goal-angle : GA' >
   if CA' := angle(moveAngle(CA, GA, DA))
   /\ GA' := angle(if D == bot then GA else D fi) .

op moveAngle : Float Float Float -> Float .
eq moveAngle(CA, GA, DA) = if abs(GA + (- CA)) > DA then CA + DA * sign(GA + (- CA)) else GA fi .
```

The function `delta` updates the goal angle according to the input `D` received from the main controller, and keeps the previous goal if it receives `bot` (i.e., $\perp$). The function `moveAngle(CA, GA, DA)` gives the angle that is increased or decreased from the current angle `CA` to the goal angle `GA` up to the maximum angle difference `DA`. Finally, its (updated) current angle `CA'` is sent to the main controller through the output port `output`.

6.3. Main Controller

The main controller is modeled as an object instance of the class `MainController` below. The `velocity` attribute denotes the speed of the aircraft. The `curr-yaw`, `curr-roll`, and `curr-dir` attributes model the position sensors of the aircraft by indicating the current yaw angle β, roll angle ϕ, and direction ψ, respectively. The `goal-dir` attribute denotes the goal direction given by the pilot.

```
class MainController | velocity : Float,
                       goal-dir : Float,
                       curr-yaw : Float,
                       curr-rol : Float,
                       curr-dir : Float .
subclass MainController < Component .
```

In each iteration, the main controller reads its input (the reported angles from the device controllers and input from the pilot), computes and sends the new desired angles to the device controllers, and updates its state attributes `curr-yaw`, `curr-roll`, and `curr-dir` according to the aeronautics equations above. The `goal-dir` is also updated if a new goal direction arrives in the `input` port from the external environment (i.e., the pilot console). The transition function `delta` of the main controller is then defined as follows:

```
vars VEL LA RA TA CY CR CD GD : Float .     vars RA' TA' CY' CR' CD' GD' : Float .
var IN OUT : Data .                         vars PI LI RI TI PO LO RO TO : List{Data} .

ops input output inLW inRW inTW outLW outRW outTW : -> PortId [ctor] .

ceq delta(< C : MainController |
              ports : < input : InPort | content : IN PI >  < output : OutPort | content : PO >
                      < inLW : InPort | content : LA LI >   < outLW : OutPort | content : LO >
                      < inRW : InPort | content : RA RI >   < outRW : OutPort | content : RO >
                      < inTW : InPort | content : TA TI >   < outTW : OutPort | content : TO >,
              velocity : VEL, period : T,
              curr-yaw : CY, curr-rol : CR,
              curr-dir : CD, goal-dir : GD >)
  =
      < C : MainController |
          ports : < input : InPort | content : PI >  < output : OutPort | content : PO OUT >
                  < inLW : InPort | content : LI >   < outLW : OutPort | content : LO (- RA') >
                  < inRW : InPort | content : RI >   < outRW : OutPort | content : RO RA' >
                  < inTW : InPort | content : TI >   < outTW : OutPort | content : TO TA' >,
          curr-yaw : CY', curr-rol : CR',
          curr-dir : CD', goal-dir : GD' >
   if
      CY' := angle( CY + dBeta(LA, RA, TA) * float(T) )
   /\ CR' := angle( CR + dPhi(LA, RA) * float(T) )
   /\ CD' := angle( CD + evalPsi(CR, dPhi(LA, RA), VEL, float(T)) )
   /\ GD' := angle( if IN == bot then GD else GD + IN fi )
   /\ RA' := angle( horizWingAngle(CR', goalRollAngle(CR', CD', GD')) )
   /\ TA' := angle( tailWingAngle(CY') )
   /\ OUT := dir: CD' roll: CR' yaw: CY' goal: GD' .
```

The first four lines in the condition compute new values for `curr-yaw`, `curr-roll`, `curr-dir`, and `goal-dir`, based on values in the input ports. A non-$\perp$ value in the `input` port is added to `goal-dir`. The variables `RA'` and `TA'` denote new angles of the ailerons and the rudder, computed by the control functions explained below. Such new angles are queued in the corresponding output ports, and will be transferred to the related subcontrollers at the next synchronous step, since they are feedback outputs. The last line in the condition gives the output for the current step (i.e., the new position information of the aircraft), which will be transferred to its container ensemble at the end of the current synchronous step.

6.3.1. The Continuous Behavior

The new values of the yaw angle β, the roll angle ϕ, and the direction angle ψ are defined by the aeronautical differential equations. Since we assume discrete movements of the ailerons and the rudder as explained in Section 5.2, their moving rate $\dot{\beta}$ and $\dot{\phi}$ are approximated as some *constants* during each period of the main controller. Therefore, given the current yaw angle β_0 and the current roll angle ϕ_0, the angles are also approximated as the linear equations $\beta(t) = \beta_0 + \dot{\beta} \cdot t$ and $\phi(t) = \phi_0 + \dot{\phi} \cdot t$. Notice that these linear approximations are closely related to the Euler's method [5] to numerically solve differential equations.

Assuming that $\dot{\phi}$ is a constant, we can actually solve the differential equation for the direction angle ψ. For the current direction ψ_0, the direction angle ψ is given by the following function:

$$\psi(x) \;=\; \psi_0 + \int_0^x \frac{g}{v} \tan(\phi_0 + \dot{\phi} \cdot t)\, \mathrm{d}t \;=\; \psi_0 + \frac{g \cdot \big(\log(\cos\phi_0) - \log(\cos(\dot{\phi} \cdot x + \phi_0))\big)}{\dot{\phi} \cdot v}$$

In the `delta` equation of the main controller, the first two lines of the condition compute the new angles using the above linear equations for $\beta(t)$ and $\phi(t)$, and the third line of the condition computes the new direction `GD'` using the function $\psi(t)$ where the `evalPsi` function evaluate its second term.

6.3.2. The Control Functions

The new angles of the ailerons and the rudder are computed by the following *control functions*.

```
op horizWingAngle : Float Float -> Float .
op goalRollAngle : Float Float Float -> Float .
op tailWingAngle : Float -> Float .
```

The function `horizWingAngle` computes the new angle for the aileron in the right wing, based on the current roll angle and the goal roll angle. The angle of the aileron in the left wing is always exactly opposite to the one of the right wing. The function `goalRollAngle` computes the desired roll angle ϕ to make a turn, based on the current roll angle and the difference between the goal direction and the current direction. Finally, in order to achieve a coordinated turn where the yaw angle is always 0, the function `tailWingAngle` computes the new rudder angle based on the current yaw angle. We define those control functions by simple linear equations as follows, where `CR` is a current roll angle and `CY` is a current yaw angle:

```
eq goalRollAngle(CR, CD, GD) = sign(angle(GD - CD)) * min(abs(angle(GD - CD)) * 0.3, 20.0) .
eq horizWingAngle(CR, GR)    = sign(angle(GR - CR)) * min(abs(angle(GR - CR)) * 0.3, 45.0) .
eq tailWingAngle(CY)         = sign(angle(- CY))    * min(abs(angle(- CY)) * 0.8, 30.0) .
```

That is, the goal roll angle is proportional to the difference `GD - CD` between the goal and current directions (with the maximum 20°). The aileron angles are also proportional to the difference `GR - CR` between the goal and current roll angles (with the maximum 45°). Finally, the rudder angle is proportional to the difference $-$ `CY` between the goal and current yaw angles (with the maximum 30°), where the goal yaw angle is 0°.

6.4. Pilot Console

The pilot console, the external environment for the aircraft turning control system, is modeled as an object instance of the following class `PilotConsole`:

```
class PilotConsole | scenario : List{Data} .
subclass PilotConsole < Component .
```

The attribute `scenario` contains a list of goal angles to be transferred to the main controller. The pilot console keeps sending goal angles in the `scenario` to its output port until no more data remains in it. In addition, a nondeterministically generated value can be *added* to the value in the `scenario` output. Note that the pilot console receives the position information from the main controller in its input port, but it does not affect the behavior of the pilot console.

In the fourth rewrite rule below, the new desired direction sent by the pilot is `angle(F + F')`, where `F` is the first angle in the pilot's `scenario` attribute, and `F'` is an angle that is nondeterministically assigned to `0.0`, `10.0`, or `-10.0`. The equation below shows that if the `scenario` is empty, then the pilot console sends $\perp$ through its `output` port.

```
op input output : -> PortId [ctor] .
op pVar : -> Data .

rl pVar => 0.0 .      rl pVar => 10.0 .      rl pVar => -10.0 .

crl delta(< C : PilotConsole | ports : < input : InPort | content : IN LI >
                                       < output : OutPort | content : LO >,
                               scenario : F LI >)
  =>
        < C : PilotConsole | ports : < input : InPort | content : LI >
                                     < output : OutPort | content : LO OUT >,
                             scenario : LI >
   if pVar => F'
   /\ OUT := angle(F + F') .
```

```
  eq delta(< C : PilotConsole | ports : < input : InPort | content : IN LI >
                                        < output : OutPort | content : LO >,
                                scenario : nil >)
    =
          < C : PilotConsole | ports : < input : InPort | content : LI >
                                        < output : OutPort | content : LO bot >,
                                scenario : nil > .
```

6.5. The Airplane System

An initial state of the airplane turning control system is then represented as the following ensemble object, where the top-level ensemble `airplane` includes the pilot console `pilot` and the ensemble `csystem` for the airplane turning control system.

```
< airplane : Ensemble |
   period : 600,
   ports :  none,
   connections : (pilot . output --> csystem . input ; csystem . output --> pilot . input),
   machines :
     (< pilot : PilotConsole |
         period : 600,
         ports : < input : InPort | content : nil >  < output : OutPort | content : bot >,
         scenario : nil >
      < csystem : Ensemble |
         period : 60,
         ports :  < input : InPort | content : nil >  < output : OutPort | content : nil >,
         connections : (input --> main . input ;  main . output --> output ;
                        left . output --> main . inLW ;  main . outLW --> left . input ;
                        right . output --> main . inRW ;  main . outRW --> right . input ;
                        rudder . output --> main . inTW ;  main . outTW --> rudder . input),
         machines :
           (< main : MainController |
               period : 60,
               ports : < input : InPort | content : nil > < output : OutPort | content : nil >
                       < inLW : InPort | content : nil >  < outLW : OutPort | content : bot >
                       < inRW : InPort | content : nil >  < outRW : OutPort | content : bot >
                       < inTW : InPort | content : nil >  < outTW : OutPort | content : bot >,
               velocity : 50.0,  goal-dir : 0.0,
               curr-yaw : 0.0,  curr-rol : 0.0,  curr-dir : 0.0 >
            < left : SubController |
               period : 15,
               ports : < input : InPort | content : nil >
                       < output : OutPort | content : 0.0 >,
               curr-angle : 0.0,  goal-angle : 0.0,  diff-angle : 1.0 >
            < right : SubController |
               period : 15,
               ports : < input : InPort | content : nil >
                       < output : OutPort | content : 0.0 >,
               curr-angle : 0.0,  goal-angle : 0.0,  diff-angle : 1.0 >
            < rudder : SubController |
               period : 20,
               ports : < input : InPort | content : nil >
                       < output : OutPort | content : 0.0 >,
               curr-angle : 0.0,  goal-angle : 0.0,  diff-angle : 0.5 >) >) >
```

Notice that each feedback output port in the ensemble contains some default value. For example, the output port `outLW` of the main controller contains the initial value `bot`, which will be delivered to the left wing subcontroller in the first synchronous step.

6.6. Input Adaptors

In the ensemble `csystem`, the left and right wing subcontrollers (with 15 ms period) are 4 times faster than the main controller (with 60 ms period), and the rudder subcontroller (with 20 ms period) is 3 times faster than the main controller. Therefore, in each step of `csystem`, the main controller receives 4-tuples from the left and right wing subcontrollers, and 3-tuples from the rudder subcontrollers. To transform such tuples of data into a single value for the main controller, we define input adaptors for the main controller that choose the last value from the input vector:

```
var D : Data .    var LI : List{Data} .

eq adaptor(main, inLW, LI D) = D .
eq adaptor(main, inRW, LI D) = D .
eq adaptor(main, inTW, LI D) = D .
```

The subcontrollers in `csystem` receive a single value from the main controller for a "slow" synchronous step, but the left and right wing subcontrollers require a 4-tuples of data, and the rudder subcontroller needs a 3-tuple of data in each step. Therefore, we define the input adaptors for the subcontrollers that generate a vector with extra $\perp$'s as follows, where the function `bots(n)` generates n $\perp$-constants:

```
eq adaptor(left,   input, D) = D bots(3) .
eq adaptor(rudder, input, D) = D bots(2) .
eq adaptor(right,  input, D) = D bots(3) .
```

Similarly, in the top ensemble `airplane`, since the ensemble `csystem` is 10 times faster than the pilot console, the input adaptor for the ensemble `csystem` generates a vector with 9 extra $\perp$'s, and the input adaptor for the pilot console takes the last value from the received input vector:

```
eq adaptor(pilot, input, LI D) = D .
eq adaptor(csystem, input,  D) = D bots(9) .
```

7. Formal Analysis of the Airplane Turning Control System

This section explains how we have formally analyzed the Real-Time Maude model of the multirate synchronous design of the airplane turning control system, and how the turning algorithm has been improved as a result of our analysis. The two main requirements that the system should satisfy are:

- *Safety*: during a turn, the yaw angle should always be close to 0.
- *Liveness*: the airplane should reach the goal direction within a reasonable time, and with both the roll angle and the yaw angle close to 0.

7.1. First Analysis Results

We first analyze deterministic[5] behaviors where the airplane turns $+60°$ to the right.[6] In this case, the pilot (console) can send different sequences of commands to the control system to achieve this ultimate goal. We consider the following variations:

1. The pilot *gradually* increases the goal direction 6 times, by adding $+10°$ each step.
2. The pilot sets the goal direction to $+60°$ immediately.
3. The goal direction is at first $-30°$, and then it is suddenly set to $+60°$.

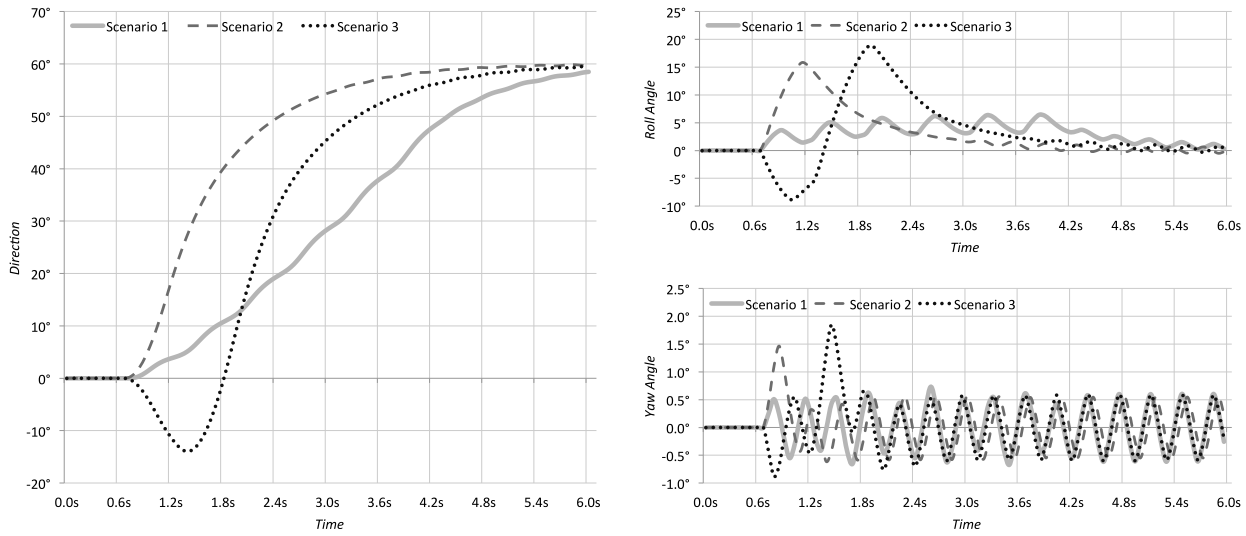

Figure 14: The simulation results for the three turning scenarios: the direction of aircraft (left), the roll angle (top right), and the yaw angle (bottom right).

Figure 14 shows the simulation results for these three scenarios up to 6 seconds (10 steps of the pilot), obtained by using the following Real-Time Maude simulation commands, where the term `model(`*Scenario*`)` gives the initial state of the system with a list of directions *Scenario* for the pilot console:

```
(tfrew {model(10.0 10.0 10.0 10.0 10.0 10.0)} in time <= 6000 .)
(tfrew {model(60.0)} in time <= 6000 .)
(tfrew {model(-30.0 90.0)} in time <= 6000 .)
```

The graphs in Figure 14 show that the airplane reaches the desired 60° direction in a fairly short time in all three scenarios, and the roll angle also becomes stable. However, the yaw angle seems to be quite unstable, in particular, when the pilot gives a sharp turn command to the main controller.

7.2. New Control Functions

There are two main reasons why the yaw angle is not sufficiently close to 0° during a turn as follows:

1. First, since the control functions `horizWingAngle`, `tailWingAngle`, and `goalRollAngle` are linear, the new angles for the ailerons and the rudder are not small enough when the yaw angle is near 0°. For example, if the current yaw angle is close enough to 0°, the rudder angle should be 0° to make the airplane stable and to keep the current yaw angle. However, since `tailWingAngle` is linear, it returns an angle whose absolute value is greater than 0°, so that the yaw angle goes beyond 0° in the other direction. Consequently, the yaw angle of the airplane shows the zigzag pattern in Figure 14.

2. Second, the roll angle is sometimes changing too fast, since the function `goalRollAngle` gives a large number, so that the rudder cannot effectively counter the adverse yaw.

Therefore, we modify the control functions as follows. When the difference between the goal and the current angles (i.e., `FR` or `FY`) is less than or equal to 1°, the functions `horizWingAngle` and `tailWingAngle` are now *not* proportional to the difference, but proportional to the *square* of the difference. This implies that if the difference is close to 0°, then the result of the function becomes much smaller than before. Furthermore, the goal roll angle can be changed at most 1.5° at a time, so that there is no more abrupt rolling.

[5]That is, we use only the single rewrite rule `pVar => 0.0` for the pilot console (see Section 6.4).
[6]In our model, a turn of positive degrees is a right turn, and one of negative degrees a left turn.

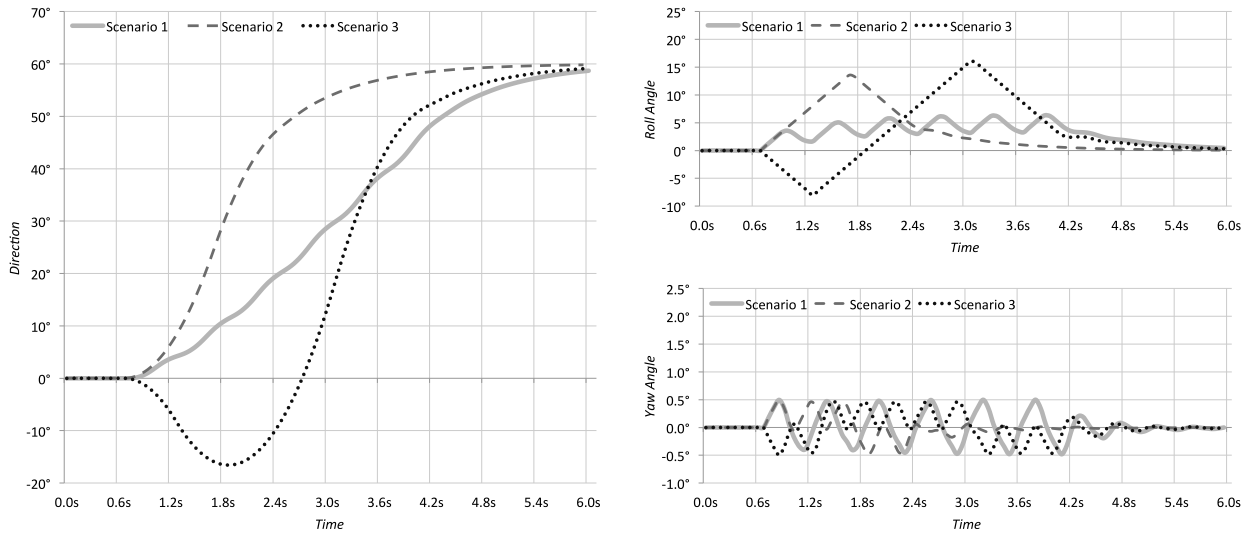

Figure 15: The simulation results of the redesigned model for the three turning scenarios: the direction of aircraft (left), the roll angle (top right), and the yaw angle (bottom right)

```
ceq horizWingAngle(CR, GR)
  = sign(FR) * (if abs(FR) > 1.0 then min(abs(FR) * 0.3, 45.0) else FR ^2.0 * 0.3 fi)
 if FR := angle(GR - CR) .

ceq tailWingAngle(CY)
  = sign(FY) * (if abs(FY) > 1.0 then min(abs(FY) * 0.8, 30.0) else FY ^2.0 * 0.8 fi)
 if FY := angle(- CY) .

ceq goalRollAngle(CR, CD, GD)
  = if abs(FD * 0.32 - CR) > 1.5 then CR + sign(FD * 0.32 - CR) * 1.5 else FD * 0.32 fi
 if FD := angle(GD - CD) .
```

By using the same simulation commands, we obtain the simulation results of the redesigned model with the new control functions in Figure 15, where the yaw angle now shows a much stabler behavior than before.

7.3. Safety Checking of the New Control Functions

To analyze whether or not the new control functions ensure a smooth turn, we define some auxiliary functions as follows. The function `systemOutput` returns the content of the output port of the airplane control system, and the function `safeYawAll(`*OutputDataList*`)` checks whether every output data in the given list has a yaw angle less than 1.0°:

```
op systemOutput : Object -> List{Data} .
eq systemOutput(
    < airplane : Ensemble |
        machines : < csystem : Ensemble |
                     ports : < output : OutPort | content : LI >  PORTS > COMPONENTS >) = LI .

op safeYawAll : List{Data} -> Bool .
eq safeYawAll(DO LI) = abs(yaw(DO)) < 1.0 and safeYawAll(LI) .
eq safeYawAll(nil)   = true .
```

Using the Real-Time Maude search command, we can verify that there is no dangerous yaw angle reachable within 27 seconds for Scenario 3 (the number of states explored is 46):

```
Maude> (tsearch [1] {model(-30.0 90.0)} =>* {SYSTEM}
           such that not safeYawAll(systemOutput(SYSTEM))
              in time <= 27000 .)

No solution
```

In fact, the set of states reachable from `{model(-30.0 60.0)}` is finite, when we only "sample" the continuous values at certain times, since all the angles will eventually become zero and will remain zero (since there is no new turn command in scenario-3) and the direction should eventually become, and remain, the desired direction 60°. We can therefore use *unbounded* reachability analysis to check that our desired property holds in the unbounded future (the number of states explored is 58):

```
Maude> (utsearch [1] {model(-30.0 60.0)} =>* {SYSTEM}
           such that not safeYawAll(systemOutput(SYSTEM)) .)

No solution
```

Although each state of the transition system captures only the slow steps for the top ensemble (i.e., every 600 ms), `safeYawAll` also checks all fast steps for the main controller (every 60 ms), since it accesses the *history* of the main controller's status in the output port of the top ensemble, which the main controller sends to the top ensemble for each of its fast steps.

7.4. Model Checking the New Control Functions

We can use Real-Time Maude's LTL model checker to verify both liveness and safety at the same time. The desired property is that the airplane reaches the desired direction with a stable status while keeping the yaw angle close to 0, which can formalized as the LTL formula

$$\Box(\neg \texttt{stable} \rightarrow (\texttt{safeYaw}\ \mathbf{U}\ (\texttt{reach} \wedge \texttt{stable})))$$

where the atomic propositions `safeYaw`, `stable`, and `reach` are defined as follows:

```
eq {SYSTEM} |= safeYaw = safeYawAll(systemOutput(SYSTEM)) .
eq {SYSTEM} |= stable  = stableAll(systemOutput(SYSTEM)) .
ceq {SYSTEM} |= reach   = abs(angle(goal(DO) - dir(DO))) < 0.5
    if DO := last(systemOutput(SYSTEM)) .
```

and the function `stableAll(`*OutputDataList*`)` returns `true` only if both the yaw angle and the roll angle are less than 0.5° for every output data in the *OutputDataList*:

```
op stableAll : List{Data} -> Bool .
eq stableAll(DO LI) = abs(roll(DO)) < 0.5 and abs(yaw(DO)) < 0.5 and stableAll(LI) .
eq stableAll(nil)   = true .
```

We have verified that all three scenarios satisfy the above LTL property with the *new* control functions, using the time-bounded LTL model checking command of Real-Time Maude. For example, the following command shows the case for Scenario 3 (the number of states explored is 13):

```
Maude> (mc {model(-30.0 90.0)}
              |=t
           [] (~stable -> (safeYaw U reach /\ stable))
           in time <= 7200 .)

Result Bool : true
```

The same formula can also be verified using the following *untimed* LTL model checking command, except that in this case we cannot check whether the airplane reaches the goal direction within a certain time (the number of states explored is 59):

```
Maude> (mc {model(-30.0 90.0)}
           |=t
           [] (~ stable -> (safeYaw U reach /\ stable)) .)

Result Bool : true
```

Finally, we have verified nondeterministic behaviors in which the pilot sends one of the turning angles $-60.0°$, $-10.0°$, $0°$, $10°$, and $60.0°$ to the main controller for 6 seconds. Such nondeterministic behaviors can be modeled by adding the following four rewrite rules for the pilot console (see Section 6.4):

```
rl pVar => 0.0 .
rl pVar => 10.0 .
rl pVar => -10.0 .
rl pVar => 60.0 .
rl pVar => -60.0 .
```

The following model checking command then shows that our redesigned system, with the new control functions, satisfies the above LTL property for all behaviors up to 18 seconds, where one of the five angles is nondeterministically chosen and added to the angle $0°$ at each step of the pilot console:

```
Maude> (mc {model(0.0 0.0 0.0 0.0 0.0 0.0)}
           |=t
           [] (~ stable -> (safeYaw U (reach /\ stable)))
           in time <= 18000 .)

Result Bool : true
```

This model checking analysis took 75 minutes on Intel Core i5 2.4 GHz with 4 GB memory, and the number of states explored was $335,363$.[7] We can also model check this property with no time bound:

```
Maude> (mc {model(0.0 0.0 0.0 0.0 0.0 0.0)}
           |=u
           [] (~ stable -> (safeYaw U (reach /\ stable))) .)

Result Bool : true
```

This command took about 4 hours and 13 minutes on the same machine, and the number of states explored in this model checking analysis was $576,590$. It is a huge state space reduction compared to the distributed asynchronous model as shown in Section 8, since:

(i) asynchronous behaviors are eliminated thanks to Multirate PALS, and

(ii) any intermediate (fast) steps for the subcomponents are merged into a single step of the system's top-level ensemble.

[7] Remember that only the outermost "big-step" transitions contribute to the state space. However, computing each such "big-step" transition of duration 600 ms involves computing many "small-step" transitions that do not contribute to the state space; i.e., 10 transitions of the main controller, 40 transitions of each left/right wing subcontroller (15 ms), and 30 transitions of the rudder controller (20 ms). Furthermore, because of nondeterministic behaviors, different computation paths can lead to the same state. Therefore, the model checking involves computing significantly more than 335,363 behaviors, which can explain the long model checking time.

8. Model Checking the Asynchronous System

To demonstrate the performance benefits obtained by using Multirate PALS, this section compares the execution times and the number of system states explored for model checking the synchronous model and the corresponding distributed asynchronous model. We consider a *highly simplified* asynchronous model with the following simplifying assumptions:

- The time domain is discrete and all clocks are perfectly synchronized (i.e., there are no clock skews).
- A machine takes zero time to perform a transition, including processing I/O (no execution times).
- Each message is *instantaneously* delivered to its recipient (no message delays).
- When a component sends output messages into the network, it sends all the messages *at the same time*, instead of sending them one by one.

In the asynchronous model, each component runs according to its own period, and communicates with other components by sending and receiving messages asynchronously. When a component begins its new local period, it reads the incoming messages from its input ports, performs its local transition, and places the generated outputs in its output ports. The input adaptor functions are applied to deal with inputs from components with different periods. To ensure that those output messages are used in the *next* round of the recipient, they are sent into the network as follows:

- A component with period $k \cdot T$ sends a k-tuple of data at the same time to the *slower* component with period T, *one time unit after* all its k local transitions are performed.
- A component sends an output message to a *faster* (or equally fast) component *one time unit after* each local transition is performed.

An output message generated in one round can therefore not be used in the same round, since it is sent one time unit after the beginning of the round.

We formally specify the *simplified* asynchronous model in Real-Time Maude. Our specification also supports a *hierarchical system design* in which a component may communicate with both faster and slower components. For example, the main controller (period 60 ms) in our airplane turning control system is connected to both the pilot console (period 600 ms) and the left wing subcontroller (period 15 ms).

8.1. The Asynchronous Model in Real-Time Maude

An asynchronous component is an object instance of (a subclass of) the following class `AsyncComponent`, which integrates all its *wrappers* and the inner component specified by an object instance of `Component`:

```
class AsyncComponent | rate : NzNat,
                       counter : Nat,
                       timer : Time,
                       fastInputs : Set{PortName},
                       slowInputs : Set{PortName},
                       fastOutputs : Set{Connection},
                       slowOutputs : Set{Connection},
                       fastOutputTimer : TimeInf,
                       slowOutputTimer : TimeInf,
                       fastBuffer : Configuration,
                       slowBuffer : Configuration .
subclass AsyncComponent < Component .
```

The `rate` attribute denotes the rate of the component compared to the slower components, and `counter` denotes the number of fast transitions that have been taken in the current slow period. The `timer` attribute denotes the time until a new *fast* period begins. That is, a new *slow* round of the asynchronous component begins when both `timer` and `counter` attributes are 0. The other attributes of `AsyncComponent` are used to control its input and output; the `fast` attributes are related to *faster* (or equal) components, and the `slow` attributes are related to *slower* components. For example, `fastInputs` denotes a set of input ports that are connected to faster components, and `fastOutputs` denotes a set of connections from output ports to input ports of faster components. The output message to a faster component generated during its round is stored in `fastBuffer`, until it is sent into the network when the `fastOutputTimer` expires. The `slow` attributes are similar to the `fast` attributes, but are used for communication with slower components.

In the aircraft turning control system, each controller component is an instance of both its corresponding controller class and `AsyncComponent`. For example, the main controller is an instance of `MainController` and `AsyncComponent`. Therefore, we define their common "asynchronous" controller classes as follows:

```
class AsyncSubController .      subclass AsyncSubController  < AsyncComponent SubController .
class AsyncMainController .     subclass AsyncMainController < AsyncComponent MainController .
class AsyncPilotConsole .       subclass AsyncPilotConsole   < AsyncComponent PilotConsole .
```

Each state of the asynchronous model is then represented as a configuration of `AsyncComponent` objects. Unlike the synchronous model, the structure is *flattened* to model distributed asynchronous components. For example, the asynchronous model of the aircraft turning control system is represented as follows:

```
< pilot : AsyncPilotConsole |
   period : 600,  rate : 1,  counter : 0,  timer : 0,
   ports : < input : InPort | content : bot >  < output : OutPort | content : nil >,
   scenario : nil,
   fastOutputTimer : INF,  fastBuffer : none,  fastInputs : input,  fastOutputs : output --> main . input,
   slowOutputTimer : INF,  slowBuffer : none,  slowInputs : empty,  slowOutputs : empty >

< left-wing : AsyncSubController |
   period : 15,  rate : 4,  counter : 0,  timer : 0,
   ports : < input : InPort | content : bot >  < output : OutPort | content : nil >,
   curr-angle : 0.0,  goal-angle : 0.0 ,  diff-angle : 1.0,
   fastOutputTimer : INF,  fastBuffer : none,  fastInputs : empty,  fastOutputs : empty,
   slowOutputTimer : INF,  slowBuffer : none,  slowInputs : input,  slowOutputs : output --> main . inLW >

< right-wing : AsyncSubController |
   period : 15,  rate : 4,  counter : 0,  timer : 0,
   ports : < input : InPort | content : bot >  < output : OutPort | content : nil >,
   curr-angle : 0.0,  goal-angle : 0.0,  diff-angle : 1.0,
   fastOutputTimer : INF,  fastBuffer : none,  fastInputs : empty,  fastOutputs : empty,
   slowOutputTimer : INF,  slowBuffer : none,  slowInputs : input,  slowOutputs : output --> main . inRW >

< vert-wing : AsyncSubController |
   period : 20,  rate : 3,  counter : 0,  timer : 0,
   ports : < input : InPort | content : bot >  < output : OutPort | content : nil >,
   curr-angle : 0.0,  goal-angle : 0.0,  diff-angle : 0.5,
   fastOutputTimer : INF,  fastBuffer : none,  fastInputs : empty,  fastOutputs : empty,
   slowOutputTimer : INF,  slowBuffer : none,  slowInputs : input,  slowOutputs : output --> main . inTW >

< main : AsyncMainController |
   period : 60,  rate : 10,  counter : 0,  timer : 0,
   ports : < input : InPort | content : bot >    < output : OutPort | content : nil >
           < inLW : InPort | content : 0.0 >  < outLW : OutPort | content : nil >
           < inRW : InPort | content : 0.0 >  < outRW : OutPort | content : nil >
           < inTW : InPort | content : 0.0 >  < outTW : OutPort | content : nil >,
   velocity : 50.0,  curr-yaw : 0.0,  curr-rol : 0.0,  curr-dir : 0.0,  goal-dir : 0.0,
   fastOutputTimer : INF,  fastBuffer : none,  fastInputs : (inLW, inRW, inTW),
   fastOutputs : outLW --> left-wing . input ; outRW --> right-wing . input ; outTW --> vert-wing . input,
   slowOutputTimer : INF,  slowBuffer : none,  slowInputs : input,  slowOutputs : output --> pilot . input >
```

Communication between different components is formalized by explicit message passing, using messages of the form (msg *Data* to *Target*) declared as follows:

```
op msg_to_ : NeList{Data} PortName -> Msg [ctor] .
```

The rewrite rules for asynchronous communications are straightforward. In the recv rule, the component C receives the incoming message (msg NDL to C . P) and puts it in the corresponding input port P:

```
rl [recv]: (msg NDL to C . P)
           < C : AsyncComponent | ports : < P : InPort | content : DL > PORTS >
        =>
           < C : AsyncComponent | ports : < P : InPort | content : DL NDL > PORTS > .
```

There are two rules for sending messages: sendFast and sendSlow. The rule sendFast sends the messages MSGS in fastBuffer generated for faster components into the network when fastOutputTimer expires, and then turns fastOutputTimer off. The sendSlow rule is similar but uses slowBuffer and slowOutputTimer for messages to slower components:

```
rl [sendFast]: < C : AsyncComponent | fastOutputTimer : 0, fastBuffer : MSGS >
            =>
               < C : AsyncComponent | fastOutputTimer : INF, fastBuffer : none >  MSGS .

rl [sendSlow]: < C : AsyncComponent | slowOutputTimer : 0, slowBuffer : MSGS >
            =>
               < C : AsyncComponent | slowOutputTimer : INF, slowBuffer : none >  MSGS .
```

When a component begins a new round (timer = 0), the appropriate input adaptors are applied to its input ports, the transition is performed by using the delta operator after setting timer to its period, and output is put into the relevant output buffers. Since we assume in our simplified asynchronous model that the execution time is zero, the operator delta is defined by rewrite rules in the same way as the synchronous semantics. The asynchronous transition of each component is specified by the following rewrite rule, where rem is the reminder function:

```
var OBJ : Object .   vars FNS SNS : Set{PortName} .   vars N : Nat .   var NZ : NzNat .

crl [step]:
    < C : AsyncComponent | timer : 0,    period : T,     rate : NZ,
                           counter : N,  ports : PORTS,  fastInputs : FNS,  slowInputs : SNS >
 =>
    procBufferOutput(OBJ)
 if delta(< C : AsyncComponent |
             timer : T,
             counter : s(N) rem NZ,
             fastOutputTimer : 1,
             slowOutputTimer : (if NZ == s(N) then 1 else INF fi),
             ports : applyAdaptors(C, (if N == 0 then (FNS, SNS) else FNS fi), PORTS) >)  =>  OBJ .
```

In the condition of the rule, fastOutputTimer is set to 1 so that the messages in fastBuffer are sent one time unit later; but slowOutputTimer is set to 1 only if all its fast transitions are performed in the slow round (i.e., rate = counter + 1). If an input port is connected to a slower component, the adaptor for the input port is applied only if the current round is a slow round (counter = 0), where the applyAdaptors function now takes an extra argument to denote a set of chosen input port names:

```
op applyAdaptors : ComponentId Set{PortName} Configuration -> Configuration .
eq applyAdaptors(C, (P, PNS), < P : InPort | content : NDL > PORTS)
 = applyAdaptors(C, PNS, PORTS) < P : InPort | content : adaptor(C,P,NDL) > .
eq applyAdaptors(C, empty, PORTS) = PORTS .
```

After performing the `delta` operator, all the data in the output ports are transferred to the corresponding output buffers according to the connection information in `fastOutputs` and `slowOutputs`:

```
vars FCONXS SCONXS : Set{Connection} .   vars FMSGS SMSGS : Configuration .

op procBufferOutput : Object -> Object .
eq procBufferOutput(< C : AsyncComponent | ports : < P : OutPort | content : NDL > PORTS,
                                           fastOutputs : FCONXS,   fastBuffer : FMSGS,
                                           slowOutputs : SCONXS,   slowBuffer : SMSGS >)
 =
   procBufferOutput(< C : AsyncComponent | ports : < P : OutPort | content : nil > PORTS,
                                           fastBuffer : merge(FMSGS, genMsgs(P, NDL, FCONXS)),
                                           slowBuffer : merge(SMSGS, genMsgs(P, NDL, SCONXS)) >) .
eq procBufferOutput(OBJECT) = OBJECT [owise] .
```

where the `genMsgs` generates messages from output ports using connections, and the `merge` function combines two multisets of messages so as to generate messages with k-tuples of data for slow components.

Finally, the following `tick` rule advances time until some asynchronous event must happen, where the function `timeEffect` defines how the system state changes according to the time elapsed, and `mte` defines the maximum amount of time that may elapse in the system until some timer expires (i.e., becomes 0):

```
crl [tick] : {< C : AsyncComponent | >}
          =>
             {timeEffect(< C : AsyncComponent | >, T)} in time T
 if T := mte(< C : AsyncComponent | >) .
```

The function `timeEffect` is distributed over the objects and messages in the state, and decreases the timers of each component in the system by the amount of time elapsed (`TE`):

```
vars NCS NCF' : NEConfiguration .    vars T TI TI' TE : Time .    var M : Msg .

op timeEffect : Configuration Time -> Configuration [frozen] .
eq timeEffect(NCF NCF', TE) = timeEffect(NCF, TE) timeEffect(NCF', TE) .
eq timeEffect(none, TE) = none .

eq timeEffect(
     < C : AsyncComponent | timer : T, fastOutputTimer : TI, slowOutputTimer : TI' >, TE)
 =
     < C : AsyncComponent | timer : T monus TE, fastOutputTimer : TI monus TE,
                            slowOutputTimer : TI' monus TE > .

eq timeEffect(M, TE) = M .
```

Similarly, `mte` of a configuration is the smallest `mte` value of an object or a message in the configuration, where `mte` of a component is the smallest value in its timers, and `mte` of a message is 0:

```
op mte : Configuration -> TimeInf [frozen] .

eq mte(NCF NCF') = min(mte(NCF), mte(NCF')) .
eq mte(none) = INF .

eq mte(< C : AsyncComponent | timer : T, fastOutputTimer : TI, slowOutputTimer : TI' >)
 = min(T, TI, TI') .

eq mte(M) = 0 .
```

Notice that time can advance only after all the messages in the configuration have arrived to their corresponding ports, since `mte` of any message is 0.

Model	N	$T \leq$ 1,200 ms		$T \leq$ 1,800 ms		$T \leq$ 2,400 ms		$T \leq$ 3,000 ms	
		#states	time (s)	#states	time (s)	#states	time (s)	#states	time (s)
Sync.	2	7	0.2	13	0.2	25	0.3	49	0.5
	3	13	0.2	28	0.3	73	0.6	202	1.6
	4	21	0.3	57	0.5	169	1.3	593	4.4
	5	31	0.3	116	0.9	471	3.3	2,111	15
Async.	2	12,552	1.6	25,088	3.2	50,144	6.3	100,256	13
	3	25,088	3.1	56,512	7.5	150,576	22	420,288	84
	4	41,816	9	117,232	30	351,680	117	1,238,648	611
	5	62,752	35	240,784	168	983,960	998	4,415,784	8,679

Table 1: The synchronous and the asynchronous models up to time bound T with N nondeterministic pilot choices.

8.2. Comparison

We first perform the same safety analysis for the three deterministic pilot scenarios to turn the airplane 60° to the right. The following command verifies that that there is no dangerous yaw angle greater than 1° within time bound 27,000 for the scenario 1:

```
Maude> (tsearch [1] {model(10.0 10.0 10.0 10.0 10.0 10.0)} =>* {SYSTEM}
          such that abs(currYaw(SYSTEM)) > 1.0
             in time <= 27000 .)

No solution
```

In all the three scenarios, the number of states explored is 93,632; the same analysis of the synchronous model explored only 46 states.

Next, we model check the LTL formula $\Box(\neg\texttt{stable} \rightarrow (\texttt{safeYaw}\ \mathbf{U}\ (\texttt{reach} \wedge \texttt{stable})))$ for Scenario 1 in our asynchronous model:

```
Maude> (mc {model(10.0 10.0 10.0 10.0 10.0 10.0)}
             |=t
          [] (~stable -> (safeYaw U reach /\ stable))
       in time <= 7200 .)

Result Bool : true
```

This analysis explored 24,992 system states in all three scenarios, even though the pilot gives the deterministic scenarios, whereas only 13 states are explored in the synchronous model.

Finally, we have compared the number of reachable states and the execution times in the both models for various *nondeterministic* pilot choices, using Real-Time Maude's time-bound search command:

```
(tsearch [1] {model(60.0 data(k,0.0))} =>* {none} in time <= TimeBound .)
```

where `data(`k`,0.0)` generates a list of k zeros. Since the term `{none}`, meaning that there are no components in the system, is never reachable, this search command explores the entire state space. In this analysis, the goal direction is initially 60°, but the pilot can nondeterministically choose one of the five angles 0°, 10°, −10°, 60°, −60.0°, and add it to the goal direction every 600 ms. These experiments were conducted on an Intel Xeon 2.93 GHz with 24 GB memory. Table 1 summarizes the experiment results with different time bounds and different nondeterministic choices. For each N in the table we use the first N angles from 0°, 10°, −10°, 60°, −60.0° in order; for example, in the experiment we use $\{0°, 10°\}$ for $N = 2$.

Table 1 illustrates how quickly the system's state space explodes, even for our unrealistically simplified asynchronous model of the airplane turning control system. The number of reachable states up to 3,000 ms in the synchronous model is 2,111 with 5 nondeterministic choices, and the model checking takes only about fifteen seconds. However, even for only 2 nondeterministic choices, the asynchronous model generates 12,552 states within time bound 1,200 ms.

9. Related Work

The PALS pattern in general is part of a broader body of work on *synchronizers*, which allow (single-rate) synchronous systems to be *simulated* by asynchronous ones. Very general synchronizers such as those in [6] place no bounds on message delays, so that *physical time* in the original synchronous system is simulated by *logical time* in its asynchronous counterpart. More recent work has developed synchronizers for the Asynchronous Bounded Delay (ABD) Network model [33], in which a bound can be given for the delay of message transmissions. PALS also assumes the ABD model (plus clock synchronization) but provides *hard real-time guarantees* needed for embedded systems, whereas in the synchronizers in [33], two nodes could be in completely different (local) rounds at the same global physical time. Work by Tripakis et al. [34] relates a synchronous Mealy machine model to a loosely timed triggered architecture with local clocks that can advance at different rates with no clock synchronization. The main difference with PALS is that it does not seem possible to give hard real time bounds for the behavior of the asynchronous system realization. In the Globally Synchronous Locally Asynchronous (GALS) architecture, e.g., [19, 30], systems may be widely distributed and it may not be possible to assume that all message communication delays are bounded, although such delays may be bounded within a synchronous subdomain. Consequently, no hard real-time guarantees can be given for a GALS implementation.

Single-rate PALS is also closely related to the *time-triggered systems* of Kopetz and Rushby [20, 31], where the goal is also to reduce an asynchronous real-time system to a synchronous one. One important difference between the work of Kopetz and Rushby and PALS comes from the somewhat different definitions of the synchronous models, which have significant repercussions in the behaviors of the corresponding asynchronous models. In particular, the smallest possible period of the asynchronous system of Kopetz and Rushby is typically significantly larger than in PALS. We refer to [25] for a thorough discussion of work related to single-rate PALS, and to [32] for an in-depth comparison between time-triggered systems and PALS.

The PALS pattern was extended to multirate systems in two different ways. In [1], the authors describe their version of multirate PALS informally using AADL. In [8, 9], three of us develop the mathematical foundations of a different version of Multirate PALS, and prove that a synchronous multirate system and the distributed asynchronous system satisfy the same temporal logic properties. The papers [8, 9] also briefly summarize the case study in this paper.

In [2, 25], the single-rate PALS pattern is applied to a single-rate avionics system for deciding which of two computer systems is active in an aircraft. This system periodically receives Boolean signals from the environment about the operating status of the two computer systems, and then decides which computer system should be the *active* system. Apart from having a completely different purpose than our case study, the active standby system is a single-rate system and is not a hybrid system.

On the hybrid systems side, Maude itself has been extended to hybrid systems in two different ways: Meseguer and Sharykin extend in [26] Maude to object-based stochastic hybrid systems that can then be statistically model checked using tools such as PVeStA [3]. Fadlisyah et a. have defined a way of decomposing the definition of the complex continuous behaviors in hybrid systems where the *continuous* behavior of a component may depend on the continuous behaviors of other components [16]. The associated HI-Maude tool [15] provides a number of numerical approximation methods for such systems, including the Euler and Runge-Kutta methods [5], and has been used to model and analyze the human thermoregulatory system [17].

Finally, although avionics systems have been subjected to formal verification for a long time (see, for example, [21] and the references therein for a brief overview of some recent work), we are not aware of any formal model checking or verification analysis of a turning algorithm that defines how to move the ailerons and rudder to achieve a smooth turn.

10. Conclusions

The present work can be seen from different perspectives. First, from the perspective of research on the PALS methodology, its main contribution is to demonstrate that Multirate PALS, when used in combination with a tool like Real-Time Maude, can be effectively applied to the formal verification of nontrivial DCPS designs, and even to the process of refining a DCPS design before it is verified. Second, from the perspective

of the formal specification and verification of distributed hybrid systems, it also shows that Real-Time Maude is an effective tool for specifying and verifying such systems.

Encouraged by the results in this paper, we have recently made the Multirate PALS modeling and verification methodology available within an industrial modeling environment through the *Multirate Synchronous AADL* language and the *MR-SynchAADL* tool [10]. Multirate Synchronous AADL allows the modelers to specify their Multirate PALS synchronous models using the industrial modeling standard AADL [18]; this model can then be verified directly from the OSATE tool environment for AADL, with Real-Time Maude as the verification back-end.

As usual much work remains ahead. More case studies should be developed for the design and verification of distributed cyber-physical systems using Multirate PALS. Furthermore, the hybrid system applications of Real-Time Maude should be further developed independently of PALS. Several such applications have been developed in the past; but many more are possible, and a richer experience will be gained.

Acknowledgments

We would like to thank the anonymous reviewers for their very careful and insightful comments on an earlier version of this paper that has led to substantial improvements. This work has been supported in part by NSF Grants CNS 08-34709, CCF 09-05584, and CNS 13-19109, the Boeing Corporation Grant C8088-557395, and AFOSR Grant FA8750-11-2-0084.

References

[1] A. Al-Nayeem, L. Sha, D.D. Cofer, S.M. Miller, Pattern-based composition and analysis of virtually synchronized real-time distributed systems, in: Proc. ICCPS'12, IEEE, 2012, pp. 65–74.
[2] A. Al-Nayeem, M. Sun, X. Qiu, L. Sha, S.P. Miller, D.D. Cofer, A formal architecture pattern for real-time distributed systems, in: Proc. RTSS'09, IEEE, 2009, pp. 161–170.
[3] M. AlTurki, J. Meseguer, PVeStA: A parallel statistical model checking and quantitative analysis tool, in: CALCO'11, volume 6859 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 386–392.
[4] J. Anderson, Introduction to flight, McGraw-Hill, 2005.
[5] K.E. Atkinson, An introduction to numerical analysis, John Wiley & Sons, 2008.
[6] B. Awerbuch, Complexity of network synchronization, J. ACM 32 (1985).
[7] K. Bae, J. Krisiloff, J. Meseguer, P.C. Ölveczky, PALS-based analysis of an airplane multirate control system in Real-Time Maude. In Proc. FTSCS'12, Electronic Proceedings in Theoretical Computer Science 105 (2012) 5–21.
[8] K. Bae, J. Meseguer, P.C. Ölveczky, Formal patterns for multi-rate distributed real-time systems, in: Proc. FACS'12, volume 7684 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 1–18.
[9] K. Bae, J. Meseguer, P.C. Ölveczky, Formal patterns for multirate distributed real-time systems, Science of Computer Programming 91, Part A (2014) 3–44.
[10] K. Bae, P.C. Ölveczky, J. Meseguer, Definition, semantics, and analysis of Multirate Synchronous AADL, in: Proc. FM'14, volume 8442 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 94–109.
[11] R. Bruni, J. Meseguer, Semantic foundations for generalized rewrite theories, Theoretical Computer Science 360 (2006) 386–414.
[12] E. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, 1999.
[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude – A High-Performance Logical Framework, volume 4350 of *Lecture Notes in Computer Science*, Springer, 2007.
[14] R.P.G. Collinson, Introduction to avionics, Chapman & Hall, 1996.
[15] M. Fadlisyah, P.C. Ölveczky, The HI-Maude tool, in: Proc. CALCO'13, volume 8089 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 322–327.
[16] M. Fadlisyah, P.C. Ölveczky, E. Ábrahám, Object-oriented formal modeling and analysis of interacting hybrid systems in HI-Maude, in: Proc. SEFM'11, volume 7041 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 415–430.
[17] M. Fadlisyah, P.C. Ölveczky, E. Ábrahám, Formal modeling and analysis of human body exposure to extreme heat in HI-Maude, in: Proc. WRLA'12, volume 7571 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 139–161.
[18] P.H. Feiler, D.P. Gluch, Model-Based Engineering with AADL, Addison-Wesley, 2012.
[19] A. Girault, C. Ménier, Automatic production of globally asynchronous locally synchronous systems, in: Proc. EMSOFT'02, volume 2491 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 266–281.
[20] H. Kopetz, G. Grünsteidl, TTP - a protocol for fault-tolerant real-time systems, IEEE Computer 27 (1994) 14–23.
[21] S.M. Loos, D. Renshaw, A. Platzer, Formal verification of distributed aircraft controllers, in: Proc. HSCC'13, ACM, 2013, pp. 125–130.
[22] N. Lynch, R. Segala, F. Vaandrager, Hybrid I/O automata, Information and Computation 185 (2003) 105–157.
[23] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theoretical Computer Science 96 (1992) 73–155.

[24] J. Meseguer, Membership algebra as a logical framework for equational specification, in: Proc. WADT'97, volume 1376 of *Lecture Notes in Computer Science*, Springer, 1998, pp. 18–61.
[25] J. Meseguer, P.C. Ölveczky, Formalization and correctness of the PALS architectural pattern for distributed real-time systems, Theoretical Computer Science 451 (2012) 1–37.
[26] J. Meseguer, R. Sharykin, Specification and analysis of distributed object-based stochastic hybrid systems, in: Proc. HSCC'06, volume 3927 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 460–475.
[27] S.P. Miller, D.D. Cofer, L. Sha, J. Meseguer, A. Al-Nayeem, Implementing logical synchrony in integrated modular avionics, in: Proc. DASC'09, IEEE, 2009.
[28] P.C. Ölveczky, J. Meseguer, Specification of real-time and hybrid systems in rewriting logic, Theoretical Computer Science 285 (2002) 359–405.
[29] P.C. Ölveczky, J. Meseguer, Semantics and pragmatics of Real-Time Maude, Higher-Order and Symbolic Computation 20 (2007) 161–196.
[30] D. Potop-Butucaru, B. Caillaud, Correct-by-construction asynchronous implementation of modular synchronous specifications, Fundamenta Informaticae 78 (2007).
[31] J. Rushby, Systematic formal verification for fault-tolerant time-triggered algorithms, IEEE Transactions on Software Engineering 25 (1999) 651–660.
[32] W. Steiner, J. Rushby, TTA and PALS: Formally verified design patterns for distributed cyber-physical systems, in: Proc. DASC'11, IEEE, 2011.
[33] G. Tel, E. Korach, S. Zaks, Synchronizing ABD networks, IEEE Transactions on Networking 2 (1994) 66–69.
[34] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, M. DiNatale, Implementing synchronous models on loosely time triggered architectures, IEEE Transactions on Computers 1 (2008).

Appendix A. The Transfer Functions in Real-Time Maude

This section explains how the functions `transferInputs` and `transferResults`, that transfer data for the synchronous composition of a multirate ensemble, are defined in Real-Time Maude. For a given ensemble component, the meaning of these functions can be described as follows:

- `transferInputs` *moves* data in the input ports of the ensemble or the feedback output ports of the subcomponents into their connected input ports at the beginning of each step. The transferred data in those source output ports are consumed.
- `transferResults` transfers data in the output ports of the subcomponents to their connected output ports of the ensemble. In this case, if an output port of a subcomponent is also connected to another subcomponent, it keeps the data for the feedback output in the next step (otherwise, it is removed from the output port to avoid generating redundant system states).

The basic idea is to model transferring data by a message passing mechanism. A message has a list of data to be delivered and a comma-separated set of its target port names. We define the two types of messages: (i) `transIn`, generated by `transferInputs`, and delivered to input ports; and (ii) `transOut`, generated by `transferResults`, and delivered to output ports:

```
ops transIn transOut : [NeList{Data}] [Set{PortName}] -> [Msg] .
```

Since messages only have a kind `[Msg]`, but no sort, no other operation can be applied until all the messages are delivered.

The following equations formalize the message passing behavior; `transIn` messages are delivered to the input ports of the subcomponents, and `transOut` messages are delivered to the output ports of the ensemble, where `KPS` and `KCS` are variables at the kind level to capture traveling messages as well as objects:

```
var PN : PortName .    var PNS : Set{PortName} .    vars KPS KCS : [Configuration] .

eq < C : Ensemble | ports : KPS transIn(NDL,PNS), machines : KCS >
 = < C : Ensemble | ports : KPS,                   machines : KCS transIn(NDL,PNS) > .

eq transIn(NDL, (C.P, PNS)) < C : Component | ports : < P : InPort | content : nil > PORTS >
 = transIn(NDL, PNS)         < C : Component | ports : < P : InPort | content : NDL > PORTS > .
eq transIn(NDL, empty) = none .

eq < C : Ensemble | ports : KPS,                    machines : transOut(NDL,PNS) KCS >
 = < C : Ensemble | ports : KPS transOut(NDL,PNS), machines : KCS > .

eq transOut(NDL, (P, PNS)) < P : OutPort | content : DL >
 = transOut(NDL, PNS)      < P : OutPort | content : DL NDL > .
eq transOut(NDL, empty) = none .
```

For efficiency reasons, instead of directly using a set of connections, we construct a connection table that defines a mapping from a port name to the set of its connected port names.

```
sort ConxTable ConxItem .
subsort ConxItem < ConxTable .
op _|->_ : PortName NeSet{PortName} -> ConxItem [ctor] .
op none : -> ConxTable [ctor] .
op __ : ConxTable ConxTable -> ConxTable [ctor comm assoc id: none] .
```

The `normalize` function returns a *normalized* connection table in which each source port name is uniquely associated with a set of its target port names.

```
var CONXS : Set{Connection} .   var CTB : ConxTable .   var NPS NPS' : NeSet{PortName} .

op normalize : ConxTable -> ConxTable .
eq normalize((PN |-> NPS) (PN |-> NPS') CTB)  =  normalize((PN |-> (NPS, NPS')) CTB) .
eq normalize(CTB) = CTB [owise] .
```

We maintain two connection tables by the *memorized* functions: `inner-tb` for environment input and feedback output connections, and `outer-tb` for environment output connections. For an operator declared with the `memo` attribute, the result of each function call is memorized and thus computed only once [13].

```
ops  inner-tb  outer-tb : Set{Connection} -> ConxTable [memo] .
```

Given a set of connections, the `inner-tb` function constructs the normalized connection table in which each port name `PN` maps to a set of the port names connected to `PN` through environment input or feedback output connections. The `outer-tb` function is similarly defined by equations.

```
eq inner-tb(CONXS) = inner-tb(CONXS, none) .

op inner-tb : Set{Connection} ConxTable -> ConxTable .
eq inner-tb((PN --> C . P) ; CONXS, CTB)  =  inner-tb(CONXS, (PN |-> C . P) CTB) .
eq inner-tb(CONXS, CTB) = normalize(CTB) [owise] .

eq outer-tb(CONXS) = outer-tb(CONXS, none) .

op outer-tb : Set{Connection} ConxTable -> ConxTable .
eq outer-tb((PN --> P) ; CONXS, CTB)
 = outer-tb(CONXS, (PN |-> P) CTB) .
eq outer-tb(CONXS, CTB) = normalize(CTB) [owise] .
```

For an ensemble component, the `transferInputs` function generates the `transIn` messages from its environment input ports and the feedback output ports of the subcomponents, by using the (memorized) connection table `inner-tb` for the ensemble component.

```
var NPS : NeSet{PortName} .  var ICTB OCTB : ConxTable .  var CONXS : Set{Connection} .

op transferInputs : Object -> Object .
eq transferInputs(< C : Ensemble | ports : PORTS,  machines : COMPS,  connections : CONXS >)
 =
    < C : Ensemble | ports : transEnvIn(PORTS, inner-tb(CONXS)),
                     machines : transFBOut(COMPS, inner-tb(CONXS)) > .
```

There are two auxiliary (partial) functions defined for `transferInputs` that are applied to the ports or the subcomponents to actually generate the desired `transIn` messages:

```
ops  transEnvIn  transFBOut : [Configuration] [ConxTable] -> [Configuration] .
```

The `transEnvIn` function creates the message `transIn(D, NPS)` from the *first* item `D` in each environment input port `P` and component table item `P |-> NPS` in the `inner-tb` table:

```
eq transEnvIn(< P : InPort | content : D DL > PORTS, (P |-> NPS) ICTB)
 = transEnvIn(< P : InPort | content : DL > PORTS, ICTB)  transIn(D, NPS) .
eq transEnvIn(PORTS, ICTB) = PORTS [owise] .
```

Similarly, the `transFBOut` function produces the message `transIn(NDL, NPS)` from data `NDL` in each input port `P` of subcomponent `C` and component table item `C . P |-> NPS` in the `inner-tb` table:

```
eq transFBOut(< C : Component | ports : < P : OutPort | content : NDL > PORTS > COMPS,
              (C . P |-> NPS) ICTB)
 = transFBOut(< C : Component | ports : < P : OutPort | content : nil > PORTS > COMPS, ICTB)
   transIn(NDL, NPS) .
eq transFBOut(COMPS, ICTB) = COMPS [owise] .
```

On the other hand, the `transferResults` function generates the `transOut` messages from the output ports of the subcomponents using the connection table `outer-tb` for the environment output connections:

```
op transferResults : Object -> Object .
eq transferResults(< C : Ensemble | machines : COMPS, connections : CONXS >)
 = < C : Ensemble | machines : transEnvOut(COMPS, outer-tb(CONXS), inner-tb(CONXS)) > .
```

The auxiliary partial function `transEnvOut` generates the message `transOut(NDL, NPS)` from data `NDL` in each output port `P` of subcomponent `C` and connection table item `C . P |-> NPS` in the `outer-tb` table. If the port `C . P` is also involved in some feedback output connections (i.e., in the `inner-tb` table), then the data `NDL` remains for the next step, and otherwise it is removed:

```
op transEnvOut : [Configuration] [ConxTable] [ConxTable] -> [Configuration] .
ceq transEnvOut(< C : Component | ports : < P : OutPort | content : NDL > PORTS > COMPS,
                (C . P |-> NPS) OCTB, ICTB)
  = transOut(NDL, NPS)
    transEnvOut(< C : Component | ports : < P : OutPort | content : DL > PORTS > COMPS,
                OCTB, ICTB)
 if DL :=  if contains?(C . P, ICTB) then NDL else nil fi .
eq transEnvOut(COMPS, OCTB, ICTB) = COMPS [owise] .
```