View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

Synchronous AADL and its Formal Analysis in Real-Time Maude

Kyungmin Bae[1], Peter C. Ölveczky[2], Abdullah Al-Nayeem[1], and José Meseguer[1]

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

Abstract. Distributed Real-Time Systems (DRTS), such as avionics systems and distributed control systems in motor vehicles, are very hard to design because of asynchronous communication, network delays, and clock skews. Furthermore, their model checking typically becomes unfeasible due to the large state spaces caused by the interleavings. For many DRTSs, we can use the PALS methodology to reduce the problem of designing and verifying asynchronous DRTSs to the much simpler task of designing and verifying their synchronous versions. AADL is an industrial modeling standard for avionics and automotive systems. We define in this paper the *Synchronous AADL* language for modeling synchronous real-time systems in AADL, and provide a formal semantics for Synchronous AADL in Real-Time Maude. We have integrated into the OSATE modeling environment for AADL a plug-in which allows us to model check Synchronous AADL models in Real-Time Maude within OSATE. We exemplify such verification on an avionics system, whose Synchronous AADL design can be model checked in less than 10 seconds, but whose asynchronous design cannot be feasibly model checked.

1 Introduction

Many real-time systems are distributed due to physical and fault-tolerance requirements. Designing, implementing, and verifying such systems is very difficult and costly. Due to their asynchrony, clock skews, and message delays, such systems may experience race conditions and violations of their safety properties that can be very difficult to uncover by testing. The alternative of automated formal verification by model checking is also unfeasible in practice. The reason is that, due to asynchrony, there are *too many interleavings*, leading to a veritable combinatorial explosion.

The above remarks apply to the verification of both designs and code. Since design errors are much more expensive than coding errors, there is general agreement that system verification should first be carried out at the level of designs by verifying the *model* representing a system design. For real-time systems such as avionics and automative systems the AADL modeling language [17] is a widely used industrial standard. To make *formal* verification of AADL models possible

two key problems have to be solved. The first problem is to have a *formal semantics* of AADL models, since without such a semantics there is no *mathematical* model satisfying any properties so that claims about satisfaction of such properties are literally meaningless. We have addressed this problem by providing a formal semantics in rewriting logic for a behavioral fragment of AADL [14], and other researchers have carried our related efforts [3, 4, 14, 2, 9]. The second problem is the one mentioned above: since AADL models have components that interact asynchronously with each other, their automatic model checking verification becomes *unfeasible* even for simple models. To illustrate the difficulty with a concrete example, using the tool presented in [14], we were unable to model check the AADL model of the avionics system presented in Section 4 with the Real-Time Maude model checker. The key point is that the inherent difficulties of verifying a distributed real-time system (DRTS) *do not disappear* at the level of models: they are common to both models and code. This leaves us with the unsolved problem of how to formally verify AADL models *in practice*.

To address the problem of drastically reducing the difficulties of designing and verifying DRTSs, we and other colleagues at Rockwell-Collins and UIUC have proposed the PALS transformation [1, 13, 11]. The key idea behind PALS is that the intended behavior of many DRTSs is that they should be *virtually synchronous*. That is, conceptually there is a logical period during which all components perform a transition and send data to each other. The PALS transformation (summarized in Section 2) achieves this virtual synchrony by reducing the design and verification of a DRTS of this nature to that of a much simpler *synchronous* one which is semantically equivalent to it [11], both systems are *bisimilar* [11] and therefore satisfy the same temporal logic properties.

Our Approach and Contributions. Motivated by the PALS ideas, our approach to solving the problem of verifying in practice AADL models that are distributed but virtually synchronous is based on the following ideas and contributions: (i) to specify a fragment of AADL called *Synchronous AADL* in which synchronous models can be defined; (Section 3); (ii) to define a formal *synchronous semantics* for this subset (Section 5); (iii) to embody this semantics in a tool called *SynchAADL2Maude* (Section 8), which is an OSATE plugin and maps models in Synchronous AADL to rewrite theories in Real-Time Maude, where such models can be simulated and formally verified by model checking (Section 6); (iv) to illustrate the effectiveness of the approach by modeling in Synchronous AADL and verifying in *SynchAADL2Maude* the same avionics example that we could not verify in its simplest possible asynchronous version: each of the formal requirements for this example can now be verified in 10 seconds or less (Section 7).

Using the PALS transformation it is then possible to transform a synchronous AADL model into a correct-by-construction asynchronous one. In fact, an OSATE implementation of PALS for AADL which can be used for this purpose has already been developed in [1]. Although the contributions (i)–(iv) were motivated by PALS, their usefulness is not restricted to PALS: they can be exploited by similar transformations relating synchronous and asynchronous systems for

other distributed real-time architectures, such as the time-triggered architecture (see, e.g., [16, 10]).

Our experience of a huge state space reduction from the verification of an asynchronous system to that of its synchronous counterpart is not specific to AADL models: for the same avionics example a similar drastic reduction was reported by Darren Cofer and Steven Miller in [13] using SMV models. We conducted a similar experiment modeling the same example directly in Real-Time Maude: the number of states of the synchronous system was 185, and all properties were verified in 0.8 seconds or less, but the simplest possible asynchronous model (no network delays, no execution time, no clock skews) had 3,047,832 states. Although it was possible to verify a property of the asynchronous system in 2000 seconds, as soon as the possibility of a one-unit delay was allowed for messages, all model checking verification became impossible. What this work achieves is to make such a huge reduction possible for AADL models to support their automatic model checking verification.

2 Preliminaries on AADL, Real-Time Maude, and PALS

AADL. The *Architecture Analysis & Design Language* (AADL) [17] is an industrial modeling standard used in avionics, aerospace, automotive, medical devices, and robotics communities 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: *thread* components modeling 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. Thread behavior is described using the *behavior annex* [8], which models thread behaviors as transition systems with local state variables. The OSATE modeling environment provides a set of Eclipse plug-ins for AADL. The current stable version 1.5.8 of OSATE, that also supports the behavior annex, supports version 1 of the AADL standard; therefore, the language Synchronous AADL defined in this paper is also based on version 1 of AADL.

In the software component subset of AADL that is the focus of this paper, a component *type* specifies the component's *interface* 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. Components may have *properties* describing its parameters and other information. 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.

The behavior of a thread is defined by guarded state transitions. The actions performed when a transition is applied may update local variables, generate new outputs, and/or suspend the thread for some time. Actions are built from such 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 is suspended).

Real-Time Maude. A Real-Time Maude [15] *timed module* specifies a *real-time rewrite theory* of the form $(\Sigma, E, \mathit{IR}, \mathit{TR})$, where:

- (Σ, E) is a *membership equational logic* [5] theory with Σ a signature[3] and E a set of *confluent and terminating conditional equations.* (Σ, E) specifies the system's states as an algebraic data type.
- *IR* is a set of (possibly conditional) *labeled instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) local transitions.[4]
- *TR* is a set of conditional *tick rewrite rules* of the form `crl [`l`] : {`u`} => {`v`} in time` τ `if` *cond*. Such a rule specifies a transition with duration τ and label l from an instance of the term u to the corresponding instance of the term v.

The Real-Time Maude syntax is fairly intuitive (see [5]). A function symbol f is declared with the syntax `op` f `:` $s_1 \dots s_n$ `->` s, where $s_1 \dots s_n$ are the sorts of its arguments, and s is its (value) *sort*. Equations are written with syntax `eq` u `=` v, and `ceq` u `=` v `if` *cond* for conditional equations. The mathematical variables in such statements are declared with the keywords `var` and `vars` (see [5]). We refer to [5] for more details on the syntax of Real-Time Maude. 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 subterm $f(\dots)$ only if no other equation with left-hand side $f(u_1, \dots, u_n)$ can be applied.[5]

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 the object's *identifier*, and where 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.

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

[4] 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.

[5] A specification with `owise` equations can be transformed to an equivalent system without such equations [5].

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, starting with a given initial state, *up to a certain duration*. It is 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. The command which searches for n states satisfying the *pattern* search criterion has syntax

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

Real-Time Maude's *linear temporal logic model checker* checks whether each behavior from an initial state, possibly up to a time bound, satisfies a linear temporal logic formula. The time-bounded model checking command has syntax

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

for initial state t and temporal logic formula *formula* . *State propositions*, possibly parametrized, are operators of sort `Prop`. Their semantics should be given by equations of the form

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

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`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), `U` ("until"), and `O` ("next"). The command `(mc` t `|=u` φ `.)` then checks whether the temporal logic formula φ holds in all behaviors starting from the initial state t.

PALS. In many systems targeted by AADL, such as avionics systems and distributed control systems in motor vehicles, the system design is essentially a *synchronous design* that must be realized in an asynchronous distributed setting. The design and verification of such distributed real-time systems is a challenging and error-prone task because of asynchronous communication, network delays, clock skews, and because the state space explosion caused by the system's concurrency can make it unfeasible to apply model checking to verify required properties.

The key idea of the *PALS architectural pattern* [11, 13, 1, 18] is to reduce the design and verification of a distributed real-time system to that of its much simpler synchronous version when the network infrastructure guarantees bounds on the messaging delays and the skews of the local clocks. For a synchronous design *SD* and network bounds Γ, PALS defines the corresponding asynchronous distributed design $PALS(SD, \Gamma)$. In [11, 12] we formalize PALS and prove that the

typically unfeasible task of model checking the asynchronous design $PALS(SD, \Gamma)$ reduces to the feasible task of model checking the synchronous design SD.

The systems we target are made up of components that communicate asynchronously and must change state and respond to environment inputs within hard real-time bounds. The synchronous PALS model of computation is therefore formalized as the synchronous composition of a collection of deterministic *typed machines*, a nondeterministic *environment*, and a *wiring diagram* that connects the machines:

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 function*, a function* $\delta_M : (D_i \times S) \rightarrow (S \times D_o)$*.*

That is, a machine 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}.

Typed machines can be "wired together" into arbitrary sequential and parallel compositions by means of a "wiring diagram," as shown in Fig. 1.

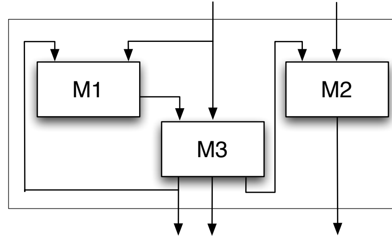

Fig. 1. A machine ensemble.

Definition 2. *A (typed)* machine ensemble $\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j \in J}, E, src)$ *has:*

- J*, a nonempty finite set of* indices*, and* $e \notin J$ *the* environment index*.*
- $\{M_j\}_{j \in J}$*, a* J*-indexed family of typed machines.*
- E*, called an* environment*, is an ordered pair* $E = (D_i^e, D_o^e)$*, where* D_i^e*, the* inputs *to the environment, is a nonempty set* $D_i^e = D_{i_1}^e \times \cdots \times D_{i_{n_e}}^e$ *and* D_o^e*, the environment's* output set*, is a nonempty set* $D_o^e = D_{o_1}^e \times \cdots \times D_{o_{m_e}}^e$*.*
- src*, a function that assigns to each input port* (j, n) *(the input port number* n *of machine* j*) its "source" output port* $src(j, n)$*. Formally, we define the set of input ports and output ports as follows:*
  - $In_{\mathcal{E}} = \{(j, n) \in (J \cup \{e\}) \times \mathbb{N} \mid 1 \leq n \leq n_j\}$*, where* M_j *has* n_j *inputs*
  - $Out_{\mathcal{E}} = \{(j, n) \in (J \cup \{e\}) \times \mathbb{N} \mid 1 \leq n \leq m_j\}$*, where* M_j *has* m_j *outputs.*

  Then src *is a surjective function* $src : In_{\mathcal{E}} \rightarrow Out_{\mathcal{E}}$ *assigning to each input port the output port to which it is connected, and such that "the types match". That is, if we denote by* $D_{i_k}^j$ *the set of data allowed as input in the* k*th input port of machine* M_j *(resp.* k*th input port of the environment if* $j = e$*), and same with output ports, then if* $src(j, q) = (k, l)$ *we should have* $D_{o_l}^k \subseteq D_{i_q}^j$*. In addition, there is no self-loop from the environment to itself.*

An ensemble $\mathcal{E}$ has a *lock-step synchronous semantics*, in the sense that the transitions of all machines are performed simultaneously, and whenever a machine has a feedback wire to itself and/or to any other machine, then the corresponding output becomes an input for any such machine at the *next* step. We assume an environment where the *constraints on the values generated by the environment* can be defined as a *satisfiable* predicate $c_e : D_o^e \rightarrow Bool$ so that $c_e(d_1, \ldots, d_{m_e})$ is *true* if and only if the environment can generate output $(d_1, \ldots, d_{m_e})$. We can associate a transition system defining the behaviors of a machine ensemble that operates in an environment as follows.

Definition 3. *Given a machine ensemble* $\mathcal{E}$ *with environment constraint* c_e*, the corresponding transition system is defined as a pair* $\mathcal{E}_{c_e} = (S^{\mathcal{E}} \times D_i^{\mathcal{E}}, \longrightarrow_{\mathcal{E}_{c_e}})$*, where the transition relation* $\longrightarrow_{\mathcal{E}_{c_e}}$ *is defined by* $(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{\mathcal{E}_{c_e}} (\boldsymbol{s'}, \boldsymbol{i'})$ *iff a machine ensemble in state* $\boldsymbol{s}$ *and with input* $\boldsymbol{i}$ *from the environment has a transition to state* $\boldsymbol{s'}$*, and the environment can generate output* $\boldsymbol{i'}$ *in the next step:*

$$(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{\mathcal{E}_{c_e}} (\boldsymbol{s'}, \boldsymbol{i'}) \iff \exists \boldsymbol{o} \; \delta_{\mathcal{E}}(\boldsymbol{i}, \boldsymbol{s})) = (\boldsymbol{s'}, \boldsymbol{o}) \wedge c_e(\boldsymbol{i'}).$$

3 Synchronous AADL

This section defines the *Synchronous AADL* language that can be used to model synchronous designs in AADL, including both synchronous PALS designs and other synchronous designs that can be mapped onto different distributed real-time architectures, such as the time-triggered architecture [10]. We have focused on making Synchronous AADL intuitive for the AADL expert by defining Synchronous AADL as an annotated sublanguage of AADL, in which the execution of each thread in each "round" is independent of the other threads, and where output generated by a thread in a round is available as input at the receiving thread exactly at the beginning at the next "round." In AADL, such threads would be executed asynchronously. However, since the threads are independent of each other in each round, the "final" states in each round are the same in both any asynchronous execution and in a synchronous execution; the latter just ignores the "intermediate" asynchronous states. Therefore, all AADL constructs in the subset have the same meaning in AADL and Synchronous AADL. Synchronous AADL also adds a *property set* `SynchAADL` to declare Synchronous AADL-specific properties as explained below.

Since Synchronous AADL is intended to model synchronous designs, as opposed to their often asynchronous implementations, it ignores the hardware, memory, and scheduling features of AADL. Synchronous AADL therefore focuses on the behavioral and structural subset of AADL, namely, hierarchical system, process, and thread components, ports and connections, and thread behaviors defined in the *behavior annex* standard. We next discuss the definition of Synchronous AADL. Section 3.1 discusses the design of Synchronous AADL, and Section 3.2 explains how a Synchronous AADL model defines a PALS typed machine ensemble.

3.1 The Definition of Synchronous AADL

Dispatch. The dispatch protocol is used to trigger an execution of an AADL thread. A *periodic* thread is dispatched at the beginning of each new time period of the thread. In the aperiodic, sporadic, timed, and hybrid dispatch protocols, the thread is dispatched when it receives an *event*. Such *event-triggered* dispatch is not suitable to define a system in which all threads (with a possible exception for the environment thread) could be seen as executing in lock-step, since the sending thread triggers the execution of the receiving thread, which would read in its ith round the output generated by the sender in the same round.[6] Therefore, each thread must have *periodic* dispatch; furthermore, since each thread must execute in each round, the *period* of all the threads must be the same.

Communication. There are three kinds of ports in AADL: *data*, *event*, and *event data* ports. Event and event data ports can be used to dispatch event-triggered threads. To have only AADL constructs that define "synchronous behaviors," the communication primitives must ensure that all output generated in an iteration is available to the receiver at the beginning of the next iteration, *and not earlier*.

Version 1 of AADL has two kinds of semantic *data* connections: *immediate* and *delayed* connections. According to [6, Section 8.1.5], an immediate connection imposes an execution order on threads with the same dispatch times: the source thread must execute before the destination thread. Not only would this break the intended "lock-step" synchronous execution, but it also implies that the destination of an immediate connection uses the output that is generated *in the same round*, which violates our requirements of synchronous models. For a *delayed* port connection,

> "the value from the sending thread is transmitted at its deadline and is available to the receiving thread at its next dispatch. [...] If the deadline of the sending thread and the dispatch of the receiving thread occur simultaneously the transmission occurs at that instant." [6, Section 8.1.5].

In our setting, where all threads have periodic dispatch with the same period, the deadline of the previous round and the dispatch of the next round coincide. Therefore, the output generated in an iteration will be available to the receivers at the start of the next iteration. Since only data ports have delayed connections, and since event-triggered dispatches are excluded, only *data* ports are used in Synchronous AADL, and all connections between non-environment threads must be delayed.

Execution Times. Since the components execute in lock-step, it is natural to assume that they use the same time to perform their execution. This is, however, not strictly necessary, since their executions are independent during the period,

[6] One could think of using event ports from the *environment* threads, but then we would need an event connection to each other thread in the system, which is undesired and therefore not included in Synchronous AADL.

so that they may in principle have different durations. For simplicity, and since the PALS synchronous model is untimed, we assume that the thread executions are instantaneous.

Deterministic Threads. In the systems targeted by PALS and Synchronous AADL, the nodes that communicate with the environment are invariably deterministic. This is reflected in Definition 1, where the behavior of a typed machine is deterministic. We therefore assume that the transition system defining the behavior of a non-environment thread is deterministic, and that each such thread has the property:

```
SynchAADL::Deterministic => true
```

Environment Thread. In PALS, the *environment thread* generates output nondeterministically in each iteration. The possible outputs can often be defined by an *environment constraint* c_e so that $c_e(\boldsymbol{o})$ is *true* if and only if the environment can nondeterministically generate output $\boldsymbol{o}$ in any iteration. The property

```
SynchAADL::IsEnvironment => true
```

denotes that the thread is an environment thread, and the property

```
SynchAADL::InputConstraints => ("Boolean formula")
```

defines an input constraint on a set of Boolean-valued outputs. We assume that a Synchronous AADL system has *at most one environment thread*.

It seems natural to regard the system as responding to the *current* environment output. We therefore support only *immediate* connections *from* the environment. According to the AADL semantics, this forces the environment to execute before the other nodes in each round.

Declaring Synchronous Systems. The top-level `system` component declares the entire system to execute synchronously by declaring the property

```
SynchAADL::Synchronous => true
```

It is also useful to declare the period of the entire system at the top level in the component hierarchy by declaring the property

```
SynchAADL::SynchPeriod => p
```

3.2 Correspondence with the PALS Synchronous Model

Each non-environment thread T in a Synchronous AADL model defines a typed machine M_T in the expected way. Since, in contrast to a typed machine, a thread may not always output a value to all output ports, the "state" S_{M_T} of the typed machine M_T also contains, for each input port, the value of the input that was last received by that port, and thus has the form $(s, v, \boldsymbol{inp})$, where s is a complete

state or initial state in the thread's transition system, v is a valuation of its local variables, and $\boldsymbol{inp}$ is a vector with the latest value received by each input port.

For each domain D of an input port, let D^* denote the domain D extended with an element '$*$' denoting "no input." Given an ordering $p_1, \ldots, p_n$ of the input ports of thread T, the input set of the typed machine M_T is the corresponding set $D^*_{i_1} \times \cdots \times D^*_{i_n}$. The output set of M_T can be defined similarly. Then, $\delta_{M_T}(\boldsymbol{i}, (s, v, \boldsymbol{inp})) = ((s', v', \boldsymbol{inp'}), \boldsymbol{o})$ iff the transition system of T, when starting from local state s with the values of local variables given by v, and having input $\boldsymbol{i}$ in its input ports (some of these may be '$*$'), and having stored $\boldsymbol{inp}$ in its data input ports, can perform a series of transitions, leading to a complete state s' and to the variables having values v', and the output ports having been assigned output values $\boldsymbol{o}$, so that in the transitions performed, a complete state was not reached in fewer steps, and where the $\boldsymbol{inp'}$ is the updated values of the input ports (which are unchanged iff the input to that port was '$*$').

A Synchronous AADL model can be seen as a machine ensemble in the obvious way, where the semantic connections define the wiring. Likewise, the environment constraint c_e is given in the input constraint property of the environment thread. The form of the single transition of environment threads then ensures that the appropriate values can be sent from the environment.

4 Avionics Examples

This section illustrates the use of Synchronous AADL on two related avionics case studies.

4.1 An Active Standby System

We first exemplify Synchronous AADL with fragments of a model of an avionics system based on a specification by Steve Miller and Darren Cofer at Rockwell-Collins [13]. A full description of this model is given in Appendix A.

In *integrated modular avionics* (IMA), a cabinet is a chassis with a power supply, internal bus, and general purpose computing, I/O, and memory cards. Aircraft applications are implemented using the resources in the cabinets. There are always two or more cabinets that are physically separated on the aircraft so that physical damage does not take out the computer system. The *active standby* system considers the case of two cabinets and focuses on the logic for deciding which side is *active*. Each side can fail, and a failed side can recover after failure. In case one side fails, the non-failed side should be the active side. In addition, the pilot can toggle the active status of these sides. The full functionality of each side depends on the two sides' perception of the availability of other system components. The architecture of the system is shown in Figure 2. Each time Environment dispatches, it nondeterministically sends 5 Boolean values, one through each ports, with the constraint that two sides cannot fail at the same time. Therefore, in each round, the environment can send any one of 24 different 5-tuples.

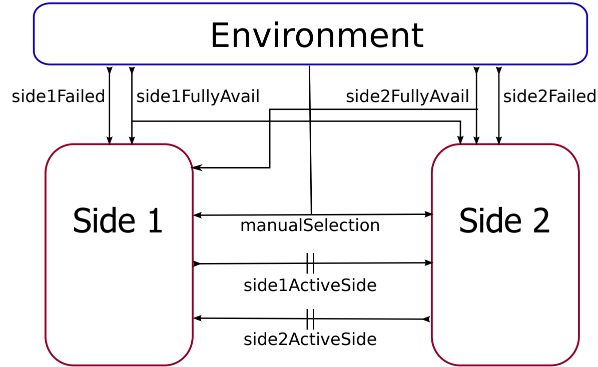

Fig. 2. The architecture of the active standby system.

The Synchronous AADL Model. The following top-level system component implementation declares the architecture of the system, with the three subcomponents `sideOne`, `sideTwo`, and `env`, and with immediate data connections (denoted by the arrow '`->`') from the environment to the two sides, and with delayed data connections ('`->>`') between the two sides (parts of the model are replaced by '`...`'):

```
system implementation ActiveStandbySystem.impl
  properties
    SynchAADL::Synchronous => true;
    SynchAADL::SynchPeriod  => 2 ms;
  subcomponents
    sideOne: system Side1.impl;
    sideTwo: system Side2.impl;
    env: system Environment.impl;
  connections
    data port sideOne.side1ActiveSide ->> sideTwo.side1ActiveSide;
    data port sideTwo.side2ActiveSide ->> sideOne.side2ActiveSide;
    data port env.side1FullyAvail -> sideOne.side1FullyAvail;
    data port env.side1FullyAvail -> sideTwo.side1FullyAvail;
    ...
    data port env.side2Failed -> sideTwo.side2Failed;
end ActiveStandbySystem.impl;
```

The environment component `env` is an instance of the following system implementation, which declares a similar *process* subcomponent `envProcess`:

```
system Environment
  features
     side1FullyAvail: out data port Behavior::boolean;
     side2FullyAvail: out data port Behavior::boolean;
     manualSelection: out data port Behavior::boolean;
     side1Failed: out data port Behavior::boolean;
     side2Failed: out data port Behavior::boolean;
end Environment;

system implementation Environment.impl
  subcomponents
```

```
    envProcess: process EnvironmentProcess.impl;
  connections
    data port envProcess.side1FullyAvail -> side1FullyAvail;
    data port envProcess.side2FullyAvail -> side2FullyAvail;
    data port envProcess.manualSelection -> manualSelection;
    data port envProcess.side1Failed -> side1Failed;
    data port envProcess.side2Failed -> side2Failed;
end Environment.impl;
```

We do not show the implementation of `envProcess`, which contains a thread instance `envThread`, ports, and connections from the output ports of `envThread` to the corresponding output ports of `envProcess`. The thread `envThread` defining the environment behavior is an instance of the following component:

```
thread EnvironmentThread
  features
    side1FullyAvail: out data port Behavior::boolean;
    side2FullyAvail: out data port Behavior::boolean;
    manualSelection: out data port Behavior::boolean;
    side1Failed: out data port Behavior::boolean;
    side2Failed: out data port Behavior::boolean;
end EnvironmentThread;

thread implementation EnvironmentThread.impl
  properties
    SynchAADL::InputConstraints => ("not (s1F and s2F)");
    SynchAADL::IsEnvironment => true;
    Dispatch_Protocol => Periodic;
    Period => 2 ms;
  annex behavior_specification {**
    states
      s0 : initial complete state;
    state variables
      s1FA: Behavior::boolean;
      s2FA: Behavior::boolean;
      mS: Behavior::boolean;
      s1F: Behavior::boolean;
      s2F: Behavior::boolean;
    transitions
      s0 -[]-> s0 {
        side1FullyAvail := s1FA;
        side2FullyAvail := s2FA;
        manualSelection := mS;
        side1Failed := s1F;
        side2Failed := s2F; };
**};
end EnvironmentThread.impl;
```

The transition system has a single transition, that in each dispatch sends the values of the state variables `s1FA`, `s2FA`, `mS`, `s1F`, and `s2F` to the corresponding output ports. These variables can be assigned any values that satisfy the constraint `not (s1F and s2F)` that states that side 1 and side 2 cannot both fail at the same time.

The following component defines the behavior of side 1:

```
thread Side1Thread
  features
    side2ActiveSide: in data port Behavior::integer;
```

```
    manualSelection: in data port Behavior::boolean;
    side1Failed: in data port Behavior::boolean;
    side1FullyAvail: in data port Behavior::boolean;
    side2FullyAvail: in data port Behavior::boolean;
    side1ActiveSide: out data port Behavior::integer;
end Side1Thread;

thread implementation Side1Thread.impl
  properties
    SynchAADL::Deterministic => true;
    Dispatch_Protocol => Periodic;
    Period => 2 ms;
  annex behavior_specification {**
    states
      preInit: initial complete state;
      initState : complete state;
      side1FailedState : complete state;
      side2FailedState : complete state;
      side1WaitState : complete state;
      side1ActiveState : complete state;
      side2ActiveState : complete state;
    state variables
      prevSide2ActiveStatus: Behavior::integer;
      prevManualSwitch: Behavior::boolean;
    initially
      prevSide2ActiveStatus := 0;
      prevManualSwitch := false;
    transitions
      ...
      side2ActiveState  -[side1Failed = false and
                          side2ActiveSide != 0 and
                          side1FullyAvail = true and
                          ((prevManualSwitch = false and
                            manualSelection = true)
                           or side2FullyAvail = false) ]-> side1ActiveState {
         side1ActiveSide := 1;
         prevSide2ActiveStatus := side2ActiveSide;
         prevManualSwitch := manualSelection; };
      ...
    **};
end Side1Thread.impl;
```

We show only one of the 20 transitions in this thread. The transition takes the thread from state `side2ActiveState` to state `side1ActiveState` if the input received in the `side1Failed` port is `false`, the value received in the port `side2ActiveSide` is different from 0, etc. As a result of applying the transition, the value 1 is sent through the output port `side1ActiveSide`, and the local variables `prevSide2ActiveStatus` and `prevManualSwitch` are assigned the values received in the ports `side2ActiveSide` and `manualSelection`, respectively.

4.2 A Three–Node Active Standby System

We also have modeled in Synchronous AADL another version of the active standby example with 3 components; this system can be used to control the aileron[7] or the rudder of an aircraft. The main difference between this example

[7] A hinged flight control surface attached to the trailing edge of an airplane wing, used to control lateral balance.

and the above active standby system is that the 3-node system uses the synchronous execution semantics to generate a consistent view of the global state. A *view* is represented in this example as the active/standby status of three sides perceived at each side. Each component then computes its active-standby status based on the current view.

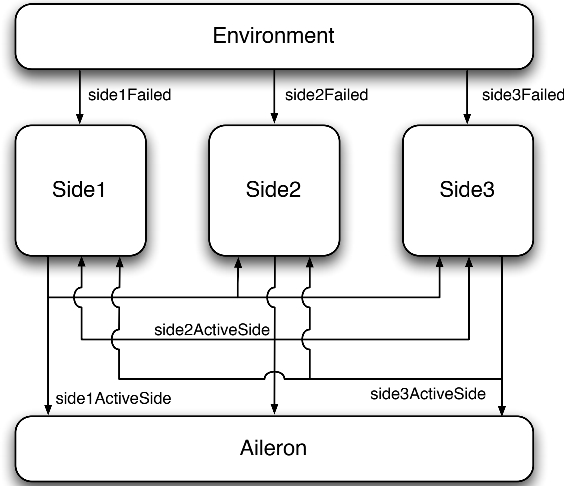

Fig. 3. The architecture of the 3-node active standby system.

The actuator, e.g. the aileron, uses the control command from an active side. The standby sides' commands are not used in normal condition when there is an active side. The most important requirement in this design is to guarantee the operational safety of the system. In order to guarantee stability, the design must satisfy that a new active side will be selected within a small number of steps after the failure of an active side. We also require that there will be no state at which more than one sides are active. This requirement is needed to simplify the logic of the actuator. Finally, this design specification requires that consistent view of the active-standby status of 3 sides at each step.

There are a total of 5 components in the system: `sideOne`, `sideTwo`, `sideThree`, `environment`, and `aileron`. The three sides (`sideOne`, `sideTwo`, and `sideThree`) agree upon which side is active. They forward their outputs to `aileron`. Based on these outputs, the aileron then selects the control output from the active side. In this example, `environment` is the `Environment` that injects failure into the components by sending 3 Boolean data nondeterministically, with the constraint that all sides cannot fail at the same time. Unlike the previous example, there is no input for the availability status and manual selection.

```
system implementation MainSystem.impl
```

```
  properties
  SynchAADL::Synchronous => true;     SynchAADL::SynchPeriod  => 2 ms;
  subcomponents
    sideOne: system Side1::Side1.impl;
    sideTwo: system Side2::Side2.impl;
    sideThree: system Side3::Side3.impl;
    environment: system Environment::Environment.impl;
    aileron: system Aileron::IOTask.impl;
  connections
    data port sideOne.side1ActiveSide ->> sideTwo.side1ActiveSide;
    data port sideOne.side1ActiveSide ->> sideThree.side1ActiveSide;
    data port sideTwo.side2ActiveSide ->> sideOne.side2ActiveSide;
    data port sideTwo.side2ActiveSide ->> sideThree.side2ActiveSide;
    data port sideThree.side3ActiveSide ->> sideOne.side3ActiveSide;
    data port sideThree.side3ActiveSide ->> sideTwo.side3ActiveSide;
    data port environment.side1Failed -> sideOne.side1Failed;
    data port environment.side2Failed -> sideTwo.side2Failed;
    data port environment.side3Failed -> sideThree.side3Failed;
    data port sideOne.side1ActiveSide ->> aileron.side1ActiveSide;
    data port sideTwo.side2ActiveSide ->> aileron.side2ActiveSide;
    data port sideThree.side3ActiveSide ->> aileron.side3ActiveSide;
end MainSystem.impl;
```

The active-standby logic of each side has two state transitions. We give the specification of the behavior of Side3 and one of its state transition below. In this design, the output of each side is determined by the local view of the active/standby status of three sides. This local view is represented by three state variables `g_side1`, `g_side2`, and `g_side3`. The synchronous behavior of this system guarantees that the local views are identical at each side so that all sides can be consistent. In the transition system, Side 3 becomes active (i.e. `g_side1` = 3) if Side1 and Side2 are failed. We use *if-elsif-endif* conditionals of the behavior annex to simplify the logic. The 'fresh' values of `side1ActiveSide` and `side2ActiveSide` are assigned to `g_side1` and `g_side2`. The value of `g_side3` is computed at the previous step.

```
thread implementation ActiveStandbyThread.impl
  annex behavior_specification {**
    state variables
      g_side1: behavior::integer;
      g_side2: behavior::integer;
      g_side3: behavior::integer;
      prev_gside3: behavior::integer;
    initially
      g_side1 := -1; g_side2 := -1; g_side3 := -1;
      prev_gside3 := -1;
    states
      s0 : initial complete state;
    transitions
      ...
      s0 -[ on side3Failed'fresh and side3Failed = false ]-> s0 {
        g_side3 := prev_gside3;
        if (side1ActiveSide'fresh)
          g_side1 := side1ActiveSide;
        else
          g_side1 := -1;
        end if;

        if (side2ActiveSide'fresh)
```

```
          g_side2 := side2ActiveSide;
        else
          g_side2 := -1;
        end if;

        if (g_side3 = -1)
          side3ActiveSide := 0;
          prev_gside3 := 0;
        elsif (g_side3 != (-1) and g_side1 = (-1) and g_side2 = -1)
          side3ActiveSide := 1;
          prev_gside3 := 1;
        else
          side3ActiveSide := prev_gside3;
        end if;
      };
   **};
end ActiveStandbyThread.impl;
```

The behavior of the other sides is specified in a similar way; see Appendix B.

5 Real-Time Maude Semantics of Synchronous AADL

This section summarizes the Real-Time Maude semantics of Synchronous AADL. Since we have previously defined the Real-Time Maude semantics of a subset of standard behavioral AADL models, we also refer to [14] for details about the Real-Time Maude representation of AADL models, and focus on Synchronous AADL-specific features of the semantics.

We first show how Synchronous AADL models are represented in Real-Time Maude and then formalize the synchronous behaviors of such models.

5.1 Representing Synchronous AADL Models in Real-Time Maude

The semantics of a component-based language can naturally be defined in an object-oriented style, where each component instance is modeled as an object. The hierarchical structure of Synchronous AADL components is reflected in the nested structure of objects, in which an attribute of an object contains its subcomponents as a multiset of objects. Any Synchronous AADL component instance is represented as an object instance of a subclass of the following class `Component`, which contains the attributes common to all kinds of components:

```
class Component | features : Configuration,
                  subcomponents : Configuration,
                  properties : Properties,
                  connections : ConnectionSet .
```

The attribute `features` denotes the ports of a component instance, represented as a multiset of `Port` objects; `subcomponents` denotes the subcomponents of the object; `properties` denotes its *properties*; and `connections` denotes its connections.

In Synchronous AADL, the classes `System` and `Process`, denoting system and process components, do not have other attributes than those they inherit from their `Component` superclass. The `Thread` class is declared as follows:

```
class Thread | behaviorRef : ComponentRef,
               variables : Valuation,
               currState : Location,
               completeStates : LocationSet .
subclass Thread < Component .
```

Since the term representing the transitions of a thread can be fairly large, and since the transitions do not change dynamically, we do not carry them around in the objects. Instead, a `memo`-ized (see [5]) function `transitions : ComponentRef ~> TransitionSet` contains the transitions of each thread component, represented as a semi-colon-separated multiset of transitions of the form s `-[`*guard*`]->` s' `{`*actions*`}`. The attribute `behaviorRef` denotes the component of the thread; `variables` denotes its local variables *and* their values; `currState` denotes the current "state" of the transition system; and `completeStates` denotes its *complete* states.

Data ports are represented as object instances of subclasses of the class `Port`, whose `content` attribute denotes the current content of the port, which is either `noMsg` (the port buffer is empty) or contains a data element e, in which case the `content` is `data(`e`)`. Thread input ports also contain a flag `fresh` denoting whether the port received data in the latest dispatch:

```
class Port | content : MsgContent .
class InDataPort .
class OutDataPort .
   subclass InDataPort OutDataPort < Port .

class InDataThreadPort | fresh : Bool .
   subclass InDataThreadPort < InDataPort .

sort MsgContent .
op noMsg : -> MsgContent [ctor] .
op data : Bool -> MsgContent [ctor] .    op data : Int -> MsgContent [ctor] .
```

A level-up *connection*, linking an outgoing port P in a subcomponent C to the outgoing port P' in the "current" component, is modeled as a term C `.` P `-->` P'. Immediate same-level and level-down connections are terms of the forms, respectively, C_1 `.` P_1 `-->` C_2 `.` P_2 and P `-->` C `.` P'. Delayed connections are denoted with the arrow `-->>`. A *connection set* is a semi-colon-separated set of connections.

For example, in our avionics example, an instance of the top-level system component in Section 4 would be represented in Real-Time Maude by the term

```
< MAIN : System | features : none,     properties : Synchronous(true) ; SynchPeriod(2),
                  subcomponents : < sideOne : System | ... >   < sideTwo : System | ... >
                                   < env : System | ... >,
                  connections : sideOne . side1ActiveSide -->> sideTwo . side1ActiveSide ;
                                sideTwo . side2ActiveSide -->> sideOne . side2ActiveSide ;
                                env . side1FullyAvail --> sideOne . side1FullyAvail ;
                                ...
                                env . side2Failed --> sideTwo . side2Failed >
```

The instance `side1Thread` of the component `Side1Thread.impl` in a particular state is represented by the term

```
< side1Thread : Thread |
    features : < side2ActiveSide : InDataThreadPort | content : data(2), fresh : true >
               < manualSelection : InDataThreadPort | content : data(false), fresh : true >
               ...
               < side1ActiveSide : OutDataPort | content : data(0) >,
    subcomponents : none,    properties : periodic-dispatch(2) ; Deterministic(true),
    connections : none,      behaviorRef : thread Side1Thread . impl,
    variables : (prevSide2ActiveStatus |-> 2) (prevManualSwitch |-> false),
    currState : side1FailedState,
    completeStates : preinit initState side1FailedState ... side2ActiveState >
```

5.2 Formalizing the Synchronous Steps

Assuming that the system contains one environment thread and that the other threads are deterministic, a synchronous step of the system is formalized by the following tick rewrite rule:

```
var SYSTEM : Object .    var VAL : Valuation .    var VALS : ValuationSet .

crl [syncStepWithTime] :
    {SYSTEM}
 => {applyTransitions(transferData(applyEnvTransitions(VAL, SYSTEM)))}
    in time period(SYSTEM)
   if containsEnvironment(SYSTEM) /\ VAL ;; VALS := allEnvAssignments(SYSTEM) .
```

The function `allEnvAssignments` uses Maude's SAT solver to find all valuations of the Boolean variables in the environment thread that satisfy the environment constraint. The union operator `_;;_` is declared to be associative and commutative; therefore, *any* of these valuations is nondeterministically assigned to the variable `VAL` in the matching condition `VAL ;; VALS := allEnvAssignments(SYSTEM)`. Then, the function `applyEnvTransitions`, which performs the environment transition that outputs the values of the variables given by the selected valuation `VAL`, is applied. The function `transferData` is then applied to the entire system; this function transfers the data in the threads' output ports to the receiving input ports and then clears the output ports. Finally, the function `applyTransitions` applies transitions in each non-environment thread until a *complete* state is reached in the thread. The function `period` extracts the period of the system. In case there is no environment thread in the system, the following rule models a synchronous step:

```
crl [syncStepWithTimeNoEnv] :
    {SYSTEM}
   =>
    {applyTransitions(transferData(SYSTEM))}  in time period(SYSTEM)
   if not containsEnvironment(SYSTEM) .
```

We refer to the full executable specification in Appendix C for the definition of these functions, and only show the definition of `applyTrans`, which distributes

to the thread objects in the state, and is defined as follows for deterministic threads:

```
ceq applyTransitions(
      < O : Thread | properties : Deterministic(true) ; PROPS,
                     features : PORTS,   currState : L1,   completeStates : LS,
                     variables : VAL,   behaviorRef : CR >)
  = if L2 in LS then < O : Thread | features : NEW-PORTS,   currState : L2,
                                    variables : NEW-VALUATION >
    else applyTransitions(< O : Thread | features : NEW-PORTS,  currState : L2,
                                         variables : NEW-VALUATION >) fi
  if ((L1 -[GUARD]-> L2 {SL}) ; TRANSITIONS) := transitions(CR)
       /\ evalGuard(GUARD, PORTS, VAL)
       /\ transResult(NEW-PORTS, NEW-VALUATION) :=
            executeTransition(L1 -[GUARD]-> L2 {SL}, PORTS, VAL) .
```

The thread is in local state `L1`, and a transition `L1 -[GUARD]-> L2 {SL}`, whose `GUARD` evaluates to `true` in the current state and input port values, is applied from the transitions in `transitions(CR)`. The function `executeTransition` executes a given transition in a state with a given set `PORTS` of ports and assignment `VAL` of the state variables. The function returns a term `transResult(`p, σ`)`, where p is the state of the ports after the execution, and σ denotes the resulting values of the state variables. If the resulting state `L2` is *not* a *complete* state, the function `applyTransitions` is applied again to the new state.

5.3 Correspondence with the PALS Model

As mentioned in Section 2, a transition of a machine ensemble $\mathcal{E}$ with environment constraint c_e has the form $(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{\mathcal{E}_{c_e}} (\boldsymbol{s'}, \boldsymbol{i'})$, where $\boldsymbol{s}$ contains the state of each machine and the contents in the "feedback wires" of the ensemble, and $\boldsymbol{i}$ is input from the environment, so that from state $\boldsymbol{s}$ with input $\boldsymbol{i}$ the system goes to state $\boldsymbol{s'}$ in one synchronous step, and so that the environment generates output $\boldsymbol{i'}$ in the *next* step. Let $\{t\} \longrightarrow \{t'\}$ be a rewrite step in the Real-Time Maude semantics. The term t contains both the states of the single threads as well as the contents in the feedback wires (since the output ports contain the output from the "previous" iteration of the threads), and therefore corresponds to $\boldsymbol{s}$. Furthermore, although the input $\boldsymbol{i}$ from the environment is generated *during* the rewrite, and hence is not present in t, the resulting state t' contains the $\boldsymbol{i}$ which brought the system from state t to state t'. This input from the environment is present in both the appropriate input ports in t' as well as in the local variables of the environment thread. Therefore, if we denote by $t(\boldsymbol{j})$ a state t where we make explicit the (previous) input $\boldsymbol{j}$ from the environment, then the transition $(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{\mathcal{E}_{c_e}} (\boldsymbol{s'}, \boldsymbol{i'})$ corresponds to a rewrite $\{t(\boldsymbol{i_p})\} \longrightarrow \{t'(\boldsymbol{i})\}$, where $\boldsymbol{i_p}$ was the input in the previous round, and where $state_{\mathcal{E}}(t) = \boldsymbol{s}$ and $state_{\mathcal{E}}(t') = \boldsymbol{s'}$ for the obvious mapping $state_{\mathcal{E}}$ which maps a term to the states of $\mathcal{E}$.

6 Formal Analysis of Synchronous AADL Models

The Real-Time Maude model that can be synthesized from a Synchronous AADL model using the semantics described in Section 5, which is supported by the *SynchAADL2Maude* tool explained in Section 8, can be formally analyzed in different ways. This section presents some functions allowing the user to define system properties in terms of a Synchronous AADL model without having to understand its formal representation.

Reachability Analysis. Real-Time Maude's search commands can be used to analyze whether or not a state matching a *state pattern* with a match that satisfies a condition can be reached from the initial state. The term

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

returns the value of the state variable *v* in the thread identified by the full component name *fullComponentName* in the system in state *globalComponent*. The full component name is a `->`-separated path of component names. Likewise, the term

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

gives the current location/state in the transition system in the given thread.

In our example, if `MAIN` is the name of the top-level component, then the following search command checks whether we can reach a state where the side one thread is in state `side1ActiveState` and the side two thread is in state `side2ActiveState`:

```
Maude> (utsearch [1] {initial} =>* {C:Configuration}
         such that
          ((location of component (MAIN -> sideOne -> sideProcess -> sideThread)
              in C:Configuration) == side1ActiveState
           and (location of component (MAIN -> sideTwo -> sideProcess -> sideThread)
                  in C:Configuration) == side2ActiveState) .)
```

LTL Model Checking. For LTL model checking purposes, our tool has useful pre-defined parametric atomic propositions, such as

full thread name `@` *location*

which holds when the thread is in state *location*, and

`value of` *port*/*variable* `in component` *fullThreadName* `is` *v*

that holds in a state if the value of the local variable or port of the thread is *v*.

7 Verifying the Active Standby System

This section show how we have verified the Synchronous AADL model of the avionics systems in Section 4.

7.1 The Active Standby System

The paper [13] lists the following important properties that the active standby system must satisfy:

R_1: Both sides should agree on which side is active (provided neither side has failed, the availability of a side has not changed, and the pilot has not made a manual selection).
R_2: A side that is not fully available should not be the active side if the other side is fully available (again, provided neither side has failed, the availability of a side has not changed, and the pilot has not made a manual selection).
R_3: The pilot can always change the active side (except if a side is failed or the availability of a side has changed).
R_4: If a side is failed the other side should become active.
R_5: The active side should not change unless the availability of a side changes, the failed status of a side changes, or manual selection is selected by the pilot.

We have verified all the property R_1–R_5 for the Synchronous AADL model using the Real-Time Maude below. A more detailed discussion of the LTL representation of the above properties can be found in [12].

Requirement R_1. *Both sides should agree on which side is active (provided neither side has failed, the availability of a side has not changed, and the pilot has not made a manual selection).*

Side i thinks that side j is active if it sends the number j to its output port `side`i`ActiveSide`. Using the predefined proposition "`value of` *port* `in` *thread* `is` v", we can easily define the formula `agreeOnActiveSide` to hold when both sides think that side 1 is active or when both sides think that side 2 is active:

```
op agreeOnActiveSide : -> Formula .
eq agreeOnActiveSide =
      ((value of side1ActiveSide in component
           (MAIN -> sideOne -> sideProcess -> sideThread) is 1) /\
       (value of side2ActiveSide in component
           (MAIN -> sideTwo -> sideProcess -> sideThread) is 1))
   \/ ((value of side1ActiveSide in component
           (MAIN -> sideOne -> sideProcess -> sideThread) is 2) /\
       (value of side2ActiveSide in component
           (MAIN -> sideTwo -> sideProcess -> sideThread) is 2)) .
```

Side i has failed if it has received the value `true` in its `side`i`Failed` port:

```
ops side1Failed side2Failed neitherSideFailed : -> Formula .
eq side1Failed
 = value of side1Failed in component
      (MAIN -> sideOne -> sideProcess -> sideThread) is true .
eq side2Failed
```

```
 = value of side2Failed in component
      (MAIN -> sideTwo -> sideProcess -> sideThread) is true .
eq neitherSideFailed = (~ side1Failed) /\ (~ side2Failed) .
```

Likewise, the proposition `side`i`FullyAvailable` holds if side i is fully available. There is no change in availability if both sides are equally available in the current state and in the next state:

```
op noChangeAvailability : -> Formula .
eq noChangeAvailability
 = (side1FullyAvailable <-> O side1FullyAvailable) /\
   (side2FullyAvailable <-> O side2FullyAvailable) .
```

We define a property that the pilot has made a manual selection, and then define a formula that says that in the *next* state, the pilot has not made a manual selection, the availability of a side has not changed, and neither side has failed:

```
ops manSelectPressed noChangeAssumptionNextState : -> Formula .
eq manSelectPressed
 = value of manualSelection in component
      (MAIN -> sideOne -> sideProcess -> sideThread) is true .
eq noChangeAssumptionNextState
 = noChangeAvailability /\ (O ~ manSelectPressed) /\
   (O neitherSideFailed) .
```

As explained in [12], the requirement $R1$ is not satisfied in the active standby system; instead, we verify the following weaker property `R1`. For each state, if in the *next* state the availability has not changed, the pilot has not pressed the button, and neither side has failed, then the sides agree on which side is active either in the next state, or in the state following the next state (unless a side then has failed):

```
op R1 : -> Formula .
eq R1 = [] (noChangeAssumptionNextState
           -> O (agreeOnActiveSide \/
                 O (neitherSideFailed -> agreeOnActiveSide))) .
```

We can then use Real-Time Maude model checking to verify the property `O R1`:

```
Maude> (mc  {initial} |=u O R1 .)

rewrites: 1211549 in 9698ms cpu (9918ms real) (124918 rewrites/second)
Result Bool :  true
```

Requirement R_2. *A side that is not fully available should not be the active side if the other side is fully available (provided neither side has failed, the availability of a side has not changed, and the pilot has not made a manual selection).*

As explained in [12], this property does not hold as stated; the standby side monitors full availability, and hence the change of active side might be delayed by one round. Instead, we can verify the following property for *side* 1:

```
op R2a : -> Formula .
eq R2a
 = O ([] ((noChangeAssumptionNextState /\ O side1FullyAvailable /\
            O ~ side2FullyAvailable) ->
        O (~ side2Active
            \/ (noChangeAssumptionNextState -> O ~ side2Active)))) .
```

where the proposition side$i$FullyAvailable holds if Side i has received the value `true` in its side$i$FullyAvail port:

```
ops side1FullyAvailable side2FullyAvailable : -> Formula .
eq side1FullyAvailable
 = value of side1FullyAvail in component
    (MAIN -> sideOne -> sideProcess -> sideThread) is true .
eq side2FullyAvailable
 = value of side2FullyAvail in component
    (MAIN -> sideTwo -> sideProcess -> sideThread) is true .
```

and side$i$FullyAvailable holds if Side i has received the value `true` in its side$i$ActiveSide port:

```
ops side1Active side2Active : -> Formula .
eq side1Active
 = value of side1ActiveSide in component
      (MAIN -> sideOne -> sideProcess -> sideThread) is 1 .
eq side2Active
 = value of side2ActiveSide in component
      (MAIN -> sideTwo -> sideProcess -> sideThread) is 2 .
```

Model checking shows that `R2a` holds in our model:

```
Maude> (mc  {initial} |=u R2a .)

rewrites: 1145326 in 7171ms cpu (7335ms real) (159711 rewrites/second)
Result Bool :  true
```

The symmetric property with the above for Side 2 does not hold, due to the fact that the sides are asymmetric in their failure recovery [12]. The best we can hope for side 2 is:

```
op R2b : -> Formula .
eq R2b
  = [] ((noChangeAssumptionNextState /\ O side2FullyAvailable /\
         O ~ side1FullyAvailable)
        -> O (~ side1Active \/
             (noChangeAssumptionNextState -> O (~ side1Active \/
                (noChangeAssumptionNextState -> O (~ side1Active \/
                   (noChangeAssumptionNextState -> O (~ side1Active \/
                      (noChangeAssumptionNextState -> O (~ side1Active))
                   )))))))) .
```

Model checking this property returns `true`:

```
Maude> (mc  {initial} |=u R2b .)

rewrites: 1146199 in 7146ms cpu (7161ms real) (160396 rewrites/second)
Result Bool :  true
```

Requirement R_3. *The pilot can always change the active side (except if a side is failed or the availability of a side has changed).*

The following proposition expresses the situation that the pilot change the active side:

```
op manSelectPressed : -> Formula .
eq manSelectPressed
 = value of manualSelection in component
    (MAIN -> sideOne -> sideProcess -> sideThread) is true .
```

Again, as explained in [12], R_3 is a problematic requirement and does not satisfied. We have verified the following variant of R_3; the property says that if the two sides are fully available and do not receive a manual switch request for two consecutive rounds, and stay faultless and receive a manual switch request in the third round, then the active side will switch *instantaneously*:

```
op R3g : -> Formula .
eq R3g
 = []((~ manSelectPressed /\ agreeOnActiveSide /\
        side1FullyAvailable /\ side2FullyAvailable /\
        noChangeAssumptionNextState)
       ->
      ((side1Active -> O O ((manSelectPressed /\ side1FullyAvailable /\
                              side2FullyAvailable) -> side2Active)) /\
       (side2Active -> O O ((manSelectPressed /\ side1FullyAvailable /\
                              side2FullyAvailable) -> side1Active)))) .
```

We can model check the property `R3g` as follows in Real-Time Maude:

```
Maude> (mc  {initial} |=u R3g .)

rewrites: 1146199 in 7118ms cpu (7148ms real) (161016 rewrites/second)
Result Bool :  true
```

Requirement R_4. *If a side is failed the other side should become active.*

Considering the one-step communication delay, this property can be represented as follows:

```
op R4 : -> Formula .
eq R4
 = [] (((side1Failed /\ ~ side2Failed) ->
         O (~ side2Failed -> side2Active)) /\
       ((side2Failed /\ ~ side1Failed) ->
         O (~ side1Failed -> side1Active))) .
```

This property holds in our model:

```
Maude> (mc  {initial} |=u R4 .)

rewrites: 1145382 in 7153ms cpu (7271ms real) (161016 rewrites/second)
Result Bool :  true
```

Requirement R_5. *The active side should not change unless the availability of a side changes, the failed status of a side changes, or manual selection is selected by the pilot.*

For active side 1, this property can be defined as follows; if side 1 is active, then it stays active forever, or until something changes [12]:

```
op R5side1 : -> Formula .
eq R5side1
 = [] ( ((side1Active /\ side1FullyAvailable /\ ~ manSelectPressed) ->
         (side1Active W (~ side1FullyAvailable \/ manSelectPressed)))
        /\
        ((side1Active /\ ~ side1FullyAvailable /\ ~ side2FullyAvailable
           /\ ~ manSelectPressed /\ ~ side1Failed)
          ->
         (side1Active W
           (side2FullyAvailable \/ manSelectPressed \/ side1Failed)))) .
```

This formula can be model checked successfully:

```
Maude> (mc  {initial} |=u R5side1 .)

rewrites: 1146223 in 7215ms cpu (7279ms real) (158864 rewrites/second)
Result Bool :  true
```

In the case of Side 2, the similar formula does not hold, since if side 2 is active, it might also be inactivated when side 1 wakes up from failure, without full availability changing, or sides failing [12]. We must therefore weaken the property for side 2, to exclude states where side 2 sends 2 only because it is some error recovery state, and consider the property only for when side 2 is in state `side2Active`:

```
op s2inStateSide2Active : -> Formula .
eq s2inStateSide2Active
 = (MAIN -> sideTwo -> sideProcess -> sideThread) @ side2ActiveState .

op R5side2X : -> Formula .
eq R5side2X
 = [] ( ((s2inStateSide2Active /\ side2FullyAvailable
           /\ ~ manSelectPressed /\ ~ side1Failed)
          -> (s2inStateSide2Active W
                (~ side2FullyAvailable \/ manSelectPressed \/
                  side1Failed))) /\
```

```
((s2inStateSide2Active /\ ~ side1FullyAvailable
   /\ ~ manSelectPressed /\ ~ side2Failed /\ ~ side1Failed)
 -> (s2inStateSide2Active
      W (side1FullyAvailable \/ manSelectPressed
          \/ side2Failed \/ side1Failed))) /\
((s2inStateSide2Active /\ ~ manSelectPressed
   /\ ~ side2Failed /\ side1Failed)
 -> (side2Active
     W (manSelectPressed \/ side2Failed \/ ~ side1Failed)))) .
```

This property is also model checked successfully in Synchronous AADL model:

```
Maude> (mc  {initial} |=u R5side2X .)

rewrites: 1211161 in 9557ms cpu (9811ms real) (126723 rewrites/second)
Result Bool :  true
```

The interesting aspect of this verification is that it seems very hard to model check the corresponding *asynchronous* design. In the paper [11] we define and analyze Real-Time Maude models of both the synchronous and the asynchronous design of the active standby system. It turns out that the synchronous system has 185 reachable states, and the different properties can be model checked in 0.8 seconds. In contrast, a *simplest possible* asynchronous model of active standby—with no message delays, no execution times, perfect local clocks, and optimal periods—has 3,047,832 reachable states and model checking a simple invariant takes 2000 seconds. If the message delay can be either 0 or 1, and everything else is optimal, then no model checking analysis terminated in reasonable time.

7.2 The Three–Node Active Standby System

The requirements of the 3-node active standby system described in Section 4.2 can be summarized as follows:

R_1: At most one side is active in any step.
R_2: If there is no active side, then after some steps a side will become active. (provided that the failure condition of some side is not flipping[8]).
R_3: In particular, if the failure status of at most one side can be flipped, whenever there is no active side, then in 4 steps, a side will become active.
R_4 The local views of the active-standby status of 3 sides are identical at each state.

We have also verified the above 4 properties for the Synchronous AADL model using the Real-Time Maude.

[8] The failure status of a side is flipped at a state if the side does not fail at the state but fails at the next state, or vice versa.

Requirement R_1. *At most one side is active in any step.*

Side i is active if the `aileron` system component receive the value 1 in its `side`i`ActiveSideout` port:

```
ops side1Active side2Active side3Active : -> Formula .
eq side1Active
 = value of side1ActiveSideout in component
      (MAIN -> aileron -> ioProcess -> outsynchThread) is 1 .
eq side2Active
 = value of side2ActiveSideout in component
      (MAIN -> aileron -> ioProcess -> outsynchThread) is 1 .
eq side3Active
 = value of side3ActiveSideout in component
      (MAIN -> aileron -> ioProcess -> outsynchThread) is 1 .
```

This property can be simply represented in Real-Time Maude as follows:

```
op R1 : -> Formula .
eq R1
 = [] ( ~(side1Active /\ side2Active) /\
        ~(side1Active /\ side3Active) /\
        ~(side2Active /\ side3Active))
```

We can model check the property `R1` as follows in Real-Time Maude:

```
Maude> (mc  {initial} |=u R1 .)

rewrites: 608387 in 5958ms cpu (5990ms real) (102105 rewrites/second)
Result Bool :  true
```

Requirement R_2. *If there is no active side, then after some steps a side will become active. (provided that the failure condition of some side is not flipping.*

Side i is flipping at the current state if Side i does not fail at the next state whenever Side i fails at this state, and vice versa:

```
ops flipOne flipTwo flipThree : -> Formula .
eq flipOne
 = (value of s1F in component
      (MAIN -> environment -> envProcess -> insynchThread) is true <->
   O (value of s1F in component
        (MAIN -> environment -> envProcess -> insynchThread) is false))
  \/
  (value of s1F in component
       (MAIN -> environment -> envProcess -> insynchThread) is false <->
   O (value of s1F in component
        (MAIN -> environment -> envProcess -> insynchThread) is true)) .

eq flipTwo
 = (value of s2F in component
```

```
     (MAIN -> environment -> envProcess -> insynchThread) is true <->
   O (value of s2F in component
        (MAIN -> environment -> envProcess -> insynchThread) is false))
  \/
  (value of s2F in component
       (MAIN -> environment -> envProcess -> insynchThread) is false <->
   O (value of s2F in component
        (MAIN -> environment -> envProcess -> insynchThread) is true)) .

eq flipThree
 = (value of s3F in component
     (MAIN -> environment -> envProcess -> insynchThread) is true <->
   O (value of s3F in component
        (MAIN -> environment -> envProcess -> insynchThread) is false))
  \/
  (value of s3F in component
       (MAIN -> environment -> envProcess -> insynchThread) is false <->
   O (value of s3F in component
        (MAIN -> environment -> envProcess -> insynchThread) is true)) .
```

Then, the property R_2 can be represented as follows:

```
op R2 : -> Formula .
eq R2 = []<> (~ flipOne /\ ~ flipTwo /\ ~ flipThree)
          ->
        []( (~ side1Active /\ ~ side2Active /\ ~ side3Active)
              ->
            <> (side1Active \/ side2Active \/ side3Active))
```

We can perform the Real-Time Maude model checking of `R2` as follows:

```
Maude> (mc  {initial} |=u R2 .)

rewrites: 357028 in 5942ms cpu (6489ms real) (60079 rewrites/second)
Result Bool : true
```

Requirement R_3. *If the failure status of at most one side can be flipped, whenever there is no active side, then in 4 steps, a side will become active.*

The antecedent of this property can be defined by the disjunction of the cases for flipping only one side and the case for flipping no side. Then using the next step operator `O`, this property can be defined as follows:

```
op R3 : -> Formula .
eq R3
 = [] ((~ flipOne /\ ~ flipTwo /\   flipThree) \/
       (~ flipOne /\   flipTwo /\ ~ flipThree) \/
       (  flipOne /\ ~ flipTwo /\ ~ flipThree) \/
       (~ flipOne /\ ~ flipTwo /\ ~ flipThree))
   ->
```

```
[] ((~ side1Active /\ ~ side2Active /\ ~ side3Active)
      ->
     (O (side1Active \/ side2Active \/ side3Active) \/
      O O (side1Active \/ side2Active \/ side3Active) \/
      O O O (side1Active \/ side2Active \/ side3Active) \/
      O O O O (side1Active \/ side2Active \/ side3Active)))
```

The model checking result shows that the property R3 is satisfied in our model:

```
Maude> (mc  {initial} |=u R3 .)

rewrites: 357034 in 5939ms cpu (6496ms real) (60108 rewrites/second)
Result Bool :  true
```

Requirement R_4. *The local views of the active-standby status of 3 sides are identical at each state.*

In the 3-node active standby model, the local view for side i is defined by the value of the variable `gside`i. Hence, the local views for Side i are identical when the variables `gside`i in each thread has the same value, we can define the propositions for the identical local view as follows:

```
ops sameOne sameTwo sameThree : -> Formula .
eq sameOne
 = (value of gside1 in component
      (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 0 <->
    value of gside1 in component
      (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is 0)
   /\
   (value of gside1 in component
      (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 0 <->
    value of gside1 in component
      (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is 0)
   /\
   (value of gside1 in component
      (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 1 <->
    value of gside1 in component
      (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is 1)
   /\
   (value of gside1 in component
      (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 1 <->
    value of gside1 in component
      (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is 1) .

eq sameTwo
 = (value of gside2 in component
      (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 0 <->
    value of gside2 in component
      (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is 0)
   /\
```

```
   (value of gside2 in component
      (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 0 <->
    value of gside2 in component
      (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is 0)
   /\
   (value of gside2 in component
      (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 1 <->
    value of gside2 in component
      (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is 1)
   /\
   (value of gside2 in component
      (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 1 <->
    value of gside2 in component
      (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is 1) .

eq sameThree
 = (value of gside3 in component
      (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 0 <->
    value of gside3 in component
      (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is 0)
   /\
   (value of gside3 in component
      (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 0 <->
    value of gside3 in component
      (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is 0)
   /\
   (value of gside3 in component
      (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 1 <->
    value of gside3 in component
      (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is 1)
   /\
   (value of gside3 in component
      (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 1 <->
    value of gside3 in component
      (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is 1) .
```

Then, the property R_4 is expressed as follws:

```
op R4 : -> Formula .
eq R4
 = [] (sameOne /\ sameTwo /\ sameThree) .
```

This property can be verified in Real Time Maude as follows:

```
Maude> (mc  {initial} |=u R4 .)

rewrites: 627563 in 6058ms cpu (6266ms real) (103577 rewrites/second)
Result Bool :  true
```

8 The SynchAADL2Maude Tool

We have integrated the Real-Time Maude verification of Synchronous AADL models into the Open Source AADL Tool Environment (OSATE), which is a set of Eclipse plug-ins. The *SynchAADL2Maude* tool is an OSATE plug-in that uses OSATE's model traversal facility to support both checking whether a model is a legal Synchronous AADL model and verifying Synchronous AADL models *within OSATE*.

When OSATE has generated an *AADL instance model* from an AADL specification, we can use the SynchAADL2Maude tool to: (i) check whether the instance model is a Synchronous AADL model, (ii) generate the corresponding Real-Time Maude model, and (iii) model check LTL properties of the instance model. Figure 4 shows the SynchAADL2Maude window for the active standby example. The `Constraints Check` button, the `Code Generation` button, and `Do Verification` button are used to perform, respectively, the static analysis, the Real-Time Maude code generation, and the model checking. The corrected versions of the active standby system requirements been entered into the tool, and are shown in the "AADL Property Requirement" table. The `Do Verification` button has been clicked and the results of the model checking are shown in the "Maude Console."

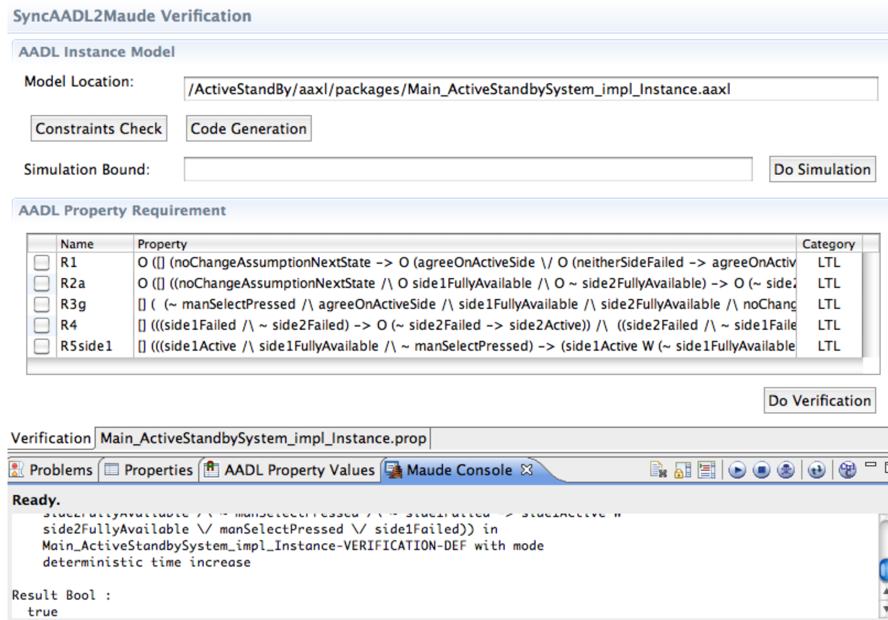

Fig. 4. SynchAADL2Maude window in OSATE.

The properties to be verified are managed by the associated XML property file. For example, to add an LTL model checking command to verify the property `R1` in Section 7, we just add the following `command` tag to the property file:

```
<command>   <name>R1</name>
  <value type = "ltl">
    O [] (noChangeAssumptionNextState
           -> O (agreeOnActiveSide \/ O (neitherSideFailed -> agreeOnActiveSide))) .
 </value>
</command>
```

New formulas can be defined in the property file using the `definition` tag. For example, the new constant `side1Failed` is defined as follows:

```
<definition>  <name>side1Failed</name>
  <value>
    value of side1Failed in component (MAIN -> sideOne -> sideProcess -> sideThread) is true
  </value>
</definition>
```

The OSATE's model traversal mechanisms are based on AADL metamodel described by the Eclipse Modeling Framework (EMF). An AADL specification is declared in OSATE in either textual or graphical format, where the abstract syntax tree of such a model is defined by the AADL metamodel that consists of seven Ecore packages explaining different aspects of AADL models, such as features, components, connections, etc. For each Ecore package, EMF provides a *switch* class that contains a collection of *case* methods. During traversing an AADL model object, OSATE just enumerates all syntactic object in the AADL model, and calls an appropriate "case" method for each of them.

Both the static analysis and the code generation are implemented by overriding such "case" methods accordingly for each corresponding syntactic object. For example, the Real-Time Maude specification is constructed by mapping each object in the system instance model to the corresponding Real-Time Maude object, as explained in Section 5, where the mapping of each component instance is defined by overriding the `caseComponentInstance` method as the following pseudo-code:

```
public Object caseComponentInstance(ComponentInstance c) {
  output("< %s : %s |\ n", getName(c), getCategory(c))\ n");
  output("  features : "); self.process(getFeatures(c)); output(",\ n");
  output("  properties : "); self.process(getProperties(c)); output(",\ n");
  output("  connections : "); self.process(getConnections(c)); output(",\ n");
  if ( isThread(c) ) {
    output("  behaviorRef : "); self.getComponentRef(c); output(",\ n");
    output("  variables : "); self.process(getVariables(c)); output(",\ n");
    output("  currState : "); self.process(getCurrentState(c)); output(",\ n");
    output("  completeStates : "); self.process(getCompleteStates(c)); output(",\ n");
  }
  output("  subcomponents : " ); self.process(getSubcomponents(c)); output(" >");
  return DONE;
}
```

Note that the `process` function in the above is a basic process function of OSATE to invoke a corresponding case method for the given syntactic object.

Algorithm 1 gives the pseudocode of the constraint checker that checks whether the input system component, S, is a valid Synchronous AADL model. The instance model of S consists of the active components (e.g. threads) and their semantic connections, as given by $threads(S)$ and $connections(S)$. We use Eclipse and OSATE support libraries for traversing through the instance model and check if the required properties of threads and connections adhere to Synchronous AADL requirements. We use two helper functions: *defined* and *value*, to check the property association and the value at an AADL component.

Algorithm 1 Constraint_Checker(S)

```
if (!defined(S, 'Synchronous') or value(S, 'Synchronous') != true) then
  addException(exceptions, S, "not synchronous")
  return exceptions
end if
for all t ∈ threads(S) do
  if (defined(t, 'Dispatch_Protocol') and value(t, 'Dispatch_Protocol') != 'Periodic')
  then
    addException(exceptions, t, "not periodic")
  end if
  if (!defined(t, 'synchPeriod')) then
    addException(exceptions, t, "SynchAADL::Period not defined")
  else if (defined(t, 'Period') and value(t, 'Period') != value(t, 'synchPeriod')) then
    addException(exceptions, t, "invalid period")
  end if
  if (!defined(t, 'isEnvironment') and value(t, 'Deterministic') = true) then
    addException(exceptions, t, "not deterministic")
  end if
end for
for all c ∈ connections(S) do
  if (type(c) != 'data port') then
    addException(exceptions, c, "not a data port connection")
  end if
  if (value(c, 'Timing') != 'Delayed' and value(src(c), 'isEnvironment') != true) then
    addException(exceptions, t, "invalid port connection timing")
  end if
end for
return exceptions
```

9 Related Work

The paper [7] formalizes the AADL data port protocol in Event-B. Despite the title of the paper, it does not define a synchronous subset of AADL and therefore does not provide an executable formal semantics of any such subset. There exist a number of formalizations and verification tools for different subsets of AADL

(see, e.g., [3, 4, 14, 2]). These approaches target ordinary (asynchronous) AADL models and do not define synchronous subsets of AADL. In [9], the behaviors of single AADL threads are given by synchronous Lustre programs. Since [9] also targets "standard" asynchronous AADL models, the authors show how asynchronous computation can be *encoded* in a synchronous language. This encoding does of course not reduce the state space of the asynchronous system.

Our work is motivated by the PALS pattern [11, 13, 1] that reduces the design and verification of an asynchronous system to that of its synchronous version. There is a fair amount of work that relates synchronous and asynchronous models in various ways; we refer to [12] for an extensive discussion on this topic.

10 Concluding Remarks

To the best of our knowledge the work we have presented is the first defining a synchronous subset of AADL. Such a subset is essential to reduce the design and verification complexity of distributed real-time systems that should operate in a virtually synchronous way. The formal semantics of synchronous AADL models we have provided is also essential for formal verification and is supported in practice by the *SynchAADL2Maude* tool in a way that preserves the AADL "look and feel" for users and minimizes the need for a detailed knowledge of the underlying Real-Time Maude tool. In summary, our work makes possible in practice the formal verification of asynchronous AADL models that are virtually synchronous by supporting the definition and verification of their semantically equivalent synchronous counterparts.

As usual much work remains ahead. One natural extension of the proposed AADL subset and the *SynchAADL2Maude* tool is the simultaneous support of synchronous subsystems with different periods and of additional AADL features. Further experimentation and development of additional case studies will also be important to improve the tool and its performance and to facilitate its use.

Acknowledgments. Our design of Synchronous AADL was inspired by our previous work on PALS with Steve Miller and Darren Cofer at Rockwell-Collins corporation, and Lui Sha at UIUC. We have also benefited from many discussions on the design of Synchronous AADL with Lui Sha and Peter Feiler; and from the feedback of the participants at several AADL meetings, where preliminary versions of these ideas were presented. This work has been partially supported by the Boeing corporation under grant C8088, by the National Science Foundation under grants CNS 08-34709 and CCF 09-05584, and by the Research Council of Norway.

References

1. Al-Nayeem, A., Sun, M., Qiu, X., Sha, L., Miller, S.P., Cofer, D.D.: A formal architecture pattern for real-time distributed systems. In: Proc. RTSS'09. IEEE (2009)

2. Berthomieu, B., Bodeveix, J.P., Chaudet, C., Dal Zilio, S., Filali, M., Vernadat, F.: Formal verification of AADL specifications in the Topcased environment. In: Reliable Software Technologies - Ada-Europe 2009, LNCS, vol. 5570. Springer (2009)
3. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V., Noll, T., Roveri, M., Wimmer, R.: A model checker for AADL. In: Proc. CAV'10, LNCS, vol. 6174. Springer (2010)
4. Chkouri, M.Y., Robert, A., Bozga, M., Sifakis, J.: Translating AADL into BIP - application to the verification of real-time systems. In: Proc. MoDELS Workshops. LNCS, vol. 5421. Springer (2008)
5. 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)
6. Feiler, H., Gluch, P., Hudak, J.: The architecture analysis & design language (AADL): An introduction. Tech. Rep. CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie-Mellon University (2006)
7. Filali, M., Lawall, J.: Development of a synchronous subset of AADL. In: Proc. ASM'10. LNCS, vol. 5977. Springer (2010)
8. França, R., Bodeveix, J.P., Filali, M., Rolland, J.F., Chemouil, D., Thomas, D.: The AADL behaviour annex - experiments and roadmap. In: ICECCS'07. IEEE (2007)
9. Jahier, E., Halbwachs, N., Raymond, P., Nicollin, X., Lesens, D.: Virtual execution of AADL models via a translation into synchronous programs. In: Proc. EMSOFT'07. ACM (2007)
10. Kopetz, H., Bauer, G.: The time-triggered architecture. Proc. of the IEEE 93(1) (2003)
11. Meseguer, J., Ölveczky, P.C.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. In: Proc. ICFEM'10. LNCS, vol. 6447. Springer (2010)
12. Meseguer, J., Ölveczky, P.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. Tech. rep., CS Dept., University of Illinois at Urbana-Champaign (September 2010), `http://hdl.handle.net/2142/17089`
13. Miller, S.P., Cofer, D.D., Sha, L., Meseguer, J., Al-Nayeem, A.: Implementing logical synchrony in integrated modular avionics. In: Proc. DASC'09. IEEE (2009)
14. 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)
15. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
16. Rushby, J.: Systematic formal verification for fault-tolerant time-triggered algorithms. IEEE Trans. Software Eng. 25(5) (1999)
17. SAE AADL Team: AADL homepage (2009), `http://www.aadl.info/`
18. Sha, L., Al-Nayeem, A., Sun, M., Meseguer, J., Ölveczky, P.C.: PALS: Physically asynchronous logically synchronous systems. Tech. rep., Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign (2009), `http://hdl.handle.net/2142/11897`

A Active Standby Model Specifications

In this section, we gives the entire Synchronous AADL source code of the Active Standby examples, and the corresponding Real-Time Maude code generated by our tool.

A.1 Synchronous AADL specification.

```
package MainModule
public
  system ActiveStandbySystem
  end ActiveStandbySystem;

  system implementation ActiveStandbySystem.impl
    subcomponents
      sideOne: system Side1::Side1.impl;
      sideTwo: system Side2::Side2.impl;
      env: system Environment::Environment.impl;
    connections
      C1: data port sideOne.side1ActiveSide ->> sideTwo.side1ActiveSide;
      C2: data port sideTwo.side2ActiveSide ->> sideOne.side2ActiveSide;
      F1: data port env.side1FullyAvail -> sideOne.side1FullyAvail;
      F2: data port env.side1FullyAvail -> sideTwo.side1FullyAvail;
      F3: data port env.side2FullyAvail -> sideOne.side2FullyAvail;
      F4: data port env.side2FullyAvail -> sideTwo.side2FullyAvail;
      C3: data port env.manualSelection -> sideOne.manualSelection;
      C4: data port env.manualSelection -> sideTwo.manualSelection;
      C5: data port env.side1Failed -> sideOne.side1Failed;
      C6: data port env.side2Failed -> sideTwo.side2Failed;
    properties
      SynchAADL::Synchronous => true;
      SynchAADL::syncPeriod => 2 Ms;
      Period => 2 Ms;
  end ActiveStandbySystem.impl;
end MainModule;

package Environment
public
  system Environment
    features
      side1FullyAvail: out data port behavior::boolean;
      side2FullyAvail: out data port behavior::boolean;
      manualSelection: out data port behavior::boolean;
      side1Failed: out data port behavior::boolean;
      side2Failed: out data port behavior::boolean;
  end Environment;

  system implementation Environment.impl
    subcomponents
      envProcess: process EnvironmentProcess.impl;
    connections
      C1: data port envProcess.side1FullyAvail -> side1FullyAvail;
      C2: data port envProcess.side2FullyAvail -> side2FullyAvail;
      C3: data port envProcess.manualSelection -> manualSelection;
      C4: data port envProcess.side1Failed -> side1Failed;
      C5: data port envProcess.side2Failed -> side2Failed;
  end Environment.impl;

  process EnvironmentProcess
```

```
    features
      side1FullyAvail: out data port behavior::boolean;
      side2FullyAvail: out data port behavior::boolean;
      manualSelection: out data port behavior::boolean;
      side1Failed: out data port behavior::boolean;
      side2Failed: out data port behavior::boolean;
  end EnvironmentProcess;

  process implementation EnvironmentProcess.impl
    subcomponents
      envThread: thread EnvironmentThread.impl;
    connections
      C1: data port envThread.side1FullyAvail -> side1FullyAvail;
      C2: data port envThread.side2FullyAvail -> side2FullyAvail;
      C3: data port envThread.manualSelection -> manualSelection;
      C4: data port envThread.side1Failed -> side1Failed;
      C5: data port envThread.side2Failed -> side2Failed;
  end EnvironmentProcess.impl;

  thread EnvironmentThread
    features
      side1FullyAvail: out data port behavior::boolean;
      side2FullyAvail: out data port behavior::boolean;
      manualSelection: out data port behavior::boolean;
      side1Failed: out data port behavior::boolean;
      side2Failed: out data port behavior::boolean;
  end EnvironmentThread;

  thread implementation EnvironmentThread.impl
    properties
      SynchAADL::InputConstraints =>
        "(not s1F and s2F and not s2FA) or (not s2F and s1F and not s1FA)
         or (not s1F and not s2F)";
      SynchAADL::IsEnvironment => true;
      Dispatch_Protocol => Periodic;
    annex behavior_specification {**
      state variables
        s1FA:behavior::boolean;
        s2FA:behavior::boolean;
        mS:behavior::boolean;
        s1F:behavior::boolean;
        s2F:behavior::boolean;
      initially
        s1FA := true;
        s2FA := true;
        mS := false;
        s1F := false;
        s2F := false;states
        s0 : initial complete state;
      transitions
        s0 -[  ]-> s0
          side1FullyAvail := s1FA;
          side2FullyAvail := s2FA;
          manualSelection := mS;
          side1Failed := s1F;
          side2Failed := s2F;;
**};
  end EnvironmentThread.impl;
end Environment;

package Side1
public
```

```
system Side1
  features
    side1FullyAvail: in data port behavior::boolean;
    side2FullyAvail: in data port behavior::boolean;
    side2ActiveSide: in data port behavior::integer;
    manualSelection: in data port behavior::boolean;
    side1Failed: in data port behavior::boolean;
    side1ActiveSide: out data port behavior::integer;
end Side1;

system implementation Side1.impl
  subcomponents
    sideProcess: process Side1Process.impl;
  connections
    data port side1FullyAvail -> sideProcess.side1FullyAvail;
    data port side2FullyAvail -> sideProcess.side2FullyAvail;
    data port side2ActiveSide -> sideProcess.side2ActiveSide;
    data port manualSelection -> sideProcess.manualSelection;
    data port side1Failed -> sideProcess.side1Failed;
    data port sideProcess.side1ActiveSide -> side1ActiveSide;
end Side1.impl;

process Side1Process
  features
    side1FullyAvail: in data port behavior::boolean;
    side2FullyAvail: in data port behavior::boolean;
    side2ActiveSide: in data port behavior::integer;
    manualSelection: in data port behavior::boolean;
    side1Failed: in data port behavior::boolean;
    side1ActiveSide: out data port behavior::integer;
end Side1Process;

process implementation Side1Process.impl
  subcomponents
    sideThread: thread Side1Thread.impl;
  connections
    data port side1FullyAvail -> sideThread.side1FullyAvail;
    data port side2FullyAvail -> sideThread.side2FullyAvail;
    data port side2ActiveSide -> sideThread.side2ActiveSide;
    data port manualSelection -> sideThread.manualSelection;
    data port sideThread.side1ActiveSide -> side1ActiveSide;
    data port side1Failed -> sideThread.side1Failed;
end Side1Process.impl;

thread Side1Thread
  features
    side1FullyAvail: in data port behavior::boolean;
    side2FullyAvail: in data port behavior::boolean;
    side2ActiveSide: in data port behavior::integer;
    manualSelection: in data port behavior::boolean;
    side1Failed: in data port behavior::boolean;
    side1ActiveSide: out data port behavior::integer;
end Side1Thread;

thread implementation Side1Thread.impl
  properties
    Dispatch_Protocol => Periodic;
    SynchAADL::Deterministic => true;
  annex behavior_specification {**
    state variables
      prevSide2ActiveSide:behavior::integer;
      prevmanualSelection:behavior::boolean;
```

```
initially
  prevSide2ActiveSide := 0;
  prevmanualSelection := false;states
  preInit : initial complete state;
  initState : complete state;
  side1FailedState : complete state;
  side2FailedState : complete state;
  side1WaitState : complete state;
  side1ActiveState : complete state;
  side2ActiveState : complete state;
transitions
  preInit -[  ]-> initState;
  initState -[ on side1Failed = true ]-> side1FailedState {
    side1ActiveSide := 0;
    prevSide2ActiveSide := 0;
    prevmanualSelection := false;};
  initState -[ on side1Failed = false ]-> side2FailedState {
    side1ActiveSide := 1;
    prevSide2ActiveSide := 0;
    prevmanualSelection := false;};
  side1FailedState -[ on side1Failed = false and side2ActiveSide = 0 ]-> side2FailedState {
    side1ActiveSide := 1;
    prevSide2ActiveSide := side2ActiveSide;
    prevmanualSelection := manualSelection;};
  side1FailedState -[ on side1Failed = false and side2ActiveSide != 0 ]-> side1WaitState {
    side1ActiveSide := 1;
    prevSide2ActiveSide := side2ActiveSide;
    prevmanualSelection := manualSelection;};
  side1FailedState -[ on side1Failed = true ]-> side1FailedState {
    side1ActiveSide := 0;
    prevSide2ActiveSide := side2ActiveSide;
    prevmanualSelection := manualSelection;};
  side2FailedState -[ on side1Failed = false and side2ActiveSide = 0 ]-> side2FailedState {
    side1ActiveSide := 1;
    prevSide2ActiveSide := side2ActiveSide;
    prevmanualSelection := manualSelection;};
  side2FailedState -[ on side1Failed = false and side2ActiveSide != 0 ]-> side1WaitState {
    side1ActiveSide := 1;
    prevSide2ActiveSide := side2ActiveSide;
    prevmanualSelection := manualSelection;};
  side2FailedState -[ on side1Failed = true ]-> side1FailedState {
    side1ActiveSide := 0;
    prevSide2ActiveSide := side2ActiveSide;
    prevmanualSelection := manualSelection;};
  side1WaitState -[ on side1Failed = false and side2ActiveSide != 0 ]-> side1ActiveState {
    side1ActiveSide := 1;
    prevSide2ActiveSide := side2ActiveSide;
    prevmanualSelection := manualSelection;};
  side1WaitState -[ on side1Failed = true ]-> side1FailedState {
    side1ActiveSide := 0;
    prevSide2ActiveSide := side2ActiveSide;
    prevmanualSelection := manualSelection;};
  side1WaitState -[ on side1Failed = false and side2ActiveSide = 0 ]-> side2FailedState {
    side1ActiveSide := 1;
    prevSide2ActiveSide := side2ActiveSide;
    prevmanualSelection := manualSelection;};
  side1ActiveState -[ on side1Failed = false and prevSide2ActiveSide != 2
                                  and side2ActiveSide = 2 ]-> side2ActiveState {
    side1ActiveSide := 2;
    prevSide2ActiveSide := side2ActiveSide;
    prevmanualSelection := manualSelection;};
  side1ActiveState -[ on side1Failed = true ]-> side1FailedState {
```

```
          side1ActiveSide := 0;
          prevSide2ActiveSide := side2ActiveSide;
          prevmanualSelection := manualSelection;};
        side1ActiveState -[ on side1Failed = false and side2ActiveSide = 0 ]-> side2FailedState {
          side1ActiveSide := 1;
          prevSide2ActiveSide := side2ActiveSide;
          prevmanualSelection := manualSelection;};
        side1ActiveState -[ on side1Failed = false and (prevSide2ActiveSide = 2 or side2ActiveSide != 2)
                                         and side2ActiveSide != 0 ]-> side1ActiveState {
          side1ActiveSide := 1;
          prevSide2ActiveSide := side2ActiveSide;
          prevmanualSelection := manualSelection;};
        side2ActiveState -[ on side1Failed = false and side2ActiveSide != 0 and side1FullyAvail = true
                                         and (prevmanualSelection = false and manualSelection = true
                                           or side2FullyAvail = false) ]-> side1ActiveState {
          side1ActiveSide := 1;
          prevSide2ActiveSide := side2ActiveSide;
          prevmanualSelection := manualSelection;};
        side2ActiveState -[ on side1Failed = true ]-> side1FailedState {
          side1ActiveSide := 0;
          prevSide2ActiveSide := side2ActiveSide;
          prevmanualSelection := manualSelection;};
        side2ActiveState -[ on side1Failed = false and side2ActiveSide = 0 ]-> side2FailedState {
          side1ActiveSide := 1;
          prevSide2ActiveSide := side2ActiveSide;
          prevmanualSelection := manualSelection;};
        side2ActiveState -[ on side1Failed = false and side2ActiveSide != 0 and
                               (side1FullyAvail = false or side2FullyAvail = true
                                  and (prevmanualSelection = true
                                    or manualSelection = false)) ]-> side2ActiveState {
          side1ActiveSide := 2;
          prevSide2ActiveSide := side2ActiveSide;
          prevmanualSelection := manualSelection;};
**};
  end Side1Thread.impl;
end Side1;


package Side2
public
  system Side2
    features
      side1FullyAvail: in data port behavior::boolean;
      side2FullyAvail: in data port behavior::boolean;
      side1ActiveSide: in data port behavior::integer;
      manualSelection: in data port behavior::boolean;
      side2Failed: in data port behavior::boolean;
      side2ActiveSide: out data port behavior::integer;
  end Side2;

  system implementation Side2.impl
    subcomponents
      sideProcess: process Side2Process.impl;
    connections
      data port side1FullyAvail -> sideProcess.side1FullyAvail;
      data port side2FullyAvail -> sideProcess.side2FullyAvail;
      data port side1ActiveSide -> sideProcess.side1ActiveSide;
      data port manualSelection -> sideProcess.manualSelection;
      data port side2Failed -> sideProcess.side2Failed;
      data port sideProcess.side2ActiveSide -> side2ActiveSide;
  end Side2.impl;
```

```
process Side2Process
  features
    side1FullyAvail: in data port behavior::boolean;
    side2FullyAvail: in data port behavior::boolean;
    side1ActiveSide: in data port behavior::integer;
    manualSelection: in data port behavior::boolean;
    side2Failed: in data port behavior::boolean;
    side2ActiveSide: out data port behavior::integer;
end Side2Process;

process implementation Side2Process.impl
  subcomponents
    sideThread: thread Side2Thread.impl;
  connections
    data port side1FullyAvail -> sideThread.side1FullyAvail;
    data port side2FullyAvail -> sideThread.side2FullyAvail;
    data port side1ActiveSide -> sideThread.side1ActiveSide;
    data port manualSelection -> sideThread.manualSelection;
    data port side2Failed -> sideThread.side2Failed;
    data port sideThread.side2ActiveSide -> side2ActiveSide;
end Side2Process.impl;

-- side2 failed will be assesed in the behavior annex
-- similarly, side1becomesActive will also be determined in annex;
-- change of manual selection will be determined in behavior annex;
-- does Maude care about non-complete state and complete state?
-- what about ambiguous transitions?
thread Side2Thread
  features
    side1FullyAvail: in data port behavior::boolean;
    side2FullyAvail: in data port behavior::boolean;
    side1ActiveSide: in data port behavior::integer;
    manualSelection: in data port behavior::boolean;
    side2Failed: in data port behavior::boolean;
    side2ActiveSide: out data port behavior::integer;
end Side2Thread;

thread implementation Side2Thread.impl
  properties
    Dispatch_Protocol => Periodic;
    SynchAADL::Deterministic => true;
  annex behavior_specification {**
    state variables
      prevSide1ActiveSide:behavior::integer;
      prevmanualSelection:behavior::boolean;
    initially
      prevSide1ActiveSide := 0;
      prevmanualSelection := false;states
      preInit : initial complete state;
      initState : complete state;
      side2FailedState : complete state;
      side1FailedState : complete state;
      side2WaitState : complete state;
      side1ActiveState : complete state;
      side2ActiveState : complete state;
    transitions
      preInit -[  ]-> initState;
      initState -[ on side2Failed = true ]-> side2FailedState {
        side2ActiveSide := 0;
        prevSide1ActiveSide := 0;
        prevmanualSelection := false;};
      initState -[ on side2Failed = false ]-> side1FailedState {
```

```
  side2ActiveSide := 2;
  prevSide1ActiveSide := 0;
  prevmanualSelection := false;};
side2FailedState -[ on side2Failed = false and side1ActiveSide = 0 ]-> side1FailedState {
  side2ActiveSide := 2;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side2FailedState -[ on side2Failed = false and side1ActiveSide != 0 ]-> side2WaitState {
  side2ActiveSide := 1;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side2FailedState -[ on side2Failed = true ]-> side2FailedState {
  side2ActiveSide := 0;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side1FailedState -[ on side2Failed = false and side1ActiveSide = 0 ]-> side1FailedState {
  side2ActiveSide := 2;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side1FailedState -[ on side2Failed = false and side1ActiveSide != 0 ]-> side2WaitState {
  side2ActiveSide := 1;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side1FailedState -[ on side2Failed = true ]-> side2FailedState {
  side2ActiveSide := 0;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side2WaitState -[ on side2Failed = false and side1ActiveSide != 0 ]-> side1ActiveState {
  side2ActiveSide := 1;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side2WaitState -[ on side2Failed = true ]-> side2FailedState {
  side2ActiveSide := 0;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side2WaitState -[ on side2Failed = false and side1ActiveSide = 0 ]-> side1FailedState {
  side2ActiveSide := 2;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side1ActiveState -[ on side2Failed = true ]-> side2FailedState {
  side2ActiveSide := 0;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side1ActiveState -[ on side2Failed = false and side1ActiveSide = 0 ]-> side1FailedState {
  side2ActiveSide := 2;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side1ActiveState -[ on side2Failed = false and side1ActiveSide != 0 and side2FullyAvail = true
                    and (prevmanualSelection = false and manualSelection = true
                     or side1FullyAvail = false) ]-> side2ActiveState {
  side2ActiveSide := 2;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side1ActiveState -[ on side2Failed = false and side1ActiveSide != 0
                    and (side2FullyAvail = false or side1FullyAvail = true
                       and (prevmanualSelection = true or manualSelection = false)) ]-> side1ActiveState {
  side2ActiveSide := 1;
  prevSide1ActiveSide := side1ActiveSide;
  prevmanualSelection := manualSelection;};
side2ActiveState -[ on side2Failed = true ]-> side2FailedState {
  side2ActiveSide := 0;
  prevSide1ActiveSide := side1ActiveSide;
```

```
          prevmanualSelection := manualSelection;};
        side2ActiveState -[ on side2Failed = false and side1ActiveSide = 0 ]-> side1FailedState {
          side2ActiveSide := 2;
          prevSide1ActiveSide := side1ActiveSide;
          prevmanualSelection := manualSelection;};
        side2ActiveState -[ on side2Failed = false and prevSide1ActiveSide != 1
                                and side1ActiveSide = 1 ]-> side1ActiveState {
          side2ActiveSide := 1;
          prevSide1ActiveSide := side1ActiveSide;
          prevmanualSelection := manualSelection;};
        side2ActiveState -[ on side2Failed = false and (prevSide1ActiveSide = 1 or side1ActiveSide != 1)
                                and side1ActiveSide != 0 ]-> side2ActiveState {
          side2ActiveSide := 2;
          prevSide1ActiveSide := side1ActiveSide;
          prevmanualSelection := manualSelection;};
**};
  end Side2Thread.impl;
end Side2;
```

A.2 Real-Time Maude specification.

```
(tomod MainActiveStandbySystemimplInstance is
  including SYNCHRONOUS-STEP .

  --- names, states, variables
  ops s2F s1FA mS s1F prevmanualSelection s2FA : -> BoolVarId [ctor] .
  ops prevSide2ActiveSide prevSide1ActiveSide : -> IntVarId [ctor] .
  ops s0 side1WaitState side2ActiveState side1FailedState side2FailedState side1ActiveState
     initState side2WaitState preInit : -> Location [ctor] .
  ops impl : -> ImplName [ctor] .
  ops Side1::Side1Thread MainModule::ActiveStandbySystem Side2::Side2
     Environment::Environment Side1::Side1 Environment::EnvironmentThread Side2::Side2Process
     Side1::Side1Process Environment::EnvironmentProcess Side2::Side2Thread : -> TypeName [ctor] .
  ops side1ActiveSide side1Failed manualSelection side2ActiveSide side2Failed side2FullyAvail
     side1FullyAvail : -> PortId [ctor] .
  ops MainActiveStandbySystemimplInstance sideThread envProcess envThread sideOne sideTwo
     env sideProcess : -> ComponentId [ctor] .

  --- the initial state
  op initial : -> Configuration .
  eq initial = MainActiveStandbySystemimplInstance : system MainModule::ActiveStandbySystem . impl .
  eq MAIN = MainActiveStandbySystemimplInstance .

  ---------------------------------------------
  --- AADL instance
  ---------------------------------------------

  var COMP : ComponentId .

  eq stateVariables(thread Side1::Side1Thread . impl) =
    (prevSide2ActiveSide |-> 0)
    (prevmanualSelection |-> false).

  eq states(thread Side1::Side1Thread . impl) =
    initial: preInit
    complete: preInit
    complete: initState
    complete: side1FailedState
    complete: side2FailedState
    complete: side1WaitState
    complete: side1ActiveState
```

```
  complete: side2ActiveState
 .

eq transitions(thread Side1::Side1Thread . impl) =
  (preInit -[]-> initState {
    nil});
  (initState -[(side1Failed = true)]-> side1FailedState {
    (side1ActiveSide := 0);
    (prevSide2ActiveSide := 0);
    (prevmanualSelection := false)});
  (initState -[(side1Failed = false)]-> side2FailedState {
    (side1ActiveSide := 1);
    (prevSide2ActiveSide := 0);
    (prevmanualSelection := false)});
  (side1FailedState -[((side1Failed = false) and (side2ActiveSide = 0))]-> side2FailedState {
    (side1ActiveSide := 1);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side1FailedState -[((side1Failed = false) and (side2ActiveSide != 0))]-> side1WaitState {
    (side1ActiveSide := 1);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side1FailedState -[(side1Failed = true)]-> side1FailedState {
    (side1ActiveSide := 0);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side2FailedState -[((side1Failed = false) and (side2ActiveSide = 0))]-> side2FailedState {
    (side1ActiveSide := 1);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side2FailedState -[((side1Failed = false) and (side2ActiveSide != 0))]-> side1WaitState {
    (side1ActiveSide := 1);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side2FailedState -[(side1Failed = true)]-> side1FailedState {
    (side1ActiveSide := 0);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side1WaitState -[((side1Failed = false) and (side2ActiveSide != 0))]-> side1ActiveState {
    (side1ActiveSide := 1);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side1WaitState -[(side1Failed = true)]-> side1FailedState {
    (side1ActiveSide := 0);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side1WaitState -[((side1Failed = false) and (side2ActiveSide = 0))]-> side2FailedState {
    (side1ActiveSide := 1);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side1ActiveState -[(((side1Failed = false) and (prevSide2ActiveSide != 2))
        and (side2ActiveSide = 2))]-> side2ActiveState {
    (side1ActiveSide := 2);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side1ActiveState -[(side1Failed = true)]-> side1FailedState {
    (side1ActiveSide := 0);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side1ActiveState -[((side1Failed = false) and (side2ActiveSide = 0))]-> side2FailedState {
    (side1ActiveSide := 1);
    (prevSide2ActiveSide := side2ActiveSide);
```

```
    (prevmanualSelection := manualSelection)});
  (side1ActiveState -[(((side1Failed = false) and (((prevSide2ActiveSide = 2)
         or (side2ActiveSide != 2)))) and (side2ActiveSide != 0))]-> side1ActiveState {
    (side1ActiveSide := 1);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side2ActiveState -[((((side1Failed = false) and (side2ActiveSide != 0))
        and (side1FullyAvail = true)) and ((((prevmanualSelection = false) and (manualSelection = true))
           or (side2FullyAvail = false))))]-> side1ActiveState {
    (side1ActiveSide := 1);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side2ActiveState -[(side1Failed = true)]-> side1FailedState {
    (side1ActiveSide := 0);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side2ActiveState -[((side1Failed = false) and (side2ActiveSide = 0))]-> side2FailedState {
    (side1ActiveSide := 1);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side2ActiveState -[(((side1Failed = false) and (side2ActiveSide != 0)) and (((side1FullyAvail = false)
        or ((side2FullyAvail = true) and (((prevmanualSelection = true)
           or (manualSelection = false)))))))]-> side2ActiveState {
    (side1ActiveSide := 2);
    (prevSide2ActiveSide := side2ActiveSide);
    (prevmanualSelection := manualSelection)}) .

eq COMP : thread Side1::Side1Thread . impl =
  < COMP : Thread |
    features :
      < side1FullyAvail : InDataThreadPort | content : noMsg, fresh : false >
      < side2FullyAvail : InDataThreadPort | content : noMsg, fresh : false >
      < side2ActiveSide : InDataThreadPort | content : noMsg, fresh : false >
      < manualSelection : InDataThreadPort | content : noMsg, fresh : false >
      < side1Failed : InDataThreadPort | content : noMsg, fresh : false >
      < side1ActiveSide : OutDataPort | content : noMsg >,
    behaviorRef : (thread Side1::Side1Thread . impl),
    variables : stateVariables(thread Side1::Side1Thread . impl),
    currState : initialState(thread Side1::Side1Thread . impl),
    completeStates : completeStates(thread Side1::Side1Thread . impl),
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2) ;
      DispatchProtocol(Periodic) ;
      Deterministic(true),
    connections :
      none,
    subcomponents :
      none
  > .

eq COMP : process Side1::Side1Process . impl =
  < COMP : Process |
    features :
      < side1FullyAvail : InDataPort | content : noMsg >
      < side2FullyAvail : InDataPort | content : noMsg >
      < side2ActiveSide : InDataPort | content : noMsg >
      < manualSelection : InDataPort | content : noMsg >
      < side1Failed : InDataPort | content : noMsg >
      < side1ActiveSide : OutDataPort | content : noMsg >,
    properties :
```

```
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (sideThread . side1ActiveSide --> side1ActiveSide) ;
      (side2ActiveSide --> sideThread . side2ActiveSide) ;
      (side1FullyAvail --> sideThread . side1FullyAvail) ;
      (side2FullyAvail --> sideThread . side2FullyAvail) ;
      (manualSelection --> sideThread . manualSelection) ;
      (side1Failed --> sideThread . side1Failed),
    subcomponents :
      (sideThread : thread Side1::Side1Thread . impl)
  > .

eq COMP : system Side1::Side1 . impl =
  < COMP : System |
    features :
      < side1FullyAvail : InDataPort | content : noMsg >
      < side2FullyAvail : InDataPort | content : noMsg >
      < side2ActiveSide : InDataPort | content : noMsg >
      < manualSelection : InDataPort | content : noMsg >
      < side1Failed : InDataPort | content : noMsg >
      < side1ActiveSide : OutDataPort | content : noMsg >,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (sideProcess . side1ActiveSide --> side1ActiveSide) ;
      (side2ActiveSide --> sideProcess . side2ActiveSide) ;
      (side1FullyAvail --> sideProcess . side1FullyAvail) ;
      (side2FullyAvail --> sideProcess . side2FullyAvail) ;
      (manualSelection --> sideProcess . manualSelection) ;
      (side1Failed --> sideProcess . side1Failed),
    subcomponents :
      (sideProcess : process Side1::Side1Process . impl)
  > .

eq stateVariables(thread Side2::Side2Thread . impl) =
  (prevSide1ActiveSide |-> 0)
  (prevmanualSelection |-> false).

eq states(thread Side2::Side2Thread . impl) =
  initial: preInit
  complete: preInit
  complete: initState
  complete: side2FailedState
  complete: side1FailedState
  complete: side2WaitState
  complete: side1ActiveState
  complete: side2ActiveState
 .

eq transitions(thread Side2::Side2Thread . impl) =
  (preInit -[]-> initState {
    nil});
  (initState -[(side2Failed = true)]-> side2FailedState {
    (side2ActiveSide := 0);
    (prevSide1ActiveSide := 0);
    (prevmanualSelection := false)});
  (initState -[(side2Failed = false)]-> side1FailedState {
    (side2ActiveSide := 2);
```

```
  (prevSide1ActiveSide := 0);
  (prevmanualSelection := false)});
(side2FailedState -[((side2Failed = false) and (side1ActiveSide = 0))]-> side1FailedState {
  (side2ActiveSide := 2);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side2FailedState -[((side2Failed = false) and (side1ActiveSide != 0))]-> side2WaitState {
  (side2ActiveSide := 1);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side2FailedState -[(side2Failed = true)]-> side2FailedState {
  (side2ActiveSide := 0);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side1FailedState -[((side2Failed = false) and (side1ActiveSide = 0))]-> side1FailedState {
  (side2ActiveSide := 2);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side1FailedState -[((side2Failed = false) and (side1ActiveSide != 0))]-> side2WaitState {
  (side2ActiveSide := 1);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side1FailedState -[(side2Failed = true)]-> side2FailedState {
  (side2ActiveSide := 0);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side2WaitState -[((side2Failed = false) and (side1ActiveSide != 0))]-> side1ActiveState {
  (side2ActiveSide := 1);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side2WaitState -[(side2Failed = true)]-> side2FailedState {
  (side2ActiveSide := 0);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side2WaitState -[((side2Failed = false) and (side1ActiveSide = 0))]-> side1FailedState {
  (side2ActiveSide := 2);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side1ActiveState -[(side2Failed = true)]-> side2FailedState {
  (side2ActiveSide := 0);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side1ActiveState -[((side2Failed = false) and (side1ActiveSide = 0))]-> side1FailedState {
  (side2ActiveSide := 2);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side1ActiveState -[((((side2Failed = false) and (side1ActiveSide != 0))
       and (side2FullyAvail = true)) and ((((prevmanualSelection = false) and (manualSelection = true))
          or (side1FullyAvail = false))))]-> side2ActiveState {
  (side2ActiveSide := 2);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side1ActiveState -[(((side2Failed = false) and (side1ActiveSide != 0)) and (((side2FullyAvail = false)
       or ((side1FullyAvail = true) and (((prevmanualSelection = true)
          or (manualSelection = false)))))))]-> side1ActiveState {
  (side2ActiveSide := 1);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
(side2ActiveState -[(side2Failed = true)]-> side2FailedState {
  (side2ActiveSide := 0);
  (prevSide1ActiveSide := side1ActiveSide);
  (prevmanualSelection := manualSelection)});
```

```
  (side2ActiveState -[((side2Failed = false) and (side1ActiveSide = 0))]-> side1FailedState {
    (side2ActiveSide := 2);
    (prevSide1ActiveSide := side1ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side2ActiveState -[(((side2Failed = false) and (prevSide1ActiveSide != 1))
         and (side1ActiveSide = 1))]-> side1ActiveState {
    (side2ActiveSide := 1);
    (prevSide1ActiveSide := side1ActiveSide);
    (prevmanualSelection := manualSelection)});
  (side2ActiveState -[(((side2Failed = false) and (((prevSide1ActiveSide = 1)
         or (side1ActiveSide != 1)))) and (side1ActiveSide != 0))]-> side2ActiveState {
    (side2ActiveSide := 2);
    (prevSide1ActiveSide := side1ActiveSide);
    (prevmanualSelection := manualSelection)}) .

eq COMP : thread Side2::Side2Thread . impl =
  < COMP : Thread |
    features :
      < side1FullyAvail : InDataThreadPort | content : noMsg, fresh : false >
      < side2FullyAvail : InDataThreadPort | content : noMsg, fresh : false >
      < side1ActiveSide : InDataThreadPort | content : noMsg, fresh : false >
      < manualSelection : InDataThreadPort | content : noMsg, fresh : false >
      < side2Failed : InDataThreadPort | content : noMsg, fresh : false >
      < side2ActiveSide : OutDataPort | content : noMsg >,
    behaviorRef : (thread Side2::Side2Thread . impl),
    variables : stateVariables(thread Side2::Side2Thread . impl),
    currState : initialState(thread Side2::Side2Thread . impl),
    completeStates : completeStates(thread Side2::Side2Thread . impl),
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2) ;
      DispatchProtocol(Periodic) ;
      Deterministic(true),
    connections :
      none,
    subcomponents :
      none
  > .

eq COMP : process Side2::Side2Process . impl =
  < COMP : Process |
    features :
      < side1FullyAvail : InDataPort | content : noMsg >
      < side2FullyAvail : InDataPort | content : noMsg >
      < side1ActiveSide : InDataPort | content : noMsg >
      < manualSelection : InDataPort | content : noMsg >
      < side2Failed : InDataPort | content : noMsg >
      < side2ActiveSide : OutDataPort | content : noMsg >,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (side1ActiveSide --> sideThread . side1ActiveSide) ;
      (sideThread . side2ActiveSide --> side2ActiveSide) ;
      (side1FullyAvail --> sideThread . side1FullyAvail) ;
      (side2FullyAvail --> sideThread . side2FullyAvail) ;
      (manualSelection --> sideThread . manualSelection) ;
      (side2Failed --> sideThread . side2Failed),
    subcomponents :
      (sideThread : thread Side2::Side2Thread . impl)
```

```
  > .

eq COMP : system Side2::Side2 . impl =
  < COMP : System |
    features :
      < side1FullyAvail : InDataPort | content : noMsg >
      < side2FullyAvail : InDataPort | content : noMsg >
      < side1ActiveSide : InDataPort | content : noMsg >
      < manualSelection : InDataPort | content : noMsg >
      < side2Failed : InDataPort | content : noMsg >
      < side2ActiveSide : OutDataPort | content : noMsg >,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (side1ActiveSide --> sideProcess . side1ActiveSide) ;
      (sideProcess . side2ActiveSide --> side2ActiveSide) ;
      (side1FullyAvail --> sideProcess . side1FullyAvail) ;
      (side2FullyAvail --> sideProcess . side2FullyAvail) ;
      (manualSelection --> sideProcess . manualSelection) ;
      (side2Failed --> sideProcess . side2Failed),
    subcomponents :
      (sideProcess : process Side2::Side2Process . impl)
  > .

eq stateVariables(thread Environment::EnvironmentThread . impl) =
  (s1FA |-> true)
  (s2FA |-> true)
  (mS |-> false)
  (s1F |-> false)
  (s2F |-> false).

eq states(thread Environment::EnvironmentThread . impl) =
  initial: s0
  complete: s0
 .

eq transitions(thread Environment::EnvironmentThread . impl) =
  (s0 -[]-> s0 {
    (side1FullyAvail := s1FA);
    (side2FullyAvail := s2FA);
    (manualSelection := mS);
    (side1Failed := s1F);
    (side2Failed := s2F)}) .

eq COMP : thread Environment::EnvironmentThread . impl =
  < COMP : Thread |
    features :
      < side1FullyAvail : OutDataPort | content : noMsg >
      < side2FullyAvail : OutDataPort | content : noMsg >
      < manualSelection : OutDataPort | content : noMsg >
      < side1Failed : OutDataPort | content : noMsg >
      < side2Failed : OutDataPort | content : noMsg >,
    behaviorRef : (thread Environment::EnvironmentThread . impl),
    variables : stateVariables(thread Environment::EnvironmentThread . impl),
    currState : initialState(thread Environment::EnvironmentThread . impl),
    completeStates : completeStates(thread Environment::EnvironmentThread . impl),
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2) ;
```

```
      DispatchProtocol(Periodic) ;
      InputConstraints((not s1F and s2F and not s2FA)
             or (not s2F and s1F and not s1FA) or (not s1F and not s2F)) ;
      IsEnvironment(true),
    connections :
      none,
    subcomponents :
      none
  > .

eq COMP : process Environment::EnvironmentProcess . impl =
  < COMP : Process |
    features :
      < side1FullyAvail : OutDataPort | content : noMsg >
      < side2FullyAvail : OutDataPort | content : noMsg >
      < manualSelection : OutDataPort | content : noMsg >
      < side1Failed : OutDataPort | content : noMsg >
      < side2Failed : OutDataPort | content : noMsg >,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (envThread . side1FullyAvail --> side1FullyAvail) ;
      (envThread . side2FullyAvail --> side2FullyAvail) ;
      (envThread . manualSelection --> manualSelection) ;
      (envThread . side1Failed --> side1Failed) ;
      (envThread . side2Failed --> side2Failed),
    subcomponents :
      (envThread : thread Environment::EnvironmentThread . impl)
  > .

eq COMP : system Environment::Environment . impl =
  < COMP : System |
    features :
      < side1FullyAvail : OutDataPort | content : noMsg >
      < side2FullyAvail : OutDataPort | content : noMsg >
      < manualSelection : OutDataPort | content : noMsg >
      < side1Failed : OutDataPort | content : noMsg >
      < side2Failed : OutDataPort | content : noMsg >,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (envProcess . side1FullyAvail --> side1FullyAvail) ;
      (envProcess . side2FullyAvail --> side2FullyAvail) ;
      (envProcess . manualSelection --> manualSelection) ;
      (envProcess . side1Failed --> side1Failed) ;
      (envProcess . side2Failed --> side2Failed),
    subcomponents :
      (envProcess : process Environment::EnvironmentProcess . impl)
  > .

eq COMP : system MainModule::ActiveStandbySystem . impl =
  < COMP : System |
    features :
      none,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
```

```
      connections :
        (sideOne . side1ActiveSide -->> sideTwo . side1ActiveSide) ;
        (sideTwo . side2ActiveSide -->> sideOne . side2ActiveSide) ;
        (env . side1FullyAvail --> sideOne . side1FullyAvail) ;
        (env . side1FullyAvail --> sideTwo . side1FullyAvail) ;
        (env . side2FullyAvail --> sideOne . side2FullyAvail) ;
        (env . side2FullyAvail --> sideTwo . side2FullyAvail) ;
        (env . manualSelection --> sideOne . manualSelection) ;
        (env . manualSelection --> sideTwo . manualSelection) ;
        (env . side1Failed --> sideOne . side1Failed) ;
        (env . side2Failed --> sideTwo . side2Failed),
      subcomponents :
        (sideTwo : system Side2::Side2 . impl)
        (env : system Environment::Environment . impl)
        (sideOne : system Side1::Side1 . impl)
    > .

endtom)
```

A.3 LTL Property Specifications in Real-Time Maude

```
(tomod Main_ActiveStandbySystem_impl_Instance-VERIFICATION-DEF is
  including MainActiveStandbySystemimplInstance .
  including LTL-MODEL-CHECK-AADL .

  op side1Active : -> Formula .
  eq side1Active
   = value of side1ActiveSide in component (MAIN -> sideOne -> sideProcess -> sideThread) is 1 .
  op side2Active : -> Formula .
  eq side2Active
   = value of side2ActiveSide in component (MAIN -> sideTwo -> sideProcess -> sideThread) is 2 .
  op agreeOnActiveSide : -> Formula .
  eq agreeOnActiveSide
   = ((value of side1ActiveSide in component (MAIN -> sideOne -> sideProcess -> sideThread) is 1)
       /\
          (value of side2ActiveSide in component (MAIN -> sideTwo -> sideProcess -> sideThread) is 1))
           \/
             ((value of side1ActiveSide in component (MAIN -> sideOne -> sideProcess -> sideThread) is 2)
        /\
           (value of side2ActiveSide in component (MAIN -> sideTwo -> sideProcess -> sideThread) is 2)) .
  op side1FullyAvailable : -> Formula .
  eq side1FullyAvailable
   = value of side1FullyAvail in component (MAIN -> sideOne -> sideProcess -> sideThread) is true .
  op side2FullyAvailable : -> Formula .
  eq side2FullyAvailable
   = value of side2FullyAvail in component (MAIN -> sideTwo -> sideProcess -> sideThread) is true .
  op noChangeAvailability : -> Formula .
  eq noChangeAvailability
   = (side1FullyAvailable <-> O side1FullyAvailable)
     /\ (side2FullyAvailable <-> O side2FullyAvailable) .
  op manSelectPressed : -> Formula .
  eq manSelectPressed
   = value of manualSelection in component (MAIN -> sideOne -> sideProcess -> sideThread) is true .
  op side1Failed : -> Formula .
  eq side1Failed
   = value of side1Failed in component (MAIN -> sideOne -> sideProcess -> sideThread) is true .
  op side2Failed : -> Formula .
  eq side2Failed
   = value of side2Failed in component (MAIN -> sideTwo -> sideProcess -> sideThread) is true .
  op neitherSideFailed : -> Formula .
  eq neitherSideFailed
```

```
   = (~ side1Failed) /\ (~ side2Failed) .
  op noChangeAssumptionNextState : -> Formula .
  eq noChangeAssumptionNextState
   = noChangeAvailability /\ (O ~ manSelectPressed) /\ (O neitherSideFailed) .
  op s2inStateSide2Active : -> Formula .
  eq s2inStateSide2Active
   = (MAIN -> sideTwo -> sideProcess -> sideThread) @ side2ActiveState .
endtom)

--- R1
(mc {initial} |=u O ([] (noChangeAssumptionNextState ->
       O (agreeOnActiveSide \/ O (neitherSideFailed -> agreeOnActiveSide)))) .)

--- R2a
(mc {initial} |=u O ([] ((noChangeAssumptionNextState /\ O side1FullyAvailable /\
            O ~ side2FullyAvailable) ->
        O (~ side2Active \/ (noChangeAssumptionNextState -> O ~ side2Active)))) .)

--- R3g
(mc {initial} |=u [] (  (~ manSelectPressed /\ agreeOnActiveSide /\ side1FullyAvailable /\
       side2FullyAvailable /\ noChangeAssumptionNextState) ->
              ( (side1Active -> O O ( (manSelectPressed /\ side1FullyAvailable /\ side2FullyAvailable) ->
                     side2Active)) /\ (side2Active -> O O (  (manSelectPressed /\ side1FullyAvailable
                            /\ side2FullyAvailable) -> side1Active)))) .)

--- R4
(mc {initial} |=u [] (((side1Failed /\ ~ side2Failed) -> O (~ side2Failed -> side2Active)) /\
       ((side2Failed /\ ~ side1Failed) -> O (~ side1Failed -> side1Active))) .)

--- R5side1
(mc {initial} |=u [] (((side1Active /\ side1FullyAvailable /\ ~ manSelectPressed) ->
       (side1Active W (~ side1FullyAvailable \/ manSelectPressed)))
     /\
     ((side1Active /\ ~ side1FullyAvailable /\ ~ side2FullyAvailable
       /\ ~ manSelectPressed /\ ~ side1Failed)
      -> (side1Active W
           (side2FullyAvailable \/ manSelectPressed \/ side1Failed)))) .)
```

B 3-node Active Standby Model

This section presents the entire Synchronous AADL source code of the 3-node Active Standby examples, and the corresponding Real-Time Maude code generated by our tool.

B.1 Synchronous AADL Specification

```
package MainModule
public
  system MainSystem
  end MainSystem;

  system implementation MainSystem.impl
    subcomponents
      sideOne: system Side1::Side1.impl;
      sideTwo: system Side2::Side2.impl;
      sideThree: system Side3::Side3.impl;
      environment: system Environment::Environment.impl;
      aileron: system Aileron::IOTask.impl;
    connections
      data port sideOne.side1ActiveSide ->> sideTwo.side1ActiveSide;
      data port sideOne.side1ActiveSide ->> sideThree.side1ActiveSide;
      data port sideTwo.side2ActiveSide ->> sideOne.side2ActiveSide;
      data port sideTwo.side2ActiveSide ->> sideThree.side2ActiveSide;
      data port sideThree.side3ActiveSide ->> sideOne.side3ActiveSide;
      data port sideThree.side3ActiveSide ->> sideTwo.side3ActiveSide;
      data port environment.side1Failed -> sideOne.side1Failed;
      data port environment.side2Failed -> sideTwo.side2Failed;
      data port environment.side3Failed -> sideThree.side3Failed;
      data port sideOne.side1ActiveSide ->> aileron.side1ActiveSide;
      data port sideTwo.side2ActiveSide ->> aileron.side2ActiveSide;
      data port sideThree.side3ActiveSide ->> aileron.side3ActiveSide;
    properties
      SynchAADL::Synchronous => true;
      SynchAADL::syncPeriod => 2 Ms;
      Period => 2 Ms;
  end MainSystem.impl;
end MainModule;

package Aileron
public
  -- controller input/output
  system IOTask
    features
      -- active standby output
      side1ActiveSide: in data port behavior::integer;
      side2ActiveSide: in data port behavior::integer;
      side3ActiveSide: in data port behavior::integer;
  end IOTask;

  system implementation IOTask.impl
    subcomponents
      ioProcess: process IOProcess.impl;
    connections
      data port side1ActiveSide -> ioProcess.side1ActiveSide_in;
      data port side2ActiveSide -> ioProcess.side2ActiveSide_in;
      data port side3ActiveSide -> ioProcess.side3ActiveSide_in;
  end IOTask.impl;
```

```
  process IOProcess
    features
      side1ActiveSide_in: in data port behavior::integer;
      side2ActiveSide_in: in data port behavior::integer;
      side3ActiveSide_in: in data port behavior::integer;
  end IOProcess;

  process implementation IOProcess.impl
    subcomponents
      outsynchThread: thread OutputSynchronizer.impl;
    connections
      data port side1ActiveSide_in -> outsynchThread.side1ActiveSide_in;
      data port side2ActiveSide_in -> outsynchThread.side2ActiveSide_in;
      data port side3ActiveSide_in -> outsynchThread.side3ActiveSide_in;
    properties
      Dispatch_Protocol => Periodic applies to outsynchThread;
      SynchAADL::Deterministic => true applies to outsynchThread;
  end IOProcess.impl;

  thread OutputSynchronizer
    features
      side1ActiveSide_in: in data port behavior::integer;
      side2ActiveSide_in: in data port behavior::integer;
      side3ActiveSide_in: in data port behavior::integer;
  end OutputSynchronizer;

  thread implementation OutputSynchronizer.impl
    annex behavior_specification {**
      state variables
        side1ActiveSide_out:behavior::integer;
        side2ActiveSide_out:behavior::integer;
        side3ActiveSide_out:behavior::integer;
      initially
        side1ActiveSide_out := -1;
        side2ActiveSide_out := -1;
        side3ActiveSide_out := -1;states
        initState : initial complete state;
        curState : complete state;
      transitions
        initState -[  ]-> curState {
          side1ActiveSide_out := -1;
          side2ActiveSide_out := -1;
          side3ActiveSide_out := -1;};
        curState -[  ]-> curState {
          side1ActiveSide_out := side1ActiveSide_in;
          side2ActiveSide_out := side2ActiveSide_in;
          side3ActiveSide_out := side3ActiveSide_in;};
**};
  end OutputSynchronizer.impl;
end Aileron;

package Environment
public
  system Environment
    features
      side1Failed: out data port behavior::boolean;
      side2Failed: out data port behavior::boolean;
      side3Failed: out data port behavior::boolean;
  end Environment;

  system implementation Environment.impl
```

```
    subcomponents
      envProcess: process EnvProcess.impl;
    connections
      C4: data port envProcess.side1Failed -> side1Failed;
      C5: data port envProcess.side2Failed -> side2Failed;
      C6: data port envProcess.side3Failed -> side3Failed;
  end Environment.impl;

  process EnvProcess
    features
      side1Failed: out data port behavior::boolean;
      side2Failed: out data port behavior::boolean;
      side3Failed: out data port behavior::boolean;
  end EnvProcess;

  process implementation EnvProcess.impl
    subcomponents
      insynchThread: thread InputSynchronizer.impl;
    connections
      C4: data port insynchThread.side1Failed -> side1Failed;
      C5: data port insynchThread.side2Failed -> side2Failed;
      C6: data port insynchThread.side3Failed -> side3Failed;
  end EnvProcess.impl;

  thread InputSynchronizer
    features
      side1Failed: out data port behavior::boolean;
      side2Failed: out data port behavior::boolean;
      side3Failed: out data port behavior::boolean;
  end InputSynchronizer;

  thread implementation InputSynchronizer.impl
    properties
      Dispatch_Protocol => Periodic;
      SynchAADL::IsEnvironment => true;
      SynchAADL::InputConstraints => "not s1F or not s2F or not s3F";
    annex behavior_specification {**
      state variables
        s1F:behavior::boolean;
        s2F:behavior::boolean;
        s3F:behavior::boolean;
      initially
        s1F := false;
        s2F := false;
        s3F := false;states
        s0 : initial complete state;
      transitions
        s0 -[  ]-> s0 {
          side1Failed := s1F;
          side2Failed := s2F;
          side3Failed := s3F;};
**};
  end InputSynchronizer.impl;
end Environment;

package Side1
public
  system Side1
    features
      side2ActiveSide: in data port behavior::integer;
      side3ActiveSide: in data port behavior::integer;
      side1Failed: in data port behavior::boolean;
```

```
    side1ActiveSide: out data port behavior::integer;
end Side1;

system implementation Side1.impl
  subcomponents
    sideProcess: process Side1Process.impl;
  connections
    data port side2ActiveSide -> sideProcess.side2ActiveSide;
    data port side3ActiveSide -> sideProcess.side3ActiveSide;
    data port side1Failed -> sideProcess.side1Failed;
    data port sideProcess.side1ActiveSide -> side1ActiveSide;
end Side1.impl;

process Side1Process
  features
    side2ActiveSide: in data port behavior::integer;
    side3ActiveSide: in data port behavior::integer;
    side1Failed: in data port behavior::boolean;
    side1ActiveSide: out data port behavior::integer;
end Side1Process;

process implementation Side1Process.impl
  subcomponents
    activeStandbyThread: thread ActiveStandbyThread.impl;
  connections
    data port side2ActiveSide -> activeStandbyThread.side2ActiveSide;
    data port side3ActiveSide -> activeStandbyThread.side3ActiveSide;
    data port activeStandbyThread.side1ActiveSide -> side1ActiveSide;
    data port side1Failed -> activeStandbyThread.side1Failed;
  properties
    Dispatch_Protocol => Periodic applies to activeStandbyThread;
    SynchAADL::Deterministic => true applies to activeStandbyThread;
end Side1Process.impl;

thread ActiveStandbyThread
  features
    side2ActiveSide: in data port behavior::integer;
    side3ActiveSide: in data port behavior::integer;
    side1Failed: in data port behavior::boolean;
    side1ActiveSide: out data port behavior::integer;
end ActiveStandbyThread;

thread implementation ActiveStandbyThread.impl
  annex behavior_specification {**
    state variables
      g_side1:behavior::integer;
      g_side2:behavior::integer;
      g_side3:behavior::integer;
      prev_gside1:behavior::integer;
    initially
      g_side1 := -1;
      g_side2 := -1;
      g_side3 := -1;
      prev_gside1 := -1;states
      s0 : initial complete state;
    transitions
      s0 -[ on side1Failed'fresh and side1Failed = true ]-> s0 {
        if (side2ActiveSide'fresh)
          g_side2 := side2ActiveSide;else
          g_side2 := -1;end if;

        if (side3ActiveSide'fresh)
```

```
          g_side3 := side3ActiveSide;else
          g_side3 := -1;end if;

        g_side1 := prev_gside1;
        side1ActiveSide := -1;
        prev_gside1 := -1;};
      s0 -[ on side1Failed'fresh and side1Failed = false ]-> s0 {
        g_side1 := prev_gside1;
        if (side2ActiveSide'fresh)
          g_side2 := side2ActiveSide;else
          g_side2 := -1;end if;

        if (side3ActiveSide'fresh)
          g_side3 := side3ActiveSide;else
          g_side3 := -1;end if;

        if (g_side2 = (-1) and g_side3 = -1)
          side1ActiveSide := 1;
          prev_gside1 := 1;
        elsif (g_side1 = (-1) and (g_side2 != (-1) or g_side3 != -1))
          side1ActiveSide := 0;
          prev_gside1 := 0;
        elsif (g_side1 = 0 and g_side2 != 1 and g_side3 != 1)
          side1ActiveSide := 1;
          prev_gside1 := 1;
  else
          side1ActiveSide := prev_gside1;
  end if;
      };
**};
  end ActiveStandbyThread.impl;
end Side1;

package Side2
public
  system Side2
    features
      side3ActiveSide: in data port behavior::integer;
      side1ActiveSide: in data port behavior::integer;
      side2Failed: in data port behavior::boolean;
      side2ActiveSide: out data port behavior::integer;
  end Side2;

  system implementation Side2.impl
    subcomponents
      sideProcess: process Side2Process.impl;
    connections
      data port side3ActiveSide -> sideProcess.side3ActiveSide;
      data port side1ActiveSide -> sideProcess.side1ActiveSide;
      data port side2Failed -> sideProcess.side2Failed;
      data port sideProcess.side2ActiveSide -> side2ActiveSide;
  end Side2.impl;

  process Side2Process
    features
      side3ActiveSide: in data port behavior::integer;
      side1ActiveSide: in data port behavior::integer;
      side2Failed: in data port behavior::boolean;
      side2ActiveSide: out data port behavior::integer;
  end Side2Process;

  process implementation Side2Process.impl
```

```
  subcomponents
    activeStandbyThread: thread ActiveStandbyThread.impl;
  connections
    data port side3ActiveSide -> activeStandbyThread.side3ActiveSide;
    data port side1ActiveSide -> activeStandbyThread.side1ActiveSide;
    data port side2Failed -> activeStandbyThread.side2Failed;
    data port activeStandbyThread.side2ActiveSide -> side2ActiveSide;
  properties
    Dispatch_Protocol => Periodic applies to activeStandbyThread;
    SynchAADL::Deterministic => true applies to activeStandbyThread;
end Side2Process.impl;

thread ActiveStandbyThread
  features
    side3ActiveSide: in data port behavior::integer;
    side1ActiveSide: in data port behavior::integer;
    side2Failed: in data port behavior::boolean;
    side2ActiveSide: out data port behavior::integer;
end ActiveStandbyThread;

thread implementation ActiveStandbyThread.impl
  annex behavior_specification {**
    state variables
      g_side1:behavior::integer;
      g_side2:behavior::integer;
      g_side3:behavior::integer;
      prev_gside2:behavior::integer;
    initially
      g_side1 := -1;
      g_side2 := -1;
      g_side3 := -1;
      prev_gside2 := -1;states
      s0 : initial complete state;
    transitions
      s0 -[ on side2Failed'fresh and side2Failed = true ]-> s0 {
        g_side2 := prev_gside2;
        if (side1ActiveSide'fresh)
          g_side1 := side1ActiveSide;else
          g_side1 := -1;end if;

        if (side3ActiveSide'fresh)
          g_side3 := side3ActiveSide;else
          g_side3 := -1;end if;

        side2ActiveSide := -1;
        prev_gside2 := -1;};
      s0 -[ on side2Failed'fresh and side2Failed = false ]-> s0 {
        g_side2 := prev_gside2;
        if (side1ActiveSide'fresh)
          g_side1 := side1ActiveSide;else
          g_side1 := -1;end if;

        if (side3ActiveSide'fresh)
          g_side3 := side3ActiveSide;else
          g_side3 := -1;end if;

        if (g_side2 = -1)
          side2ActiveSide := 0;
          prev_gside2 := 0;
        elsif (g_side2 = 0 and g_side1 = (-1) and g_side3 != 1)
          side2ActiveSide := 1;
          prev_gside2 := 1;
```

```
  else
            side2ActiveSide := prev_gside2;
  end if;
        };
**};
  end ActiveStandbyThread.impl;
end Side2;

package Side3
public
  system Side3
    features
      side1ActiveSide: in data port behavior::integer;
      side2ActiveSide: in data port behavior::integer;
      side3Failed: in data port behavior::boolean;
      side3ActiveSide: out data port behavior::integer;
  end Side3;

  system implementation Side3.impl
    subcomponents
      sideProcess: process Side3Process.impl;
    connections
      data port side1ActiveSide -> sideProcess.side1ActiveSide;
      data port side2ActiveSide -> sideProcess.side2ActiveSide;
      data port side3Failed -> sideProcess.side3Failed;
      data port sideProcess.side3ActiveSide -> side3ActiveSide;
  end Side3.impl;

  process Side3Process
    features
      side1ActiveSide: in data port behavior::integer;
      side2ActiveSide: in data port behavior::integer;
      side3Failed: in data port behavior::boolean;
      side3ActiveSide: out data port behavior::integer;
  end Side3Process;

  process implementation Side3Process.impl
    subcomponents
      activeStandbyThread: thread ActiveStandbyThread.impl;
    connections
      data port side1ActiveSide -> activeStandbyThread.side1ActiveSide;
      data port side2ActiveSide -> activeStandbyThread.side2ActiveSide;
      data port activeStandbyThread.side3ActiveSide -> side3ActiveSide;
      data port side3Failed -> activeStandbyThread.side3Failed;
    properties
      Dispatch_Protocol => Periodic applies to activeStandbyThread;
      SynchAADL::Deterministic => true applies to activeStandbyThread;
  end Side3Process.impl;

  thread ActiveStandbyThread
    features
      side1ActiveSide: in data port behavior::integer;
      side2ActiveSide: in data port behavior::integer;
      side3Failed: in data port behavior::boolean;
      side3ActiveSide: out data port behavior::integer;
  end ActiveStandbyThread;

  thread implementation ActiveStandbyThread.impl
    annex behavior_specification {**
      state variables
        g_side1:behavior::integer;
        g_side2:behavior::integer;
```

```
        g_side3:behavior::integer;
        prev_gside3:behavior::integer;
      initially
        g_side1 := -1;
        g_side2 := -1;
        g_side3 := -1;
        prev_gside3 := -1;states
        s0 : initial complete state;
      transitions
        s0 -[ on side3Failed'fresh and side3Failed = true ]-> s0 {
          g_side3 := prev_gside3;
          if (side1ActiveSide'fresh)
            g_side1 := side1ActiveSide;else
            g_side1 := -1;end if;

          if (side2ActiveSide'fresh)
            g_side2 := side2ActiveSide;else
            g_side2 := -1;end if;

          side3ActiveSide := -1;
          prev_gside3 := -1;};
        s0 -[ on side3Failed'fresh and side3Failed = false ]-> s0 {
          g_side3 := prev_gside3;
          if (side1ActiveSide'fresh)
            g_side1 := side1ActiveSide;else
            g_side1 := -1;end if;

          if (side2ActiveSide'fresh)
            g_side2 := side2ActiveSide;else
            g_side2 := -1;end if;

          if (g_side3 = -1)
            side3ActiveSide := 0;
            prev_gside3 := 0;
          elsif (g_side3 != (-1) and g_side1 = (-1) and g_side2 = -1)
            side3ActiveSide := 1;
            prev_gside3 := 1;
  else
            side3ActiveSide := prev_gside3;
  end if;
        };
**};
  end ActiveStandbyThread.impl;
end Side3;
```

B.2 Real-Time Maude Specification

```
load rtmaude/synch-aadl-interpreter.maude
(tomod MainMainSystemimplInstance is
  including SYNCHRONOUS-STEP .

  --- names, states, variables
  ops s2F s3F s1F : -> BoolVarId [ctor] .
  ops prevgside2 prevgside1 side1ActiveSideout side3ActiveSideout gside1 side2ActiveSideout
      prevgside3 gside2 gside3 : -> IntVarId [ctor] .
  ops s0 initState curState : -> Location [ctor] .
  ops impl : -> ImplName [ctor] .
  ops Side1::ActiveStandbyThread Aileron::OutputSynchronizer Environment::EnvProcess
     Aileron::IOProcess Side2::Side2 Side3::Side3 Side1::Side1 Aileron::IOTask Environment::Environment
     Side3::Side3Process Side3::ActiveStandbyThread Side2::Side2Process MainModule::MainSystem
     Side1::Side1Process Environment::InputSynchronizer Side2::ActiveStandbyThread : -> TypeName [ctor] .
```

```
ops side3Failed side2ActiveSidein side3ActiveSidein side3ActiveSide side1ActiveSidein side1ActiveSide
   side1Failed side2ActiveSide side2Failed : -> PortId [ctor] .
ops environment sideOne outsynchThread insynchThread sideTwo envProcess aileron activeStandbyThread
   MainMainSystemimplInstance ioProcess sideThree sideProcess : -> ComponentId [ctor] .

--- the initial state
op initial : -> Configuration .
eq initial = MainMainSystemimplInstance : system MainModule::MainSystem . impl .
eq MAIN = MainMainSystemimplInstance .

---------------------------------------------
--- AADL instance
---------------------------------------------

var COMP : ComponentId .

eq stateVariables(thread Side1::ActiveStandbyThread . impl) =
  (gside1 |-> -1)
  (gside2 |-> -1)
  (gside3 |-> -1)
  (prevgside1 |-> -1) .

eq states(thread Side1::ActiveStandbyThread . impl) =
  initial: s0
  complete: s0
 .

eq transitions(thread Side1::ActiveStandbyThread . impl) =
  (s0 -[(fresh(side1Failed) and (side1Failed = true))]-> s0 {
    (if (fresh(side2ActiveSide))
      (
      (gside2 := side2ActiveSide))
    else
      (
      (gside2 := -1))end if);
    (if (fresh(side3ActiveSide))
      (
      (gside3 := side3ActiveSide))
    else
      (
      (gside3 := -1))end if);
    (gside1 := prevgside1);
    (side1ActiveSide := -1);
    (prevgside1 := -1)});
  (s0 -[(fresh(side1Failed) and (side1Failed = false))]-> s0 {
    (gside1 := prevgside1);
    (if (fresh(side2ActiveSide))
      (
      (gside2 := side2ActiveSide))
    else
      (
      (gside2 := -1))end if);
    (if (fresh(side3ActiveSide))
      (
      (gside3 := side3ActiveSide))
    else
      (
      (gside3 := -1))end if);
    (if (((gside2 = -1) and (gside3 = -1)))
      (
      (side1ActiveSide := 1);
      (prevgside1 := 1))
```

```
    ((elsif (((gside1 = -1) and (((gside2 != -1) or (gside3 != -1)))))
      (
      (side1ActiveSide := 0);
      (prevgside1 := 0)))
    (elsif ((((gside1 = 0) and (gside2 != 1)) and (gside3 != 1)))
      (
      (side1ActiveSide := 1);
      (prevgside1 := 1))))
    else
      (side1ActiveSide := prevgside1) end if)}) .


eq COMP : thread Side1::ActiveStandbyThread . impl =
  < COMP : Thread |
    features :
      < side2ActiveSide : InDataThreadPort | content : noMsg, fresh : false >
      < side3ActiveSide : InDataThreadPort | content : noMsg, fresh : false >
      < side1Failed : InDataThreadPort | content : noMsg, fresh : false >
      < side1ActiveSide : OutDataPort | content : noMsg >,
    behaviorRef : (thread Side1::ActiveStandbyThread . impl),
    variables : stateVariables(thread Side1::ActiveStandbyThread . impl),
    currState : initialState(thread Side1::ActiveStandbyThread . impl),
    completeStates : completeStates(thread Side1::ActiveStandbyThread . impl),
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2) ;
      DispatchProtocol(Periodic) ;
      Deterministic(true),
    connections :
      none,
    subcomponents :
      none
  > .

eq COMP : process Side1::Side1Process . impl =
  < COMP : Process |
    features :
      < side2ActiveSide : InDataPort | content : noMsg >
      < side3ActiveSide : InDataPort | content : noMsg >
      < side1Failed : InDataPort | content : noMsg >
      < side1ActiveSide : OutDataPort | content : noMsg >,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (activeStandbyThread . side1ActiveSide --> side1ActiveSide) ;
      (side2ActiveSide --> activeStandbyThread . side2ActiveSide) ;
      (side3ActiveSide --> activeStandbyThread . side3ActiveSide) ;
      (side1Failed --> activeStandbyThread . side1Failed),
    subcomponents :
      (activeStandbyThread : thread Side1::ActiveStandbyThread . impl)
  > .

eq COMP : system Side1::Side1 . impl =
  < COMP : System |
    features :
      < side2ActiveSide : InDataPort | content : noMsg >
      < side3ActiveSide : InDataPort | content : noMsg >
      < side1Failed : InDataPort | content : noMsg >
      < side1ActiveSide : OutDataPort | content : noMsg >,
```

```
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (sideProcess . side1ActiveSide --> side1ActiveSide) ;
      (side2ActiveSide --> sideProcess . side2ActiveSide) ;
      (side3ActiveSide --> sideProcess . side3ActiveSide) ;
      (side1Failed --> sideProcess . side1Failed),
    subcomponents :
      (sideProcess : process Side1::Side1Process . impl)
  > .

eq stateVariables(thread Side2::ActiveStandbyThread . impl) =
  (gside1 |-> -1)
  (gside2 |-> -1)
  (gside3 |-> -1)
  (prevgside2 |-> -1) .

eq states(thread Side2::ActiveStandbyThread . impl) =
  initial: s0
  complete: s0
 .

eq transitions(thread Side2::ActiveStandbyThread . impl) =
  (s0 -[(fresh(side2Failed) and (side2Failed = true))]-> s0 {
    (gside2 := prevgside2);
    (if (fresh(side1ActiveSide))
      (
      (gside1 := side1ActiveSide))
    else
      (
      (gside1 := -1))end if);
    (if (fresh(side3ActiveSide))
      (
      (gside3 := side3ActiveSide))
    else
      (
      (gside3 := -1))end if);
    (side2ActiveSide := -1);
    (prevgside2 := -1)});
  (s0 -[(fresh(side2Failed) and (side2Failed = false))]-> s0 {
    (gside2 := prevgside2);
    (if (fresh(side1ActiveSide))
      (
      (gside1 := side1ActiveSide))
    else
      (
      (gside1 := -1))end if);
    (if (fresh(side3ActiveSide))
      (
      (gside3 := side3ActiveSide))
    else
      (
      (gside3 := -1))end if);
    (if ((gside2 = -1))
      (
      (side2ActiveSide := 0);
      (prevgside2 := 0))
    ((elsif ((((gside2 = 0) and (gside1 = -1)) and (gside3 != 1)))
      (
      (side2ActiveSide := 1);
```

```
      (prevgside2 := 1))))
    else
      (
      (side2ActiveSide := prevgside2))end if)}) .

eq COMP : thread Side2::ActiveStandbyThread . impl =
  < COMP : Thread |
    features :
      < side3ActiveSide : InDataThreadPort | content : noMsg, fresh : false >
      < side1ActiveSide : InDataThreadPort | content : noMsg, fresh : false >
      < side2Failed : InDataThreadPort | content : noMsg, fresh : false >
      < side2ActiveSide : OutDataPort | content : noMsg >,
    behaviorRef : (thread Side2::ActiveStandbyThread . impl),
    variables : stateVariables(thread Side2::ActiveStandbyThread . impl),
    currState : initialState(thread Side2::ActiveStandbyThread . impl),
    completeStates : completeStates(thread Side2::ActiveStandbyThread . impl),
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2) ;
      DispatchProtocol(Periodic) ;
      Deterministic(true),
    connections :
      none,
    subcomponents :
      none
  > .

eq COMP : process Side2::Side2Process . impl =
  < COMP : Process |
    features :
      < side3ActiveSide : InDataPort | content : noMsg >
      < side1ActiveSide : InDataPort | content : noMsg >
      < side2Failed : InDataPort | content : noMsg >
      < side2ActiveSide : OutDataPort | content : noMsg >,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (side1ActiveSide --> activeStandbyThread . side1ActiveSide) ;
      (activeStandbyThread . side2ActiveSide --> side2ActiveSide) ;
      (side3ActiveSide --> activeStandbyThread . side3ActiveSide) ;
      (side2Failed --> activeStandbyThread . side2Failed),
    subcomponents :
      (activeStandbyThread : thread Side2::ActiveStandbyThread . impl)
  > .

eq COMP : system Side2::Side2 . impl =
  < COMP : System |
    features :
      < side3ActiveSide : InDataPort | content : noMsg >
      < side1ActiveSide : InDataPort | content : noMsg >
      < side2Failed : InDataPort | content : noMsg >
      < side2ActiveSide : OutDataPort | content : noMsg >,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (side1ActiveSide --> sideProcess . side1ActiveSide) ;
      (sideProcess . side2ActiveSide --> side2ActiveSide) ;
```

```
      (side3ActiveSide --> sideProcess . side3ActiveSide) ;
      (side2Failed --> sideProcess . side2Failed),
    subcomponents :
      (sideProcess : process Side2::Side2Process . impl)
  > .

eq stateVariables(thread Side3::ActiveStandbyThread . impl) =
  (gside1 |-> -1)
  (gside2 |-> -1)
  (gside3 |-> -1)
  (prevgside3 |-> -1) .

eq states(thread Side3::ActiveStandbyThread . impl) =
  initial: s0
  complete: s0
 .

eq transitions(thread Side3::ActiveStandbyThread . impl) =
  (s0 -[(fresh(side3Failed) and (side3Failed = true))]-> s0 {
    (gside3 := prevgside3);
    (if (fresh(side1ActiveSide))
      (
      (gside1 := side1ActiveSide))
    else
      (
      (gside1 := -1))end if);
    (if (fresh(side2ActiveSide))
      (
      (gside2 := side2ActiveSide))
    else
      (
      (gside2 := -1))end if);
    (side3ActiveSide := -1);
    (prevgside3 := -1)});
  (s0 -[(fresh(side3Failed) and (side3Failed = false))]-> s0 {
    (gside3 := prevgside3);
    (if (fresh(side1ActiveSide))
      (
      (gside1 := side1ActiveSide))
    else
      (
      (gside1 := -1))end if);
    (if (fresh(side2ActiveSide))
      (
      (gside2 := side2ActiveSide))
    else
      (
      (gside2 := -1))end if);
    (if ((gside3 = -1))
      (
      (side3ActiveSide := 0);
      (prevgside3 := 0))
    ((elsif ((((gside3 != -1) and (gside1 = -1)) and (gside2 = -1)))
      (
      (side3ActiveSide := 1);
      (prevgside3 := 1))))
    else
      (
      (side3ActiveSide := prevgside3))end if)}) .

eq COMP : thread Side3::ActiveStandbyThread . impl =
  < COMP : Thread |
```

```
    features :
      < side1ActiveSide : InDataThreadPort | content : noMsg, fresh : false >
      < side2ActiveSide : InDataThreadPort | content : noMsg, fresh : false >
      < side3Failed : InDataThreadPort | content : noMsg, fresh : false >
      < side3ActiveSide : OutDataPort | content : noMsg >,
    behaviorRef : (thread Side3::ActiveStandbyThread . impl),
    variables : stateVariables(thread Side3::ActiveStandbyThread . impl),
    currState : initialState(thread Side3::ActiveStandbyThread . impl),
    completeStates : completeStates(thread Side3::ActiveStandbyThread . impl),
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2) ;
      DispatchProtocol(Periodic) ;
      Deterministic(true),
    connections :
      none,
    subcomponents :
      none
  > .

eq COMP : process Side3::Side3Process . impl =
  < COMP : Process |
    features :
      < side1ActiveSide : InDataPort | content : noMsg >
      < side2ActiveSide : InDataPort | content : noMsg >
      < side3Failed : InDataPort | content : noMsg >
      < side3ActiveSide : OutDataPort | content : noMsg >,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (side1ActiveSide --> activeStandbyThread . side1ActiveSide) ;
      (side2ActiveSide --> activeStandbyThread . side2ActiveSide) ;
      (activeStandbyThread . side3ActiveSide --> side3ActiveSide) ;
      (side3Failed --> activeStandbyThread . side3Failed),
    subcomponents :
      (activeStandbyThread : thread Side3::ActiveStandbyThread . impl)
  > .

eq COMP : system Side3::Side3 . impl =
  < COMP : System |
    features :
      < side1ActiveSide : InDataPort | content : noMsg >
      < side2ActiveSide : InDataPort | content : noMsg >
      < side3Failed : InDataPort | content : noMsg >
      < side3ActiveSide : OutDataPort | content : noMsg >,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (side1ActiveSide --> sideProcess . side1ActiveSide) ;
      (side2ActiveSide --> sideProcess . side2ActiveSide) ;
      (sideProcess . side3ActiveSide --> side3ActiveSide) ;
      (side3Failed --> sideProcess . side3Failed),
    subcomponents :
      (sideProcess : process Side3::Side3Process . impl)
  > .

eq stateVariables(thread Environment::InputSynchronizer . impl) =
```

```
  (s1F |-> false)
  (s2F |-> false)
  (s3F |-> false) .

eq states(thread Environment::InputSynchronizer . impl) =
  initial: s0
  complete: s0
 .

eq transitions(thread Environment::InputSynchronizer . impl) =
  (s0 -[]-> s0 {
    (side1Failed := s1F);
    (side2Failed := s2F);
    (side3Failed := s3F)}) .

eq COMP : thread Environment::InputSynchronizer . impl =
  < COMP : Thread |
    features :
      < side1Failed : OutDataPort | content : noMsg >
      < side2Failed : OutDataPort | content : noMsg >
      < side3Failed : OutDataPort | content : noMsg >,
    behaviorRef : (thread Environment::InputSynchronizer . impl),
    variables : stateVariables(thread Environment::InputSynchronizer . impl),
    currState : initialState(thread Environment::InputSynchronizer . impl),
    completeStates : completeStates(thread Environment::InputSynchronizer . impl),
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2) ;
      DispatchProtocol(Periodic) ;
      IsEnvironment(true) ;
      InputConstraints(not s1F or not s2F or not s3F),
    connections :
      none,
    subcomponents :
      none
  > .

eq COMP : process Environment::EnvProcess . impl =
  < COMP : Process |
    features :
      < side1Failed : OutDataPort | content : noMsg >
      < side2Failed : OutDataPort | content : noMsg >
      < side3Failed : OutDataPort | content : noMsg >,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (insynchThread . side1Failed --> side1Failed) ;
      (insynchThread . side2Failed --> side2Failed) ;
      (insynchThread . side3Failed --> side3Failed),
    subcomponents :
      (insynchThread : thread Environment::InputSynchronizer . impl)
  > .

eq COMP : system Environment::Environment . impl =
  < COMP : System |
    features :
      < side1Failed : OutDataPort | content : noMsg >
      < side2Failed : OutDataPort | content : noMsg >
      < side3Failed : OutDataPort | content : noMsg >,
```

```
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (envProcess . side1Failed --> side1Failed) ;
      (envProcess . side2Failed --> side2Failed) ;
      (envProcess . side3Failed --> side3Failed),
    subcomponents :
      (envProcess : process Environment::EnvProcess . impl)
  > .

eq stateVariables(thread Aileron::OutputSynchronizer . impl) =
  (side1ActiveSideout |-> -1)
  (side2ActiveSideout |-> -1)
  (side3ActiveSideout |-> -1) .

eq states(thread Aileron::OutputSynchronizer . impl) =
  initial: initState
  complete: initState
  complete: curState
 .

eq transitions(thread Aileron::OutputSynchronizer . impl) =
  (initState -[]-> curState {
    (side1ActiveSideout := -1);
    (side2ActiveSideout := -1);
    (side3ActiveSideout := -1)});
  (curState -[]-> curState {
    (side1ActiveSideout := side1ActiveSidein);
    (side2ActiveSideout := side2ActiveSidein);
    (side3ActiveSideout := side3ActiveSidein)}) .

eq COMP : thread Aileron::OutputSynchronizer . impl =
  < COMP : Thread |
    features :
      < side1ActiveSidein : InDataThreadPort | content : noMsg, fresh : false >
      < side2ActiveSidein : InDataThreadPort | content : noMsg, fresh : false >
      < side3ActiveSidein : InDataThreadPort | content : noMsg, fresh : false >,
    behaviorRef : (thread Aileron::OutputSynchronizer . impl),
    variables : stateVariables(thread Aileron::OutputSynchronizer . impl),
    currState : initialState(thread Aileron::OutputSynchronizer . impl),
    completeStates : completeStates(thread Aileron::OutputSynchronizer . impl),
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2) ;
      DispatchProtocol(Periodic) ;
      Deterministic(true),
    connections :
      none,
    subcomponents :
      none
  > .

eq COMP : process Aileron::IOProcess . impl =
  < COMP : Process |
    features :
      < side1ActiveSidein : InDataPort | content : noMsg >
      < side2ActiveSidein : InDataPort | content : noMsg >
      < side3ActiveSidein : InDataPort | content : noMsg >,
    properties :
```

```
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (side1ActiveSidein --> outsynchThread . side1ActiveSidein) ;
      (side2ActiveSidein --> outsynchThread . side2ActiveSidein) ;
      (side3ActiveSidein --> outsynchThread . side3ActiveSidein),
    subcomponents :
      (outsynchThread : thread Aileron::OutputSynchronizer . impl)
  > .

eq COMP : system Aileron::IOTask . impl =
  < COMP : System |
    features :
      < side1ActiveSide : InDataPort | content : noMsg >
      < side2ActiveSide : InDataPort | content : noMsg >
      < side3ActiveSide : InDataPort | content : noMsg >,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (side1ActiveSide --> ioProcess . side1ActiveSidein) ;
      (side2ActiveSide --> ioProcess . side2ActiveSidein) ;
      (side3ActiveSide --> ioProcess . side3ActiveSidein),
    subcomponents :
      (ioProcess : process Aileron::IOProcess . impl)
  > .

eq COMP : system MainModule::MainSystem . impl =
  < COMP : System |
    features :
      none,
    properties :
      Synchronous(true) ;
      syncPeriod(2) ;
      Period(2),
    connections :
      (sideOne . side1ActiveSide -->> sideTwo . side1ActiveSide) ;
      (sideOne . side1ActiveSide -->> sideThree . side1ActiveSide) ;
      (sideOne . side1ActiveSide -->> aileron . side1ActiveSide) ;
      (sideTwo . side2ActiveSide -->> sideOne . side2ActiveSide) ;
      (sideTwo . side2ActiveSide -->> sideThree . side2ActiveSide) ;
      (sideTwo . side2ActiveSide -->> aileron . side2ActiveSide) ;
      (sideThree . side3ActiveSide -->> sideOne . side3ActiveSide) ;
      (sideThree . side3ActiveSide -->> sideTwo . side3ActiveSide) ;
      (sideThree . side3ActiveSide -->> aileron . side3ActiveSide) ;
      (environment . side1Failed --> sideOne . side1Failed) ;
      (environment . side2Failed --> sideTwo . side2Failed) ;
      (environment . side3Failed --> sideThree . side3Failed),
    subcomponents :
      (sideTwo : system Side2::Side2 . impl)
      (sideThree : system Side3::Side3 . impl)
      (aileron : system Aileron::IOTask . impl)
      (environment : system Environment::Environment . impl)
      (sideOne : system Side1::Side1 . impl)
  > .

endtom)
```

B.3 LTL Property Specification in Real-Time Maude

```
(tomod Main_MainSystem_impl_Instance-VERIFICATION-DEF is
  including MainMainSystemimplInstance .
  including LTL-MODEL-CHECK-AADL .

  op side1Active : -> Formula .
  eq side1Active
   = value of side1ActiveSideout in component (MAIN -> aileron -> ioProcess -> outsynchThread) is 1 .
  op side2Active : -> Formula .
  eq side2Active
   = value of side2ActiveSideout in component (MAIN -> aileron -> ioProcess -> outsynchThread) is 1 .
  op side3Active : -> Formula .
  eq side3Active
   = value of side3ActiveSideout in component (MAIN -> aileron -> ioProcess -> outsynchThread) is 1 .
  op side1Standby : -> Formula .
  eq side1Standby
   = value of side1ActiveSideout in component (MAIN -> aileron -> ioProcess -> outsynchThread) is 0 .
  op side2Standby : -> Formula .
  eq side2Standby
   = value of side2ActiveSideout in component (MAIN -> aileron -> ioProcess -> outsynchThread) is 0 .
  op side3Standby : -> Formula .
  eq side3Standby
   = value of side3ActiveSideout in component (MAIN -> aileron -> ioProcess -> outsynchThread) is 0 .
  op flipOne : -> Formula .
  eq flipOne
   = (value of s1F in component (MAIN -> environment -> envProcess -> insynchThread) is true
     <-> O (value of s1F in component (MAIN -> environment -> envProcess -> insynchThread) is false))
    \/
    (value of s1F in component (MAIN -> environment -> envProcess -> insynchThread) is false
     <-> O (value of s1F in component (MAIN -> environment -> envProcess -> insynchThread) is true)) .
  op flipTwo : -> Formula .
  eq flipTwo
   = (value of s2F in component (MAIN -> environment -> envProcess -> insynchThread) is true
     <-> O (value of s2F in component (MAIN -> environment -> envProcess -> insynchThread) is false))
    \/
    (value of s2F in component (MAIN -> environment -> envProcess -> insynchThread) is false
     <-> O (value of s2F in component (MAIN -> environment -> envProcess -> insynchThread) is true)) .
  op flipThree : -> Formula .
  eq flipThree
   = (value of s3F in component (MAIN -> environment -> envProcess -> insynchThread) is true
     <-> O (value of s3F in component (MAIN -> environment -> envProcess -> insynchThread) is false))
    \/
    (value of s3F in component (MAIN -> environment -> envProcess -> insynchThread) is false
     <-> O (value of s3F in component (MAIN -> environment -> envProcess -> insynchThread) is true)) .
  op sameOne : -> Formula .
  eq sameOne
   = (value of gside1 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 0
     <->
   value of gside1 in component (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is 0)
  /\
  (value of gside1 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 0
     <->
   value of gside1 in component (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is 0) /\
  (value of gside1 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 1
     <->
   value of gside1 in component (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is 1)
  /\
  (value of gside1 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 1
     <->
   value of gside1 in component (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is 1) /\
  (value of gside1 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is -1
```

```
    <->
   value of gside1 in component (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is -1)
  /\
  (value of gside1 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is -1
    <->
   value of gside1 in component (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is -1) .
  op sameTwo : -> Formula .
  eq sameTwo
   = (value of gside2 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 0
    <->
   value of gside2 in component (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is 0)
  /\
  (value of gside2 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 0
    <->
   value of gside2 in component (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is 0) /\
  (value of gside2 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 1
    <->
   value of gside2 in component (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is 1)
  /\
  (value of gside2 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 1
    <->
   value of gside2 in component (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is 1) /\
  (value of gside2 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is -1
    <->
   value of gside2 in component (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is -1)
  /\
  (value of gside2 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is -1
    <->
   value of gside2 in component (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is -1) .
  op sameThree : -> Formula .
  eq sameThree
   = (value of gside3 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 0
    <->
   value of gside3 in component (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is 0)
  /\
  (value of gside3 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 0
    <->
   value of gside3 in component (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is 0) /\
  (value of gside3 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 1
    <->
   value of gside3 in component (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is 1)
  /\
  (value of gside3 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is 1
    <->
   value of gside3 in component (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is 1) /\
  (value of gside3 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is -1
    <->
   value of gside3 in component (MAIN -> sideThree -> sideProcess -> activeStandbyThread) is -1)
  /\
  (value of gside3 in component (MAIN -> sideOne -> sideProcess -> activeStandbyThread) is -1
    <->
   value of gside3 in component (MAIN -> sideTwo -> sideProcess -> activeStandbyThread) is -1) .
endtom)

(mc {initial} |=u [] ( ~(side1Active /\ side2Active) /\
         ~(side1Active /\ side3Active) /\
         ~(side2Active /\ side3Active)) .)    --- label: R1
(mc {initial} |=u (([]<> (~ flipOne /\ ~ flipTwo /\ ~ flipThree) )->
     ([]( (~ side1Active /\ ~ side2Active /\ ~ side3Active) ->
      (
      <> (side1Active \/ side2Active \/ side3Active)
      )
```

```
      ))) .)    --- label: R2
(mc {initial} |=u (([] ((~ flipOne /\ ~ flipTwo /\  flipThree) \/
                        (~ flipOne /\  flipTwo /\ ~ flipThree) \/
                        ( flipOne /\ ~ flipTwo /\ ~ flipThree) \/
                        (~ flipOne /\ ~ flipTwo /\ ~ flipThree)) )
                        ->
     ([]( (~ side1Active /\ ~ side2Active /\ ~ side3Active) ->
      (
      O (side1Active \/ side2Active \/ side3Active)
      \/
      O O (side1Active \/ side2Active \/ side3Active)
      \/
      O O O (side1Active \/ side2Active \/ side3Active)
            \/
      O O O O (side1Active \/ side2Active \/ side3Active)
      )
      ))) .)    --- label: R3
(mc {initial} |=u [] (sameOne /\ sameTwo /\ sameThree) .)    --- label: R4
```

C Real-Time Maude semantics of Synchronous AADL

In this section, the entire Real-Time Maude semantics of Synchronous AADL is given.

```
(omod BASIC-SORTS is
  protecting INT .
  sorts PortId ComponentId .    subsorts PortId ComponentId < Oid .
  sort PortName .  subsort PortId < PortName .
  op _._ : ComponentId PortId -> PortName [ctor] .
endom)

(omod COMPONENT-REF is including BASIC-SORTS .
  sort TypeName .   --- name of qualified component type name
  sort ImplName .   --- name of implementation
  sort CompCategory . --- name of component category
  ops system thread process : -> CompCategory [ctor] .

  --- top module name ID, where the corresponding equation is generated by code-gen.
  op MAIN : -> ComponentId [ctor] .

  --- component type
  sort CompType .    subsort TypeName < CompType .
  op _._ : TypeName ImplName -> CompType [ctor prec 0] .

  --- reference to a component
  sort ComponentRef .
  op __ : CompCategory CompType -> ComponentRef [ctor prec 1] .

  --- component instance operator, where each equation is generated by code generation.
  op _:_ : ComponentId ComponentRef -> Object [prec 2] .
endom)

(omod PORTS is inc BASIC-SORTS .
  class Port | content : MsgContent .    --- Ports are represented as objects ...
  sort MsgContent .
  op noMsg : -> MsgContent [ctor] .
endom)

(omod DATA-PORTS is including PORTS .
 op data : Bool -> MsgContent [ctor] .
 op data : Int -> MsgContent [ctor] .

 class InDataPort .    class OutDataPort .
 subclass InDataPort OutDataPort < Port .
endom)

(omod THREAD-PORTS is including DATA-PORTS .
 class InDataThreadPort  | fresh : Bool .    subclass InDataThreadPort < InDataPort .

 var P : PortId . var PORTS : Configuration . var B : Bool . var I : Int . var MSGC : MsgContent .

 --- Functions on input thread ports
 op newMsgInPort : PortId Configuration ~> Bool .
 eq newMsgInPort(P,  < P : InDataThreadPort | fresh : B >  PORTS) = B .
 op intFromPort : PortId Configuration ~> Int .
 op boolFromPort : PortId Configuration ~> Bool .
 eq intFromPort(P,  < P : InDataThreadPort | content : data(I) > PORTS) = I .
 eq boolFromPort(P,  < P : InDataThreadPort | content : data(B) > PORTS) = B .

 --- OUTPUT PORTS FROM THREADS
```

```
 op writeData : Bool PortId Configuration ~> Configuration .
 op writeData : Int PortId Configuration ~> Configuration .
 eq writeData(I, P, < P : OutDataPort | >  PORTS)  = < P : OutDataPort | content : data(I) >  PORTS .
 eq writeData(B, P, < P : OutDataPort | >  PORTS)  = < P : OutDataPort | content : data(B) >  PORTS .
endom)

(omod BEHAVIOR-LANGUAGE is pr BASIC-SORTS + INT .
  sorts BoolVarId IntVarId .
  sorts BoolExpression IntExpression .
  subsort BoolVarId Bool < BoolExpression .
  subsort IntVarId Int < IntExpression .
  op fresh : PortId -> BoolExpression [ctor] .    --- p'fresh

  op _=_ : IntExpression IntExpression -> BoolExpression [ctor] .
  op _>=_ : IntExpression IntExpression -> BoolExpression [ctor ditto] .
  op _>_ : IntExpression IntExpression -> BoolExpression [ctor ditto] .
  op _<=_ : IntExpression IntExpression -> BoolExpression [ctor ditto] .
  op _<_ : IntExpression IntExpression -> BoolExpression [ctor ditto] .
  op _!=_ : IntExpression IntExpression -> BoolExpression [ctor] .

  op _=_ : PortId IntExpression -> BoolExpression [ctor] .
  op _=_ : IntExpression PortId -> BoolExpression [ctor] .
  op _>=_ : PortId IntExpression -> BoolExpression [ctor] .
  op _>=_ : IntExpression PortId -> BoolExpression [ctor] .
  op _>_ : PortId IntExpression -> BoolExpression [ctor] .
  op _>_ : IntExpression PortId -> BoolExpression [ctor] .
  op _<=_ : PortId IntExpression -> BoolExpression [ctor] .
  op _<=_ : IntExpression PortId -> BoolExpression [ctor] .
  op _<_ : PortId IntExpression -> BoolExpression [ctor] .
  op _<_ : IntExpression PortId -> BoolExpression [ctor] .
  op _!=_ : PortId IntExpression -> BoolExpression [ctor] .
  op _!=_ : IntExpression PortId -> BoolExpression [ctor] .

  op _-_ : IntExpression IntExpression -> IntExpression [ctor ditto] .
  op _+_ : IntExpression IntExpression -> IntExpression [ctor ditto] .

  op _=_ : PortId Bool -> BoolExpression [ctor] .
  op _=_ : BoolVarId Bool -> BoolExpression [ctor] .

  op _and_ : BoolExpression BoolExpression -> BoolExpression [ctor ditto] .
  op _or_ : BoolExpression BoolExpression -> BoolExpression [ctor ditto] .
  op not_ : BoolExpression -> BoolExpression [ctor ditto] .

  sort Statement .
  op _:=_ : IntVarId IntExpression -> Statement [ctor] .
  op _:=_ : BoolVarId BoolExpression -> Statement [ctor] .
  op _:=_ : PortId IntExpression -> Statement [ctor] .
  op _:=_ : PortId BoolExpression -> Statement [ctor] .
  op _:=_ : PortId PortId -> Statement [ctor] .
  op _:=_ : IntVarId PortId -> Statement [ctor] .
  op _:=_ : BoolVarId PortId -> Statement [ctor] .

  sort StatementList .
  op nil : -> StatementList [ctor] .
  subsort Statement < StatementList .
  op _;_ : StatementList StatementList -> StatementList  [ctor assoc id: nil] .

  op if`(_`)_end`if : BoolExpression StatementList -> Statement [ctor] .
  op if`(_`)_else_end`if : BoolExpression StatementList  StatementList -> Statement [ctor] .
  op if`(_`)__end`if : BoolExpression StatementList ElseIfs -> Statement [ctor] .
  op if`(_`)__else_end`if : BoolExpression StatementList ElseIfs StatementList -> Statement [ctor] .
```

```
  sort ElseIfs .
  op __ : ElseIfs ElseIfs -> ElseIfs [ctor assoc] .
  op elsif`(_`)_ : BoolExpression StatementList -> ElseIfs [ctor] .
  op while`(_`)`{_`} : BoolExpression StatementList -> Statement [ctor] .
endom)

(omod BEHAVIOR-PROG is inc BEHAVIOR-LANGUAGE + COMPONENT-REF .

  var CR : ComponentRef .  var SL : StatementList .
  vars L L1 L2 : Location .  vars LS LS1 LS2 : LocationSet .
  var LDS : LocationDeclSet .

  op stateVariables : ComponentRef ~> Valuation .
  op transitions : ComponentRef ~> TransitionSet [memo] .
  op states : ComponentRef ~> LocationDeclSet [memo] .

  --- state lookup ops
  op initialState : ComponentRef ~> Location .
  op completeStates : ComponentRef ~> LocationSet .

  --- assume only 1 initial state.
  ceq initialState(CR) = L if (initial: L) LDS := states(CR) .
  ceq completeStates(CR) = LS if (complete: LS) LDS := states(CR) .

--- state variables
  sort VarAssignment .
  op _|->_ : IntVarId Int -> VarAssignment [ctor] .
  op _|->_ : BoolVarId Bool -> VarAssignment [ctor] .

  sort Valuation .    --- set of variable assignments
  op emptyValuation : -> Valuation [ctor] .
  subsort VarAssignment < Valuation .
  op __ : Valuation Valuation -> Valuation  [ctor assoc comm id: emptyValuation] .

--- Locations and location sets:
  sorts Location LocationSet .
  subsort Location < LocationSet .
  op emptyLocationSet : -> LocationSet [ctor] .
  op __ : LocationSet LocationSet -> LocationSet  [ctor assoc comm id: emptyLocationSet] .

  sorts LocationDecl LocationDeclSet .
  subsort LocationDecl < LocationDeclSet .
  ops initial:_ : Location -> LocationDecl [ctor] .
  ops complete:_ other:_ : LocationSet -> LocationDecl [ctor] .
  op noLocDecl : -> LocationDeclSet [ctor] .
  op __ : LocationDeclSet LocationDeclSet -> LocationDeclSet [ctor assoc comm id: noLocDecl] .

  ceq complete: LS1 complete: LS2 = complete: (LS1 LS2)
      if LS1 =/= emptyLocationSet and LS2 =/= emptyLocationSet .
  ceq other: LS1 other: LS2 = other: (LS1 LS2)
      if LS1 =/= emptyLocationSet and LS2 =/= emptyLocationSet .

*** Transitions

  sort Transition .
  op _-`[_`]->_`{_`} : Location TransGuard Location StatementList  -> Transition [ctor] .

  sort TransitionSet .
  subsort Transition < TransitionSet .
  op emptyTransitionSet : -> TransitionSet [ctor] .
  op _;_ : TransitionSet TransitionSet -> TransitionSet [ctor assoc comm id: emptyTransitionSet] .
```

```
  --- GUARDS for transitions
  sort TransGuard .
  op on`dispatch : -> TransGuard [ctor] .

  --- Execution guards:
  subsort BoolExpression < TransGuard .
  op otherwise : -> TransGuard [ctor] .
  op _-`[`]->_`{_`}  : Location  Location StatementList -> Transition .
  eq L1 -[]-> L2 {SL} = L1 -[ true ]-> L2 {SL} .
endom)

(omod COMPONENT is
  including BASIC-SORTS .

  class Component | features : Configuration,    --- PORT objects!
                    properties : Properties,
                    subcomponents : Configuration,
                    connections : ConnectionSet .

  --- properties
  sorts Property Properties .
  subsort Property < Properties .
  op noProperty : -> Properties [ctor] .
  op _;_ : Properties Properties -> Properties  [ctor assoc comm id: noProperty] .

  --- Connections: immediate and delayed connections
  sort Connection .
  ops _-->_  _-->>_ : PortName PortName -> Connection [ctor] .

  sort ConnectionSet .
  subsort Connection < ConnectionSet .
  op none : -> Connection [ctor] .
  op _;_ : ConnectionSet ConnectionSet -> ConnectionSet [ctor assoc comm id: none] .
endom)

(omod SYSTEM is including COMPONENT .
  class System .       subclass System < Component .
endom)

(omod PROCESS is including COMPONENT .
  class Process .       subclass Process < Component .
endom)

(tomod THREAD is including COMPONENT + BEHAVIOR-PROG + THREAD-PORTS .

  class Thread | behaviorRef : ComponentRef,     --- to retrieve transitions
                 variables : Valuation,         --- behavior state variables
                 currState : Location,           --- behavior current state
                 completeStates : LocationSet .  --- behavior complete states
  subclass Thread < Component .
endtom)

*** Execute a transition.
(omod EXEC-TRANS is
  including BEHAVIOR-PROG .
  including THREAD-PORTS .

  vars P P1 P2 : PortId .
  vars BE BE2 BE3 BEXP1 BEXP2 : BoolExpression .
  var VAL : Valuation .
  var PORTS : Configuration .
  vars B B2 : Bool .
```

```
var BVAR : BoolVarId .
vars IE E1 E2 : IntExpression .
var I : Int .
var IVAR : IntVarId .
var GUARD : TransGuard .
vars L1 L2 : Location .
vars SL SL1 SL2 SL3 SL4 : StatementList .
var ELSIFS : ElseIfs .

--- Evaluate transition guards:
op evalGuard : TransGuard Configuration Valuation ~> Bool .

eq evalGuard(BE, PORTS, VAL) = evalBoolExpression(BE, VAL, PORTS) .
eq evalGuard(otherwise, PORTS, VAL) = true .     *** Be aware of this one!
eq evalGuard(on dispatch, PORTS, VAL) = true .

--- Evaluate boolean expressions:
op evalBoolExpression : BoolExpression Valuation Configuration ~> Bool .

eq evalBoolExpression(B, VAL, PORTS) = B .
eq evalBoolExpression(BVAR, (BVAR |-> B) VAL, PORTS) = B .
eq evalBoolExpression(fresh(P), VAL, < P : InDataThreadPort | fresh : B >  PORTS) = B .

eq evalBoolExpression(E1 = E2, VAL, PORTS) =
    (evalIntExpression(E1, VAL) == evalIntExpression(E2, VAL)) .
eq evalBoolExpression(E1 <= E2, VAL, PORTS) =
    (evalIntExpression(E1, VAL) <= evalIntExpression(E2, VAL)) .
eq evalBoolExpression(E1 < E2, VAL, PORTS) =
    (evalIntExpression(E1, VAL) < evalIntExpression(E2, VAL)) .
eq evalBoolExpression(E1 >= E2, VAL, PORTS) =
    (evalIntExpression(E1, VAL) >= evalIntExpression(E2, VAL)) .
eq evalBoolExpression(E1 > E2, VAL, PORTS) =
    (evalIntExpression(E1, VAL) > evalIntExpression(E2, VAL)) .
eq evalBoolExpression(E1 != E2, VAL, PORTS) =
    (evalIntExpression(E1, VAL) =/= evalIntExpression(E2, VAL)) .

eq evalBoolExpression(P = IE, VAL, PORTS) =
    (intFromPort(P, PORTS) == evalIntExpression(IE, VAL)) .
eq evalBoolExpression(IE = P, VAL, PORTS) = evalBoolExpression(P = IE, VAL, PORTS) .
eq evalBoolExpression(P >= IE, VAL, PORTS) =
    (intFromPort(P, PORTS) >= evalIntExpression(IE, VAL)) .
eq evalBoolExpression(IE >= P, VAL, PORTS) = evalBoolExpression(P <= IE, VAL, PORTS) .
eq evalBoolExpression(P > IE, VAL, PORTS) =
    (intFromPort(P, PORTS) > evalIntExpression(IE, VAL)) .
eq evalBoolExpression(IE > P, VAL, PORTS) = evalBoolExpression(P < IE, VAL, PORTS) .
eq evalBoolExpression(P <= IE, VAL, PORTS) =
    (intFromPort(P, PORTS) <= evalIntExpression(IE, VAL)) .
eq evalBoolExpression(IE <= P, VAL, PORTS) = evalBoolExpression(P >= IE, VAL, PORTS) .
eq evalBoolExpression(P < IE, VAL, PORTS) =
    (intFromPort(P, PORTS) <= evalIntExpression(IE, VAL)) .
eq evalBoolExpression(IE < P, VAL, PORTS) = evalBoolExpression(P > IE, VAL, PORTS) .
eq evalBoolExpression(P != IE, VAL, PORTS) =
    (intFromPort(P, PORTS) =/= evalIntExpression(IE, VAL)) .
eq evalBoolExpression(IE != P, VAL, PORTS) = evalBoolExpression(P != IE, VAL, PORTS) .

eq evalBoolExpression(P = B, VAL, PORTS) = boolFromPort(P, PORTS) == B .
eq evalBoolExpression(BVAR = B, (BVAR |-> B2) VAL, PORTS) = (B == B2) .

ceq evalBoolExpression(BEXP1 and BEXP2, VAL, PORTS) =
      (evalBoolExpression(BEXP1, VAL, PORTS) and evalBoolExpression(BEXP2, VAL, PORTS))
    if not (BEXP1 :: Bool and BEXP2 :: Bool) .
ceq evalBoolExpression(BEXP1 or BEXP2, VAL, PORTS) =
```

```
      (evalBoolExpression(BEXP1, VAL, PORTS) or evalBoolExpression(BEXP2, VAL, PORTS))
    if not (BEXP1 :: Bool and BEXP2 :: Bool) .
ceq evalBoolExpression(not BEXP1, VAL, PORTS) =
       not evalBoolExpression(BEXP1, VAL, PORTS)
    if not (BEXP1 :: Bool) .

op evalIntExpression : IntExpression Valuation ~> Int .

eq evalIntExpression(IVAR, (IVAR |-> I) VAL) = I .
eq evalIntExpression(I, VAL) = I .
eq evalIntExpression(E1 - E2, VAL) =
    (evalIntExpression(E1, VAL) - evalIntExpression(E2, VAL)) .
eq evalIntExpression(E1 + E2, VAL) =
    (evalIntExpression(E1, VAL) + evalIntExpression(E2, VAL)) .


*** Execute a transition.
op executeTransition : Transition Configuration Valuation  ~> PortsValuation .

eq executeTransition(L1 -[ GUARD ]-> L2 {SL}, PORTS, VAL) =
    if evalGuard(GUARD, PORTS, VAL) then
       executeStatements(SL, PORTS, VAL)
    else transResult(PORTS, VAL)
    fi .

op executeStatements : StatementList Configuration Valuation ~>
                                  PortsValuation [strat (2 3 1 0)] .  --- just to be sure

eq executeStatements(nil, PORTS, VAL) = transResult(PORTS, VAL) .

eq executeStatements((IVAR := IE) ; SL, PORTS, (IVAR |-> I) VAL) =
      executeStatements(SL, PORTS,
                        (IVAR |-> evalIntExpression(IE, (IVAR |-> I) VAL)) VAL) .

--- same thing for Boolean assignment:
eq executeStatements((BVAR := BE) ; SL, PORTS, (BVAR |-> B) VAL) =
      executeStatements(SL, PORTS,
                        (BVAR |-> evalBoolExpression(BE, (BVAR |-> B) VAL, PORTS)) VAL) .

eq executeStatements((P := IE) ; SL, PORTS, VAL) =
      executeStatements(SL, writeData(evalIntExpression(IE, VAL), P, PORTS), VAL) .

eq executeStatements((P := BE) ; SL, PORTS, VAL) =
      executeStatements(SL,
                        writeData(evalBoolExpression(BE, VAL, PORTS), P, PORTS),
                        VAL) .

--- A little trickier one:
ceq executeStatements((P1 := P2) ; SL, PORTS, VAL) =
      executeStatements(SL, writeData(B, P1, PORTS), VAL)
    if B := boolFromPort(P2, PORTS) .
ceq executeStatements((P1 := P2) ; SL, PORTS, VAL) =
      executeStatements(SL, writeData(I, P1, PORTS), VAL)
    if I := intFromPort(P2, PORTS) .

eq executeStatements((BVAR := P) ; SL, PORTS, (BVAR |-> B) VAL) =
      executeStatements(SL, PORTS,
                        (BVAR |-> boolFromPort(P, PORTS)) VAL) .
eq executeStatements((IVAR := P) ; SL, PORTS, (IVAR |-> I) VAL) =
      executeStatements(SL, PORTS,
                        (IVAR |-> intFromPort(P, PORTS)) VAL) .
```

```
  --- The various if-constructs:
  eq executeStatements((if ( BE ) SL1 end if) ; SL2, PORTS, VAL) =
     if evalBoolExpression(BE, VAL, PORTS) then
        executeStatements(SL1 ; SL2, PORTS, VAL)
     else executeStatements(SL2, PORTS, VAL) fi .

  eq executeStatements((if ( BE ) SL1 else SL2 end if) ; SL3, PORTS, VAL) =
     if evalBoolExpression(BE, VAL, PORTS) then
        executeStatements(SL1 ; SL3, PORTS, VAL)
     else executeStatements(SL2 ; SL3, PORTS, VAL) fi .

  --- the elsifs ... there might be a nicer way of doing this ...
  eq executeStatements((if ( BE ) SL1 (elsif ( BE2 ) SL2) end if) ; SL3, PORTS, VAL) =
     if evalBoolExpression(BE, VAL, PORTS) then
        executeStatements(SL1 ; SL3, PORTS, VAL)
     else executeStatements((if ( BE2 ) SL2 end if) ; SL3, PORTS, VAL) fi .

  eq executeStatements((if ( BE ) SL1 (elsif ( BE2 ) SL2) else SL3 end if) ; SL4, PORTS, VAL) =
     if evalBoolExpression(BE, VAL, PORTS) then
        executeStatements(SL1 ; SL4, PORTS, VAL)
     else executeStatements((if ( BE2 ) SL2 else SL3 end if) ; SL4, PORTS, VAL) fi .

  eq executeStatements((if ( BE ) SL1 ((elsif ( BE2 ) SL2) ELSIFS) end if) ; SL3, PORTS, VAL) =
     if evalBoolExpression(BE, VAL, PORTS) then
        executeStatements(SL1 ; SL3, PORTS, VAL)
     else executeStatements((if ( BE2 ) SL2 ELSIFS end if) ; SL3, PORTS, VAL) fi .

  eq executeStatements((if ( BE ) SL1 ((elsif ( BE2 ) SL2) ELSIFS) else SL3 end if) ; SL4,
                     PORTS, VAL) =
     if evalBoolExpression(BE, VAL, PORTS) then
        executeStatements(SL1 ; SL4, PORTS, VAL)
     else executeStatements((if ( BE2 ) SL2 ELSIFS else SL3 end if) ; SL4, PORTS, VAL) fi .

  --- while loop:
  eq executeStatements((while ( BE ) {SL1}) ; SL2, PORTS, VAL) =
     if evalBoolExpression(BE, VAL, PORTS) then
        executeStatements(SL1 ; (while ( BE ) {SL1}) ; SL2, PORTS, VAL)
     else executeStatements(SL2, PORTS, VAL) fi .
endom)


(omod TRANSFER-DATA is
 including BASIC-SORTS .      --- Definition of ConnectionSet
 including THREAD-PORTS .
 including COMPONENT .
 including THREAD .

 op transferData : Configuration -> Configuration [strat (1 0)] .

 vars C C1 C2 : ComponentId .
 vars P P1 P2 : PortId .
 var PN PN1 : PortName .
 vars PORTS PORTS2 OBJECTS : Configuration .
 vars MC MC' : MsgContent .
 var CONXS : ConnectionSet .
 vars REST COMPONENTS : Configuration .
 var I : Int . var B : Bool .

 op transfer : MsgContent -> MsgContent [ctor] . --- the flag that makes the equations move the data.
 eq transferData(COMPONENTS) = initializeTransfer(COMPONENTS) .

 op initializeTransfer : Configuration -> Configuration .
```

```
eq initializeTransfer(< C : Component | features : PORTS,
                                        subcomponents : COMPONENTS >
                      REST)
 = < C : Component | features : initializeTransfer(PORTS),
                     subcomponents : initializeTransfer(COMPONENTS) >
   initializeTransfer(REST) .

--- now for ports:
eq initializeTransfer(< P : InDataThreadPort | >  PORTS)
 = < P : InDataThreadPort | fresh : false >
   initializeTransfer(PORTS) .

eq initializeTransfer(< P : OutDataPort | content : data(I) >
                      PORTS)
 = < P : OutDataPort | content : transfer(data(I)) >
   initializeTransfer(PORTS) .

eq initializeTransfer(< P : OutDataPort | content : data(B) >
                      PORTS)
 = < P : OutDataPort | content : transfer(data(B)) >
   initializeTransfer(PORTS) .

eq initializeTransfer(REST) = REST [owise] .
--- covers both the empty configuration and non-thread ports and
--- thread ports that need not change due to lack of output to transfer.

*** Connections of the form inport --> subcomponent . inport.
eq < C : Component | features :
                      < P : InDataPort | content : transfer(MC) > PORTS,
                     subcomponents : COMPONENTS,
                     connections : (P --> C1 . P1) ; CONXS  >
 =
   < C : Component | features :
                       < P : InDataPort | content : noMsg >  PORTS,
                     subcomponents :
                        (transfer MC from P to COMPONENTS
                         using ((P --> C1 . P1) ; (P,CONXS))) > .

op transfer_from_to_using_ : MsgContent PortName Configuration  ConnectionSet -> Configuration [memo] .

--- Notice 'InPort' here. Never connection to subcomponent . outport!
eq transfer MC from PN to
       (< C : Component | features : < P : InDataPort | >   PORTS >
        COMPONENTS)
   using ((PN --> C . P) ; CONXS)
 =
  transfer MC from PN to (
    < C : Component | features :
                       < P : InDataPort | content : transfer(MC) > PORTS >
    COMPONENTS)
  using CONXS .


*** Connections of the form subcomponent . outport --> outport  and
---                   subcomponent1 . outport --> subcomponent2 . inport
eq < C : Component |
      features : PORTS,
      subcomponents :
        < C1 : Component |
            features :
              < P1 : OutDataPort | content : transfer(MC) > PORTS2 >
        COMPONENTS,
```

```
      connections : (C1 . P1 --> PN) ; CONXS  >
     =
    < C : Component | features : (transfer MC from (C1 . P1) to PORTS
                                  using ((C1 . P1 --> PN) ; ((C1 . P1),CONXS))),
                      subcomponents :
                        < C1 : Component |
                            features :
                              < P1 : OutDataPort | content : noMsg >
                              PORTS2 >
                        (transfer MC from (C1 . P1) to COMPONENTS
                         using ((C1 . P1 --> PN) ; ((C1 . P1),CONXS)))  > .

 eq transfer MC from PN to (< P : OutDataPort | > PORTS)
    using (PN --> P) ; CONXS
  = < P : OutDataPort | content : transfer(MC) >
    transfer MC from PN to PORTS using CONXS .

 *** Same thing, with delayed connections:
 eq < C : Component |
       features : PORTS,
       subcomponents :
         < C1 : Component |
             features :
               < P1 : OutDataPort | content : transfer(MC) > PORTS2 >
         COMPONENTS,
       connections : (C1 . P1 -->> PN) ; CONXS  >
     =
    < C : Component | features : (transfer MC from (C1 . P1) to PORTS
                                  using CONXS),
                      subcomponents :
                        < C1 : Component |
                            features :
                              < P1 : OutDataPort | content : noMsg >
                              PORTS2 >
                        (transfer MC from (C1 . P1) to COMPONENTS
                         using ((C1 . P1 -->> PN) ; ((C1 . P1), CONXS)))  > .

 eq transfer MC from PN to
        (< C : Component | features : < P : InDataPort | >   PORTS >
         COMPONENTS)
    using ((PN -->> C . P) ; CONXS)
  =
    transfer MC from PN to (
      < C : Component | features :
                        < P : InDataPort | content : transfer(MC) > PORTS >
      COMPONENTS)
    using CONXS .

 eq transfer MC from PN to OBJECTS using CONXS = OBJECTS [owise] .

 *** Finally, a message has arrived at its destination:
 eq < C : Thread | features :
                     < P : InDataThreadPort | content : transfer(MC) >
                     PORTS >
   =
    < C : Thread | features :
                     < P : InDataThreadPort | content : MC, fresh : true >
                     PORTS > .
endom)

(omod AADL-PROPERTIES is
  including TIME-DOMAIN .
```

```
  including COMPONENT .
  including BEHAVIOR-LANGUAGE .

*** Some SynchAADL properties:
  sort SynchAADLProperty .
  subsort SynchAADLProperty < Property .

  op Synchronous : Bool -> SynchAADLProperty [ctor] .
  op syncPeriod : Time -> SynchAADLProperty [ctor] .

  op Deterministic : Bool -> Property [ctor] .

  --- Environment properties; will be dealt with more thoroughly later:
  op IsEnvironment : Bool -> Property [ctor] .
  op InputConstraints : BoolExpression -> Property [ctor] .

*** Dispatch methods
  sort DispatchMethod .
  subsort DispatchMethod < Property .
  op periodic-dispatch : Time -> DispatchMethod [ctor] .

  --- Dispatch:
  sort ADispatchMethod .
  op Periodic : -> ADispatchMethod .
  op DispatchProtocol : ADispatchMethod -> Property .

  op Period : Time -> Property .
  eq (DispatchProtocol(Periodic) ; Period(T:Time)) = periodic-dispatch(T:Time) .
endom)

(tomod EXECUTE-THREADS is
  including THREAD .
  including EXEC-TRANS .
  protecting EXT-BOOL .
  including SYSTEM .
  including PROCESS .
  including AADL-PROPERTIES .

  var CR : ComponentRef .
  var LDS : LocationDeclSet .
  var O : Oid .
  vars PORTS NEW-PORTS CONF1 CONF2 COMPONENTS : Configuration .
  var P : PortId .
  vars L L1 L2 : Location .
  vars LS LS1 LS2 : LocationSet .
  var GUARD : TransGuard .
  vars VAL NEW-VALUATION : Valuation .
  var TRANSITIONS TRANSES : TransitionSet .
  var SL : StatementList .
  vars TP PROPS : Properties .

  op applyTransitions : Configuration ~> Configuration [strat (1 0)] .

  *** Distribute down to threads:
  eq applyTransitions(< O : System | subcomponents : COMPONENTS >)
   = < O : System | subcomponents : applyTransitions(COMPONENTS) > .

  eq applyTransitions(< O : Process | subcomponents : COMPONENTS >)
   = < O : Process | subcomponents : applyTransitions(COMPONENTS) > .

  ceq applyTransitions(CONF1 CONF2) = applyTransitions(CONF1) applyTransitions(CONF2)
       if CONF1 =/= none and CONF2 =/= none .
```

```
  *** Threads: environment thread is not executed by this one:
  eq applyTransitions(< O : Thread | properties : (IsEnvironment(true) ; PROPS) >)
   = < O : Thread | > .

  *** Now, we come to deterministic thread, which should be executed.
  ceq applyTransitions(
        < O : Thread | properties : Deterministic(true) ; PROPS,
                       features : PORTS,
       currState : L1,
       completeStates : LS,
       variables : VAL,
       behaviorRef : CR >)
   = if L2 in LS then   --- done!
        < O : Thread | features :  NEW-PORTS,
               currState : L2,
       variables : NEW-VALUATION >
     else        --- not a complete state: continue!
       applyTransitions(< O : Thread | features :  NEW-PORTS,
                                       currState : L2,
                                       variables : NEW-VALUATION >)
     fi
   if ((L1 -[ GUARD ]-> L2 {SL})  ; TRANSITIONS) := transitions(CR)
         /\ evalGuard(GUARD, PORTS, VAL)
         /\ GUARD =/= otherwise or-else
             not someTransEnabled(TRANSITIONS, L1, VAL, PORTS)
         /\ transResult(NEW-PORTS, NEW-VALUATION) :=
              executeTransition( L1 -[ GUARD ]-> L2 {SL}, PORTS, VAL) .

  op environmentThread : Properties -> Bool .
  eq environmentThread(IsEnvironment(true) ; PROPS) = true .
  eq environmentThread(PROPS) = false [owise] .

  op someTransEnabled : TransitionSet Location Valuation Configuration -> Bool .
  eq someTransEnabled((L1 -[ GUARD ]-> L2 {SL})  ; TRANSITIONS, L, VAL, PORTS)
   = if L == L1 and evalGuard(GUARD, PORTS, VAL) then true
     else someTransEnabled(TRANSITIONS, L, VAL, PORTS) fi .

 eq someTransEnabled(emptyTransitionSet, L, VAL, PORTS) = false .

  op _in_ : Location LocationSet -> Bool .
  eq L in L LS = true .
  eq L in LS = false [owise] .
endtom)

(omod APPLY-ENV-TRANS is
  including EXECUTE-THREADS .
  including GENERATE-ENV-INPUTS .

  var CR : ComponentRef .
  var LDS : LocationDeclSet .
  var O : Oid .
  vars PORTS NEW-PORTS CONF1 CONF2 COMPONENTS : Configuration .
  var P : PortId .
  vars L L1 L2 : Location .
  vars LS LS1 LS2 : LocationSet .
  var GUARD : TransGuard .
  vars VAL OLD-VAL NEW-VALUATION : Valuation .
  var TRANSITIONS TRANSES : TransitionSet .
  var SL : StatementList .
  vars TP PROPS : Properties .
  var BE : BoolExpression .  var BVAR : BoolVarId .  var B : Bool .
```

```
op applyEnvTransitions : Valuation Configuration ~> Configuration .

*** Distribute down to threads:
eq applyEnvTransitions(VAL, < O : System | subcomponents : COMPONENTS >)
 = < O : System | subcomponents : applyEnvTransitions(VAL, COMPONENTS) > .

eq applyEnvTransitions(VAL, < O : Process | subcomponents : COMPONENTS >)
 = < O : Process | subcomponents : applyEnvTransitions(VAL, COMPONENTS) > .

ceq applyEnvTransitions(VAL, CONF1 CONF2)
  = applyEnvTransitions(VAL, CONF1) applyEnvTransitions(VAL, CONF2)
      if CONF1 =/= none and CONF2 =/= none .

*** Threads: non-environment thread is not executed by this one:
ceq applyEnvTransitions(VAL, < O : Thread | properties : PROPS >) = < O : Thread | >
      if not environmentThread(PROPS) .

*** Now, we come to environment thread, which should be executed:
ceq applyEnvTransitions(VAL,
      < O : Thread | properties : IsEnvironment(true) ; PROPS,
                     features : PORTS,
     currState : L1,
     completeStates : LS,
     variables : OLD-VAL,
     behaviorRef : CR >)
 = if L2 in LS then   --- done!
      < O : Thread | features :  NEW-PORTS,
             currState : L2,
     variables : NEW-VALUATION >
   else        --- not a complete state: continue!
     applyEnvTransitions(NEW-VALUATION,
         < O : Thread | features :  NEW-PORTS,
                currState : L2,
variables : NEW-VALUATION >)
   fi
   if ((L1 -[ GUARD ]-> L2 {SL})  ; TRANSITIONS) := transitions(CR)
       /\ evalGuard(GUARD, PORTS, VAL)
       /\ GUARD =/= otherwise or-else
           not someTransEnabled(TRANSITIONS, L1, VAL, PORTS)
       /\ transResult(NEW-PORTS, NEW-VALUATION) :=
            executeTransition( L1 -[ GUARD ]-> L2 {SL}, PORTS, VAL) .

*** A main function, that gives all the possible variable assignments in a system:
op allEnvAssignments : Configuration ~> ValuationSet .  --- assumes ONE environment!

op containsEnvironment : Configuration -> Bool .
eq containsEnvironment(< O : System | subcomponents : COMPONENTS >)
   = containsEnvironment(COMPONENTS) .
eq containsEnvironment(< O : Process | subcomponents : COMPONENTS >)
   = containsEnvironment(COMPONENTS) .
ceq containsEnvironment(CONF1 CONF2) =
     containsEnvironment(CONF1) or-else containsEnvironment(CONF2)
      if CONF1 =/= none and CONF2 =/= none .
eq containsEnvironment(< O : Thread | properties : PROPS >) = environmentThread(PROPS) .
eq containsEnvironment(none) = false .

op getEnvironment : Configuration ~> Object .
  eq getEnvironment(< O : System | subcomponents : COMPONENTS >)
 = getEnvironment(COMPONENTS) .
eq getEnvironment(< O : Process | subcomponents : COMPONENTS >)
 = getEnvironment(COMPONENTS) .
```

```
  ceq getEnvironment(CONF1 CONF2) =
       if containsEnvironment(CONF1) then getEnvironment(CONF1)
       else getEnvironment(CONF2) fi
        if CONF1 =/= none and CONF2 =/= none .
  eq getEnvironment(< O : Thread | properties : IsEnvironment(true) ; PROPS >)
   =  < O : Thread | > .

  eq allEnvAssignments(< O : System | subcomponents : COMPONENTS >) =
        allEnvAssignments(getEnvironment(COMPONENTS)) .

  eq allEnvAssignments(< O : Thread | properties : IsEnvironment(true) ; InputConstraints(BE) ; TP,
                                      variables : VAL >)
   = allAssignments(initializeToFalse(VAL), BE) .

  op initializeToFalse : Valuation ~> Valuation .
  eq initializeToFalse((BVAR |-> B) VAL) = (BVAR |-> false) initializeToFalse(VAL) .
  eq initializeToFalse(VAL) = VAL [owise] .
endom)

(tomod SYNCHRONOUS-STEP is
  including APPLY-ENV-TRANS .
  including TRANSFER-DATA .

  vars SYSTEM OBJ : Object .
  var VAL : Valuation .   var VALS : ValuationSet .
  var NZT : NzTime .  var O : Oid .
  var PROPS : Properties .  var COMPONENTS CONF : Configuration .

  crl [syncStepWithTime] :
      {SYSTEM}               --- top-level object
     =>
      {applyTransitions(transferData(applyEnvTransitions(VAL, SYSTEM)))}
     in time period(SYSTEM)
     if containsEnvironment(SYSTEM)
        /\ VAL ;; VALS := allEnvAssignments(SYSTEM) .

  crl [syncStepWithTimeNoEnv] :
      {SYSTEM}
     =>
      {applyTransitions(transferData(SYSTEM))}
     in time period(SYSTEM)
     if not containsEnvironment(SYSTEM) .

  --- We assume that all perids are the same, but it period may be declared
  --- in different places, either explicitly at the top level, or impicitly
  --- by the period in some thread.
  op period : Configuration -> Time .
  eq period(< O : Component | properties : syncPeriod(NZT) ; PROPS >) = NZT .
  eq period(< O : Thread | properties : periodic-dispatch(NZT) ; PROPS >) = NZT .
  eq period(< O : Component | subcomponents : COMPONENTS >) =
       period(COMPONENTS) [owise] .
  ceq period(OBJ CONF) = if period(OBJ) == 0 then period(CONF) else period(OBJ) fi
         if CONF =/= none .
  eq period(none) = 0 .   --- default value
endtom)
```