Definition, Semantics, and Analysis of Multirate Synchronous AADL

Kyungmin Bae[1], Peter Csaba Ölveczky[2], and José Meseguer[1]

[1] University of Illinois at Urbana-Champaign
[2] University of Oslo

Abstract. A number of cyber-physical systems are hierarchical distributed control systems whose components operate with different rates, and that should behave in a virtually synchronous way. Designing such systems is hard due to asynchrony, skews of the local clocks, and network delays; furthermore, their model checking verification is typically unfeasible due to the state space explosion caused by the interleavings. The Multirate PALS formal pattern reduces the problem of designing and model checking such virtually synchronous multirate systems to the much simpler tasks of specifying and verifying their underlying synchronous design. To make the Multirate PALS design and verification methodology available within an industrial modeling environment, we define in this paper the modeling language *Multirate Synchronous AADL*, which can be used to specify multirate synchronous designs using the AADL modeling standard. We then define the formal semantics of Multirate Synchronous AADL in Real-Time Maude, and integrate Real-Time Maude verification into the OSATE tool environment for AADL. Finally, we show how an algorithm for smoothly turning an airplane can be modeled and analyzed using Multirate Synchronous AADL.

1 Introduction

Modeling languages are widely used but tend to be weak on the formal analysis side. If they can be endowed with formal analysis capabilities "under the hood" with minimal disruption to the established modeling processes, formal methods can be more easily adopted and many design errors can be detected early in the design phase, resulting in higher quality systems and in substantial savings of effort in the development and verification processes. This work reports on a significant advance within a long-term effort to intimately connect formal methods and modeling languages for cyber-physical systems. The advance consists in supporting model checking analysis of multirate distributed cyber-physical systems within the industrial modeling standard AADL [12].

Our previous work in this area [7, 8, 15] has focused on endowing AADL with formal analysis capabilities, using Real-Time Maude [16] as an "under the hood" formal tool, by developing plugins to AADL's OSATE modeling environment.[3] Our goal is the *automated* analysis of AADL models by model checking. Such models describe systems made up of *distributed components* that communicate with each other through

[3] A similar approach has intimately connected the Ptolemy II modeling language and Real-Time Maude to model check Ptolemy II models [9].

ports in various time- or event-triggered ways. Because of combinatorial explosion, distribution can quickly make a naive model checking analysis unfeasible. This problem is caused by the *distributed* nature of many cyber-physical systems, not by AADL.

To tame this combinatorial explosion we have investigated general *formal patterns* that, by drastically reducing the state space, can support the model checking analysis of distributed cyber-physical systems. A broad class of such systems is that of distributed control systems that, while asynchronous, must be *virtually synchronous*, since they are controlled in a periodic way. The *PALS* pattern [2, 14] achieves such a drastic state space reduction by reducing the design of a distributed system of this kind to that of its much simpler synchronous counterpart.[4] However, PALS is limited by the requirement that all components have the same period, which is unrealistic in practice. Typically, components closer to sensors and actuators have a faster period than components higher up in the control hierarchy. This has led us to develop the *Multirate PALS* pattern [6] (see also [1] for related work), which generalizes PALS to the multi-rate caseand also achieves a drastic state space reduction.

The benefits of PALS and Multirate PALS for model checking distributed system designs in AADL are obvious. Since the distributed model and its synchronous counterpart are *bisimilar* [6, 14], they satisfy the same temporal logic properties. Therefore, we should model check the much simpler *synchronous* model, since model checking the asynchronous model is typically unfeasible. This requires: (i) defining appropriate extensions of AADL where such synchronous models can be specified; (ii) giving a *formal semantics* in rewriting logic to such language extensions;[5] and (iii) building tools as OSATE plugins that *automate* such a formal semantics and invoke Real-Time Maude to model check the synchronous models. Methodologically this is very useful because: (a) synchronous designs are much easier to understand by engineers; (b) they are much easier to model check; and (c) generation of their more complex distributed versions can be automated and made *correct by construction* using PALS and Multirate PALS.

For PALS, steps (i)–(iii) were taken in the Synchronous AADL language and tool [7, 8]. This paper greatly broadens the class of AADL models that can be model checked in this way by extending AADL to support the Multirate PALS methodology. This involves the following steps: (i) in Section 3 we define the *Multirate Synchronous AADL* language; (ii) in Section 4 we illustrate its modeling capabilities by defining (the synchronous version of) a distributed control system for turning an aircraft. (iii) in Section 5 we define the formal semantics of Multirate Synchronous AADL in Real-Time Maude; and (iv) in Section 6 and Section 7 we describe the Multirate Synchronous AADL tool as an OSATE plugin, illustrating its use in model checking the aircraft controller model. This is quite a complex model, showing the effectiveness of the Multirate Synchronous AADL tool to analyze systems in practice.

[4] For example, for an avionics case study in [14], the number of system states for their simplest possible distributed version with perfect local clocks and no network delays was 3,047,832, but PALS reduced the number of states to be analyzed by model checking to 185.

[5] In efforts of this kind there is a clear danger to produce an engineering artifact with no explicit clarification of the semantic assumptions, which are implicitly embedded in the tool's code. However, without a clear mathematical semantics formal analysis is *meaningless*. To avoid this danger, in all our previous work [7, 9, 8, 15] we have made formal tools for a modeling language *semantics-based* by defining a formal semantics of the language in rewriting logic.

2 Preliminaries

This section summarizes preliminary notions on Multirate PALS, synchronous designs, AADL, and Real-Time Maude.

2.1 Multirate PALS

The *Multirate PALS* pattern [6] can drastically simplify the design and verification of distributed cyber-physical systems whose architecture is one of *hierarchical distributed control*. Systems of this nature are very common in avionics, motor vehicles, robotics, and automated manufacturing. Although these systems are distributed, they must achieve virtual synchrony *in real time*, since actual deadlines must be met in physical time for physical reasons. More specifically, given a multirate synchronous design *SD* and performance bounds Γ on the clock skews, computation times, and network delays, Multirate PALS maps *SD* to the corresponding distributed real-time system $\mathcal{MA}(SD, \Gamma)$ that is stuttering bisimilar to *SD* as made precise in [6].

The topic of this paper is the specification of such multirate *designs* in AADL, and their formal analysis through the OSATE toolset. We therefore only describe the *synchronous* designs *SD*, called *multirate ensembles*, in this paper. A single component in such an ensemble is formalized as a *typed machine* that receives inputs, changes its local state, and produces output in each "iteration":

Definition 1. *A* typed machine $M = (D_i, S, D_o, \delta_M)$ *consists of:*

- D_i*, called the* input set*, a nonempty set of the form* $D_i = D_{i_1} \times \cdots \times D_{i_n}$*,*
- S*, a nonempty set, called the* set of states.
- D_o*, called the* output set*, a nonempty set of the form* $D_o = D_{o_1} \times \cdots \times D_{o_m}$*,*
- δ_M*, called the* transition relation*, a total relation* $\delta_M \subseteq (D_i \times S) \times (S \times D_o)$*.*

That is, a 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 an element of D_{o_j}.[6] We consider multirate systems in which a set of components with the same rate may communicate with each other and with a number of faster components, so that the period of the higher-level components is a multiple of the period of each fast component, as illustrated in Fig. 1.

To compose machines with different periods into a synchronous system in which *all* components operate in lock-step, we "slow down" the fast components so that all components run at the slow rate. A fast machine that is slowed down by a factor k *performs k internal transitions* during one (slow) period; since it consumes an input and produces an output at each port in each of these internal steps, it consumes and produces k-tuples of inputs and outputs in a slow step. Such a k-tuple output from a fast machine must be *transformed* into a *single* value by an *input adaptor* function $\alpha_p : D_{i_p}^k \to D_{i_p}$ so that it can be read by the slow component. Likewise, since the fast component expects a k-tuple of input values in each input port, the single-value output from a slow component must be transformed to a k-tuple of inputs to the fast machine

[6] We can regard a machine without any output (resp. input) to have a "dummy" output (resp. input) singleton set $\{*\}$.

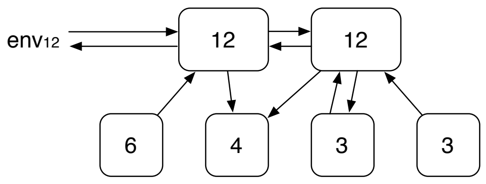
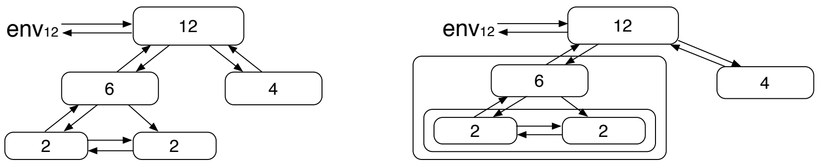


Fig. 1: A multirate system where each machine is annotated by its period.

by an input adaptor $\alpha'_q : D_{i_q} \to D^k_{i_q}$. Such an input adaptor may, for example, transform an input d to a k-tuple $(d, \perp, \ldots, \perp)$ for some "don't care" value $\perp$.

A *multirate machine ensemble* is a network of typed machines with different rates and input adaptorsthat satisfies the above constraints. Such an ensemble has a *synchronous semantics*: all machines perform a transition (possibly consisting of multiple "internal transitions") simultaneously, and the output becomes an input at the *next* (global) step. Its *synchronous composition* of an multirate ensemble defines another typed machine, which can be a component in another ensemble, giving rise to *hierarchical multirate ensembles* formalized in [6]. For example, the "system" in Fig. 2a can be seen as the hierarchical multirate ensemble in Fig. 2b. We assume that the observable behavior of an environment can be defined by a (possibly) *nondeterministic* machine, and that all other machines are deterministic, i.e., their transition relation δ is a total function. For a hierarchical multirate ensembles, an environment of a sub-ensemble, which can be considered as a nondeterministic machine, can be slower than the sub-ensemble, when the environment is located at a "higher" level of the system hierarchy.



(a) Multirate control system.

(b) Hierarchical multirate ensemble.

Fig. 2: A multirate control system and the corresponding multirate ensemble.

2.2 AADL

The *Architecture Analysis & Design Language* (AADL) [12] is an industrial modeling standard used in avionics, aerospace, automotive, medical devices, and robotics to describe an embedded real-time system as an assembly of software components mapped onto an execution platform. An AADL model describes a system of hardware and software components. Hardware components include: *processor* components that schedule and execute threads, *memory* components, *device* components, and *bus* components that interconnect processors, memory, and devices. Software components include *threads* that model the application software to be executed; *process* components defining protected memory that can be accessed by its thread and data subcomponents; and *data* components representing data types. The OSATE modeling environment provides a set of Eclipse plug-ins for AADL.

Since we want to use AADL to specify the underlying *synchronous design*, as opposed to the actual *distributed real-time system* with network delays, computation times, and so on, this paper focuses on the software component subset of AADL. In this subset, a component *type* specifies the component's *interface* (i.e., ports) and *properties*, and a component *implementation* specifies the internal structure of the component as a set of *subcomponents* and a set of *connections* linking their ports. *System* components are the top level components, and a set of *thread* components define their dynamic behaviors. Each AADL construct may have *properties* describing its parameters and other information. A user may define new domain-specific properties in *property sets*. The *dispatch protocol* of a thread determines when the thread is executed. For example, a *periodic* thread is activated at fixed time intervals, and an *aperiodic* thread is activated when it receives an event.

Thread behavior is described using the *behavior annex* [13], which models thread behaviors as a guarded transition system with local variables. The actions performed when a transition is applied may update local variables, call methods (subprograms), generate new outputs, and/or suspend the thread. Actions are built from basic actions using sequencing, conditionals, and finite loops. When a thread is activated, an enabled transition is applied; if the resulting state is not a *complete* state, another transition is applied, until a complete state is reached (or the thread suspends).

Real-Time Maude. A Real-Time Maude [16] *module* is a tuple (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational theory* [10] with Σ a signature[7] and E a set of confluent and terminating conditional *equations*, specifying the system's states as an algebraic data type;
- IR is a set of *instantaneous rewrite rules* `crl` $[l]$ `:` t `=>` t' `if` *condition* specifying the system's *instantaneous* (i.e., zero-time) transitions;[8]
- TR is a set of *tick rewrite rules* of the form `crl` $[l]$`:` `{`t`}` `=>` `{`t'`}` `in time` τ `if` *condition*, which specifies a transition with duration τ and label l from an instance of the term t to the corresponding instance of t'.

A conjunct in *condition* may be an equation $u = v$, a rewrite u `=>` v (which holds if u can be rewritten to v in zero or more steps), or a matching equation u `:=` v (which can be used to instantiate the variables in u).

The Real-Time Maude syntax is fairly intuitive (see [10]). A function symbol f is declared with the syntax `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ldots s_n$ are the sorts of its arguments, and s is its (value) *sort*. Maude supports the declaration of partial functions

[7] i.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols*.

[8] E is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo A*. Operationally, a term is reduced to its E'-normal form modulo A before any rewrite rule is applied.

using the arrow '~>' (e.g., `op` f `:` $s_1 \dots s_n$ `~>` s), so that a term containing a partial function may *not* have a sort. Equations are written with syntax `eq` $u = v$, and `ceq` $u = v$ `if` *condition* for conditional equations. The mathematical variables in such statements are declared with the keywords `var` and `vars`. We make extensive use of the fact that an equation $f(t_1, \dots, t_n) = t$ with the `owise` (for "otherwise") attribute can be applied to a term $f(\dots)$ only if no other equation with left-hand side $f(u_1, \dots, u_n)$ can be applied.[9] We refer to [10] for more details on the syntax of Real-Time Maude.

A *class* declaration `class` C `|` att_1 `:` s_1`, ... ,` att_n `:` s_n declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C is represented as a term $< O : C \mid att_1 : val_1, ..., att_n : val_n >$ where O is its *identifier*, and val_1 to val_n are the current values of the attributes att_1 to att_n. The global state has the form $\{t\}$, where t is a term of sort `Configuration` that has the structure of a *multiset* of objects and messages, with multiset union denoted by a juxtaposition operator that is declared associative and commutative, so that rewriting is *multiset rewriting* supported in Real-Time Maude. A *subclass* inherits all the attributes and rules of its superclasses.

A Real-Time Maude specification is *executable*, and the tool offers a variety of formal analysis methods. The *rewrite* command simulates *one* behavior of the system from an initial state, written with syntax

```
(trew t in time <= τ .)
```

where t is the initial state and τ is a term of sort `Time`. The *search* command uses a breadth-first strategy to analyze all possible behaviors of the system from an initial state, by checking whether a state matching a *pattern* and satisfying a *condition* can be reached from the initial state, written with syntax:

```
(utsearch [n] t =>* pattern such that condition .)
```

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

```
ceq {statePattern} |= prop = b if condition
```

for b a term of sort `Bool`, which defines the state proposition *prop* to hold in all states $\{t\}$ such that $\{t\}$ `|=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), `U` ("until"), and `O` ("next"). Given an initial state t, the following model checking commands checks whether the formula φ holds in all behaviors (up to duration τ):

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

[9] A specification with `owise` can be transformed to an equivalent system without them [10].

3 Multirate Synchronous AADL

This section introduces the *Multirate Synchronous AADL* language for specifying hierarchical multirate ensembles in AADL. Multirate Synchronous AADL is a subset of AADL extended with a *property set* `MR_SynchAADL`. Our goals when designing Multirate Synchronous AADL were: (i) keeping the new property set small, and (ii) letting the AADL constructs in the subset have the same meaning in AADL and Multirate Synchronous AADL as much as possible.

3.1 Subset of AADL

Since Multirate Synchronous AADL is intended to model synchronous *designs*, it ignores the hardware and scheduling featuresof AADL, and focuses on the behavioral and structural subset: hierarchical system, process, and thread components; ports and connections; and thread behaviors defined in the *behavior annex* standard.

Dispatch. The dispatch protocol is used to trigger an execution of a thread. An aperiodic, sporadic, timed, or hybrid thread is dispatched when it receives an *event*. Such *event-triggered* dispatch, where the execution of one thread triggers the execution of another thread, is not suitable to define a system in which all threads must execute in lock-step. Therefore, each thread must have *periodic* dispatch. This means that, *in the absence of immediate connections*, the thread is dispatched at the beginning of each period of the thread. In AADL, they are declared by the component properties:

```
Dispatch_Protocol => Periodic;
Period => time;
```

Ports. There are three kinds of ports in AADL: *data* ports, *event* ports, and *event data* ports. Event and event data ports can be used to dispatch event-triggered threads, but can also be used with periodic dispatch in version 2 of AADL. The main difference between event (or event data) ports and data ports is that an event (or event data) port may contain a buffer of untreated received events, whereas a data port always contains (at most) one element. To emphasize that in multirate ensembles, each component gets only one piece of data in each input port, Multirate Synchronous AADL only allows *data* ports. (The user should only specify single machines and the input adaptors; the Real-Time Maude execution environment will deal with the *k*-tuples of inputs/outputs.)

Connections. We must make sure that all outputs generated in one iteration is available at the beginning of the next iteration, and not before. As explained in [7] for (single-rate) Synchronous AADL, this is achieved in AADL by having *delayed* connections, declared by the connection property:

```
Timing => Delayed
```

3.2 New Features

The new features in Multirate Synchronous AADL are given in the following property set `MR_SynchAADL`:

```
property set MR_SynchAADL is
  Synchronous: inherit aadlboolean
               applies to (system, process, thread group, thread);
  Nondeterministic: aadlboolean applies to (thread);
  InputAdaptor: aadlstring applies to (port);
end MR_SynchAADL;
```

The main system component should declare the property `Synchronous` to be `true`, to state that it should be executed synchronously:

```
MR_SynchAADL::Synchronous  => true;
```

As mentioned in Section 2, we assume that the observable behavior of an environment can be defined by a *nondeterministic* machine, and that all other threads are *deterministic*. A nondeterministic environment component should add the property:

```
MR_SynchAADL::Nondeterministic  => true;
```

The most important new feature to define a multirate ensemble is *input adaptors*. Multirate Synchronous AADL provides a number of *predefined input adaptors*. The predefined 1-to-k input adaptors, mapping a single value to a k-vector of values, are:

`"repeat_input"`	(maps v to $(v, v, \ldots, v)$)
`"use in first iteration"`	(maps v to $(v, \perp, \ldots, \perp)$)
`"use in last iteration"`	(maps v to $(\perp, \ldots, \perp, v)$)
`"use in iteration` i`"`	(maps v to $(\underbrace{\perp, \ldots, \perp}_{i-1}, v, \perp, \ldots, \perp)$)

and the predefined k-to-1 input adaptors, mapping a k-vector to a single value, are:

`"first"`	(maps $(v_1, \ldots, v_k)$ to v_1)
`"last"`	(maps $(v_1, \ldots, v_k)$ to v_k)
`"use element` i`"`	(maps $(v_1, \ldots, v_k)$ to v_i)
`"average"`	(maps $(v_1, \ldots, v_k)$ to $(v_1 + \cdots + v_k)/k$)
`"max"`	(maps $(v_1, \ldots, v_k)$ to $\max(v_1, \ldots, v_k)$)
`"min"`	(maps $(v_1, \ldots, v_k)$ to $\min(v_1, \ldots, v_k)$)
`"sum"`	(maps $(v_1, \ldots, v_k)$ to $v_1 + \cdots + v_k$)

where the first two adaptors are special cases of the third one, and the last four adaptors can be only applied for numerical inputs. Such an input adaptor is assigned to an input port as a property `MR_SynchAADL::InputAdaptor` => *input adaptor*, e.g.:

```
goal_angle: in data port Base_Types::Float
            {MR_SynchAADL::InputAdaptor => "use in first iteration";};
```

The `"use in ..."` 1-to-k adaptors generate some “don't care” values $\perp$. Instead of explicitly having to define such default values, the fact that a port p has a input “$\perp$” is manifested by p`'fresh` being `false`.

4 Case Study: Turning an Airplane

This section shows how the *design* of a virtually synchronous control system for turning an airplane in a desired direction can be specified in Multirate Synchronous AADL. We have also *directly* defined a Real-Time Maude model of a control algorithm (under a few simplifying assumptions) in [4], and refer to that paper for more details of the turning control algorithm and the physical properties of airplanes.

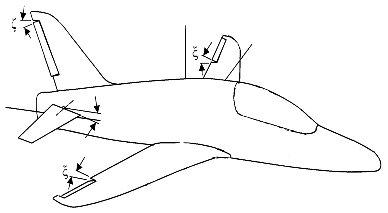


Fig. 3: The ailerons and the rudder of an aircraft.

To achieve a smooth turn of the airplane, the controller must synchronize the movements of the airplane's two *ailerons* and its *rudder* (an aileron is a flap attached to the end of the left or the right wing, and a rudder is a flap attached to the vertical tail). More precisely, a turning control algorithm must give commands to the (subcontrollers of the) ailerons and the rudder. This is a prototypical multirate distributed control system, since the subcontrollers for the ailerons and the rudder typically have different periods,[10] yet must synchronize in real time to achieve a smooth turn.

4.1 Airplane Dynamics

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 $d\psi$ can be given by $d\psi = (g/v) * \tan\phi$, where ψ is the direction of the aircraft, ϕ is the aircraft's roll angle, g is the gravity constant, and v is the velocity of the aircraft [11]. The *ailerons* are used to control the roll angle ϕ of the aircraft by generating different amounts of lift force in the left and the right wings. 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*. It makes the aircraft sideslip in a wrong direction with the amount of the yaw angleβ, as shown in Fig. 4. 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 should be increased for the desired direction while the yaw angle β stays at 0.

[10] In a commercial aircraft, the ailerons are controlled at 30-100 Hz, and the rudder is controlled at 30-50 Hz [1].

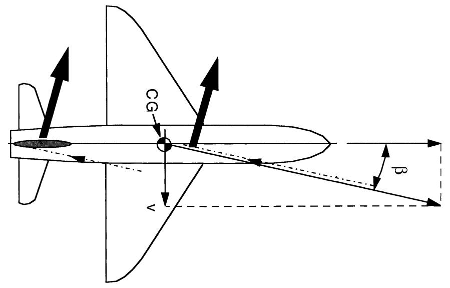


Fig. 4: Adverse yaw.

Under some simplifying assumptions [4], the roll angle ϕ and the yaw angle β can be modeled by the following differential equations [3]:

$$\mathrm{d}\phi^2 = (\mathit{Lift\ Right} - \mathit{Lift\ Left}) \,/\, (\mathit{Weight} * \mathit{Length\ of\ Wing}) \tag{1}$$

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

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

$$\mathit{Lift} = \mathit{Lift\ constant} * \mathit{Angle} \tag{3}$$

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 wing, and the drag ratio is given by the size and the shape of the entire aircraft.

4.2 System Architecture and Behavior

As shown in Fig. 5, our system consists of four periodic controllers with different periods. The *environment* is the pilot console that allows the pilot to select a new desired direction every 600 ms. Their behavior can be summarized as follows.

- The *left wing controller* receives the desired angle $goal_L$ of the left wing aileron from the main controller, moves the aileron towards that angle, and sends the current angle α_L to the main controller.
- The *right wing* (resp., the *rudder*) *controller* operates in the same way for the right wing aileron (resp., the rudder).
- The *main controller* is responsible for achieving the desired turn. It receives the desired direction from the pilot console, and receives the current angle of each device (aileron or rudder) from the device controllers. Based on these inputs, and on its estimate of the state of the airplane, the main controller computes the new desired device angles and sends them to the device controllers. The main controller also sends the current direction to the pilot. Furthermore, the main controller maintain the current state (ψ, ϕ, β) of the aircraft, where ψ is the current direction, ϕ is the roll angle, and β is the yaw angle.

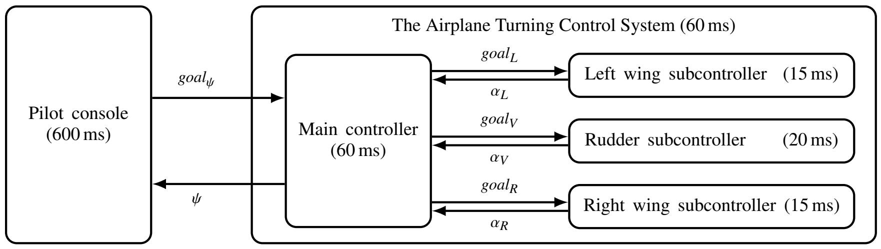

The Airplane Turning Control System (60 ms)
Pilot console (600 ms)
$goal_\psi$
ψ
Main controller (60 ms)
$goal_L$
α_L
$goal_V$
α_V
$goal_R$
α_R
Left wing subcontroller (15 ms)
Rudder subcontroller (20 ms)
Right wing subcontroller (15 ms)

Fig. 5: The architecture of our airplane turning control system.

4.3 Control Laws and Continuous Behavior

As mentioned, the job of the main controller is two-fold:

1. Compute the current direction/roll/yaw (ψ, ϕ, β) of the aircraft.
2. Decide the new desired angles $goal_L$, $goal_R$, and $goal_V$ of the left wing aileron, the right wing aileron, and the tail (vertical) wing rudder, respectively.

The new values of the yaw angle β, the roll angle ϕ, and the direction angle ψ are defined by the aeronautical differential equations above. Their moving rates $\dot{\beta}$ and $\dot{\phi}$ are approximated by 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 $\phi(t) = \phi_0 + \dot{\phi} \cdot t$. 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:

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

As for the second task, the new goal direction for the right wing aileron is defined as follows in the (improved) turning algorithm in [4]:

$$\begin{aligned}&goal_R(\phi, \psi, goal_\psi) = sign(goal_\phi - \phi) \cdot \\ &\quad (\texttt{if } |goal_\phi - \phi| > 1 \texttt{ then } \min(|goal_\phi - \phi| \cdot 0.3, 45) \texttt{ else } (goal_\phi - \phi)^2 \cdot 0.3)\end{aligned}$$

where the desired roll angle $goal_\phi$ is given by

$$\begin{aligned}&\texttt{if } |(goal_\psi - \psi) \cdot 0.32 - \phi| > 1.5 \texttt{ then } \phi + sign((goal_\psi - \psi) \cdot 0.32 - \phi) \cdot 1.5 \\ &\texttt{else } (goal_\psi - \psi) \cdot 0.32.\end{aligned}$$

The left aileron should move in exactly the opposite direction: $goal_L(\phi, \psi, goal_\psi) = -goal_R(\phi, \psi, goal_\psi)$. We refer to [4] for the goal angle of the rudder.

4.4 The Multirate Synchronous AADL Model

This section presents parts of our Multirate Synchronous AADL model of the airplane system. The system consists of the environment `pilotConsole` and the whole control system `turnCtrl`. The first two lines declares the type of the system, i.e., its interface, and the rest describes its "implementation" in terms of connections and subcomponents:

```
system Airplane   -- closed system; no ports
end Airplane;

system implementation Airplane.impl
  subcomponents
    pilotConsole: system PilotConsole.impl;  turnCtrl: system TurningController.impl;
  connections
    port pilotConsole.goal_dr -> turnCtrl.pilot_goal   {Timing => Delayed;};
    port turnCtrl.curr_dr     -> pilotConsole.curr_dr  {Timing => Delayed;};
  properties
    MR_SynchAADL::Synchronous => true;              Period => 600 ms;
    Data_Model::Initial_Value => ("0.0") applies to    -- initial feedback output
      pilotConsole.goal_dr, turnCtrl.curr_dr;
end Airplane.impl;
```

Pilot. The pilot may in any round nondeterministically *add* to the current desired direction 0° (keep the current goal direction), 10°, or −10°, where a negative angle denotes a turn to the left. This modification of the existing desired direction is sent out through the output port `goal_dr`. The input port `curr_dr` receives the current direction ψ from the turning system, which operates 10 times faster than the pilot; we must therefore use an input adaptor to map the 10-tuple of directions into a single value, for which it is natural to use the *last* value.

```
system PilotConsole                  -- "interface" of the pilot console
  features
    curr_dr: in data port Base_Types::Float {MR_SynchAADL::InputAdaptor => "last";};
    goal_dr: out data port Base_Types::Float;
end PilotConsole;

system implementation PilotConsole.impl
 subcomponents
    pilotConsoleProc: process PilotConsoleProc.impl;
 connections
   port curr_dr -> pilotConsoleProc.curr_dr; port pilotConsoleProc.goal_dr -> goal_dr;
end PilotConsole.impl;
```

A *thread* must be encapsulated by a *process* in AADL as follows, which is also encapsulated by a *system*:

```
process PilotConsoleProc
 features
    curr_dr: in data port Base_Types::Float;  goal_dr: out data port Base_Types::Float;
end PilotConsoleProc;

process implementation PilotConsoleProc.impl
 subcomponents
    pilotConsoleThread: thread PilotConsoleThread.impl;
 connections
   port curr_dr -> pilotConsoleThread.curr_dr; port pilotConsoleThread.goal_dr -> goal_dr;
end PilotConsoleProc.impl;
```

The following `PilotConsoleThread` implementation defines the behavior of the pilot. Each time the thread dispatches, the transition from state `idle` to `select` is taken. Since the latter state is not a *complete* state, the thread continues executing, by nondeterministically selecting one of the other transitions, which assigns the selected angle change to the output port `goal_dr`. Since the resulting state `idle` is a complete state, the execution of the thread in the current dispatch ends.

```
thread PilotConsoleThread
  features
    curr_dr: in data port Base_Types::Float;  goal_dr: out data port Base_Types::Float;
end PilotConsoleThread;

thread implementation PilotConsoleThread.impl
  properties
    MR_SynchAADL::Nondeterministic => true;   Dispatch_Protocol => Periodic;
  annex behavior_specification {**
    states
      idle: initial complete state;                  select: state;
    transitions
      idle -[on dispatch]-> select;             select -[ ]-> idle {goal_dr := 0.0};
      select -[ ]-> idle {goal_dr := 10.0};   select -[ ]-> idle {goal_dr := -10.0};
  **};
end PilotConsoleThread.impl;
```

Turning Controller. The *turning controller* consists of the main controller, which computes the desired device angles, and the three subcontrollers. We could of course have specified the three different device controllers in three different specifications. However, since their behavior is the same (move a flap $x°$ towards the received desired angle in each period, and report back the current flap angle), we instead choose to specify the device controllers in the same way, as instances of `Subcontroller.impl`. Since the periods of the device controllers are different, and since the rudder can move at most 0.5° in each 20 ms period, whereas the ailerons can move 1° in each 15 ms period, we must define these values in the turning controller.

The desired *change* in the direction is received from the pilot console in the input port `pilot_goal`. Since the turning controller is 10 times faster than the pilot console, it will execute 10 "internal" iterations in a global period; hence the single input in `pilot_goal` from the pilot must be mapped into 10 values, and we choose to use the input in the first local iteration:

```
system TurningController             -- "interface" of the turning controller
  features
    pilot_goal: in data port Base_Types::Float
                {MR_SynchAADL::InputAdaptor => "use in first iteration";};
    curr_dr: out data port Base_Types::Float;
end TurningController;

system implementation TurningController.impl
 subcomponents
   mainCtrl: system Maincontroller.impl;    rudderCtrl: system Subcontroller.impl;
   leftCtrl: system Subcontroller.impl;     rightCtrl: system Subcontroller.impl;
 connections
   port leftCtrl.curr_angle   -> mainCtrl.left_angle    {Timing => Delayed;};
   port rightCtrl.curr_angle  -> mainCtrl.right_angle   {Timing => Delayed;};
   port rudderCtrl.curr_angle -> mainCtrl.rudder_angle  {Timing => Delayed;};
   port mainCtrl.left_goal    -> leftCtrl.goal_angle    {Timing => Delayed;};
   port mainCtrl.right_goal   -> rightCtrl.goal_angle   {Timing => Delayed;};
   port mainCtrl.rudder_goal  -> rudderCtrl.goal_angle  {Timing => Delayed;};
   port pilot_goal -> mainCtrl.goal_angle;   port mainCtrl.curr_dr -> curr_dr;
 properties
   Period => 60 ms;
   Period => 15 ms applies to leftCtrl, rightCtrl;
   Period => 20 ms applies to rudderCtrl;
   Data_Model::Initial_Value => ("1.0") applies to    -- ailerons can move 1° in 15ms
     leftCtrl.ctrlProc.ctrlThread.diffAngle, rightCtrl.ctrlProc.ctrlThread.diffAngle;
   Data_Model::Initial_Value => ("0.5") applies to    -- rudder can move 0.5° in 20ms
     rudderCtrl.ctrlProc.ctrlThread.diffAngle;
   Data_Model::Initial_Value => ("0.0") applies to    -- initial feedback output
     leftCtrl.curr_angle,  rightCtrl.curr_angle, rudderCtrl.curr_angle,
     mainCtrl.left_goal, mainCtrl.right_goal, mainCtrl.rudder_goal;
end TurningController.impl;
```

Device Subcontrollers. The behavior of the subcontrollers is straightforward: move the device toward the goal angle up to `diffAngle` (denoting how much the flap can be moved in single period of the device, and declared in `TurningController.impl` above), update the goal angle if a new value has received in the input port `goal_angle`, and report back the current angle through the output port `curr_angle`. Since the main controller is slower than the device controller, the single input in `goal_angle` received from the main controller must be adapted to a *k*-tuple; in this case, we use the input in the first of the *k* internal iterations:

```
system Subcontroller                -- "interface" of a device controller
  features
    goal_angle: in data port Base_Types::Float
                {MR_SynchAADL::InputAdaptor => "use in first iteration";};
    curr_angle: out data port Base_Types::Float;
end Subcontroller;

system implementation Subcontroller.impl
  subcomponents
    ctrlProc: process SubcontrollerProc.impl;
  connections
    port goal_angle -> ctrlProc.goal_angle;
    port ctrlProc.curr_angle -> curr_angle;
end Subcontroller.impl;


-- a thread should be encapsulated by a process in AADL.
process SubcontrollerProc
  features
    goal_angle: in data port Base_Types::Float;
    curr_angle: out data port Base_Types::Float;
end SubcontrollerProc;

process implementation SubcontrollerProc.impl
  subcomponents
    ctrlThread: thread SubcontrollerThread.impl;
  connections
    port goal_angle -> ctrlThread.goal_angle;
    port ctrlThread.curr_angle -> curr_angle;
end SubcontrollerProc.impl;


thread SubcontrollerThread
  features
    goal_angle: in data port Base_Types::Float;
    curr_angle: out data port Base_Types::Float;
  properties
    Dispatch_Protocol => Periodic;
end SubcontrollerThread;
```

```
thread implementation SubcontrollerThread.impl
  subcomponents
    currAngle : data Base_Types::Float {Data_Model::Initial_Value => ("0.0");};
    goalAngle : data Base_Types::Float {Data_Model::Initial_Value => ("0.0");};
    diffAngle : data Base_Types::Float;
  annex behavior_specification {**
    states
      init: initial complete state;       move, update: state;
    transitions
      init -[on dispatch]-> move;
      move -[abs(goalAngle - currAngle) > diffAngle]-> update {
       if (goalAngle - currAngle >= 0) currAngle := currAngle + diffAngle
       else currAngle := currAngle - diffAngle end if };
      move -[otherwise]-> update {currAngle := goal_angle};
      update -[ ]-> init {
       if (goal_angle'fresh) goalAngle := goal_angle end if; curr_angle := currAngle};
  **};
end SubcontrollerThread.impl;
```

The subcontroller thread has three state variables to keep the current status of the controller: `currAngle`, `goalAngle`, and `diffAngle`. In the transition system, the state `init` is the only complete state; therefore, in each dispatch, the transition from `init` to `move` is taken, followed by one of the transitions from `move` to `update` that moves the flap up to the maximum difference `diffAngle`, which checks whether the remaining movement can be done in one step or not, followed by the transition from `update` to `init` that sets the output port `curr_angle` to the value of the state variable `currAngle` and updates `goalAngle` if the received goal angle is not $\bot$. Notice that if the system is *not* in the first internal transition, then the value of the input is set to "$\bot$" by the input adaptor, which is reflected in the test `goal_angle'fresh`.

Main Controller. The *main controller* is responsible for deciding the desired angles of the devices, and also for updating the current state of the aircraft. The main controller must adapt the tuples (denoting the current flap angle) received from the device controllers; the controller naturally chooses the *last* value in these tuples, which denote the most recent angle of the flap. The input from the pilot has already been adapted in the turning control system, which has the same period as the main controller. Therefore, no adaptation is needed for the port `goal_angle`.

```
system Maincontroller
 features
    goal_angle: in data port Base_Types::Float;
    left_angle: in data port Base_Types::Float {MR_SynchAADL::InputAdaptor => "last";};
    right_angle: in data port Base_Types::Float {MR_SynchAADL::InputAdaptor => "last";};
    rudder_angle: in data port Base_Types::Float {MR_SynchAADL::InputAdaptor => "last";};
    curr_dr: out data port Base_Types::Float;
    left_goal: out data port Base_Types::Float;
    right_goal: out data port Base_Types::Float;
    rudder_goal: out data port Base_Types::Float;
end Maincontroller;
```

We do not show the corresponding system implementation, process and process implementation, and thread type,[11] and show parts of the implementation of the thread (the entire specification can be found in Appendix A):

[11] The ports in these modules have the same names and types as in the `Maincontroller` *system*.

```
thread implementation MaincontrollerThread.impl
 subcomponents
    currDir : data Base_Types::Float {Data_Model::Initial_Value => ("0.0");};
    currRol : data Base_Types::Float {Data_Model::Initial_Value => ("0.0");};
    currYaw : data Base_Types::Float {Data_Model::Initial_Value => ("0.0");};
    goalDir : data Base_Types::Float {Data_Model::Initial_Value => ("0.0");};
```

The above data components ("state variables") denote the current state (ψ, ϕ, β) of the aircraft, and the speed of the plane. We continue with the transitions that computes the new values of: (i) the state variables `currDir`, `currRol`, and `currYaw`; and (ii) the output ports `curr_dr`, `left_goal`, `right_goal`, and `rudder_goal`:

```
 annex behavior_specification {**
   variables
      d, x, y, z, w : Base_Types::Float;
   states
      init : initial complete state;
      yaw, rollNdir, goal, aileron, rudder, output : state;
   transitions
      init -[on dispatch]-> yaw;
      yaw -[ ]-> rollNdir  { ... };   -- computes current yaw
      rollNdir -[ ]-> goal { ... };   -- computes current roll and direction

      goal -[ ]-> aileron  {          -- updates desired direction
         if (goal_angle'fresh)
           goalDir := goalDir + goal_angle
      end if };

      aileron -[ ]-> rudder {         -- computes desired aileron angles
        MathLib::angle!(goalDir - currDir, x);
        y := x * 0.32 - currRol;
        if (abs(y) > 1.5)
          if (y >= 0) d := currRol + 1.5 else d := currRol - 1.5 end if
        else
          d := x * 0.32
        end if;   -- d = goalRoll
        MathLib::angle!(d - currRol, y);
        if (abs(y) > 1.0)
          MathLib::min!(abs(y) * 0.3, 45.0, x);
          if (y >= 0) w := x else w := -x end if
        else
          w := z * y * y * 0.3
        end if;
        right_goal := w;     -- set goal angle of right aileron
        left_goal := -w      -- the left aileron should move in the opposite direction
      };

      rudder -[ ]-> output { ... };   -- compute desired  rudder angle
      output -[ ]-> init {curr_dr := currDir};
  **};
end MaincontrollerThread.impl;
```

Finally, the transitions not shown use standard functions on the floating-point numbers, such as the trigonometric functions, the square root function, and so on. We have defined a library `MathLib` of such functions, so that they can be seen as AADL *subprograms*, and called as such (as is done in the calls to `min` above). While these functions are *declared* as AADL subprograms, they are currently *implemented* in Maude.

5 Real-Time Maude Semantics

This section summarizes the Real-Time Maude semantics of Multirate Synchronous AADLwhich is also employed to formally verify Multirate Synchronous AADL models in our MR-SynchAADL tool in the following section. The entire semantics can be found in Appendix B. Our semantics is very different from the semantics of *single-rate* Synchronous AADL [7], which could consider a *flattened* structure of (single-rate) components, in order to explicitly deal with the *hierarchical* structure of components with different ratesand version 2 of the AADL standard.

5.1 Real-Time Maude Representations

The Real-Time Maude semantics is defined in an object-oriented style, where a Multirate Synchronous AADL component instance is represented as a (hierarchical) object instance of a subclass of the base class `Component`:

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

The attribute `features` denotes the ports of a component, represented as a multiset of `Port` objects; `subcomponents` denotes its subcomponents as a multiset of `Component` objects; `properties` denotes its *properties*; and `connections` denotes its connections. Notice that the hierarchical structure of an AADL component is indicated by the fact that the attribute `subcomponents` contains its subcomponents as a multiset of objects.

A component whose behavior is completely determined by its subcomponents, such as a *system* or a *process*, is represented as an object instance of a subclass of `Ensemble`:

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

The `Thread` class contains the attributes for the thread's behavior:

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

The attribute `variables` denotes the local *temporary* variables of the thread component, `transitions` denotes its behavior transitions, `currState` denotes the current state of the transition system, and `completeStates` denotes its *complete* states. Such a transition system is represented as a semi-colon-separated multiset of transitions, each of which has the form s `-[`*guard*`]->` s' `{`*actions*`}` with s a source state, s' a destination state, *guard* a boolean condition, and `{`*actions*`}` an action block.

The *data* subcomponents of a thread can specify the thread's local *state* variables, whose `value` attribute denotes its current value v, expressed as the term `[`v`]`, where `bot` denotes the "don't care" value $\bot$:

```
class Data | value : DataContent .      subclass Data < Component .
sorts DataContent Value .               subsort Value < DataContent .
op bot : -> DataContent [ctor] .        op [_] : Bool -> Value [ctor] .
op [_] : Int -> Value [ctor] .          op [_] : Float -> Value [ctor] .
```

A *data port* is represented as an object instance of a subclass of the class `Port`, whose `content` attribute contains a *list* of data contents (either a value or $\bot$) and `properties` denotes its properties, such as an input adaptor. The subclasses `InPort` and `OutPort` denote input and output data ports, respectively. An input data port also contains the attribute `cache` to keep the previously received "value"; if an input port `p` received $\bot$ in the latest dispatch, the thread can use a value in `cache`, while the behavior annex expression `p'fresh` becomes *false*:

```
class Port   | content : List{DataContent},  properties : PropertyAssociation .
class InPort | cache   : DataContent .       class OutPort .
subclass InPort OutPort < Port .
```

A *connection set* of a component is a semi-colon-separated set, each of which has the form p_i `-->` p_o, where p_i and p_o denotes the source and target ports, respectively. A connection from an output port p_1 in a subcomponent c_1 to an input port p_2 in c_2 is represented as a term c_1`..`p_1 `-->` c_2`..`p_2. Similarly, a connection c`..`p `-->` p' (resp., p' `-->` c`..`p) represents a level-up (resp., level-down) connection, linking a port p in a subcomponent c with the corresponding port p' in the "current" component (the double dots `..` is used to avoid parsing problems for "feature references"):

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

sort FeatureRef .
subsort FeatureId < FeatureRef .
op _.._ : ComponentRef FeatureId -> FeatureRef .

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

For example, an instance of the `TurningController.impl` system component in our airplane controller example can be represented by an object

```
< turnCtrl : System |
    features : < pilot_goal : InPort | content : [0.0], cache : [0.0],
                             properties : InputAdaptor => {use in first iteration} >
               < curr_dr : OutPort | content : [0.0], properties : none >
    subcomponents : < mainCtrl : System | ... >   < leftCtrl : System | ... >
                    < rightCtrl : System | ... >  < rudderCtrl : System | ... >,
    connections : leftCtrl..curr_angle --> mainCtrl..left_angle ;
                  rightCtrl..curr_angle --> mainCtrl..right_angle ;
                  rudderCtrl..curr_angle --> mainCtrl..rudder_angle ;
                  mainCtrl..left_goal --> leftCtrl..goal_angle ;
                  mainCtrl..right_goal --> rightCtrl..goal_angle ;
                  mainCtrl..rudder_goal --> rudderCtrl..goal_angle ;
                  pilot_goal --> mainCtrl..goal_angle ;
                  mainCtrl..curr_dr --> curr_dr,
    properties : Period => {60} >
```

Similarly, an instance of the thread component `SubcontrollerThread.impl` would be represented by the following term, where some identifiers in behavior transitions are enclosed by `{...}` or `[...]` for parsing purposes:

```
< ctrlThread : Thread |
    features : < goal_angle : InPort | content : bot, cache : [0.0], properties : none >
               < curr_angle : OutPort | content : [10.0], properties : none >
    subcomponents : < currAngle : Data | value : [10.0], ... >
                    < goalAngle : Data | value : [0.0], ... >
                    < diffAngle : Data | value : [1.0], ... >,
    connections : none,    properties : Period => {15},
    variables : empty,     currState : move,     completeStates : init,
    transitions : init -[on dispatch]-> move { skip } ;
                  move -[abs([goalAngle] - [currAngle]) > [diffAngle]]-> update {
                    if (([goalAngle] - [currAngle]) >= [0])
                      {currAngle} := [currAngle] + [diffAngle]
                    else
                      {currAngle} := [currAngle] - [diffAngle]
                    end if
                  } ;
                  move -[ otherwise ]-> update { {currAngle} := [goalAngle] } ;
                  update -[ [true] ]-> init {
                    {curr_angle} := [currAngle] ;
                    if (fresh(goal_angle)) {goalAngle} := [goal_angle] end if } >
```

5.2 Thread Behavior

The behavior of a single AADL component is specified using the *partial function* operator `executeStep`, by means of equations (for deterministic components) or rewrite rules (for nondeterministic components).

```
op executeStep : Object ~> Object .
```

Since a term containing `executeStep` will *not* have a sort, this is used to ensure that a transition equation/rule is only applied to an object of sort `Object` in which the transitions have already been performed in all subcomponents. For example, the following rule defines the behavior of *nondeterministic* threads:[12]

```
crl [execute]:
    executeStep(
      < C : Thread | features : PORTS, subcomponents : DATA,
                     currState : L,    completeStates : LS,   transitions : TRS,
                     variables : VARS,  properties : PROPS >)
 =>   < C : Thread | features : writeFeature(FMAP',PORTS'),
                     subcomponents : DATA',  currState : L' >
if Nondeterministic => {true} in PROPS
/\ (PORTS' | FMAP) := readFeature(PORTS)
/\ execTrans(L, LS, TRS, VARS, FMAP | DATA | PROPS)  =>  L' | FMAP' | DATA' .
```

The function `readFeature` returns a map from each input port to its current value (i.e., the first value of the data content list), while removing the value from the input port and using the *cached* value if the value is ⊥. In the rewrite condition, any possible computation result of the thread's transition system —based on the temporary variables `VARS`, the port values `FMAP`, the state variable values `DATA`, and the property values `PROPS`— is nondeterministically assigned to the pattern `L' | FMAP' | DATA'`. The function `writeFeature` updates the content of each output port from the result, while adding ⊥ if no value is assigned.

[12] The semantics of *deterministic* threads is given in Appendix B; it is quite similar to the nondeterministic case, but instead of rewrite rules, their equation versions are used.

Reading Features. Given a set of port objects, the `readFeature` function "consumes" the current value of each input port, constructs a map from port identifiers to their current values, and returns the pair of the resulting set of ports and the map. It is defined by using an auxiliary function with extra arguments to carry intermediate results:

```
eq readFeature(PORTS) = readFeature(PORTS, none, empty) .
eq readFeature(none, PORTS', FMAP) = PORTS' | FMAP .
```

If the current value of an input port `P` is a value `V`, then the port `P` is related to the pair `V : true` in the resulting map `FMAP`, which also indicates that the `fresh` flag of `P` is true, while the `cache` argument of the input port `P` is also updated to the value `V`:

```
eq readFeature(< P : InPort | content : V DCL > PORTS, PORTS', FMAP)
 = readFeature(PORTS, < P : InPort | content : DCL, cache : V > PORTS',
               insert(P, V : true, FMAP)) .
```

On the other hand, if the current value of `P` is `bot` (i.e., no "actual" value has been received in the latest dispatch), the `P` is related to the pair `V : false`, where `V` is the previously received value in `cache`:

```
eq readFeature(< P : InPort | content : bot DCL, cache : V > PORTS, PORTS', FMAP)
 = readFeature(PORTS, < P : InPort | content : DCL > PORTS',
               insert(P, V : false, FMAP)) .
```

Finally, each output port `P` is related to the "don't care" value `bot`, since behavior transitions cannot read a value from such an output port `P`:

```
eq readFeature(< P : OutPort | > PORTS, PORTS', FMAP)
 = readFeature(PORTS, < P : OutPort | > PORTS', insert(P, bot, FMAP)) .
```

Executing Transitions. The meaning of `execTrans` is defined by the following rewrite rule, which repeatedly applies transitions until a *complete* state is reached:

```
crl [trans]: execTrans(L, LS, TRS, VARS, FMAP | DATA | PROPS)
          => if (L' in LS) then  L' | FMAP' | DATA'
             else  execTrans(L', LS, TRS, VARS, FMAP' | DATA' | PROPS) fi
 if (L -[GUARD]-> L' ACTION) ; TRS' := enabledTrans(L, TRS, FMAP | DATA | PROPS)
 /\ FMAP' | DATA' | PROPS := execAction(ACTION, VARS, FMAP | DATA | PROPS) .
```

The function `enabledTrans` finds all *enabled* transitions from the current state `L` whose `GUARD` evaluates to *true*. Since the multiset union operator `_;_` is declared to be associative and commutative, *any* of these transitions is nondeterministically assigned to the pattern `(L -[GUARD]-> L' ACTION)`in the matching condition. The function `execAction` executes the actions of the chosen transition and returns a new configuration. If the next state `L'` is *not* a *complete* state (`else` branch), then `execTrans` is applied again with the new configuration.

The function `enabledTrans` is defined by the following equations. Any transition guarded by `on dispatch` is enabled (the second equation). A transition guarded by a boolean expression `E` is enabled only if `E` is evaluated to `true` (the third equation). If there are no enabled transitions from the current state `L` guarded by the above cases, then all transitions from `L` guarded by `otherwise` are enabled (the fourth equation).

```
eq enabledTrans(L, TRS, FMAP | DATA | PROPS)
 = enabledTrans(L, TRS, FMAP | DATA | PROPS, empty) .
eq enabledTrans(L, (L -[on dispatch]-> L' ACTION) ; TRS, FMAP | DATA | PROPS, TRS')
 = enabledTrans(L, TRS, FMAP | DATA | PROPS, TRS' ; (L -[on dispatch]-> L' ACTION)) .
eq enabledTrans(L, (L -[E]-> L' ACTION) ; TRS, FMAP | DATA | PROPS, TRS')
 = if eval(E, empty | FMAP | DATA | PROPS) == [true]
   then enabledTrans(L, TRS, FMAP | DATA | PROPS, TRS' ; (L -[E]-> L' ACTION))
   else enabledTrans(L, TRS, FMAP | DATA | PROPS, TRS') fi .
eq enabledTrans(L, TRS, FMAP | DATA | PROPS, TRS')
 = if TRS' == empty then owiseTransitions(L, TRS, empty) else TRS' fi [owise] .

eq owiseTransitions(L, (L -[otherwise]-> L' ACTION) ; TRS, ETRS)
 = owiseTransitions(L, TRS, ETRS ; (L -[otherwise]-> L' ACTION)) .
eq owiseTransitions(L, TRS, ETRS) = ETRS [owise] .
```

Notice that an equation with the `owise` attribute can be applied to a term only if no other equations of the same kind can be applied to the term.

Whenever a transition executes its action `ACTION`, since `VARS` is a set of *temporary* variables, it uses the *default* valuation in which each variable `VI` in `VARS` is mapped to `bot` as follows (the actual semantics of `ACTION` is given below):

```
eq execAction(ACTION, VARS, FMAP | DATA | PROPS)
 = execAction(ACTION, defaultValuation(VARS) | FMAP | DATA | PROPS)
eq defaultValuation(VI ; VARS) = (VI |-> bot) ; defaultValuation(VARS) .
eq defaultValuation(empty)     = empty .
```

Writing Features. The definition of `writeFeature` is straightforward; for each output port `P`, if some value `V` (other than `bot`) is written for `P` in the map `FMAP`, then `V` is added to the end of the data content of `P`, and otherwise, `bot` is added:

```
eq writeFeature(FMAP, PORTS) = writeFeature(FMAP, PORTS, none) .
eq writeFeature(FMAP, < P : OutPort | content : DCL > PORTS, PORTS')
 = if $hasMapping(FMAP,P) and FMAP[P] :: Value
   then writeFeature(FMAP, PORTS, < P : OutPort | content : DCL FMAP[P] > PORTS')
   else writeFeature(FMAP, PORTS, < P : OutPort | content : DCL bot > PORTS') fi .
eq writeFeature(FMAP, PORTS, PORTS') = PORTS PORTS' [owise] .
```

Evaluating Behavior Expressions. A behavior expression *E* is evaluated to a value by the function `eval`, based on the configuration of the temporary variable values `VAL`, the port values `FMAP`, the state variable values `DATA`, and the property values `PROPS`. The following equations defines the basic cases: a value `V`, a temporary variable `VI`, a state variable `C`, a port identifier `P`, a property name `PR`, and a fresh expression:

```
eq eval(V, VAL | FMAP | DATA | PROPS) = V .
eq eval([VI], (VI |-> V) ; VAL | FMAP | DATA | PROPS) = V .
eq eval([C], VAL | FMAP | < C : Data | value : V > DATA | PROPS) = V .
eq eval([P], VAL | (P |-> (V : B), FMAP) | DATA | PROPS) = V .
eq eval([PR], VAL | FMAP | DATA | (PR => PV) ; PROPS) = value(PV) .
eq eval(fresh(P), VAL | (P |-> (V : B), FMAP) | DATA | PROPS) = [B] .
```

The cases for the other expressions are defined by propagating `eval` to their subexpressions; for example, the semantics of an addition expression is defined by the equation:

```
eq eval(E1 + E2, VAL | FMAP | DATA | PROPS)
 = eval(E1, VAL | FMAP | DATA | PROPS) + eval(E2, VAL | FMAP | DATA | PROPS) .
```

Executing Behavior Actions. The function `execAction` executes a behavior action `ACTION` based on the current configuration `VAL | FMAP | DATA | PROPS`, and returns a new configuration. For example, an assignment action *id* `:=` *exp* assigns the evaluated value of *exp* to the identifier *id*, and the meaning is defined by the following equations:

```
ceq execAction({VI} := E, (VI |-> DC) ; VAL | FMAP | DATA | PROPS)
  = (VI |-> V) ; VAL | FMAP | DATA | PROPS .
 if V := eval(E, (VI |-> DC) ; VAL | FMAP | DATA | PROPS) .

ceq execAction({P} := E, VAL | (P |-> DC, FMAP) | DATA | PROPS)
 = VAL | (P |-> V, FMAP) | DATA |  PROPS .
 if V := eval(E, VAL | (P |-> DC, FMAP) | DATA | PROPS) .

ceq execAction({C} := E, VAL | FMAP | < C : Data | value : DC > DATA | PROPS)
  = VAL | FMAP | < C : Data | value : V > DATA | PROPS .
 if V := eval(E, VAL | FMAP | < C : Data | value : DC > DATA | PROPS)
```

For a sequence of actions $\{Action_1\ ;\ \cdots\ ;\ Action_n\}$, an action $Action_k$, $1 \leq k \leq n$, in the sequence is executed based on the execution results of the previous actions:

```
eq execAction({ACTION ; SEQ}, VAL | FMAP | DATA | PROPS)
 = execAction({SEQ}, execAction(A, VAL | FMAP | DATA | PROPS)) .
eq execAction({ACTION}, VAL | FMAP | DATA | PROPS)    --- single action block
 = execAction(ACTION, VAL | FMAP | DATA | PROPS) .
```

5.3 Ensemble Behavior

For *ensemble* components such as processes and systems, their synchronous behavior is also defined by using `executeStep`:

```
crl [execute]: executeStep(< C : Ensemble | >)  =>  transferResults(OBJ')
 if OBJ := applyAdaptors( transferInputs(< C : Ensemble | >) )
 /\ prepareExec(OBJ) => OBJ' .
```

This rule explicitly specifies the *multirate* synchronous composition of its all subcomponents. First, each input port of the subcomponents receives a value from its source, either an input port of `C` or an output port of another subcomponent (`transferInputs`). Second, appropriate input adaptors are applied to each input port (`applyAdaptors`), and the resulting term is assigned to the variable `OBJ`. Third, for each subcomponent, `executeStep` is applied multiple times according to its period (`prepareExec`).[13] Next, any term of sort `Object` resulting from rewriting `prepareExec(OBJ)` in zero or more steps is nondeterministically assigned to `OBJ'` of sort `Object`. Since `executeStep` does *not* yield terms of this sort, `OBJ'` will only capture an object where `executeStep` has been completely evaluated in each subcomponent. Finally, the new outputs of the subcomponents are transferred to the output ports of `C` (`transferResults`).

[13] Notice that for *deterministic* subcomponent of `C` the operator `executeStep` can be executed at this point, since its behavior is declared as equations.

Applying Input Adaptors. Given an ensemble C, for each input port P of its subcomponents, if an input adaptor is declared, the function applyAdaptors applies such an input adaptor to the input port P. This function is declared by using several auxiliary functions. First, it calls the auxiliary function applyAdaptorsAux(GT, COMPS), where GT is the period of C and COMPS is the subcomponents of C:

```
eq applyAdaptors(
    < C : Ensemble | subcomponents : COMPS, properties : (Period => GT) ; PROPS >)
  =
    < C : Ensemble | subcomponents : applyAdaptorsAux(GT, COMPS) > .
```

Next, for each subcomponent C' with period T and ports PORTS, it calls another auxiliary function applyAdaptorsAux(GT quo T, PORTS, none), where quo is the integer quotient function (i.e., GT quo T is the "rate" of the component C' in the ensemble C):

```
eq applyAdaptorsAux(GT,
     < C' : Component | features : PORTS,
                        properties : (Period => T) ; PROPS > REST)
 =
     < C' : Component | features : applyAdaptorsAux(GT quo T, PORTS, none) >
     applyAdaptorsAux(GT, REST) .

eq applyAdaptors(GT, none) = none .
```

Then, for each input port P that defines an input adaptor IA as its property, the input adaptor IA is applied to the data content list DL of P:

```
eq applyAdaptorsAux(N,
     < P : InPort | content : DL,
                    properties : (MRSynchAADL::InputAdaptor => IA); PROPS > PORTS,
     PORTS')
 =
   applyAdaptorsAux(N, PORTS,
     PORTS' < P : InPort | content : adaptor(IA, DL, N) >)  .

eq applyAdaptorsAux(N, PORTS, PORTS') = PORTS PORTS' [owise] .
```

The semantics of predefined input adaptors is defined by the function adaptor, which takes the three arguments: an input adaptor identifier IA, a data content list DL, and a rate N. For example, the meaning of the 1-to-k input adaptor repeat input can be defined by the following equations:

```
eq adaptor(repeat input, D, s(N), DL) = adaptor(repeat input, D, N, DL D) .
eq adaptor(repeat input, D,    0, DL) = DL .
```

Preparing Executions. Given an ensemble C with period GT, for each subcomponent of C with period T, the function prepareExec applies the operator executeStep to C as many times as the integer quotient GT quo T. In a similar way to applyAdaptors, this function prepareExec is also declared using several auxiliary functions as follows:

```
eq prepareExec(
    < C : Ensemble | subcomponents : COMPS, properties : (Period => GT) ; PROPS >)
 =
    < C : Ensemble | subcomponents : prepareExecAux(GT, COMPS, none) > .

eq prepareExecAux(GT,
    < C : Component | properties : (Period => T) ; PROPS > COMPS, COMPS')
 =
   prepareExecAux(GT, COMPS,
                  k-executeStep(GT quo T, < C : Component | >) COMPS') .
eq prepareExecAux(GT, COMPS, COMPS') = COMPS COMPS' [owise] .

eq k-executeStep(s(N), OBJ) = executeStep(k-executeStep(N, OBJ)) .
eq k-executeStep(   0, OBJ) = KOBJ .
```

where the function `k-executeStep(N, OBJ)` applies the `executeStep` operator to the component `OBJ` as many times as `N`.

Message Passing. We model transferring data by a message passing mechanism for the `transferInputs` and `transferResults` functions. A message contains a list of data to be delivered and its target port name. We define the two types of messages:

- `transIn` messages, delivered to input ports of subcomponents; and
- `transOut` messages, delivered to output ports of an ensemble.

The following equations formalize such message passing behavior:

```
eq < C : Ensemble | features : PORTS transIn(DL,TARGET), subcomponents : COMPS >
 = < C : Ensemble | features : PORTS, subcomponents : transIn(DL,TARGET) COMPS > .

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

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

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

Transferring Inputs. Given an ensemble `C`, the `transferInputs` function *moves* data in the input ports of `C` or the feedback output ports of its subcomponents into their connected input ports. This function generates `transIn` messages by using the two auxiliary functions `transEnvIn` and `transFBOut`:

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

For each input port P of the ensemble connected to an input port of a subcomponent, transEnvIn creates a transIn message with its current data (i.e., the *first* item):

```
eq transEnvIn((P --> C' .. P') ; CONXS, < P : InPort | content : D DL > PORTS)
 = transInAux(D, P, (P --> C' .. P') ; CONXS)
   transEnvIn(remove(P, CONXS), < P : InPort | content : DL > PORTS ) .
eq transEnvIn(CONXS, PORTS) = PORTS [owise] .

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

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

The auxiliary function remove(PN,CONXS) removes any connection with source PN from CONXS, and transInAux is defined to deal with "fan-out" connections in which a single source port is connected to several target ports.

For each output port P of a subcomponent connected to an input port of another subcomponent, transFBOut produces a transIn message with its *whole* data DL :

```
ceq transFBOut((C .. P --> C' .. P') ; CONXS,
      < C : Component | features : < P : OutPort | content : DL > PORTS > COMPS)
  = transInAux(DL, C .. P, (C .. P --> C' .. P') ; CONXS)
    transFBOut(remove(C .. P, CONXS),
      < C : Component | features : < P : OutPort | content : nil > PORTS > COMPS)
 if DL =/= nil .
eq transFBOut(CONXS, COMPS) = COMPS [owise] .
```

Transferring Results. Given an ensemble C, the function transferResults transfers data in the output ports of the subcomponents to their connected output ports of C. If such an output port is also connected to another subcomponent, it keeps the data for the feedback output in the next step. Similar to the transferInputs function, this function generates transOut messages as follows:

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

ceq transEnvOut((C .. P --> P') ; CONXS,
     < C : Component | features : < P : OutPort | content : DL > PORTS > COMPS)
  = transOutAux(DL, C .. P,  (C .. P --> P') ; CONXS)
    transEnvOut(remove(C .. P, CONXS),
      < C : Component | features : < P : OutPort | content : DL' > PORTS > COMPS)
 if DL =/= nil
 /\ DL' := if feedback?(C .. P, CONXS) then DL else nil fi .
eq transEnvOut(CONXS, COMPS) = COMPS [owise] .

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

eq feedback?(C .. P, (C .. P --> C' .. P') ; CONXS) = true .
eq feedback?(C .. P, CONXS) = false [owise] .
```

5.4 Multirate Synchronous Steps

When the entire system is specified by one top-level *closed* system component with no ports, a synchronous step of the entire system is formalized by the following conditional tick rewrite rule:

```
crl [step]:
     {< C : System | properties : Period => {T} ; Synchronous => {true} ; PROPS,
                     features : none >}
  => {SYSTEM}  in time T
   if executeStep(< C : System | >) => SYSTEM .
```

In a similar way to the `execute` rule, any term of sort `Object`, in which `executeStep` is completely evaluated, resulting from rewriting `executeStep(< C : System | >)` in zero or more steps can be nondeterministically assigned to the variable `SYSTEM`.

6 Formal Analysis using the MR-SynchAADL Tool

The Real-Time Maude semantics makes it possible to formally verify Multirate Synchronous AADL models using the Real-Time Maude tool. However, it is still required to translate *both* Multirate Synchronous AADL models and their system properties into Real-Time Maude terms. To support the convenient modeling and verification of Multirate Synchronous AADL models within the OSATE tool environment, we have developed the *MR-SynchAADL* OSATE plugin that:

- checks whether a given model is a *valid* Multirate Synchronous AADL model;
- provides an intuitive language for specifying *system requirements*; and
- automatically synthesizes a Real-Time Maude model from a Multirate Synchronous AADL model and uses Real-Time Maude model checking to analyze whether the Multirate Synchronous AADL model satisfies the given requirements.

The tool is available at `http://formal.cs.illinois.edu/kbae/MR-SynchAADL`.

6.1 Requirement Specification Language

The MR-SynchAADL tool provides a requirement specification language that allows the user to define system requirements in an intuitive way, without having to understand Real-Time Maude. The requirement specification language defines several parametric atomic propositions. The proposition

full component name `@` *location*

holds in a state when the thread identified by the full component name is in state *location*. A full component name is a component path in the AADL syntax, a period-separated path of component identifiers. Similarly, the proposition

full component name `|` *boolean expression* holds in a state if *boolean expression* evaluates to *true* in the component. We can use any boolean expression in the AADL behavior annex syntax involving data components, feedback output data ports, and property values.

In MR-SynchAADL, we can easily declare *formulas* and *requirements* for Multirate Synchronous AADL models as LTL formulas, using the usual Boolean connectives and temporal logic operators `[]`, <>, `U`, etc. Such formulas and requirements are declared using the following syntax, where LTL formulas can also contain references to other "formulas" defined by `formula` statements:

formula *name*: *proposition*;
formula *name*: *LTL formula*;
requirement *name*: *LTL formula*;

In our airplane example, the following formula declaration

```
formula safeYaw:  turnCtrl.mainCtrl.ctrlProc.ctrlThread | abs(currYaw) < 1.0;
```

states that `safeYaw` holds when the current yaw angle in the main controller is less than 1°. The following *requirement* defines the safety requirement of the system: *the yaw angle should always be close to* 0°.

```
requirement safety:  [] safeYaw;
```

6.2 Real-Time Maude Semantics of Requirement Specification Language

The semantics of the requirement specification language is defined by equations in Real-Time Maude. For example, the meaning of state-lockup propositions are defined by the following equations using the auxiliary function `lookupState`:

```
eq {< C : Ensemble | subcomponents : COMPS >} |= PATH @ L
   = lookupState(COMPS, PATH, L) .

eq lookupState(< C : Ensemble | subcomponents : COMPS > REST, C . PATH, L)
 = lookupState(COMPS, PATH, L) .
eq lookupState(< C : Thread | currState : L > REST, C, L) = true .
eq lookupState(REST, PATH, L) = false [owise] .
```

Next, each formula declaration in the MR-SynchAADL tool automatically adds the corresponding equation. For example, the `safeYaw` formula for the airplane controller example generates the following Real-Time Maude declarations:

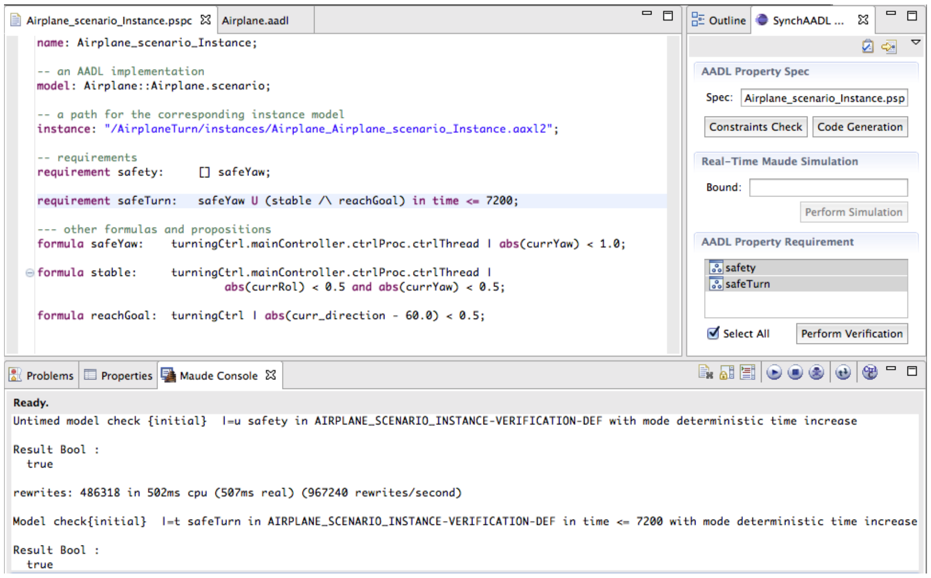
```
op safeYaw : -> Formula .
eq safeYaw
 = turningCtrl . mainController . ctrlProc . ctrlThread | abs([currYaw]) < [1.0] .
```

Finally, each requirement declaration gives the corresponding Real-Time Maude verification command. For example, the `safety` requirement generate the following command, where `initial` is reduced to the term representation of the entire model:

```
(mc {initial} |=u [] safeYaw .)
```

6.3 Tool Interface.

Figure 6 shows the MR-SynchAADL window for the airplane example. In the editor part, two system requirements, explained below, are specified using the requirement specification language. Those requirements are also listed in the "AADL Property Requirement" table. The `Constraints Check`, the `Code Generation`, and the `Perform Verification` buttons are used to perform, respectively, the syntactic validation of the model, the Real-Time Maude code generation, and the LTL model checking. The `Perform Verification` button has been clicked and the results of the model checking are shown in the "Maude Console."



```
name: Airplane_scenario_Instance;

-- an AADL implementation
model: Airplane::Airplane.scenario;

-- a path for the corresponding instance model
instance: "/AirplaneTurn/instances/Airplane_Airplane_scenario_Instance.aaxl2";

-- requirements
requirement safety:      [] safeYaw;

requirement safeTurn:   safeYaw U (stable /\ reachGoal) in time <= 7200;

--- other formulas and propositions
formula safeYaw:    turningCtrl.mainController.ctrlProc.ctrlThread | abs(currYaw) < 1.0;

formula stable:     turningCtrl.mainController.ctrlProc.ctrlThread |
                            abs(currRol) < 0.5 and abs(currYaw) < 0.5;

formula reachGoal:  turningCtrl | abs(curr_direction - 60.0) < 0.5;
```

```
Ready.
Untimed model check {initial}  |=u safety in AIRPLANE_SCENARIO_INSTANCE-VERIFICATION-DEF with mode deterministic time increase

Result Bool :
  true

rewrites: 486318 in 502ms cpu (507ms real) (967240 rewrites/second)

Model check{initial}  |=t safeTurn in AIRPLANE_SCENARIO_INSTANCE-VERIFICATION-DEF in time <= 7200 with mode deterministic time increase

Result Bool :
  true
```

Fig. 6: MR-SynchAADL window in OSATE.

7 Verifying the Airplane Turing Controller

This section shows how our Multirate Synchronous AADL model of the airplane controller can be verified using the MR-SynchAADL tool. As explained in [4], the system must satisfy the following requirement: *the airplane should reach the desired direction with a stable status within a reasonable time, while keeping the yaw angle close to* 0.

In order to verify whether the airplane can reach a *specific* goal direction or not, we first consider a *deterministic* pilot environment given by the following implementation, where the pilot gradually turns the airplane 60° to the right by adding 10° to the goal direction 6 times, instead of using the nondeterministic pilot in Section 4:

```
thread implementation PilotConsoleThread.scenario
  subcomponents
    counter: data Base_Types::Integer  {Data_Model::Initial_Value => ("0");};
  annex behavior_specification {**
    states
      idle: initial complete state;        select: state;
    transitions
      idle -[on dispatch]-> select;    select -[counter >= 6]-> idle;
      select -[counter < 6]-> idle  {goal_dr := 10.0; counter := counter + 1};
  **};
end PilotConsoleThread.scenario;
```

The desired requirement, with the additional constraint that the desired state must always be reached within 7,200 ms, can be formalized as an LTL formula using the requirement specification language in MR-SynchAADL as follows:

```
requirement safeTurn:    safeYaw U (stable /\ reachGoal) in time <= 7200;
formula stable:          turnCtrl.mainCtrl.ctrlProc.ctrlThread |
                                   abs(currRol) < 0.5 and abs(currYaw) < 0.5;
formula reachGoal:       turnCtrl | abs(curr_dr - 60.0) < 0.5;
```

where `safeYaw` holds if the yaw angle is close to 0, `stable` holds if both roll and yaw angles are close to 0, and `reachGoal` holds if the current direction is close to 60°.

Figure 6 shows the model checking results for the two system requirements `safety` ($\Box$ `safeYaw`, declared in Section 6) and `safeTurn`. In the deterministic scenario, the airplane controller satisfies both properties as displayed in the Maude console. These model checking analyses, respectively, took 1.6 and 0.5 seconds on Intel Core i5 2.4 GHz with 4 GB memory and the numbers of states explored are 59 and 13.

We have verified the `safety` requirement for the nondeterministic pilot who sends one of the turning angles $-10.0°$, $0°$, and $10°$ for each step, and have summarized the model checking in the table below, which shows a huge state space reduction compared to the asynchronous model: for the same pilot behavior and time bound 3,000 ms, the number of reachable states in the *simplest possible* distributed asynchronous model, with perfect local clocks and no network delays, was already 420,288 [6], whereas the number of reachable states is 364 in the synchronous model.

Bound (ms)	# States	Time (s)	Bound (ms)	# States	Time (s)	Bound (ms)	# States	Time (s)
≤ 3,000	364	7.3	≤ 4,200	3,280	62.1	≤ 5,400	29,524	599.8
≤ 3,600	1,093	21.0	≤ 4,800	9,841	189.1	≤ 6,000	88,573	2,323.8

8 Related Work and Conclusions

There are a number of synchronizers relating synchronous and asynchronous systems; see [14] for an overview and comparison with PALS. To the best of our knowledge, only Multirate PALS and the work in [1] propose synchronizers for multirate systems where tight time bounds must be met. The paper [1] proposes a different multirate extension of PALS, without general input adaptors; however, they do not provide a formal model of the synchronous or asynchronous systems, and—the main difference with this paper—they do not propose a language for defining synchronous models, or any way of formally analyzing the synchronous designs. We formalize Multirate PALS in [6, 5], but that work does not consider AADL. On the other hand, [7, 8] define the single-rate Synchronous AADL language and a Real-Time Maude-based analysis tool for Synchronous AADL. The current paper significantly generalizes that work to account for hierarchical *multirate* systems. In particular, in addition to needing input adaptors, one significant difference is that the single-rate case allows a very simple Real-Time Maude semantics, where we can consider a flattened system, whereas in the hierarchical multirate case we need to maintain the hierarchy, which makes the Real-Time Maude semantics quite complex. The paper [4] performs the airplane case study using (only) Real-Time Maude instead of using Multirate Synchronous AADL and our OSATE plug-in. Finally, [15] presents a "standard" (i.e., asynchronous) semantics for a subset of AADL in Real-Time Maude, but does not consider a language extension or a synchronous semantics of AADL.

In this work we have made the complexity-reducing Multirate PALS modeling and verification methodology for virtually synchronous hierarchical multirate systems available to AADL modelers by: (i) defining the Multirate Synchronous AADL language, which allows the modeler to specify his/her synchronous designs using AADL; (ii) giving a Real-Time Maude semantics for Multirate Synchronous AADL, which not only defines the language precisely but also allows formal analysis of Multirate Synchronous AADL models; (iii) providing an intuitive way of specifying temporal logic *requirements* that such models should satisfy; and (iv) integrating both modeling and automated model checking into the OSATE tool environment for AADL. We have illustrated the effectiveness of our methodology, language, and tool on a control system for smoothly turning an airplane.

Future work includes applying our language and tool on more case studies, and on automatically generating a correct-by-construction AADL model of the distributed implementation from a verified model of the synchronous design.

References

1. Al-Nayeem, A., Sha, L., Cofer, D.D., Miller, S.M.: Pattern-based composition and analysis of virtually synchronized real-time distributed systems. In: Proc. ICCPS'12. IEEE (2012)
2. Al-Nayeem, A., Sun, M., Qiu, X., Sha, L., Miller, S.P., Cofer, D.D.: A formal architecture pattern for real-time distributed systems. In: Proc. 30th IEEE Real-Time Systems Symposium. IEEE (2009)
3. Anderson, J.: Introduction to flight. McGraw-Hill (2005)
4. Bae, K., Krisiloff, J., Meseguer, J., Ölveczky, P.C.: PALS-based analysis of an airplane multirate control system in Real-Time Maude. In Proc. FTSCS'12. Electronic Proceedings in Theoretical Computer Science 105, 5–21 (2012)
5. Bae, K., Meseguer, J., Ölveczky, P.C.: Formal patterns for multi-rate distributed real-time systems. In: Proc. FACS'12. LNCS, vol. 7684. Springer (2012)
6. Bae, K., Meseguer, J., Ölveczky, P.C.: Formal patterns for multirate distributed real-time systems. Science of Computer Programming (2013), to appear. http://dx.doi.org/10.1016/j.scico.2013.09.010

7. Bae, K., Ölveczky, P.C., Al-Nayeem, A., Meseguer, J.: Synchronous AADL and its formal analysis in Real-Time Maude. In: Proc. ICFEM'11. LNCS, vol. 6991. Springer (2011)
8. Bae, K., Ölveczky, P.C., Meseguer, J., Al-Nayeem, A.: The SynchAADL2Maude tool. In: Proc. FASE'12. LNCS, vol. 7212. Springer (2012)
9. Bae, K., Ölveczky, P.C., Feng, T.H., Lee, E.A., Tripakis, S.: Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude. Science of Computer Programming 77(12), 1235–1271 (2012)
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework, LNCS, vol. 4350. Springer (2007)
11. Collinson, R.P.G.: Introduction to avionics. Chapman & Hall (1996)
12. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL. Addison-Wesley (2012)
13. França, R., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL behaviour annex - experiments and roadmap. In: Proc. ICECCS'07. IEEE (2007)
14. Meseguer, J., Ölveczky, P.C.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. Theor. Comp. Sci. 451, 1–37 (2012)
15. Ölveczky, P.C., Boronat, A., Meseguer, J.: Formal semantics and analysis of behavioral AADL models in Real-Time Maude. In: Proc. FMOODS/FORTE'10. LNCS, vol. 6117. Springer (2010)
16. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)

A The Entire AADL Specification of the Airplane Controller

```
property set MR_SynchAADL is
  Synchronous: inherit aadlboolean applies to (system, process, thread group, thread);
  Nondeterministic: aadlboolean applies to (thread);
  InputAdaptor: aadlstring applies to (port);
end MR_SynchAADL;

property set AirplaneSpec is
  gravityConstant: constant aadlreal => 9.80555;
  weight: constant aadlreal => 1000.0;
  wingSize: constant aadlreal => 2.0;
  planeSize: constant aadlreal => 4.0;
  dragRatio: constant aadlreal => 0.05;
  horzLiftConstant: constant aadlreal => 0.4;
  virtLiftConstant: constant aadlreal => 0.6;
end AirplaneSpec;

package Airplane
public
  with TurningController;
  with PilotConsole;
  with MR_SynchAADL;
  with Base_Types;
  with Data_Model;

  system Airplane
    properties
      Period => 600 ms;
      MR_SynchAADL::Synchronous => true;
  end Airplane;

  -- with the nondeterministic environment
  system implementation Airplane.impl
    subcomponents
      pilotConsole: system PilotConsole::PilotConsole.impl;
      turningCtrl: system TurningController::TurningController.impl;
    connections
      port pilotConsole.goal_dr -> turningCtrl.pilot_goal {Timing => Delayed;};
      port turningCtrl.curr_dr -> pilotConsole.curr_dr  {Timing => Delayed;};
    properties
      Data_Model::Initial_Value => ("0.0") applies to -- initial feedback output
        pilotConsole.goal_dr, turningCtrl.curr_dr;
  end Airplane.impl;

  -- with the deterministic environment
  system implementation Airplane.scenario
    subcomponents
      pilotConsole: system PilotConsole::PilotConsole.scenario;
      turningCtrl: system TurningController::TurningController.impl;
    connections
      port pilotConsole.goal_dr -> turningCtrl.pilot_goal {Timing => Delayed;};
      port turningCtrl.curr_dr -> pilotConsole.curr_dr {Timing => Delayed;};
    properties
      Data_Model::Initial_Value => ("0.0") applies to -- initial feedback output
        pilotConsole.goal_dr, turningCtrl.curr_dr;
  end Airplane.scenario;

end Airplane;
```

```
package PilotConsole
public
  with MR_SynchAADL;
  with Base_Types;
  with Data_Model;

  system PilotConsole
    features
      curr_dr: in data port Base_Types::Float {MR_SynchAADL::InputAdaptor => "last";};
      goal_dr: out data port Base_Types::Float;
  end PilotConsole;

  system implementation PilotConsole.impl
    subcomponents
      pilotConsoleProc: process PilotConsoleProc.impl;
    connections
      port curr_dr -> pilotConsoleProc.curr_dr;
      port pilotConsoleProc.goal_dr -> goal_dr;
  end PilotConsole.impl;

  system implementation PilotConsole.scenario
    subcomponents
      pilotConsoleProc: process PilotConsoleProc.scenario;
    connections
      port curr_dr -> pilotConsoleProc.curr_dr;
      port pilotConsoleProc.goal_dr -> goal_dr;
  end PilotConsole.scenario;

  process PilotConsoleProc
    features
      curr_dr: in data port Base_Types::Float;
      goal_dr: out data port Base_Types::Float;
  end PilotConsoleProc;

  process implementation PilotConsoleProc.impl
    subcomponents
      pilotConsoleThread: thread PilotConsoleThread.impl;
    connections
      port curr_dr -> pilotConsoleThread.curr_dr;
      port pilotConsoleThread.goal_dr -> goal_dr;
  end PilotConsoleProc.impl;

  process implementation PilotConsoleProc.scenario
    subcomponents
      pilotConsoleThread: thread PilotConsoleThread.scenario;
    connections
      port curr_dr -> pilotConsoleThread.curr_dr;
      port pilotConsoleThread.goal_dr -> goal_dr;
  end PilotConsoleProc.scenario;

  thread PilotConsoleThread
    features
      curr_dr: in data port Base_Types::Float;
      goal_dr: out data port Base_Types::Float;
    properties
      Dispatch_Protocol => Periodic;
  end PilotConsoleThread;

  -- nondeterministically changes the direction
  thread implementation PilotConsoleThread.impl
    properties
      MR_SynchAADL::Nondeterministic => true;
    annex behavior_specification {**
      states
        idle : initial complete state;
        select : state;
      transitions
        idle -[ on dispatch ]-> select;
        select -[ ]-> idle { goal_dr := 0.0 };
        select -[ ]-> idle { goal_dr := 10.0 };
        select -[ ]-> idle { goal_dr := -10.0 };
        **};
  end PilotConsoleThread.impl;

  -- gradually turns 60 to the right
  thread implementation PilotConsoleThread.scenario
    subcomponents
      counter : data Base_Types::Integer  {Data_Model::Initial_Value => ("0");};
    annex behavior_specification {**
      states
        idle : initial complete state;
        select : state;
      transitions
        idle -[ on dispatch ]-> select;
        select -[ counter < 6 ]-> idle { goal_dr := 10.0; counter := counter + 1 };
        select -[ counter >= 6 ]-> idle;
        **};
  end PilotConsoleThread.scenario;

end PilotConsole;

package TurningController
public
  with Maincontroller;
  with Subcontroller;
  with MR_SynchAADL;
  with Base_Types;
  with Data_Model;

  system TurningController
    features
      pilot_goal: in data port Base_Types::Float
              {MR_SynchAADL::InputAdaptor => "use in first iteration";};
      curr_dr: out data port Base_Types::Float;
  end TurningController;

  system implementation TurningController.impl
    subcomponents
      mainCtrl: system MainController::Maincontroller.impl;
      leftCtrl: system SubController::Subcontroller.impl;
      rightCtrl: system SubController::Subcontroller.impl;
      rudderCtrl: system SubController::Subcontroller.impl;
    connections
      port leftCtrl.curr_angle -> mainCtrl.left_angle   {Timing => Delayed;};
      port rightCtrl.curr_angle -> mainCtrl.right_angle  {Timing => Delayed;};
      port rudderCtrl.curr_angle -> mainCtrl.rudder_angle {Timing => Delayed;};
      port mainCtrl.left_goal -> leftCtrl.goal_angle    {Timing => Delayed;};
      port mainCtrl.right_goal -> rightCtrl.goal_angle  {Timing => Delayed;};
      port mainCtrl.rudder_goal -> rudderCtrl.goal_angle {Timing => Delayed;};
      port pilot_goal -> mainCtrl.goal_angle;
      port mainCtrl.curr_dr -> curr_dr;
    properties
      Period => 60 ms;
      Period => 15 ms applies to leftCtrl, rightCtrl;
      Period => 20 ms applies to rudderCtrl;

      Data_Model::Initial_Value => ("1.0") applies to
        leftCtrl.ctrlProc.ctrlThread.diffAngle, rightCtrl.ctrlProc.ctrlThread.diffAngle;
      Data_Model::Initial_Value => ("0.5") applies to
        rudderCtrl.ctrlProc.ctrlThread.diffAngle;
      Data_Model::Initial_Value => ("0.0") applies to  -- initial feedback output
        leftCtrl.curr_angle, rightCtrl.curr_angle, rudderCtrl.curr_angle,
        mainCtrl.left_goal, mainCtrl.right_goal, mainCtrl.rudder_goal;
    end TurningController.impl;

 end TurningController;


 package Maincontroller
 public
    with MR_SynchAADL;
    with AirplaneSpec;
    with MathLib;
    with Base_Types;
    with Data_Model;

    system Maincontroller
      features
      goal_angle: in data port Base_Types::Float;
      left_angle: in data port Base_Types::Float {MR_SynchAADL::InputAdaptor => "last";};
      right_angle: in data port Base_Types::Float {MR_SynchAADL::InputAdaptor => "last";};
      rudder_angle: in data port Base_Types::Float {MR_SynchAADL::InputAdaptor =>"last";};
      curr_dr: out data port Base_Types::Float;
      left_goal: out data port Base_Types::Float;
      right_goal: out data port Base_Types::Float;
      rudder_goal: out data port Base_Types::Float;
    end Maincontroller;

    system implementation Maincontroller.impl
      subcomponents
      ctrlProc: process MaincontrollerProc.impl;
      connections
      port goal_angle -> ctrlProc.goal_angle;
      port left_angle -> ctrlProc.left_angle;
      port right_angle -> ctrlProc.right_angle;
      port rudder_angle -> ctrlProc.rudder_angle;
      port ctrlProc.curr_dr -> curr_dr;
      port ctrlProc.left_goal -> left_goal;
      port ctrlProc.right_goal -> right_goal;
      port ctrlProc.rudder_goal -> rudder_goal;
    end Maincontroller.impl;

    process MaincontrollerProc
      features
      goal_angle: in data port Base_Types::Float;
      left_angle: in data port Base_Types::Float;
      right_angle: in data port Base_Types::Float;
      rudder_angle: in data port Base_Types::Float;
      curr_dr: out data port Base_Types::Float;
      left_goal: out data port Base_Types::Float;
      right_goal: out data port Base_Types::Float;
      rudder_goal: out data port Base_Types::Float;
    end MaincontrollerProc;

    process implementation MaincontrollerProc.impl
      subcomponents
      ctrlThread: thread MaincontrollerThread.impl;
      connections
      port goal_angle -> ctrlThread.goal_angle;
      port left_angle -> ctrlThread.left_angle;
      port right_angle -> ctrlThread.right_angle;
      port rudder_angle -> ctrlThread.rudder_angle;
      port ctrlThread.curr_dr -> curr_dr;
      port ctrlThread.left_goal -> left_goal;
      port ctrlThread.right_goal -> right_goal;
      port ctrlThread.rudder_goal -> rudder_goal;
    end MaincontrollerProc.impl;

    thread MaincontrollerThread
      features
      goal_angle: in data port Base_Types::Float;
      left_angle: in data port Base_Types::Float;
      right_angle: in data port Base_Types::Float;
      rudder_angle: in data port Base_Types::Float;
      curr_dr: out data port Base_Types::Float;
      left_goal: out data port Base_Types::Float;
      right_goal: out data port Base_Types::Float;
      rudder_goal: out data port Base_Types::Float;
      properties
      Dispatch_Protocol => Periodic;
    end MaincontrollerThread;

    thread implementation MaincontrollerThread.impl
      subcomponents
      currYaw : data Base_Types::Float {Data_Model::Initial_Value => ("0.0");};
      currRol : data Base_Types::Float {Data_Model::Initial_Value => ("0.0");};
      currDir : data Base_Types::Float {Data_Model::Initial_Value => ("0.0");};
      goalDir : data Base_Types::Float {Data_Model::Initial_Value => ("0.0");};
      annex behavior_specification {**
      variables
        d, x, y, z, w : Base_Types::Float;
      states
        init : initial complete state;
        yaw, rollNdir, goal, aileron, rudder, output : state;
      transitions
        init -[ on dispatch ]-> yaw;

        yaw -[ ]-> rollNdir {
          x := AirplaneSpec::dragRatio *
                  (right_angle * AirplaneSpec::horzLiftConstant
                   - left_angle * AirplaneSpec::horzLiftConstant) /
                          (AirplaneSpec::weight * AirplaneSpec::wingSize)
                  + rudder_angle * AirplaneSpec::virtLiftConstant /
                     (AirplaneSpec::weight * AirplaneSpec::planeSize);
          MathLib::sqrt!(abs(x), d);
          if (x < 0) d := - d end if; -- d = dBeta

          -- set the currYaw
          MathLib::angle!(currYaw + d * Period, currYaw)
        };

        rollNdir -[ ]-> goal {
          x := (right_angle * AirplaneSpec::horzLiftConstant
                  - left_angle * AirplaneSpec::horzLiftConstant) /
                          (AirplaneSpec::weight * AirplaneSpec::wingSize);
          MathLib::sqrt!(abs(x), d);
          if (x < 0) d:= - d end if; -- d = dPhi

          if (abs(d) > 0)
```

```
          MathLib::cos!(currRol * 3.1415926535897931 / 180.0, y);
          MathLib::log!(y, y);
          MathLib::cos!((d * Period + currRol) * 3.1415926535897931 / 180.0, z);
          MathLib::log!(z, z);

          -- set the currDir
          MathLib::angle!(currDir +
                AirplaneSpec::gravityConstant * (y - z) /
                (d * 3.1415926535897931 / 180.0 * 50.0), currDir)
        end if;

        -- set the currRol
        MathLib::angle!(currRol + d * Period, currRol)
      };

      goal -[ ]-> aileron {
        if (goal_angle'fresh)
                MathLib::angle!(goalDir + goal_angle, goalDir)
        end if
      };

      aileron -[ ]-> rudder {
        MathLib::angle!(goalDir - currDir, x);
        y := x * 0.32 - currRol;
        if (abs(y) > 1.5)
          if (y >= 0) d := currRol + 1.5 else d := currRol - 1.5 end if
        else
          d := x * 0.32
        end if;  -- d = goalRoll

        MathLib::angle!(d - currRol, z);
        if (abs(z) > 1.0)
          MathLib::min!(abs(z) * 0.3, 45.0, w)
        else
          w := z * z * 0.3
        end if;
        if (z < 0) w := -w end if; -- hwAngle

        -- output the left/right goal angles
        MathLib::angle!(w, right_goal);
        MathLib::angle!(- w, left_goal)
      };

      rudder -[ ]-> output {
        MathLib::angle!(- currYaw, x);
        if (abs(x) > 1.0)
          MathLib::min!(abs(x) * 0.8, 30.0, d)
        else
          d := x * x * 0.8
        end if;
        if (x < 0) d := -d end if;

        -- output the rudder goal angle
        MathLib::angle!(d, rudder_goal)
      };

      output -[ ]-> init { curr_dr := currDir };
      **};
  end MaincontrollerThread.impl;
end Maincontroller;

package Subcontroller
```

```
public
  with MR_SynchAADL;
  with Base_Types;
  with Data_Model;

  system Subcontroller
    features
      goal_angle: in data port Base_Types::Float
          {MR_SynchAADL::InputAdaptor => "use in first iteration";};
      curr_angle: out data port Base_Types::Float;
  end Subcontroller;

  system implementation Subcontroller.impl
    subcomponents
      ctrlProc: process SubcontrollerProc.impl;
    connections
      port goal_angle -> ctrlProc.goal_angle;
      port ctrlProc.curr_angle -> curr_angle;
  end Subcontroller.impl;

  process SubcontrollerProc
    features
      goal_angle: in data port Base_Types::Float;
      curr_angle: out data port Base_Types::Float;
  end SubcontrollerProc;

  process implementation SubcontrollerProc.impl
    subcomponents
      ctrlThread: thread SubcontrollerThread.impl;
    connections
      port goal_angle -> ctrlThread.goal_angle;
      port ctrlThread.curr_angle -> curr_angle;
  end SubcontrollerProc.impl;

  thread SubcontrollerThread
    features
      goal_angle: in data port Base_Types::Float;
      curr_angle: out data port Base_Types::Float;
    properties
      Dispatch_Protocol => Periodic;
  end SubcontrollerThread;

  thread implementation SubcontrollerThread.impl
    subcomponents
      currAngle : data Base_Types::Float  {Data_Model::Initial_Value => ("0.0");};
      goalAngle : data Base_Types::Float  {Data_Model::Initial_Value => ("0.0");};
      diffAngle : data Base_Types::Float;
    annex behavior_specification {**
      states
        init : initial complete state;
        move, update : state;
      transitions
        init -[ on dispatch ]-> move;

        move -[ abs(goalAngle - currAngle) > diffAngle ]-> update {
          if (goalAngle - currAngle >= 0)
            currAngle := currAngle + diffAngle
          else
            currAngle := currAngle - diffAngle
          end if
        };
        move -[ otherwise ]-> update {
```

```
        currAngle := goalAngle
      };

      update -[ ]-> init {
        curr_angle := currAngle;
        if (goal_angle'fresh)
          goalAngle := goal_angle
        end if
      };

      **};
  end SubcontrollerThread.impl;

end Subcontroller;


package MathLib
public
  with Base_Types;

  subprogram sqrt
    features
      input : in parameter Base_Types::Float; output : out parameter Base_Types::Float;
  end sqrt;

  subprogram sin
    features
      input : in parameter Base_Types::Float; output : out parameter Base_Types::Float;
  end sin;

  subprogram cos
    features
      input : in parameter Base_Types::Float; output : out parameter Base_Types::Float;
  end cos;

  subprogram tan
    features
      input : in parameter Base_Types::Float; output : out parameter Base_Types::Float;
  end tan;

  subprogram log
    features
      input : in parameter Base_Types::Float; output : out parameter Base_Types::Float;
  end log;

  subprogram angle
    features
      input : in parameter Base_Types::Float; output : out parameter Base_Types::Float;
  end angle;

 subprogram min
    features
      arg1 : in parameter Base_Types::Float; arg2 : in parameter Base_Types::Float;
      output : out parameter Base_Types::Float;
  end min;
end MathLib;
```

B The Entire Real-Time Maude Semantics

```
--- All the identifiers will be automatically generated by the tool.
(fmod FEATURE-ID is sort FeatureId . endfm)
(fmod COMPONENT-ID is sort ComponentId . endfm)
(fmod PROPERTY-ID is sort PropertyId . endfm)
(fmod BEHAVIOR-VAR-ID is sort VarId . endfm)
(fmod BEHAVIOR-LOCATION-ID is sort Location . endfm)

--- Full component names
(fmod COMPONENT-REF is
  including COMPONENT-ID .
  sort ComponentRef .
  subsort ComponentId < ComponentRef .
  op _._ : ComponentRef ComponentRef -> ComponentRef [ctor assoc] .
endfm)

--- Full feature (e.g., port) names
(fmod FEATURE-REF is
  including FEATURE-ID .
  including COMPONENT-REF .
  sort FeatureRef .
  subsort FeatureId < FeatureRef .
  op _.._ : ComponentRef FeatureId -> FeatureRef [ctor] .
endfm)

--- The views for parameterized modules
(view FeatureId from TRIV to FEATURE-ID is sort Elt to FeatureId . endv)
(view ComponentId from TRIV to COMPONENT-ID is sort Elt to ComponentId . endv)
(view Location from TRIV to BEHAVIOR-LOCATION-ID is sort Elt to Location . endv)
(view VarId from TRIV to BEHAVIOR-VAR-ID is sort Elt to VarId . endv)
(view FeatureRef from TRIV to FEATURE-REF is sort Elt to FeatureRef . endv)
(view ComponentRef from TRIV to COMPONENT-REF is sort Elt to ComponentRef . endv)


--- Data value
(fmod DATA-VALUE is sort Value . endfm)

--- Data content = data value + bot (the "don't care" value)
(fmod DATA-CONTENT is
  including DATA-VALUE .
  sort DataContent .
  subsort Value < DataContent .
  op bot : -> DataContent [ctor] .
endfm)

--- The views for parameterized modules
(view Value from TRIV to DATA-VALUE is sort Elt to Value . endv)
(view DataContent from TRIV to DATA-CONTENT is sort Elt to DataContent . endv)

--- Basic data types in AADL, defined in the Data Modeling Annex. They are enclosed by
--- the operator [_] for parsing purposes.
(fmod BASIC-VALUE is
  including DATA-VALUE .
  protecting CONVERSION .

  sort BoolValue IntValue FloatValue CharValue StringValue .
  subsort BoolValue IntValue FloatValue CharValue StringValue < Value .
  subsort CharValue < StringValue .

  op `[_`] : Bool -> BoolValue [ctor] .
  op `[_`] : Int -> IntValue [ctor] .
  op `[_`] : Float -> FloatValue [ctor] .
  op `[_`] : Char -> CharValue [ctor] .
  op `[_`] : String -> StringValue [ctor] .

  var B : Bool . var I : Int . var F : Float . var S : String .

  op bool : BoolValue -> Bool .          eq bool([B]) = B .
  op int : IntValue -> Int .             eq int([I]) = I .
  op float : IntValue -> Float .         eq float([I]) = float(I) .
  op float : FloatValue -> Float .       eq float([F]) = F .
  op string : StringValue -> String .    eq string([S]) = S .
endfm)

--- The syntax of the predefined input adaptors
(fmod BUILTIN-INPUT-ADAPTORS is
  sorts BuiltinInputAdaptor OneToManyInputAdaptor ManyToOneInputAdaptor .
  subsorts OneToManyInputAdaptor ManyToOneInputAdaptor < BuiltinInputAdaptor .

  ops repeat`input
      use`in`first`iteration
      use`in`last`iteration : -> OneToManyInputAdaptor [ctor] .
  op use`in`iteration_ : Nat -> OneToManyInputAdaptor [ctor] .

  ops first last max min sum average : -> ManyToOneInputAdaptor [ctor] .
  op use`element_ : Nat -> ManyToOneInputAdaptor [ctor] .
endfm)

--- Basic property values, which are enclosed by the operator {...} for parsing.
(fmod AADL-PROPERTY-VALUE is
  including BASIC-VALUE .

  sort PropertyValue .
  op `{_`} : Bool -> PropertyValue [ctor] .
  op `{_`} : Int -> PropertyValue [ctor] .
  op `{_`} : Float -> PropertyValue [ctor] .
  op `{_`} : String -> PropertyValue [ctor] .

  op value : PropertyValue -> Value .
  eq value({B:Bool}) = [B:Bool] .
  eq value({I:Int}) = [I:Int] .
  eq value({F:Float}) = [F:Float] .
  eq value({S:String}) = [S:String] .
endfm)

--- AADL property assignments
(fmod AADL-PROPERTY is
  including PROPERTY-ID .
  including AADL-PROPERTY-VALUE .
  sorts Property .
  op _=>_ : PropertyId PropertyValue -> Property [ctor] .
endfm)

--- The view for parameterized modules
(view Property from TRIV to AADL-PROPERTY is sort Elt to Property . endv)

--- A set of AADL properties, where the constructor operators are renamed as follows.
(fmod AADL-PROPERTY-ASSOCIATION is
  including SET{Property} * (sort Set{Property} to PropertyAssociation,
                             sort NeSet{Property} to NePropertyAssociation,
                             op _`,_ to _;_,
                             op empty to none) .
endfm)
```

```
--- Default AADL properties; currently, only Period is explicitly used in the semantics.
(fmod DEFAULT-PROPERTIES is
  including AADL-PROPERTY .
  op TimingProperties::Period : -> PropertyId [ctor] .
endfm)

--- the Multirate Synchronous AADL properties
(fmod SYNCHAADL-PROPERTIES is
  including BUILTIN-INPUT-ADAPTORS .
  including AADL-PROPERTY .
  ops MRSynchAADL::Synchronous
      MRSynchAADL::Nondeterministic MRSynchAADL::InputAdaptor : -> PropertyId [ctor] .
  op `{_`} : BuiltinInputAdaptor -> PropertyValue [ctor] .
endfm)

--- AADL connections, which are assumed to be delayed.
(fmod CONNECTION is
  including FEATURE-REF .
  sort Connection .
  op _-->_ : FeatureRef FeatureRef -> Connection [ctor] .
endfm)

--- The view for parameterized modules
(view Connection from TRIV to CONNECTION is sort Elt to Connection . endv)

--- The connection set, with the set union operator _;_
(fmod CONNECTION-SET is
  including SET{Connection} * (op _`,_ to _;_) .
endfm)

--- Features
(omod FEATURE is
  including FEATURE-ID .
  including AADL-PROPERTY-ASSOCIATION .
  class Feature | properties : PropertyAssociation .
  subsort FeatureId < Oid .
endom)

--- Data ports which can have a list of values. An input port has an extra argument
--- "cache" that stores the previously received value.
(omod PORT is
  including FEATURE .
  including LIST{DataContent} .
  class Port | content : List{DataContent} .
  subclass Port < Feature .

  class InPort | cache : DataContent .
  class OutPort .
  subclass InPort OutPort < Port .
endom)

--- AADL parameters for subprograms
(omod PARAMETER is
  including FEATURE .
  including DATA-CONTENT .
  class Parameter | content : DataContent .
  subclass Parameter < Feature .

  class InParameter .  class OutParameter .
  subclass InParameter OutParameter < Parameter .
endom)
```

```
--- The base class for any AADL components
(omod COMPONENT is
  including FEATURE .
  including CONNECTION-SET .

  class Component | features : Configuration, subcomponents : Configuration,
                    connections : Set{Connection}, properties : PropertyAssociation .
  subsort ComponentRef < Oid .
endom)

--- A component with period (or rate).
(omod PERIODIC-COMPONENT is
  including COMPONENT .
  class PeriodicComponent | rate : NzNat .
  subclass PeriodicComponent < Component .
endom)

--- A container class whose behavior is defined by its subcomponents.
(omod ENSEMBLE-COMPONENTS is
  including PERIODIC-COMPONENT .
  class Ensemble .
  subclass Ensemble < PeriodicComponent .

  class System . class Process . class ThreadGroup .
  subclass System Process ThreadGroup < Ensemble .
endom)

--- A thread with behavior
(omod THREAD-COMPONENT is
  including PERIODIC-COMPONENT .
  including BEHAVIOR-TRANSITION-SET .
  including SET{VarId} * (op _‘,_ to _;_) .

  class Thread | currState : Location, completeStates : Set{Location},
                 variables : Set{VarId}, transitions : Set{Transition} .
  subclass Thread < PeriodicComponent .
endom)

--- A data component
(omod DATA-COMPONENT is
  including COMPONENT .
  including DATA-CONTENT .
  class Data | value : DataContent .
  subclass Data < Component .
endom)

--- The syntax of the behavior annex language in AADL.
(fmod BEHAVIOR-EXPRESSION-SYNTAX is
  including BEHAVIOR-VAR-ID .
  including FEATURE-ID .
  including PROPERTY-ID .
  including COMPONENT-REF .
  including BASIC-VALUE .
  sort Expression .
  subsort Value < Expression .

  sort VarExpression .
  subsort VarExpression < Expression .
  op ‘[_‘] : FeatureId -> Expression [ctor] .
  op ‘[_‘] : VarId -> Expression [ctor] .
  op ‘[_‘] : ComponentRef -> Expression [ctor] .
  op ‘[_‘] : PropertyId -> Expression [ctor] .
  op count : FeatureId -> VarExpression [ctor] . --- p'count
  op fresh : FeatureId -> VarExpression [ctor] . --- p'fresh

--- logical binary operators
  op _and_ : Expression Expression -> Expression [ctor] .
  op _or_ : Expression Expression -> Expression [ctor] .
  op _xor_ : Expression Expression -> Expression [ctor] .

--- relational binary operators
  op _=_ : Expression Expression -> Expression [ctor] .
  op _!=_ : Expression Expression -> Expression [ctor] .
  op _<_ : Expression Expression -> Expression [ctor] .
  op _<=_ : Expression Expression -> Expression [ctor] .
  op _>_ : Expression Expression -> Expression [ctor] .
  op _>=_ : Expression Expression -> Expression [ctor] .

--- arithmetic binary operators
  op _+_ : Expression Expression -> Expression [ctor] .
  op _-_ : Expression Expression -> Expression [ctor] .
  op _*_ : Expression Expression -> Expression [ctor] .
  op _/_ : Expression Expression -> Expression [ctor] .
  op _mod_ : Expression Expression -> Expression [ctor] .
  op _rem_ : Expression Expression -> Expression [ctor] .
  op _**_ : Expression Expression -> Expression [ctor] .

--- unary operators
  op not : Expression -> Expression [ctor] .
  op plus : Expression -> Expression [ctor] .
  op minus : Expression -> Expression [ctor] .
  op abs : Expression -> Expression [ctor] .
endfm)

--- A transition guard
(fmod BEHAVIOR-CONDITION-SYNTAX is
  including BEHAVIOR-EXPRESSION-SYNTAX .

  sorts TransGuard DispatchCond .
  subsort DispatchCond < TransGuard .
  op on`dispatch : -> DispatchCond [ctor] .

  sort ExecuteCond .
  subsort Expression < ExecuteCond < TransGuard .
  op otherwise : -> ExecuteCond [ctor] .
endfm)

--- The syntax of the behavior action language
(fmod BEHAVIOR-ACTION-SYNTAX is
  including BEHAVIOR-EXPRESSION-SYNTAX .
  including CLASSIFIER-ID .
  sort Action .

--- action block for action sequences/sets
  sort ActionBlock .
  subsort ActionBlock < Action .
  op `{_`} : ActionGroup -> ActionBlock [ctor] .

  sort ActionGroup ActionSequence ActionSet .
  subsort Action < ActionSequence ActionSet < ActionGroup .
  op skip : -> ActionGroup [ctor] . --- no action
  op _;_ : ActionSequence ActionSequence -> ActionSequence [ctor assoc] .
  op _&_ : ActionSet ActionSet -> ActionSet [ctor comm assoc] .
```

```
--- assignment: local variables, output ports/params, and data (sub)components
  sort AssignmentAction .
  subsort AssignmentAction < Action .
  op _:=_ : AssignmentTarget Expression -> AssignmentAction [ctor] .

  sort AssignmentTarget .
  op `{_`} : VarId -> AssignmentTarget [ctor] .
  op `{_`} : FeatureId -> AssignmentTarget [ctor] .
  op `{_`} : ComponentRef -> AssignmentTarget [ctor] .

--- communication: MR-SynchAADL only supports subprogram component/classifier calls
  sort CommunicationAction .
  subsort CommunicationAction < Action .

  op _! : ComponentId -> CommunicationAction [ctor] .
  op _!`(_`) : ComponentId ParameterList -> CommunicationAction [ctor] .

  op _! : ClassifierId -> CommunicationAction [ctor] .
  op _!`(_`) : ClassifierId ParameterList -> CommunicationAction [ctor] .

  sort ParameterList .
  subsort Expression < ParameterList .
  op _`,_ : ParameterList ParameterList -> ParameterList [ctor assoc] .

--- branch action
  sort BranchAction .
  subsort BranchAction < Action .
  op if`(_`)_end`if : Expression ActionGroup -> BranchAction [ctor] .
  op if`(_`)_else_end`if : Expression ActionGroup ActionGroup -> BranchAction [ctor] .
  op if`(_`)__end`if : Expression ActionGroup ElseIfs -> BranchAction [ctor] .
  op if`(_`)__else_end`if : Expression ActionGroup ElseIfs ActionGroup -> BranchAction
      [ctor] .

  sort ElseIfs .
  op __ : ElseIfs ElseIfs -> ElseIfs [ctor assoc] .
  op elsif`(_`)_ : Expression ActionGroup -> ElseIfs [ctor] .

--- loop action. NOTE: The for loop is not supported yet.
  sort LoopAction .
  subsort LoopAction < Action .
  op while`(_`)`{_`} : Expression ActionGroup -> LoopAction [ctor] .
  op do_until`(_`) : ActionGroup Expression -> LoopAction [ctor] .
endfm)

--- Behavior transitions of the form: source -[ guard ]-> destination {action}
(fmod BEHAVIOR-TRANSITION is
  including BEHAVIOR-CONDITION-SYNTAX .
  including BEHAVIOR-ACTION-SYNTAX .
  including SET{Location} * (op _`,_ to __) .

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

--- The view for parameterized modules
(view Transition from TRIV to BEHAVIOR-TRANSITION is sort Elt to Transition . endv)

--- A behavior transition set with the set union operator _;_
(fmod BEHAVIOR-TRANSITION-SET is
  including SET{Transition} * (op _`,_ to _;_) .
endfm)
```

```
--- The semantics for value expressions (with no variables)
(fmod BEHAVIOR-EXPRESSION-VALUE-SEMANTICS is
 including BEHAVIOR-EXPRESSION-SYNTAX .

 vars V1 V2 : Value . vars B1 B2 : Bool . vars I1 I2 : Int .
 vars F1 F2 : Float . vars S1 S2 : String .

 eq [B1] and [B2] = [B1 and B2] .
 eq [B1] or [B2] = [B1 or B2] .
 eq [B1] xor [B2] = [B1 xor B2] .

 eq (V1 = V2)  = [V1 == V2] [owise] .
 eq ([I1] = [F2]) = [float(I1) == F2] .
 eq ([F1] = [I2]) = [F1 == float(I2)] .

 eq [I1] < [I2] = [I1 < I2] .
 eq [F1] < [F2] = [F1 < F2] .
 eq [I1] < [F2] = [float(I1) < F2] . eq [F1] < [I2] = [F1 < float(I2)] .

 eq [I1] > [I2] = [I1 > I2] .
 eq [F1] > [F2] = [F1 > F2] .
 eq [I1] > [F2] = [float(I1) > F2] . eq [F1] > [I2] = [F1 > float(I2)] .

 eq (V1 != V2) = not (V1 = V2) .
 eq (V1 <= V2) = not (V1 > V2) .
 eq (V1 >= V2) = not (V1 < V2) .

 eq [I1] + [I2] = [I1 + I2] .
 eq [F1] + [F2] = [F1 + F2] .
 eq [I1] + [F2] = [float(I1) + F2] . eq [F1] + [I2] = [F1 + float(I2)] .

 eq [I1] - [I2] = [I1 - I2] .
 eq [F1] - [F2] = [F1 - F2] .
 eq [I1] - [F2] = [float(I1) - F2] . eq [F1] - [I2] = [F1 - float(I2)] .

 eq [I1] * [I2] = [I1 * I2] .
 eq [F1] * [F2] = [F1 * F2] .
 eq [I1] * [F2] = [float(I1) * F2] . eq [F1] * [I2] = [F1 * float(I2)] .

 eq [I1] / [I2] = [float(I1) / float(I2)] .
 eq [F1] / [F2] = [F1 / F2] .
 eq [I1] / [F2] = [float(I1) / F2] . eq [F1] / [I2] = [F1 / float(I2)] .

 eq [I1] mod [I2] = [I1 rem I2] .
 eq [I1] rem [I2] = [I1 rem I2] .

 eq [I1] ** [I2] = [I1 ^ I2] .
 eq [F1] ** [F2] = [F1 ^ F2] .
 eq [I1] ** [F2] = [float(I1) ^ F2] . eq [F1] ** [I2] = [F1 ^ float(I2)] .

 eq not([B1]) = [not B1] .

 eq plus([I1]) = [I1] .
 eq plus([F1]) = [F1] .

 eq minus([I1]) = [- I1] .
 eq minus([F1]) = [- F1] .

 eq abs([I1]) = [abs(I1)] .
 eq abs([F1]) = [abs(F1)] .
endfm)
```

```
--- A map for local temporary variables
(fmod VAR-VALUATION is
  including SET{VarId} * (op _‘,_ to _;_) .
  including MAP{VarId,DataContent}
          * (sort Map{VarId,DataContent} to VarValuation, op _‘,_ to _;_) .

  var VI : VarId . var VIS : Set{VarId} . var DC : DataContent .
  vars VAL VAL’ : VarValuation .

--- a default valuation where every variable is mapped to bot.
  op defaultValuation : Set{VarId} -> VarValuation .
  op defaultValuation : Set{VarId} VarValuation -> VarValuation .
  eq defaultValuation(VIS) = defaultValuation(VIS, empty) .
  eq defaultValuation(VI ; VIS, VAL) = defaultValuation(VIS, (VI |-> bot) ; VAL) .
  eq defaultValuation(empty, VAL) = VAL .
endfm)

--- A map for features (i.e., ports)
(omod FEATURE-MAP is
  including PORT .
  including PARAMETER .
  including MAP{FeatureId,DataContent} *(sort Map{FeatureId,DataContent} to FeatureMap) .
  sort PortValue .
  subsort PortValue < DataContent .
  op _:_ : Value Bool -> PortValue [ctor] . --- a pair of (value : fresh)

  sort Pair{Configuration,FeatureMap} .
  op _|_ : Configuration FeatureMap -> Pair{Configuration,FeatureMap} [ctor] .

  var FMAP : FeatureMap . vars FTS FTS’ : Configuration . var PI : FeatureId .
  vars V : Value . var DCL : List{DataContent} . var B : Bool .

  op readFeature : Configuration -> Pair{Configuration,FeatureMap} .
  op readFeature : Configuration Configuration FeatureMap
                -> Pair{Configuration,FeatureMap} .
  eq readFeature(FTS) = readFeature(FTS, none, empty) .
  eq readFeature(< PI : InPort | content : V DCL > FTS, FTS’, FMAP)
   = readFeature(FTS, < PI : InPort | content : DCL, cache : V > FTS’,
                insert(PI, V : true, FMAP)) .
  eq readFeature(< PI : InPort | content : bot DCL, cache : V > FTS, FTS’, FMAP)
   = readFeature(FTS, < PI : InPort | content : DCL > FTS’,insert(PI, V : false, FMAP)) .
  eq readFeature(< PI : InParameter | content : V > FTS, FTS’, FMAP)
   = readFeature(FTS, < PI : InParameter | content : bot > FTS’, insert(PI, V, FMAP)) .
  eq readFeature(< PI : OutPort | > FTS, FTS’, FMAP)
   = readFeature(FTS, < PI : OutPort | > FTS’, insert(PI, bot, FMAP)) .
  eq readFeature(< PI : OutParameter | > FTS, FTS’, FMAP)
   = readFeature(FTS, < PI : OutParameter | > FTS’, insert(PI, bot, FMAP)) .
  eq readFeature(< PI : InPort | content : bot DCL, cache : bot > FTS, FTS’, FMAP)
   = readFeature(FTS, < PI : InPort | content : DCL > FTS’, FMAP) .
  eq readFeature(none, FTS’, FMAP) = FTS’ | FMAP .

  op writeFeature : FeatureMap Configuration -> Configuration .
  op writeFeature : FeatureMap Configuration Configuration -> Configuration .
  eq writeFeature(FMAP, FTS) = writeFeature(FMAP, FTS, none) .
  eq writeFeature(FMAP, < PI : OutPort | content : DCL > FTS, FTS’)
   = if $hasMapping(FMAP,PI) and FMAP[PI] :: Value
     then writeFeature(FMAP, FTS, < PI : OutPort | content : DCL FMAP[PI] > FTS’)
     else writeFeature(FMAP, FTS, < PI : OutPort | content : DCL bot > FTS’) fi .
  eq writeFeature((PI |-> V, FMAP), < PI : OutParameter | content : bot > FTS, FTS’)
   = writeFeature(FMAP, FTS, < PI : OutParameter | content : V > FTS’) .
  eq writeFeature(FMAP, FTS, FTS’) = FTS FTS’ [owise] .
endom)
```

```
--- The configuration of the behavior annex expression/action language
(omod BEHAVIOR-CONF is
  including VAR-VALUATION . including FEATURE-MAP .
  including DATA-COMPONENT .
  sorts GlobalBehaviorConf LocalBehaviorConf .
  op _|_|_ : FeatureMap Configuration PropertyAssociation -> GlobalBehaviorConf [ctor] .
  op _|_ : VarValuation GlobalBehaviorConf -> LocalBehaviorConf [ctor] .
endom)

--- The semantics of the behavior expression language.
(omod BEHAVIOR-EXPRESSION-SEMANTICS is
  protecting BEHAVIOR-CONF .
  including BEHAVIOR-EXPRESSION-VALUE-SEMANTICS .

  var VAL : VarValuation . var FMAP : FeatureMap . vars COMPS : Configuration .
  var PROPS : PropertyAssociation . vars V : Value . var B : Bool .
  var LCF : LocalBehaviorConf . var GCF : GlobalBehaviorConf .
  var CR : ComponentRef . var PI : FeatureId . var VI : VarId .
  var PR : PropertyId . var PV : PropertyValue . vars E1 E2 : Expression .

  op eval : Expression LocalBehaviorConf -> Value .
  eq eval(V, LCF) = V . --- values

--- variable expressions
  eq eval([VI], (VI |-> V) ; VAL | GCF) = V . --- temporary variables
  eq eval([PI], VAL | (PI |-> (V : B), FMAP) | COMPS | PROPS) = V . --- ports
  eq eval([PI], VAL | (PI |-> V, FMAP) | COMPS | PROPS) = V .      --- parameters
  eq eval([CR], VAL | FMAP | < CR : Data | value : V > COMPS | PROPS) = V . --- data
  eq eval([PR], VAL | FMAP | COMPS | (PR => PV) ; PROPS) = value(PV) . --- properties
  eq eval(count(PI), VAL | (PI |-> (V : B), FMAP) | COMPS | PROPS)
   = [if B then 1 else 0 fi] .
  eq eval(fresh(PI), VAL | (PI |-> (V : B), FMAP) | COMPS | PROPS) = [B] .

--- logical binary expressions
  eq eval(E1 and E2, LCF) = eval(E1, LCF) and eval(E2, LCF) .
  eq eval(E1 or E2, LCF) = eval(E1, LCF) or eval(E2, LCF) .
  eq eval(E1 xor E2, LCF) = eval(E1, LCF) xor eval(E2, LCF) .

--- relational expressions
  eq eval(E1 = E2, LCF) = (eval(E1, LCF) = eval(E2, LCF)) .
  eq eval(E1 != E2, LCF) = (eval(E1, LCF) != eval(E2, LCF)) .
  eq eval(E1 < E2, LCF) = (eval(E1, LCF) < eval(E2, LCF)) .
  eq eval(E1 <= E2, LCF) = (eval(E1, LCF) <= eval(E2, LCF)) .
  eq eval(E1 > E2, LCF) = (eval(E1, LCF) > eval(E2, LCF)) .
  eq eval(E1 >= E2, LCF) = (eval(E1, LCF) >= eval(E2, LCF)) .

--- numeric binary expressions
  eq eval(E1 + E2, LCF) = (eval(E1, LCF) + eval(E2, LCF)) .
  eq eval(E1 - E2, LCF) = (eval(E1, LCF) - eval(E2, LCF)) .
  eq eval(E1 * E2, LCF) = (eval(E1, LCF) * eval(E2, LCF)) .
  eq eval(E1 / E2, LCF) = (eval(E1, LCF) / eval(E2, LCF)) .
  eq eval(E1 mod E2, LCF) = (eval(E1, LCF) mod eval(E2, LCF)) .
  eq eval(E1 rem E2, LCF) = (eval(E1, LCF) rem eval(E2, LCF)) .
  eq eval(E1 ** E2, LCF) = (eval(E1, LCF) ** eval(E2, LCF)) .

--- unary operators
  eq eval(not(E1), LCF) = not(eval(E1, LCF)) .
  eq eval(plus(E1), LCF) = plus(eval(E1, LCF)) .
  eq eval(minus(E1), LCF) = minus(eval(E1, LCF)) .
  eq eval(abs(E1), LCF) = abs(eval(E1, LCF)) .
endom)
```

```
--- The semantics of the behavior action language.
(omod BEHAVIOR-ACTION-SEMANTICS is
  including BEHAVIOR-ACTION-SYNTAX .
  including BEHAVIOR-EXPRESSION-SEMANTICS .

  var VAL : VarValuation . var FMAP : FeatureMap . vars COMPS : Configuration .
  var PROPS : PropertyAssociation . var LCF : LocalBehaviorConf .
  var CR : ComponentRef . var PI : FeatureId . var VI : VarId . var A : Action .
  vars AS AS' AS'' : ActionGroup . var ASQ : ActionSequence . var AST : ActionSet .
  vars F F' : Float . vars V : Value . vars E E' E'' : Expression .
  var ELSIFS : ElseIfs . var DC : DataContent . var ATTS : AttributeSet .

  op execAction : Action LocalBehaviorConf -> LocalBehaviorConf .

--- action blocks/sets/sequences
  eq execAction({A ; ASQ}, LCF) = execAction({ASQ}, execAction(A, LCF)) .
  eq execAction({A & AST}, LCF) = execAction({AST}, execAction(A, LCF)) .
  eq execAction({A},    LCF) = execAction(A, LCF) .
  eq execAction({skip}, LCF) = LCF . --- empty action

--- assignment: local variables, output ports/params, and data (sub)components
 ceq execAction({VI} := E, (VI |-> DC) ; VAL | FMAP | COMPS | PROPS)
   = (VI |-> V) ; VAL | FMAP | COMPS | PROPS
 if V := eval(E, (VI |-> DC) ; VAL | FMAP | COMPS | PROPS) .

 ceq execAction({PI} := E, VAL | (PI |-> DC, FMAP) | COMPS | PROPS)
   = VAL | (PI |-> V, FMAP) | COMPS | PROPS .
 if V := eval(E, VAL | (PI |-> DC, FMAP) | COMPS | PROPS) .

 ceq execAction({CR} := E, VAL | FMAP | < CR : Data | value : DC > COMPS | PROPS)
   = VAL | FMAP | < CR : Data | value : V > COMPS | PROPS .
 if V := eval(E, VAL | FMAP | < CR : Data | value : DC > COMPS | PROPS) .

  op target : VarExpression -> AssignmentTarget .
  eq target([VI]) = {VI} .
  eq target([PI]) = {PI} .
  eq target([CR]) = {CR} .

--- subprogram call: currently, we only consider predefined functions.
  ops MathLib::sqrt MathLib::sin MathLib::cos MathLib::tan
      MathLib::log MathLib::angle MathLib::min : -> ClassifierId [ctor] .
  eq execAction(MathLib::sqrt ! (E, E'), LCF)
   = execAction(target(E') := [sqrt(float(eval(E,LCF)))], LCF) .
  eq execAction(MathLib::sin ! (E, E'), LCF)
   = execAction(target(E') := [sin(float(eval(E,LCF)))], LCF) .
  eq execAction(MathLib::cos ! (E, E'), LCF)
   = execAction(target(E') := [cos(float(eval(E,LCF)))], LCF) .
  eq execAction(MathLib::tan ! (E, E'), LCF)
   = execAction(target(E') := [tan(float(eval(E,LCF)))], LCF) .
  eq execAction(MathLib::log ! (E, E'), LCF)
   = execAction(target(E') := [log(float(eval(E,LCF)))], LCF) .
  eq execAction(MathLib::angle ! (E, E'), LCF)
   = execAction(target(E') := [angle(float(eval(E,LCF)))], LCF) .
  eq execAction(MathLib::min ! (E, E', E''), LCF)
   = execAction(target(E'') := [min(float(eval(E,LCF)),float(eval(E',LCF)))], LCF) .

  op angle : Float -> Float .
  eq angle(F)
   = if F > 180.0 then angle(F - 360.0) else
       if F <= -180.0 then angle(F + 360.0) else F fi fi .
```

```
--- branch action
  eq execAction(if (E) AS end if, LCF)
   = if eval(E, LCF) == [true] then execAction({AS}, LCF) else LCF fi .
  eq execAction(if (E) AS else AS' end if, LCF)
   = if eval(E, LCF) == [true] then execAction({AS},LCF) else execAction({AS'},LCF) fi .
  eq execAction(if (E) AS (elsif (E') AS') end if, LCF)
   = if eval(E, LCF) == [true] then execAction({AS}, LCF)
     else execAction(if (E') AS' end if, LCF) fi .
  eq execAction(if (E) AS ((elsif (E') AS') ELSIFS) end if, LCF)
   = if eval(E, LCF) == [true] then execAction({AS}, LCF)
     else execAction(if (E') AS' ELSIFS end if, LCF) fi .
  eq execAction(if (E) AS (elsif (E') AS') else AS'' end if, LCF)
   = if eval(E, LCF) == [true] then execAction({AS}, LCF)
     else execAction(if (E') AS' else AS'' end if, LCF) fi .
  eq execAction(if (E) AS ((elsif (E') AS') ELSIFS) else AS'' end if, LCF)
   = if eval(E, LCF) == [true] then execAction({AS}, LCF)
     else execAction(if (E') AS' ELSIFS else AS'' end if, LCF) fi .

--- loop action.
  eq execAction(while (E) {AS}, LCF)
   = if eval(E, LCF) == [true]
     then execAction({while (E) {AS}}, execAction({AS}, LCF))
     else LCF fi .
  eq execAction(do AS until (E), LCF)
   = execAction(while (not(E)) {AS}, execAction({AS}, LCF)) .
endom)

--- The semantics of behavior transitions
(omod BEHAVIOR-TRANSITION-SEMANTICS is
  including BEHAVIOR-TRANSITION-SET .
  including BEHAVIOR-ACTION-SEMANTICS .

  sort Tuple{Location,FeatureMap,Configuration} .
  op _|_|_ : Location FeatureMap Configuration
         -> Tuple{Location,FeatureMap,Configuration} [ctor] .

  vars VAL VAL' : VarValuation . var FMAP : FeatureMap . vars COMPS : Configuration .
  var PROPS : PropertyAssociation . vars GCF GCF' : GlobalBehaviorConf .
  var LCF : LocalBehaviorConf . vars TRS TRS' ETRS : Set{Transition} .
  var GUARD : TransGuard . var ACTION : ActionBlock . var E : Expression .
  vars L L' L'' : Location . var LS : Set{Location} .

--- execute (nondeterministic) transitions until reaching a complete state.
--- the following equations/rules are "unconditional" versions of those in the paper.
  op execTrans : Location Set{Location} Set{Transition} VarValuation GlobalBehaviorConf
         ~> Tuple{Location,FeatureMap,Configuration} .
  op execTrans : Location Set{Location} Set{Transition} Set{Transition} VarValuation
         GlobalBehaviorConf ~> Tuple{Location,FeatureMap,Configuration} .

  eq execTrans(L, LS, TRS, VAL, GCF)
   = execTrans(L, LS, enabledTrans(L, TRS, VAL | GCF, empty), TRS, VAL, GCF) .

  rl [trans]:
     execTrans(L, LS, (L -[GUARD]-> L' ACTION) ; TRS', TRS, VAL, GCF)
   =>
     if L' in LS
     then transitionResult(L', execAction(ACTION, VAL | GCF))
     else execTrans(L', LS, TRS, VAL, global(execAction(ACTION, VAL | GCF))) fi .

--- execute deterministic transitions until reaching a complete state.
  op execDetTrans : Location Set{Location} Set{Transition} VarValuation
           GlobalBehaviorConf ~> Tuple{Location,FeatureMap,Configuration} .
  op execDetTrans : Location Set{Location} Set{Transition} Set{Transition} VarValuation
            GlobalBehaviorConf ~> Tuple{Location,FeatureMap,Configuration} .

  eq execDetTrans(L, LS, TRS, VAL, GCF)
   = execDetTrans(L, LS, enabledTrans(L, TRS, VAL | GCF, empty), TRS, VAL, GCF) .

  eq execDetTrans(L, LS, (L -[GUARD]-> L' ACTION) ; TRS', TRS, VAL, GCF)
   =
    if L' in LS
    then transitionResult(L', execAction(ACTION, VAL | GCF))
    else execDetTrans(L', LS, TRS, VAL, global(execAction(ACTION, VAL | GCF))) fi .

  --- aux functions
  op transitionResult : Location LocalBehaviorConf
                    -> Tuple{Location,FeatureMap,Configuration} .
  eq transitionResult(L, VAL | FMAP | COMPS | PROPS) = L | FMAP | COMPS .

  op global : LocalBehaviorConf ~> GlobalBehaviorConf .
  eq global(VAL | GCF) = GCF .

--- return a set of enabled transitions
  op enabledTrans : Location Set{Transition} LocalBehaviorConf Set{Transition}
                 -> Set{Transition} .
  eq enabledTrans(L, (L -[on dispatch]-> L' ACTION) ; TRS, LCF, TRS')
   = enabledTrans(L, TRS, LCF, TRS' ; (L -[on dispatch]-> L' ACTION)) .
  eq enabledTrans(L, (L -[E]-> L' ACTION) ; TRS, LCF, TRS')
   = if eval(E, LCF) == [true]
    then enabledTrans(L, TRS, LCF, TRS' ; (L -[E]-> L' ACTION))
    else enabledTrans(L, TRS, LCF, TRS') fi .
  eq enabledTrans(L, TRS, LCF, TRS')
   = if TRS' == empty then owiseTransitions(L, TRS, empty) else TRS' fi [owise] .

  op owiseTransitions : Location Set{Transition} Set{Transition} -> Set{Transition} .
  eq owiseTransitions(L, (L -[otherwise]-> L' ACTION) ; TRS, ETRS)
   = owiseTransitions(L, TRS, ETRS ; (L -[otherwise]-> L' ACTION)) .
  eq owiseTransitions(L, TRS, ETRS) = ETRS [owise] .

endom)

--- The behavior of components
(tomod COMPONENT-DYNAMICS is
  protecting PERIODIC-COMPONENT .
  protecting PORT .
  protecting SYNCHAADL-PROPERTIES .
  including TIME-DOMAIN .

  var CR : ComponentRef . var P : FeatureId .
  vars PORTS PORTS' REST : Configuration . var PROPS : PropertyAssociation .
  var NZ : NzNat . var NDL : NeList{DataContent} . var IA : BuiltinInputAdaptor .

--- A transition relation of each component, defined by either equations or rules.
  op executeStep : Object ~> Object .

--- input adaptors (name, input, output length)
  op adaptor : BuiltinInputAdaptor NeList{DataContent} NzNat -> NeList{DataContent} .

--- apply adaptors to the input ports of a component
  op applyAdaptors : Configuration -> Configuration .
  eq applyAdaptors(< CR : PeriodicComponent | rate : NZ, features : PORTS > REST)
   = < CR : PeriodicComponent | features : applyAdaptors(NZ,PORTS,none) >
    applyAdaptors(REST) .
  eq applyAdaptors(none) = none .
  op applyAdaptors : NzNat Configuration Configuration -> Object .
  eq applyAdaptors(NZ,
         < P : InPort | content : NDL,
              properties : (MRSynchAADL::InputAdaptor => {IA}); PROPS > PORTS, PORTS')
   = applyAdaptors(NZ, PORTS, PORTS' < P : InPort | content : adaptor(IA, NDL, NZ) >) .
  eq applyAdaptors(NZ, PORTS, PORTS') = PORTS PORTS' [owise] .
endtom)

--- The thread behavior. Note that the equations/rules are unconditional versions
--- of those in the paper, with additional aux functions.
(tomod THREAD-DYNAMICS is
  including COMPONENT-DYNAMICS .
  including THREAD-COMPONENT .
  including BEHAVIOR-TRANSITION-SEMANTICS .

  vars COMPS COMPS' PORTS PORTS' : Configuration . var PROPS : PropertyAssociation .
  vars VAL VAL' : VarValuation . vars FMAP FMAP' : FeatureMap . var CR : ComponentRef .
  var VIS : Set{VarId} . vars L L' : Location . vars LS : Set{Location} .
  var GUARD : TransGuard . var TRS TRS' : Set{Transition} . var ACTION : ActionBlock .

  op executeStepRead : Pair{Configuration,FeatureMap} Object ~> Object .
  op executeStepTrans : Configuration Tuple{Location,FeatureMap,Configuration} Object
                    ~> Object .

--- first, read data in its input ports
  eq executeStep(< CR : Thread | features : PORTS >)
   = executeStepRead(readFeature(PORTS), < CR : Thread | features : PORTS >) .

--- then, execute its transition system; for nondeterministic cases,
--- the term with the operator "execTrans" will be rewritten by rules.
  eq executeStepRead(PORTS' | FMAP,
        < CR : Thread | subcomponents : COMPS, properties : PROPS,
                        currState : L, completeStates : LS,
                        variables : VIS, transitions : TRS >)
   =
    if MRSynchAADL::Nondeterministic => {true} in PROPS
    then executeStepTrans(PORTS',
            execTrans(L,LS,TRS,defaultValuation(VIS), FMAP | COMPS | PROPS),
              < CR : Thread | >)
    else executeStepTrans(PORTS',
            execDetTrans(L,LS,TRS,defaultValuation(VIS), FMAP | COMPS | PROPS),
              < CR : Thread | >) fi .

--- finally, its ports, data, and state will be update accordingly
  eq executeStepTrans(PORTS', L' | FMAP' | COMPS', < CR : Thread | >)
   = < CR : Thread | features : writeFeature(FMAP',PORTS'),
                    subcomponents : COMPS', currState : L' > .
endtom)

--- The behavior of ensembles. Similarly, any conditional rules/equations are
--- transformed to their unconditional versions.
(tomod ENSEMBLE-DYNAMICS is
  including COMPONENT-DYNAMICS .
  including ENSEMBLE-COMPONENTS .
  including DEFAULT-PROPERTIES .
  protecting TRANSFER-DATA .

  vars CR : ComponentRef . var P : FeatureId . var N : Nat . var NZ : NzNat .
  vars OBJ : Object . vars COMPS : Configuration . var PROPS : PropertyAssociation .
  var PROPS : PropertyAssociation . var NDL : NeList{DataContent} .
  var QUEUE : [ObjectQueue] . var KOBJ : [Object] .
```

```
--- the frozen attribute gives a deterministic order in rewriting.
  sort ObjectQueue .
  op nil : -> ObjectQueue [ctor] .
  op _::_ : Object ObjectQueue -> ObjectQueue [ctor frozen(2)] .
  op _|_ : ObjectQueue Configuration ~> Configuration [ctor] .

--- the result of prepareExecSub will be rewritten to an object .
  eq executeStep(< CR : Ensemble | >)
   = transferResults(
       prepareExecSub(
           applyAdaptorsSub(
              transferInputs(< CR : Ensemble | >)))) .

--- apply adaptors to subcomponents
   op applyAdaptorsSub : Object -> Object .
   eq applyAdaptorsSub(< CR : Ensemble | subcomponents : COMPS >)
    = < CR : Ensemble | subcomponents : applyAdaptors(COMPS) > .

--- prepare to execute.
  op prepareExecSub : Object ~> Object .
  eq prepareExecSub(< CR : Ensemble | subcomponents : COMPS >)
   = < CR : Ensemble | subcomponents : prepareExec(COMPS, nil) > .

--- generate a "serialized" queue. This is generally needed since a system
--- component can contain several system components in AADL.
  op prepareExec : Configuration ObjectQueue ~> Configuration .
  eq prepareExec(< CR : PeriodicComponent | rate : NZ > COMPS, QUEUE)
   = prepareExec(COMPS, k-executeStep(NZ, < CR : PeriodicComponent | >) :: QUEUE) .
  eq prepareExec(COMPS, QUEUE) = QUEUE | COMPS [owise] .

--- performs "executeStep" k times, after applying adaptors.
  op k-executeStep : Nat Object ~> Object .
  eq k-executeStep(s(N), OBJ) = executeStep(k-executeStep(N, OBJ)) .
  eq k-executeStep(0, OBJ) = OBJ .

--- if the first item finishes its execution, then the next item is scheduled
  eq OBJ :: QUEUE | COMPS = QUEUE | COMPS OBJ .
  eq nil | COMPS = COMPS .
endtom)

--- The semantics of the predefined input adaptors
(tomod BUILTIN-INPUT-ADAPTOR-SEMANTICS is
  including COMPONENT-DYNAMICS .

  var OTMA : OneToManyInputAdaptor . var MTOA : OneToManyInputAdaptor .
  var NZ : NzNat . vars N M : Nat . vars I1 I2 : Int . vars F1 F2 : Float .
  var D : DataContent . var DL : List{DataContent} . var NDL : NeList{DataContent} .

--- one-to-many adaptors
  op adaptor : OneToManyInputAdaptor DataContent Nat NeList{DataContent}
             -> NeList{DataContent} .

  eq adaptor(OTMA, D, NZ) = adaptor(OTMA, D, NZ, nil) .
  eq adaptor(OTMA, D, 0, NDL) = NDL .

  eq adaptor(repeat input, D, s(N), DL)
   = adaptor(repeat input, D, N, DL D) .

  eq adaptor(use in first iteration, D, N, DL)
   = adaptor(use in iteration 1, D, N, DL) .
  eq adaptor(use in last iteration, D, N, DL)
   = adaptor(use in iteration N, D, N, DL) .

  eq adaptor(use in iteration s(s(M)), D, s(N), DL)
   = adaptor(use in iteration s(M), D, N, DL bot) .

  eq adaptor(use in iteration s(0), D, s(N), DL)
   = adaptor(use in iteration 0, D, N, DL D) .

  eq adaptor(use in iteration 0,  D, s(N), DL)
   = adaptor(use in iteration 0, D, N, DL bot) .

--- many to one adaptors
  op numAdap : ManyToOneInputAdaptor DataContent List{DataContent} -> DataContent .

  eq adaptor(first, D DL, 1) = D .
  eq adaptor(last, DL D, 1) = D .

  eq adaptor(max, D DL, 1)   = numAdap(max, D, DL) .
  eq numAdap(max, [I1], nil) = [I1] .
  eq numAdap(max, [F1], nil) = [F1] .
  eq numAdap(max, [I1], [I2] DL) = numAdap(max, [max(I1,I2)], DL) .
  eq numAdap(max, [F1], [F2] DL) = numAdap(max, [max(F1,F2)], DL) .

  eq adaptor(min, D DL, 1)   = numAdap(min, D, DL) .
  eq numAdap(min, [I1], nil) = [I1] .
  eq numAdap(min, [F1], nil) = [F1] .
  eq numAdap(min, [I1], [I2] DL) = numAdap(min, [min(I1,I2)], DL) .
  eq numAdap(min, [F1], [F2] DL) = numAdap(min, [min(F1,F2)], DL) .

  eq adaptor(sum, D DL, 1)   = numAdap(sum, D, DL) .
  eq numAdap(sum, [I1], nil) = [I1] .
  eq numAdap(sum, [F1], nil) = [F1] .
  eq numAdap(sum, [I1], [I2] DL) = numAdap(sum, [I1 + I2], DL) .
  eq numAdap(sum, [F1], [F2] DL) = numAdap(sum, [F1 + F2], DL) .

  eq adaptor(average, NDL, 1) = [float(adaptor(sum, NDL, 1)) / float(size(NDL))] .
endtom)

--- the connection table for the "optimized" message passing.
(fmod CONX-TABLE is
  protecting CONNECTION-SET .
  protecting SET{FeatureRef} .

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

  var CR : ComponentRef . var P : FeatureId .
  var CONXS : Set{Connection} . var CTB : ConxTable .
  vars PN : FeatureRef . var NPS NPS' : NeSet{FeatureRef} .

  op contains? : FeatureRef ConxTable ~> Bool [memo format (m! o)] .
  eq contains?(PN, (PN |-> NPS) CTB) = true .
  eq contains?(PN, CTB) = false [owise] .

  op normalize : ConxTable ~> ConxTable .
  eq normalize((PN |-> NPS) (PN |-> NPS') CTB)
   = normalize((PN |-> (NPS,NPS')) CTB) .
  eq normalize(CTB) = CTB [owise] .
  op inner-tb : Set{Connection} ~> ConxTable [memo format (m! o)] .
  op inner-tb : Set{Connection} ConxTable ~> ConxTable .
  eq inner-tb(CONXS) = inner-tb(CONXS, none) .
  eq inner-tb((PN --> CR .. P) ; CONXS, CTB)
   = inner-tb(CONXS, (PN |-> CR .. P) CTB) .
  eq inner-tb(CONXS, CTB) = normalize(CTB) [owise] .

  op outer-tb : Set{Connection} ~> ConxTable [memo format (m! o)] .
  op outer-tb : Set{Connection} ConxTable ~> ConxTable .
  eq outer-tb(CONXS) = outer-tb(CONXS, none) .
  eq outer-tb((PN --> P) ; CONXS, CTB)
   = outer-tb(CONXS, (PN |-> P) CTB) .
  eq outer-tb(CONXS, CTB) = normalize(CTB) [owise] .
endfm)

--- messages
(omod TRANSFER-FUNCTIONS is
  protecting ENSEMBLE-COMPONENTS .
  protecting CONX-TABLE .
  protecting PORT .

  var CR : ComponentRef . var P : FeatureId . var PNS : Set{FeatureRef} .
  var DL : List{DataContent} . var NDL : NeList{DataContent} .
  var PORTS : Configuration . vars KPS KCS : [Configuration] .

  --- transfer into subcomponents
  op transIn : NeList{DataContent} Set{FeatureRef} ~> Msg [format (b! o)] .
  eq < CR : Ensemble | features : KPS transIn(NDL,PNS), subcomponents : KCS >
   = < CR : Ensemble | features : KPS, subcomponents : transIn(NDL,PNS) KCS > .
  eq transIn(NDL, (CR .. P, PNS))
     < CR : Component | features : < P : InPort | content : nil > PORTS >
   = transIn(NDL, PNS)
     < CR : Component | features : < P : InPort | content : NDL > PORTS > .
  eq transIn(NDL, empty) = none .

  --- transfer to the wrapper
  op transOut : NeList{DataContent} Set{FeatureRef} ~> Msg [format (b! o)] .
  eq < CR : Ensemble | features : KPS, subcomponents : transOut(NDL,PNS) KCS >
   = < CR : Ensemble | features : KPS transOut(NDL,PNS), subcomponents : KCS > .
  eq transOut(NDL, (P, PNS)) < P : OutPort | content : DL >
   = transOut(NDL, PNS) < P : OutPort | content : DL NDL > .
  eq transOut(NDL, empty) = none .
endom)

--- defining the transferInputs and transferResults functions
(omod TRANSFER-DATA is
  protecting TRANSFER-FUNCTIONS .

  var CR : ComponentRef . vars P : FeatureId .
  var D : DataContent . vars DL DL' : List{DataContent} . var NDL : NeList{DataContent} .
  var PORTS COMPS : Configuration . var CONXS : Set{Connection} .
  var NPS : NeSet{FeatureRef} . var ICTB OCTB : ConxTable .

--- transfer ensemble inputs and feedback outputs
  op transferInputs : Object ~> Object .
  eq transferInputs(
     < CR : Ensemble | features : PORTS,
                    subcomponents : COMPS, connections : CONXS >)
   =
     < CR : Ensemble | features : transEnvIn(PORTS,inner-tb(CONXS)),
                    subcomponents : transFBOut(COMPS,inner-tb(CONXS)) > .
  --- 1. transfer first inputs from the ensemble's input ports to subcomponents.
  op transEnvIn : Configuration ConxTable ~> Configuration .
  eq transEnvIn(< P : InPort | content : D DL > PORTS, (P |-> NPS) ICTB)
   = transIn(D,NPS) transEnvIn(< P : InPort | content : DL > PORTS,ICTB) .
  eq transEnvIn(PORTS, ICTB) = PORTS [owise] .

  --- 2. transfer feedback outputs between subcomponents.
  op transFBOut : Configuration ConxTable ~> Configuration .
  eq transFBOut(
      < CR : Component | features : < P : OutPort | content : NDL > PORTS >
      COMPS, (CR .. P |-> NPS) ICTB)
   =
    transIn(NDL,NPS)
    transFBOut(
      < CR : Component | features : < P : OutPort | content : nil > PORTS >
      COMPS, ICTB) .
  eq transFBOut(COMPS,ICTB) = COMPS [owise] .

--- transfer outputs to the ensemble' output ports
  op transferResults : Object ~> Object .
  eq transferResults(< CR : Ensemble | subcomponents : COMPS, connections : CONXS >)
   = < CR : Ensemble |
        subcomponents : transEnvOut(COMPS, outer-tb(CONXS), inner-tb(CONXS)) > .

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

--- The top-level synchronous step
(tomod MODEL-TRANSITION-SYSTEM is
  including THREAD-DYNAMICS .
  including ENSEMBLE-DYNAMICS .
  including BUILTIN-INPUT-ADAPTOR-SEMANTICS .
  including COLLAPSE-SINGLE-RATE .

  var C : ComponentId . var PROPS : PropertyAssociation .
  var T : Time . var OBJ : Object .

--- Assume that there is no port in the top-level component
 crl [step]:
     {< C : System | features : none,
                  properties : (TimingProperties::Period => {T}) ;
                               (MRSynchAADL::Synchronous => {true}) ; PROPS >}
   =>
     {OBJ} in time T
   if executeStep(< C : System | >) => OBJ .
endtom)

--- the semantics of the requirement specification language
(tomod AADL-LTL-PROPOSITION is
  including MODEL-TRANSITION-SYSTEM .
  including TIMED-MODEL-CHECKER .
  var CR : ComponentRef . var P : FeatureId .
  vars COMPS PORTS PORTS' REST : Configuration .
  var FMAP : FeatureMap . var PROPS : PropertyAssociation .
  var PATH : ComponentRef . var L : Location .
  var E : Expression . var V : Value . var DCL : List{DataContent} .

--- component data expressions
  op _|_ : ComponentRef Expression -> Prop [ctor] .

  eq {< CR : Ensemble | subcomponents : COMPS >} |= PATH | E
   = lookupExp(COMPS, PATH, E) .

  op lookupExp : Configuration ComponentRef Expression -> Bool .
  eq lookupExp(< CR : Ensemble | subcomponents : COMPS > REST, CR . PATH, E)
   = lookupExp(COMPS, PATH, E) .
  eq lookupExp(< CR : Component | features : PORTS,
                           subcomponents : COMPS, properties : PROPS > REST, CR, E)
   = eval(E, empty | feedbackOutputs(PORTS,empty) | COMPS | PROPS) == [true] .
  eq lookupExp(REST, PATH, E) = false [owise] .

  --- return the last value in a given feedback output port
  op feedbackOutputs : Configuration FeatureMap -> FeatureMap .
  eq feedbackOutputs(< P : OutPort | content : DCL V > PORTS, FMAP)
   = feedbackOutputs(PORTS, insert(P, V, FMAP)) .
  eq feedbackOutputs(PORTS, FMAP) = FMAP [owise] .

--- thread states
  op _@_ : ComponentRef Location -> Prop [ctor] .
  eq {< CR : Ensemble | subcomponents : COMPS >} |= PATH @ L
   = lookupState(COMPS, PATH, L) .

  op lookupState : Configuration ComponentRef Location -> Bool .
  eq lookupState(< CR : Ensemble | subcomponents : COMPS > REST, CR . PATH, L)
   = lookupState(COMPS, PATH, L) .
  eq lookupState(< CR : Thread | currState : L > REST, CR, L) = true .
  eq lookupState(REST, PATH, L) = false [owise] .
endtom)
```