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

Formalization and Correctness of the PALS Architectural Pattern for Distributed Real-Time Systems

José Meseguer[1] and Peter Csaba Ölveczky[2]

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

Abstract. Many Distributed Real-Time Systems (DRTS), such as integrated modular avionics systems and distributed control systems in motor vehicles, are made up of a collection of components communicating asynchronously among themselves and with their environment that must change their state and respond to environment inputs within hard real-time bounds. Such systems are often safety-critical and need to be certified; but their certification is currently very hard due to their distributed nature. The *Physically Asynchronous Logically Synchronous* (PALS) architectural pattern can greatly reduce the design and verification complexities of achieving virtual synchrony in a DTRS. This work presents a formal specification of PALS as a formal model transformation that maps a synchronous design, together with a set of performance bounds of the underlying infrastructure, to a formal DRTS specification that is semantically equivalent to the synchronous design. This semantic equivalence is proved, showing that the formal verification of temporal logic properties of the DRTS can be reduced to their verification on the much simpler synchronous design. An avionics system case study is used to illustrate the usefulness of PALS for formal verification purposes.

1 Introduction

Many Distributed Real-Time Systems (DRTS), such as integrated modular avionics systems and distributed control systems in motor vehicles, are made up of a collection of components communicating asynchronously among themselves and with their environment that must change their state and respond to environment inputs within hard real-time bounds. Because of physical and fault tolerance requirements, such systems are asynchronous, with each component having its own local clock. Yet, overall system behavior must ensure *virtual synchrony*, in the sense that each cycle of interaction of each system component with the environment and with the other components should result in a proper state change and proper outputs being produced at each component within hard real-time bounds. That is, the system, although asynchronous, must behave *as if it were synchronous*, not in some fictional logical time, but *in actual physical time*.

The design, verification, and implementation of such systems is a challenging and error-prone task for several reasons. The main danger is for a DRTS of

this nature to enter an *inconsistent state* due to race conditions, network delays, and clock skews in the asynchronous communication between components that can easily fool one component into mistakenly acting on inputs from the wrong cycle, or sending its outputs to other components at the wrong time; that is, the intrinsically asynchronous nature of the system makes it hard to ensure its virtual synchrony. Furthermore, since such a system is often safety-critical, its must undergo a stringent certification process that requires full coverage of the verification of its design and the validation of its implementation. Such a certification effort can be very demanding and time consuming because: (i) the state space explosion caused by the system's concurrency can easily make it unfeasible to apply automatic model checking techniques to verify that its design satisfies the required safety properties; and (ii) proper testing of an implementation is particularly challenging, due to the serious difficulty of testing many different interleaving that might reveal unforeseen errors and the insidious *no fault found* problem, where an error observed only once may be extremely hard to reproduce and therefore to diagnose by subsequent testing.

A useful way to meet engineering challenges such as the one described above is to amortize the use of formal methods not on an individual design, but on a *generic family of system designs* by means of a *formal architectural pattern*, that is, a generic formal specification of an engineering solution to a generic design problem that: (i) is shown to be correct by construction; (ii) comes with strong formal guarantees; and (iii) greatly reduces system complexity, making system verification and correct system implementation orders of magnitude simpler than if the pattern were not used. In this paper we present a formal specification and a proof of correctness for one such pattern, namely the *Physically Asynchronous Logically Synchronous*[3] (PALS) architectural pattern, which we have developed in collaboration with colleagues at Rockwell-Collins and UIUC (see [20, 2, 31]). This pattern provides a generic engineering solution to the problem of designing a DRTS that must be virtually synchronous in spite of its asynchronous nature.

1.1 The PALS Formal Model in a Nutshell

The key idea of PALS is to drastically reduce the effort of designing, verifying, and implementing a DRTS of this kind by *reducing its design and verification to that of its much simpler synchronous version*. This is achieved by assuming that the DRTS can rely on an underlying Asynchronous Bounded Delay (ABD) Network infrastructure, so that a bound can be given for the delay of any message transmission from any process to any other process.[4] Similary, it is assumed that the *clock skew* between the different local clocks of the DRTS is bounded. The

[3] An alternative acronym would be PAVS, for *Physically Asynchronous Virtually Synchronous* pattern, to emphasize that the pattern guarantees synchronous behavior not in some fictional logical time but in actual physical time; however we will stick with the PALS acronym as used in [20, 2, 31].

[4] See [6, 33] for the ABD theoretical model, and [28] for a detailed discussion of several commercial network architectures used in avionics and automotive systems that support the ABD model.

PALS pattern can then be formalized as a *model transformation* which sends a synchronous system design to its correct-by-construction asynchronous design. Specifically, PALS is a formal model transformation of the form:

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

where:

1. $\mathcal{E}$ is a synchronous system, which is formally defined as a *synchronous ensemble of state machines* connected together by a *wiring diagram*.
2. Γ specifies the following *performance bounds*: (i) the *clock skew* of the local asynchronous clocks for each state machine is strictly smaller than ϵ; (ii) the minimum and maximum duration times $0 \leq \alpha_{min} \leq \alpha_{max}$ for any machine to consume inputs, make a transition, and produce outputs; and (iii) the minimum and maximum *message transmission delays* $0 \leq \mu_{min} \leq \mu_{max}$ for communication between any to processes in the ABD network,

and where $\mathcal{A}(\mathcal{E}, \Gamma)$ then denotes the corresponding asynchronous design guaranteed to behave like $\mathcal{E}$ in a virtually synchronous way under the assumption that the performance bounds Γ are met by the underlying infrastructure. As we further discuss below, a key advantage of PALS for formal verification purposes is that the, typically unfeasible, verification of formal requirements for $\mathcal{A}(\mathcal{E}, \Gamma)$ can be reduced to the much simpler verification of such requirements for $\mathcal{E}$.

Main Contributions. This work complements other research on PALS such as [20, 2, 31] by providing both a formal specification of the PALS architecture and a detailed proof of its correctness that justifies why a formal verification of the synchronous design also verifies its PALS asynchronous version. Specifically, it presents the following contributions:

1. A formal model in rewriting logic [18] of the PALS transformation, expressed in the Real-Time Maude formal specification language [23], including precise requirements about the allowable synchronous designs to which PALS can be applied and about the real-time bounds assumed for the network and clock synchronization infrastructures.
2. A precise derivation of the PALS period based on the formal model, as well as a proof of its *optimality*, showing that it is shortest possible under the given assumptions about the asynchronous implementation, message format, and network and clock synchronization infrastructures.
3. A *bisimulation theorem*, showing that the original synchronous design and the so-called stable states of the corresponding PALS asynchronous design constitute bisimilar systems, and two further generalizations of such a theorem.
4. A mathematical justification of a method that *reduces* the formal verification of temporal logic safety and liveness properties of an asynchronous PALS design —typically unfeasible due to state space explosion— to the model checking verification of its much simpler synchronous counterpart.
5. An avionics case study illustrating the usefulness of the PALS patern for formal verification purposes.

The rest of this paper is organized as follows. In Section 2 we summarize the basic ideas about rewriting logic and Real-Time Maude needed to define our formal model of PALS. In Section 3 we give a formal definition of the synchronous models that are legal input designs for the PALS transformation, including precise formal definitions of typed machines, synchronous ensembles, and synchronous composition. In Section 4 we define in detail the assumptions about clock drift, network delays, and machine execution times that, together with the given synchronous ensemble, are the inputs to the PALS transformation. We then give a precise time-line analysis of the period that must be chosen, based on these parameters, for the PALS asynchronous system to achieve logical synchrony. Based on this analysis, we then give in Section 5 a formal specification in Real-Time Maude (parametric on the input ensemble $\mathcal{E}$ and the performance bounds Γ) of the resulting PALS-transformed asynchronous system, and collect in Section 6 some facts about the consequences of the extra generality gained by allowing local clock functions that are only piecewise continuous instead of just continuous. In Section 7 we state and prove the main bisimulation result, connecting the states of the synchronous system with the so-called stable states of its PALS-transformed asynchronous counterpart; some key lemmas for this theorem are collected in Appendix A. We also give in Section 7 two theorems that make explicit the temporal logic properties that would have to be verified in the asynchronous model $\mathcal{A}(\mathcal{E}, \Gamma)$ but are reduced to the much simpler verification of corresponding properties in the much simpler $\mathcal{E}$. A detailed proof of the optimality, under appropriate assumptions, of the PALS period used to achieve logical synchrony is given in Section 8. Related work is discussed in Section 10, and some final conclusions are drawn in Section 11.

Acknowledgments. This work is part of a broader collaboration with Steve Miller and Darren Cofer at Rockwell-Collins and with Lui Sha, Abdullah Al-Nayeem, and Mu Sun at UIUC on the PALS architecture. The PALS ideas have been developed in close interaction with all these people, who have provided very useful comments on earlier versions of this work. We thank particularly Lui Sha and Mu Sun for their very careful and insightful comments on an earlier version of this paper that has led to substantial improvements. We also thank Camilo Rocha for his kind help with some of the figures. We gratefully acknowledge funding for this research from the Rockwell-Collins corporation. Partial support has also been provided by the National Science Foundation under Grants IIS 07-20482 and CNS 08-34709, by the Boeing Company under Grant C8088-557395, and by the Research Council of Norway.

2 Real-Time Maude

A Real-Time Maude [23] *timed module* specifies a *real-time rewrite theory* of the form (Σ, E, IR, TR), where:

- (Σ, E) is a *membership equational logic* [8] theory with Σ a signature[5] and E a set of *confluent and terminating conditional equations*. (Σ, E) specifies the

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

system's states as an algebraic data type, and must contain a specification of a sort `Time` modeling the (discrete or dense) time domain.

- *IR* is a set of (possibly conditional) *labeled instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) local transitions, written `rl [`l`] :` t `=>` t', where l is a *label*. Such a rule specifies a *one-step transition* from an instance of term t to the corresponding instance of term t'. The rules are applied *modulo* the equations E.[6]
- *TR* is a set of *tick (rewrite) rules*, written with syntax
  `rl [`l`] : {`t`} => {`t'`} in time` τ `.`
  that model time elapse. `{_}` is a built-in constructor of sort `GlobalSystem`, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

The initial state must be a ground term of sort `GlobalSystem` and must be reducible to a term of the form `{`t`}` using the equations in the specification.

The Real-Time Maude syntax is fairly intuitive. For example, a function symbol f in Σ is declared with the syntax `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ldots s_n$ are the sorts of its arguments, and s is its (value) *sort*. Equations are written with syntax `eq` t `=` t', and `ceq` t `=` t' `if` *cond* for conditional equations. The mathematical variables in such statements are declared with the keywords `var` and `vars`. We refer to [8] for more details on the syntax of Real-Time Maude.

In object-oriented Real-Time Maude modules, 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 in a given state is represented as a term `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>` of sort `Object`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In a concurrent object-oriented system, the state is a term of sort `Configuration`, and has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Real-Time Maude.

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

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

defines a parametrized family of transitions (one for each substitution instance) in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions have the effect of altering the attribute `a1` of the object `O` and of sending a new message `m'(O')` *with delay* `x` (see [23]). "Irrelevant" attributes (such as `a3`, and the *right-hand side occurrence* of `a2`) need not be mentioned in a rule (or equation).

A *subclass* inherits all the attributes and rules of its superclasses.

[6] 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.

Formal Analysis. A Real-Time Maude specification is *executable*, and the tool offers a variety of formal analysis methods. The *rewrite* command simulates *one* fair behavior of the system *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, 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 also extends Maude's *linear temporal logic model checker* to check whether each behavior, possibly up to a certain time bound, satisfies a temporal logic formula. *State propositions*, possibly parametrized, can be predicates characterizing properties of the state and/or properties of the global time of the system. They are operators of sort `Prop`, and their semantics is defined by (possibly conditional) equations of the form `{`*statePattern*`} |=` *prop* `=` b, for b a term of sort `Bool`, which defines the state proposition *prop* to hold in all states `{`t`}` where `{`t`} |=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), and `U` ("until"). The time-bounded model checking command has syntax

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

for initial state t and temporal logic formula *formula* .

3 Formal Definition of the Synchronous Model

This section formally defines the synchronous model of computation as a collection of *nondeterministic typed machines* and an *environment*, and a set of connections that connect output ports of the machines and the environment to input ports.

3.1 Typed Machines

A *typed machine* is a component in the synchronous model.

Definition 1. *A* (nondeterministic) typed machine *is a 4-tuple*

$$M = (D_i, S, D_o, \delta_M)$$

where

- D_i*, called the* input set*, is a nonempty set of the form* $D_i = D_{i_1} \times \cdots \times D_{i_n}$*, for* $n \geq 1$*, where* $D_{i_1}, \ldots, D_{i_n}$ *are called the* input data types*.*
- S *is a nonempty set, called the* set of states*.*

– D_o, *called the* output set, *is a nonempty set of the form* $D_o = D_{o_1} \times \cdots \times D_{o_m}$, *for* $m \geq 1$, *where* $D_{o_1}, \ldots, D_{o_m}$ *are called the* output data types.
– δ_M, *called the* input-output-transition (i-o-t) relation, *is a total relation*

$$\delta_M \subseteq (D_i \times S) \times (S \times D_o).$$

That is, for any input $\boldsymbol{d_i}$ *and state* s, *there exist at least one state* s' *and output* $\boldsymbol{d_o}$ *such that* $((\boldsymbol{d_i}, s), (s', \boldsymbol{d_o})) \in \delta_M$.

We call M finite *iff* D_i, S, *and* D_o *are all finite sets, and call* M deterministic *if the transition relation* δ_M *is a function.*

That is, such a machine has n input ports and m output ports; an input to port k should be an element of the set D_{i_k}, and an output from port j should be an element of the set D_{o_j}. Pictorially, we represent a typed machine as a box with typed input and output wires as shown in Fig. 1.

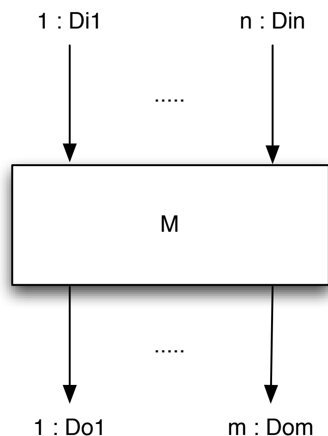

Fig. 1. Graphical representation of a machine.

3.2 Synchronous Ensembles of Typed Machines

Typed machines can be "wired together" into arbitrary sequential and parallel compositions by means of a "wiring diagram," as the one shown in Fig. 2, where the types are left implicit, but where it is assumed that the type in an output wire must match any types in the input wires connected with it:

Definition 2. *A (typed)* machine ensemble *is a 4-tuple*

$$\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j \in J}, E, src)$$

where

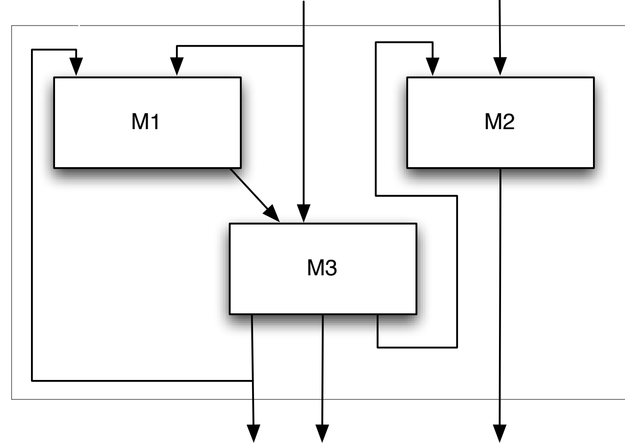

Fig. 2. A machine ensemble.

– *J is a nonempty finite set, called the set of* indices, *and e is an element, called the* environment index, *with $e \notin J$.*
– *$\{M_j\}_{j \in J}$ is a J-indexed family of typed machines.*
– *E, called a* typed environment, *is an ordered pair of sets*

$$E = (D_i^e, D_o^e)$$

where D_i^e, called the environment's input set *(*inputs *to the environment), is a nonempty set of the form*

$$D_i^e = D_{i_1}^e \times \cdots \times D_{i_{n_e}}^e, \text{ for } n_e \geq 1$$

and D_o^e, called the environment's output set *(*outputs *from the environment), is a nonempty set of the form*

$$D_o^e = D_{o_1}^e \times \cdots \times D_{o_{m_e}}^e, \text{ for } m_e \geq 1$$

– *src is a function that assigns to each input port (j, n) (the input port number n of machine j) the corresponding output port (or "source" for that input) $src(j, n)$. Formally, we define the set of input ports and output ports, respectively, as follows:*
  - $In_{\mathcal{E}} = \{(j, n) \in (J \cup \{e\}) \times \mathbb{N} \mid 1 \leq n \leq n_j\}$
  - $Out_{\mathcal{E}} = \{(j, n) \in (J \cup \{e\}) \times \mathbb{N} \mid 1 \leq n \leq m_j\}$

  Then src is a surjective function

$$src : In_{\mathcal{E}} \to Out_{\mathcal{E}}$$

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 kth input port of machine M_j (resp. kth input port of the environment if $j = e$), and same with output ports, then if $src(j, q) = (k, l)$ we should have $D^k_{o_l} \subseteq D^j_{i_q}$.
In addition, we require that there are no self-loops from the environment to itself; that is, for $(e, q) \in In_{\mathcal{E}}$, if $src(e, q) = (k, p)$, then $k \in J$.

As its name suggests, a synchronous ensemble $\mathcal{E}$ has a *lock-step synchronous semantics*, in the sense that the state-and-output transitions of all the 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* instant. As explained below, what this means mathematically is that any ensemble $\mathcal{E}$ is semantically equivalent to a *single state machine*, called the *synchronous composition* of all the machines in the ensemble $\mathcal{E}$. This has enormous practical importance for formal verification purposes, since the composed state machine is much simpler than an asynchronous system realizing such a design in a distributed way. In particular, model checking a single state machine is much more efficient and feasible than verifying a system of asynchronously interacting machines, which can easily become unfeasible due to the combinatorial explosion caused by the system's asynchronous concurrency.

Example 1. In the machine ensemble in Fig. 2,

- $J = \{1, 2, 3\}$
- $\{M_j\}_J$ is the mapping $1 \mapsto M_1,\ 2 \mapsto M_2,\ 3 \mapsto M_3$.
- $n_e = 3$ (the number of inputs to the environment) and $m_e = 2$ (number of outputs from the environment).
- With an ordering of ports from left to right, the wiring function *src* is:
  - $(1, 1) \mapsto (3, 1)$
  - $(1, 2) \mapsto (e, 1)$
  - $(2, 1) \mapsto (3, 3)$
  - $(2, 2) \mapsto (e, 2)$
  - $(3, 1) \mapsto (1, 1)$
  - $(3, 2) \mapsto (e, 1)$
  - $(e, 1) \mapsto (3, 1)$
  - $(e, 2) \mapsto (3, 2)$
  - $(e, 3) \mapsto (2, 1)$

Intuitively, we can enclose the typed machines M_1, M_2, and M_3 in the box with thin lines, hiding the internal details of how the machine ensemble is decomposed. The single machine resulting in this way from the composition of machines M_1, M_2, and M_3 is called the *synchronous composition* of M_1, M_2, and M_3, according to the given wiring diagram, and can itself be seen as a typed machine. The general definition is as follows.

Definition 3. *Given a synchronous machine ensemble* $\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j \in J}, E, src)$, *its* synchronous composition *is the typed machine*

$$M_{\mathcal{E}} = (D^{\mathcal{E}}_i, S^{\mathcal{E}}, D^{\mathcal{E}}_o, \delta_{\mathcal{E}})$$

where

- $D_i^{\mathcal{E}} = D_o^e$ *(the input set of the composed machine is the output set of the environment)*
- $D_o^{\mathcal{E}} = D_i^e$ *(the output set is the input set of the environment)*
- $S^{\mathcal{E}} = (\Pi_{j \in J} S_j) \times (\Pi_{j \in J} D_{OF}^j)$, *where if* $D_o^j = D_{o_1}^j \times \cdots \times D_{o_{m_j}}^j$ *is the output set of* M_j, *then* D_{OF}^j *is the set* $D_{OF}^j = D_{OF_1}^j \times \cdots \times D_{OF_{m_j}}^j$, *where, for* $1 \leq m \leq m_j$, $D_{OF_m}^j = D_{o_m}^j$ *if* $(j, m) = src(l, q)$ *for some* $l \in J$, *and* $D_{OF_m}^j = 1$ *otherwise, with* $1 = \{*\}$ *a one point set. Intuitively,* D_{OF}^j *stores the "feedback outputs" of machine* M_j. *We then have an obvious* "feedback output" *function*
$$fout_j : D_o^j \to D_{OF}^j$$
where for $1 \leq m \leq m_j$, *we have* $\pi_m(fout_j(d_1, \ldots, d_{m_j})) = d_m$ *if* $(j, m) = src(l, q)$ *for some* $l \in J$, *and* $\pi_m(fout_j(d_1, \ldots, d_{m_j})) = *$ *otherwise, with* π_m *the* m*-th projection from the Cartesian product* D_{OF}^j. *Similarly, for each* $k \in J$ *we have an obvious* input *function*
$$in_k : D_o^e \times \Pi_{j \in J} D_{OF}^j \to D_i^k$$
where for $1 \leq n \leq n_k$, *with* $src(k, n) = (l, q)$, *we have* $\pi_n(in_k(\boldsymbol{d}, \{\boldsymbol{d_j}\}_{j \in J})) =$ *if* $l = e$ *then* $\pi_q(\boldsymbol{d})$ *else* $\pi_q(\boldsymbol{d_l})$ *fi, where* π_q *denotes the* q*-th projection from the corresponding Cartesian product.*
- *The i-o-t relation for* $M_{\mathcal{E}}$ *is the relation*
$$\delta_{\mathcal{E}} \subseteq (D_i^{\mathcal{E}} \times S^{\mathcal{E}}) \times (S^{\mathcal{E}} \times D_o^{\mathcal{E}})$$
where for each $(\boldsymbol{d}, (\{s_j\}_{j \in J}, \{\boldsymbol{d_j}\}_{j \in J})) \in D_i^{\mathcal{E}} \times S^{\mathcal{E}}$, *we define*
$$((\boldsymbol{d}, (\{s_j\}_{j \in J}, \{\boldsymbol{d_j}\}_{j \in J})), ((\{s'_j\}_{j \in J}, \{\boldsymbol{d'_j}\}), \boldsymbol{d'})) \in \delta_{\mathcal{E}}$$
iff, for each $l \in J$, *there exists* $(s'_l, \boldsymbol{d''_l})$ *such that* $((in_l(\boldsymbol{d}, \{\boldsymbol{d_j}\}_{j \in J}), s_l), (s'_l, \boldsymbol{d''_l})) \in \delta_{M_l}$, *and where* $\boldsymbol{d'_l} = fout_l(\boldsymbol{d''_l})$ *and the output to the environment* $\boldsymbol{d'}$ *is defined for each* $1 \leq n \leq n_e$ *with* $src(e, n) = (j', r)$ *by* $\pi_n(\boldsymbol{d'}) = \pi_r(\boldsymbol{d''_{j'}})$. *Note that* $\delta_{\mathcal{E}}$ *is a total relation, since each* δ_{M_l} *is a total relation; therefore, some desired* $(s'_l, \boldsymbol{d''_l})$ *always exists. Furthermore, if each machine* M_i *is a deterministic typed machine, then* $M_{\mathcal{E}}$ *is also a deterministic typed machine.*

Note that the above notion of synchronous composition of a machine ensemble supports a *hierarchical design methodology*, in which entire sub-ensembles can be "closed off" and regarded from the outside as a single machine, which can then be synchronously composed at a higher level with other such machines, which may themselves also be synchronous compositions of other sub-ensembles.

3.3 Environment Constraints

In our model of the behaviors of a system, we assume a nondeterministic environment where there could be some constraints on the values generated by this

environment. In this work, we assume that the environment constraint can be defined as a predicate

$$c_e : D_o^e \rightarrow Bool$$

so that $c_e(d_1^e, \ldots, d_{o_{m_e}}^e)$ is *true* if and only if the environment can generate output $(d_1^e, \ldots, d_{o_{m_e}}^e)$. We also assume that the constraint c_e is satisfiable.

If desired, the environment could be regarded as another nondeterministic machine with a single state, $*$, or even with a set of states S^e, and would then have the form $E = (D_i^e, S^e, D_o^e, \delta_E)$. The particular case of a constraint c_e then corresponds to the case where $S^e = \{*\}$ and $((\boldsymbol{d_i}, *), (*, \boldsymbol{d_o})) \in \delta_E$ iff $c_e(\boldsymbol{d_o}) = true$. By viewing E as another machine, a synchronous composition would then have no explicit external environment and would then yield a "closed" system. We prefer to make the environment explicit in our model, since this is useful both for design purposes and for hierarchical composition.

3.4 The Transition System and the Kripke Structure Associated to a Machine Ensemble

To each machine ensemble that operates in an environment with a given environment constraint, we can associate a transition system defining the behaviors of the system.

Definition 4. *Given a machine ensemble $\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j \in J}, E, src)$ 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:

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

The set $Paths(\mathcal{E}_{c_e})_{(\boldsymbol{s},\boldsymbol{i})}$ is the set of all infinite sequences $(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{\mathcal{E}_{c_e}} (\boldsymbol{s'}, \boldsymbol{i'}) \longrightarrow_{\mathcal{E}_{c_e}} (\boldsymbol{s''}, \boldsymbol{i''}) \longrightarrow_{\mathcal{E}_{c_e}} \cdots$ of transition steps starting in state $(\boldsymbol{s}, \boldsymbol{i})$.

Let $\mathcal{E}$ be a typed machine ensemble with environment constraint c_e, AP a set of *atomic propositions*, and $L : S^{\mathcal{E}} \times D_i^{\mathcal{E}} \rightarrow \mathcal{P}(AP)$ a *labeling function* that assigns to each state $(s, i) \in S^{\mathcal{E}} \times D_i^{\mathcal{E}}$ the set $L(s, i)$ of atomic propositions that hold in (s, i). Then $(\mathcal{E}_{c_e}, L) = (S^{\mathcal{E}} \times D_i^{\mathcal{E}}, \longrightarrow_{\mathcal{E}_{c_e}}, L)$ is the *Kripke structure* associated to $(\mathcal{E}, c_e, L)$.

4 Overview of the PALS Asynchronous Model

This section gives a high-level overview of the asynchronous PALS transformation of a synchronous machine ensemble. Section 4.1 makes explicit some assumptions about clock drift and computation and communication times, and defines some constant values. Section 4.2 gives a high-level overview of the asynchronous system, and Section 4.3 focuses on the time line.

4.1 Some System Assumptions

Time Domain. The type of time (discrete or dense) is a parameter of the model. It could be $\mathbb{N}$, or $\mathbb{Q}_{\geq 0}$, or $\mathbb{R}_{\geq 0}$, for example. (If it is $\mathbb{N}$, we can scale up things so that one "logical time step" can correspond to many basic steps.) For simplicity and fullest generality, in what follows we will assume that all is done in $\mathbb{R}_{\geq 0}$.

Clock Drift and Clock Synchronization. A basic assumption is that a clock synchronization algorithm is executing "in the background" and guarantees a certain bound on the imprecision of the local clocks. We assume that this is achieved by an *external clock synchronization* mechanism; that is, the difference between the time of a local clock and "real" global time is assumed to be always *strictly less* than a given bound ϵ.

To reason about clock drift in a general way, we assume a local clock function c_j for each machine M_j that assigns to each global instant r the local clock value $c_j(r)$:

Definition 5. *A function $c : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ is called an* ϵ-drift local clock function *if and only if the following conditions are satisfied:*

1. *c is monotonic w.r.t. the $\leq$ relation and piecewise continuous,*
2. *$\forall x \in \mathbb{R}_{\geq 0},\ |c(x) - x| < \epsilon$, and*
3. *$\forall x \in \mathbb{R}_{\geq 0},\ \inf\{t \mid c(t) \geq x\} \in \{\ t \mid c(t) \geq x\}$.*

The assumption of monotonicity is very good to have, particularly to have a clear idea of when and where to tick the global clock. Synchronization can be achieved while preserving monotonicity. The idea is the same as when one has an expensive clock that typically does not allow to be wound *backwards*. How do you adjust such a clock to the precise time? Well, if the clock is too fast, then you can just "stop" your clock and wait until "real" time has caught up with your clock time. If the clock is slow, then you quickly adjust it forward to the "real" time. This is an isolated discontinuity which happens only from time to time. Furthermore, the third condition above ensures that the time at which a discontinuity of a local clock function c occurs is well-defined. Alternatively, one can achieve the effect of the ϵ-drift local clock function c to always be *continuous* by increasing the "ticking rate" of the local clock by a small factor whenever the clock is detected to be slow, and likewise decreasing its "ticking rate" by a small factor whenever it is detected to be fast. Continuity is preferable for applications where control of physical parameters is involved; but our results, although having a slightly simpler formulation for the continuous case, do not require continuity, but only piecewise continuity of the local clocks satisfying (1)–(3) above.

Execution Times. The shortest, respectively longest, time required for *processing input, executing a transition, and generating output* is supposed to be, respectively, α_{min} and α_{max}. Therefore, if a given execution of a machine M_j takes time α_j, then $\alpha_{min} \leq \alpha_j \leq \alpha_{max}$.

Network Delays. The point-to-point message transmission time is assumed to always be greater than or equal to a minimum value $\mu_{min} \geq 0$, and smaller than or equal to some maximum time value $\mu_{max} \geq \mu_{min}$.

The constants $\Gamma = (\epsilon, \alpha_{min}, \alpha_{max}, \mu_{min}, \mu_{max})$ make up the *performance parameters* of the PALS transformation.

4.2 The Asynchronous System with Clock Drifts: Overview

Given an ensemble $\mathcal{E}$ and performance parameters Γ, the asynchronous system $\mathcal{A}(\mathcal{E}, \Gamma)$ is made up of a J-indexed family of objects and an environment, with each object behaving like an "asynchronous typed machine" whose inputs and outputs are received and sent by *asynchronous message passing.* The system is supposed to execute in rounds according to "ticks" of a "logical clock." Let T denote the time between such "ticks of the logical clock" (or the *period* of the logical clock). We often write t_i for the time of the ith tick of the logical clock; i.e., $t_i = i \cdot T$.

Each object j is equipped with two timers:

`roundTimer` is a timer that should expire at the tick of each logical clock; that is, at the end of each period.

`outputBackoffTimer` is a backoff timer used to ensure that output from a machine is not sent into the network too early.

The actions of each object j can be summarized as follows:

- When `roundTimer` expires, input from the input buffer is read, a transition is executed, and the generated output is put in the output buffer. In addition, this timer is reset to the value T (denoting the period of the "logical clock").
- When the `outputBackoffTimer` timer expires, the messages in the output buffer are sent, *provided that they have been generated.* If the execution of the transition generating the outgoing messages is not yet finished when the timer expires, then the messages are sent when the execution of the transition is finished. This timer is started and set to $dly_{out} = 2 \cdot \epsilon \texttt{ monus } \mu_{min}$, where `monus` is defined by $x \texttt{ monus } y = \max(0, x - y)$, each time the `roundTimer` expires.

4.3 Time-line Analysis

The "time-line analysis" for object j in $\mathcal{A}(\mathcal{E}, \Gamma)$ is therefore as follows:

1. At each *local* logical clock tick (that is, when the local clock c_j shows t_i), the object gets the messages from the input buffer, executes a transition, and puts the output messages in the output buffer. This starts somewhere in the global time interval $(t_i - \epsilon, t_i + \epsilon]$ for round i. This process may end at any global time in the interval $(t_i - \epsilon + \alpha_{min}, t_i + \epsilon + \alpha_{max}]$.

2. Since the messages cannot be sent into the network before the backoff timer expires, and before the messages are "ready," the messages from the output buffers are therefore sent into the network at a global time that is strictly greater than $\max(t_i + \epsilon - \mu_{min},\ (t_i - \epsilon) + \alpha_{min})$ and is less than or equal to $\max(t_i + 3 \cdot \epsilon - \mu_{min},\ (t_i + \epsilon) + \alpha_{max})$.
3. At any global time in the time interval $(\max(t_i + \epsilon,\ (t_i - \epsilon) + \alpha_{min} + \mu_{min}),\ \max(t_i + 3 \cdot \epsilon - \mu_{min},\ (t_i + \epsilon) + \alpha_{max}) + \mu_{max}]$, a message could arrive at an object, at which time it is entered into the object's input buffer.
4. When the local clock shows t_{i+1}, the object starts all over from the first point above.

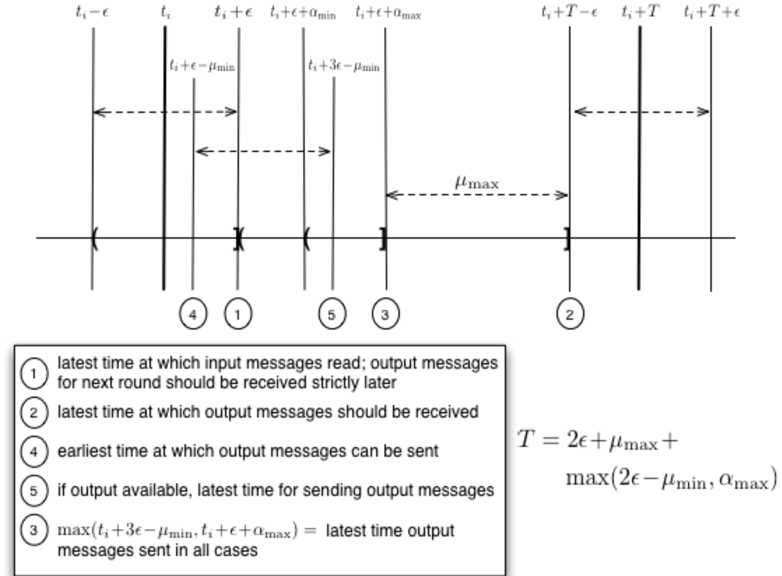

Fig. 3. PALS timeline.

An overview of this timeline is depicted in Fig. 3. It is worth remarking that some of the time intervals are right-closed. For example, the "local clock tick" in item (1) above could happen at any time in the global time interval $(t_i - \epsilon, t_i + \epsilon]$ instead of the right-open interval $(t_i - \epsilon, t_i + \epsilon)$ that might seem more intuitive,

given that the clock synchronization should ensure a difference that is *strictly smaller* than ϵ between the local clock time and global time. However, the "local tick" could happen at time $t_i + \epsilon$ as well: At all global times $t_i + \epsilon - \Delta$ for small $\Delta > 0$, the local clock shows $t_i - \Delta/2$, and at global time $t_i + \epsilon$ the local clock jumps to, say $t_i + \epsilon$. This scenario satisfies the assumptions on the clock synchronization, yet the "local logical tick" happens at time $t_i + \epsilon$. Section 6 gives further explanations on this issue.

Constraints. For this to work, we must ensure that messages generated for round $i+1$ should be received sometime in the global time interval $(t_i+\epsilon, t_{i+1}-\epsilon]$, which is $(t_i+\epsilon, t_i+T-\epsilon]$. Therefore, the message arrival interval $(\max(t_i+\epsilon,\ (t_i-\epsilon) + \alpha_{min} + \mu_{min}),\ \max(t_i + 3 \cdot \epsilon - \mu_{min},\ (t_i + \epsilon) + \alpha_{max}) + \mu_{max}]$ must be a subset of $(t_i + \epsilon, t_i + T - \epsilon]$. This implies that we must have

1. $t_i + \epsilon \leq (\max(t_i + \epsilon,\ (t_i - \epsilon) + \alpha_{min} + \mu_{min})$, and
2. $\max(t_i + 3 \cdot \epsilon - \mu_{min},\ (t_i + \epsilon) + \alpha_{max}) + \mu_{max} \leq t_i + T - \epsilon$.

(1) holds trivially. (2) implies that

$$T \geq \mu_{max} + 2 \cdot \epsilon + \max(2 \cdot \epsilon - \mu_{min},\ \alpha_{max}).$$

Note finally that, although this paper presents the "optimal" PALS transformation, all correctness results hold as long as the backoff timer is always initialized to a value $b \geq 2 \cdot \epsilon$ `monus` μ_{min} and the PALS period $T \geq \mu_{max} + 2 \cdot \epsilon + \max(b,\ \alpha_{max})$, which both hold in the avionics case study in Section 9.

5 PALS Formal Model in Real-Time Maude

This section presents the formal specification of the asynchronous PALS system $\mathcal{A}(\mathcal{E}, \Gamma)$ associated to a synchronous machine ensemble $\mathcal{E}$ with environment constraint c_e, under the assumptions in Section 4.1, as a rewrite theory in Real-Time Maude. In particular, Sections 5.1 to 5.5 formally specify $\mathcal{A}(\mathcal{E}, \Gamma)$ as a rewrite theory in Real-Time Maude, and Section 5.6 defines the initial states.

5.1 Some Sorts

We start by discussing some sorts used in this specification.

Local States. The state component of machine j has sort S_j. For convenience, we add a supersort `State` of all such states:

```
sort State .     --- supersort of local states
subsorts S_1 ... S_|J| < State .
```

It may take some time to compute the next local state of a machine. During this transition computation time, the local state has the value [s, t], where s is the next state, and t is the time remaining until the execution of the transition is finished. Such a term [s, t] is called a *delayed state*, where the sort `DlyState` is defined as follows:

```
sort DlyState .
subsort State < DlyState .

op [_,_] : State Time ->  DlyState [ctor right id: 0] .
```

Note that the fact that `0` has been defined as a *right identity* mens that [s,0]$= s$.

Likewise, during the execution of a transition generating the messages for the next round, these messages are not yet ready to be sent, and hence the output buffer has the value [$msgs$, t], which we call a *delayed configuration*:

```
sort DlyConfiguration .
subsort Configuration < DlyConfiguration .

op [_,_] : Configuration Time ->  DlyConfiguration [ctor] .
```

The sort `Configuration` denotes multisets of objects and messages, formed with an "empty syntax" multiset union operator "`__`" (juxtaposition) which is declared to be *associative* (`assoc`) and *commutative* (`comm`) and have identity `none` (`id: none`).

We also introduce a supersort `Data` of the sorts $D_1 \dots D_n$ of the data in the wires:

```
sort Data .
subsorts D1 ... Dn < Data .
```

Each object is also assumed to know its local wiring diagram; that is, which objects and ports are connected to its output ports in the synchronous system. This knowledge is stored in a data structure called a *local wiring*, where the sort `LocalWiring` is defined as follows:

```
sort LocalWiring .

op _-->_._ : Nat Oid Nat -> LocalWiring [ctor] .
op noWiring : -> LocalWiring [ctor] .
op _;_ : LocalWiring LocalWiring -> LocalWiring
                                      [ctor assoc comm id: noWiring] .
```

Here, a connection p `-->` j `.` p' says that the output port p of the current object is connected to the input port p' of object j. A *local wiring* is then a set of such single connections formed with the associative-commutative union operator `_;_` with identity the empty set constant `noWiring`.

5.2 The Class Declarations

Each machine M_j is translated into an object instance of a subclass $C_{[j]}$ of the class `Machine` declared as follows:

```
class Machine | state : DlyState,
                inBuffer : MsgConfiguration,
                outBuffer : DlyConfiguration,
                roundTimer : Time,
                outputBackoffTimer : TimeInf,
                clock : Time,
                localWiring : LocalWiring .

class C_1 .
...
class C_k .

subclass C_1 ... C_k < Machine .
```

Note that several typed machines, say, $M_{j_1}, \ldots, M_{j_r}$, can all be of the same type, and can therefore all belong to the same subclass, i.e., $C_{[j_1]} = \cdots = C_{[j_r]}$. The `state` attribute denotes the local state of the machine. The `inBuffer` attribute is the buffer of incoming messages. `outBuffer` is the output message buffer. The timers `roundTimer` and `outputBackoffTimer` have been explained in Section 4.2. The `clock` attribute shows the value of the local clock of the object. Finally, the `localWiring` attribute assigns to each output port number the set of input ports to which this port is connected. However, notice that here a connection is only a *reference* for asynchronous message passing, and not a real "wired" connection as in the synchronous model.

We assume that the environment also has input and output buffers, and that it satisfies the same timing requirement as all the other objects. The environment is therefore modeled as an object instance of a class `Env` that is declared as a subclass of `Machine`:

```
class Env .

subclass Env < Machine .
```

Since we do not explicitly represent the internal "state" of the environment, the `state` attribute for the environment is given the constant default value `*`:

```
op * : -> State [ctor] .
```

Note that treating the environment as another "wrapped" machine is crucial for PALS, since in a real application the actual timing of inputs from the environment may be quite unpredictable. Therefore, it is crucial for the environment inputs to be buffered and synchronized by the object wrapping it in the exact same way as all other machines are thus synchronized by their wrapper objects.

5.3 Messages

Messages have the general form

to j from j' (p, d)

where $j, j' \in J \cup \{e\}, 1 \leq p \leq n_j$, and $d \in D^j_{i_p}$. Therefore, p is the input port of the intended recipient j, where data d from j' is to be received.

We also use the `dly` operator on messages to model the delay of such messages when they are in transit through the network, as explained in [23].

5.4 The Instantaneous Rewrite Rules

The following actions of the system are modeled by corresponding *instantaneous* rewrite rules:

1. Receive an incoming message and put it into the `inBuffer`.
2. When the `roundTimer` expires, the `inBuffer` is emptied, a transition is applied, and the output is put into the `outBuffer`. Note that, since performing a transition takes time, as already mentioned this is manifested by the fact that the results of the tansition are "delayed".
3. When the `outputBackoffTimer` expires, if the generated output is ready to be sent, then the contents in the output buffer are sent into the network, with appropriate message delays.
4. Otherwise, as soon as the generated output is ready to be sent *after* the `outputBackoffTimer` has expired (i.e., the `outputBackoffTimer` has the infinity value `INF`, as in the rule `outputMsg2` below), then the generated output is sent into the network.

Receive a Message. A message is received by an object and is inserted into its `inBuffer`:

```
vars j j' : Oid .
var B : MsgConfiguration .   --- multiset of messages
var p : Nat .
var d : Data .

rl [receiveMsg] :
   (to j from j' (p, d))
   < j : Machine | inBuffer : B >
  =>
   < j : Machine | inBuffer : B (to j from j' (p, d)) > .
```

Note that the rule can also be applied when the machine is in a "delayed" state; that is, the wrapper can buffer messages while the internal machine is executing a transition.

Reading Input and Executing a Transition. When `roundTimer` expires, the messages `B` in the `inBuffer` are read, and a transition is taken. Since different classes will have different transitions, executing transitions is modeled by a *family* of rewrite rules, one for each class $C_{[j]}$. Notice that the resulting state and messages are delayed by a value $\alpha_{min} \leq$ `X-DLY` $\leq \alpha_{max}$. In addition, the `roundTimer` must be reset to expire at the same time in the next round; i.e., it must be reset to the round time T, adjusted for possible clock jumps as explained in Section 6. Likewise, the `outputBackoffTimer` must be set to $2 \cdot \epsilon - \mu_{min}$:

```
vars X-DLY LT : Time .
vars S NEXT-STATE : State .
var W : LocalWiring .
var d_{j_1} : D^j_{o_1} .
...
var d_{j_{m_j}} : D^j_{o_{m_j}} .

crl [applyTrans] :
    < j : C_[j] | inBuffer : B, roundTimer : 0, state : S,
                  localWiring : W, clock : LT >
   =>
    < j : C_[j] | inBuffer : none,
                  state : [NEXT-STATE, X-DLY],
                  roundTimer : T - adjust(LT, T, 0),
                  outputBackoffTimer :
                    (2·ε monus μ_min) monus adjust(LT,T,0),
                  outBuffer : [makeMsg(j, W, (d_{j_1}, ..., d_{j_{m_j}})), X-DLY] >
   if X-DLY >= α_min and X-DLY <= α_max
      /\ ((vect_[j](B),S), (NEXT-STATE,(d_{j_1}, ..., d_{j_{m_j}}))) ∈ δ_{M_j} .
```

Here, given a complete set `B` of messages of the form

$$(\texttt{to}\ j\ \texttt{from}\ j'_1(1, d_1))\quad \dots\quad (\texttt{to}\ j\ \texttt{from}\ j'_{n_j}(n_j, d_{n_j}))\quad (\dagger)$$

the function $vect_{[j]}$(`B`) maps `B` to the vector of inputs $(d_1, \dots, d_{n_j})$. `makeMsg` is the obvious (but a bit tedious to spell out in detail) function that looks at the local wiring diagram `W`, takes the vector of output data from j, and produces the set of messages for the machines and environment getting inputs from that wire. For example, for the system in Fig. 2, `makeMsg`$(3, src, (d1, d2, d3))$ produces the message configuration

```
(to 1 from 3 (1, d1))
(to e from 3 (1, d1))
(to e from 3 (2, d2))
(to 2 from 3 (3, d3))
```

Expiration of the `outputBackoffTimer`. When the `outputBackoffTimer` expires, and the messages are already generated (that is, the output buffer matches

MSG MSGS), the messages in the output buffer are sent into the network one by one, each message with its own nondeterministic delay (NTW-DLY). The timer is turned off (i.e., set to the infinity value INF) after the last message has been sent:

```
var MSGS : Configuration .
var MSG : Msg .
var NTW-DLY : Time .

crl [outputMsg1] :
    < j : Machine | outBuffer : MSG MSGS, outputBackoffTimer : 0 >
   =>
    < j : Machine | outBuffer : MSGS,
                    outputBackoffTimer :
                      if MSGS == none then INF else 0 fi >
    dly(MSG, NTW-DLY)
   if μmin <= NTW-DLY and NTW-DLY <= μmax .
```

If the execution of the transition and the generation of the outgoing messages is not finished when the outputBackoffTimer expires (that is, the output buffer has the form $[msgs, t]$ for $t > 0$), the timer is just turned off:

```
var NZT : NzTime .

rl [turnOffOutTimer] :
   < j : Machine | outBuffer : [MSGS, NZT], outputBackoffTimer : 0 >
  =>
   < j : Machine | outputBackoffTimer : INF > .
```

End of a Transition with Possible Immediate Output. When the execution of a transition and the generation of outgoing messages is finished, the "delay" of the generated messages in the output buffer is 0. If, in addition, the outputBackoffTimer has expired (is INF), then the messages in the output buffer are immediately sent into the network one by one, each message with its own delay (rule outPutMsg2); otherwise the outgoing messages are kept in the output buffer, but the delay wrappers on the generated messages disappear (rule transitionFinished):

```
crl [outputMsg2] :
    < j : Machine | outBuffer : [MSG MSGS, 0],
                    outputBackoffTimer : INF >
   =>
    < j : Machine | outBuffer :
                      if MSGS == none then none else [MSGS, 0] fi >
    dly(MSG, NTW-DLY)
   if μmin <= NTW-DLY and NTW-DLY <= μmax .
```

```
var TI : TimeInf .

rl [transitionFinished] :
   < j : Machine | outBuffer : [MSGS, 0], outputBackoffTimer : T >
  =>
   < j : Machine | outBuffer : MSGS > .
```

Environment Behavior. Since the environment class `Env` is a subclass of `Machine`, the environment inherits the rules for receiving and sending messages. The "machine" rules for reading the input buffer and executing a transition are replaced by one rule that consumes the messages in the input buffer, and generates the output nondeterministically, but ensuring that the environment constraint c_e is satisfied:

```
var D1 : D^e_{o_1} .
...
var DME : D^e_{o_{m_e}} .

crl [consumeInputAndGenerateOutput] :
    < e : Env | inBuffer : B, roundTimer : 0, clock : LT, wiring : W >
   =>
    < e : Env | inBuffer : none,
                roundTimer : T - adjust(LT, T, 0),
                outputBackoffTimer :
                   (2·ϵ monus μ_min)  monus  adjust(LT,T,0),
                outBuffer : [makeMsg(e,W,(D1, ..., DME)), X-DLY] >
  if c_e(D1, ..., DME)  /\  X-DLY >= α_min and X-DLY <= α_max .
```

5.5 Time Behavior

This section describes the time behavior of the asynchronous system.

State and Tick Rule. The global state of the system has the form $\{C; t\}$, where C is the configuration consisting of the objects and messages in the asynchronous system, and t is the global time.

The tick rule, advancing the global time in the system, is the following slight modification of the "usual" tick rule for object-oriented systems [23]:

```
var C : Configuration .
vars T T' : Time .

crl [tick] :
    {C ; T} => {delta(C, T, T') ; T + T'}  in time T'
       if T' <= mte(C, T) .
```

Here, delta is the function that defines how the *passage of time affects the state*, and mte is the function that defines the *smallest time until a timer becomes zero.* These functions are declared in the expected way:

```
op delta : Configuration Time Time -> Configuration [frozen (1)] .
op mte : Configuration Time -> TimeInf [frozen (1)] .
```

These functions distribute over the objects and messages in the state in the expected way, and must be defined for individual objects and messages.

We first define delta: how does time elapse affect the timers? If from time t, time advances by Δ, then the local clock has advanced from $c_j(t)$ to $c_j(t+\Delta)$, that is, by $c_j(t+\Delta) - c_j(t)$, where $c_j(t)$ is assumed to be the value T4 given in the clock attribute:

```
vars t Δ T1 T2 T3 T4 T5 : Time .

eq delta(< j : Machine | roundTimer : T1, outputBackoffTimer : TI,
                          state : [S, T3], clock : T4,
                          outBuffer : [MSGS, T3] >, t, Δ) =
   < j : Machine | roundTimer : T1 monus (cj(t + Δ) - T4),
                   outputBackoffTimer : TI monus (cj(t + Δ) - T4),
                   state : [S, T3 monus Δ],
                   clock : cj(t + Δ),
                   outBuffer : [MSGS, T3 monus Δ] > .

eq delta(< j : Machine | roundTimer : T1, outputBackoffTimer : TI,
                          clock : T2,
                          outBuffer : MSGS >, t, Δ) =
   < j : Machine | roundTimer : T1 monus (cj(t + Δ) - T2),
                   outputBackoffTimer : TI monus (cj(t + Δ) - T2),
                   clock : cj(t + Δ) > .
```

As for mte, it will be the smallest of the mte's of objects and messages in the configuration, where the mte of an object is just defined as the smallest time until one of the two timers become zero, or until the delay on the outgoing messages in the output buffer reaches 0. This is not a constructive definition, but this just reflects the fact that we do not model the underlying clock synchronization "constructively." The assumption of monotonicity of the local clock functions is crucial to make this definition of mte well defined.

Finally, defining delta and mte on messages is trivial:

```
eq delta(dly(MSG, T1), T2, T) = dly(MSG, T1 monus T) .
eq mte(dly(MSG, T1), T2) = T1 .
```

5.6 Initial States

We define the initial states of the system to start at time t_0, defined by $t_0 = T - \epsilon$. We do not start at time 0 since:

- local clocks could be less than the global clock;
- transitions are only taken (and hence output produced) when input is available.

At global times $(i \cdot T) + t_0$, for all $i \in \mathbb{N}$, the state components are undelayed and consistent, and all the input buffers are full.

What are the "initial" values of the object attributes at time t_0?

- The `clock` attribute of each object is $c_j(T - \epsilon)$, and we have $0 \leq T - 2\epsilon < c_j(T - \epsilon) < T$.
- The `outputBackoffTimer` should be turned off at this time, as all outgoing messages have been sent.
- The `roundTimer`, which is supposed to expire at each (local) time $i \cdot T$ should be initialized to $T - c_j(t_0)$, where it follows from the above clock values in the initial state that $0 < T - c_j(t_0) < 2\epsilon$.
- The `state` attribute should be an initial state of the expected sort.
- The `inBuffer`s are full of messages.
- The `outBuffer`s should be empty.
- There are no messages in transit in the network.
- The local wiring is constant and maps the ports to the input wires as illustrated below.
- The initial input $(d^e_{o_1}, \ldots, d^e_{o_{m_e}})$ from the environment should satisfy the environment constraint c_e.

In addition, the input buffers should be consistent w.r.t. the *src* function. That is, if $src(j, l) = src(j', l') = (j'', l'')$, then the data value d in the message `(to` j `from` j'' `(`l`,`d`))` in the `inBuffer` of object j must equal the data value d' in the message `(to` j' `from` j'' `(`l'`,`d'`))` in the `inBuffer` of object j'.

For example, an initial state corresponding to the synchronous machine in Fig. 2 could be the following:

```
{< 1 : C_1 | clock : c_1(t_0),  roundTimer : T - c_1(t_0),
             outputBackoffTimer : INF,
             inBuffer : (to 1 from 3 (1, d^3_{o_1}))
                        (to 1 from e (2, d^e_{o_1})),
             outBuffer : none,  state : s_1,
             localWiring : 1 --> 3.1 >
 < 2 : C_2 | clock : c_2(t_0),  roundTimer : T - c_2(t_0),
             outputBackoffTimer : INF,
             inBuffer : (to 2 from 3 (1, d^3_{o3}))
                        (to 2 from e (2, d^e_{o_2})),
             outBuffer : none,  state : s_2,
             localWiring : 1 --> e.3 >
 < 3 : C_3 | clock : c_3(t_0),  roundTimer : T - c_3(t_0),
             outputBackoffTimer : INF,
             inBuffer : (to 3 from 1 (1, d^1_{o_1}))
                        (to 3 from e (2, d^e_{o_1})),
```

```
           outBuffer : none, state : s3,
           localWiring : 1 --> 1.1 ; 1 --> e.1 ;
                         2 --> e.2 ; 3 --> 2.1 >
 < e : Env | clock : ce(t0),  roundTimer : T - ce(t0),
            outputBackoffTimer : INF,
            inBuffer : (to e from 3 (1, d^3_o1))
                       (to e from 3 (2, d^3_o2))
                       (to e from 2 (3, d^2_o1)),
            outBuffer : none,
            localWiring : 1 --> 1.2 ; 1 --> 3.2 ;  2 --> 2.2 >
    ;
 t0 }
```

for values $s_1, d^1_{o_1}, \ldots$ of appropriate sorts.

6 Consequences of Clock Synchronization

The following, perhaps slightly surprising, facts are two consequences of having local clock functions that are only piecewise continuous. They do not apply when the clock functions are continuous.

Fact 1. *Although the clock synchronization ensures that the difference between any local clock and the global time is always* strictly less *than ϵ, it may happen that a timer that is set to expire at (local) time t expires* exactly *ϵ time "units" later; that is, the timer could expire at global time $t + \epsilon$.*

Proof. We can have a clock $c(t)$ obeying $|c(t) - t| < \epsilon$ and such that it has a discontinuity at global time 11 but gets arbitrarily close to the value 10 for $t < 11$, i.e., $\lim_{t<11,\, t\to 11} c(t) = 10$. Instead at time 11, the clock jumps to a value $10 < c(11) < 12$.

The above fact has a practical consequence in that timer-driven events that are supposed to happen at ("local") time T may happen at any global time in the interval $(T - \epsilon, T + \epsilon]$ (instead of in the right-open interval $(T - \epsilon, T + \epsilon)$).

Note that if all clocks are *continuous*, this behavior is impossible so that a timer set to expire at local time t must indeed expire within the global time interval $(t - \epsilon, t + \epsilon)$.

Fact 2. *In the presence of only piecewise continuous clocks with maximum drift strictly less than ϵ, a periodic timer of period p which is reset when the local timer expires can drift from the ideal time strictly beyond ϵ.*

Proof. It is sufficient to show a concrete example. Let $\epsilon = 1$, $p = 10$, and $c(0) = 0$. As shown by Fact 1, we can have $\lim_{t<11,\, t\to 11} c(t) = 10$ but $c(11) = 11.9$. Then the timer is reset to 10, and we can have $c(21.9) = 22.8$. Therefore, the second time the timer is set, namely at time 21.9, the time has drifted 1.9 time units from the ideal timer. Note that, by repeating this type of behavior, it is possible for the timer to drift an arbitrary distance from the ideal timer.

This second fact implies that we must set the timers not to the round time T, but must adjust the new timer value to account for the possible "clock jump". Fortunately, in our asynchronous model, this "offset" can be computed without knowing anything except the local clock value, the length of the period, and the "starting time" of the first period:

Fact 3. *It is possible to keep a timer with a local clock $c(t)$ set to expire at times $k \cdot p + \tau$ (with $2\epsilon \leq p$) with a drift less than or equal to ϵ from the ideal timer by resetting it each time it expires, say at time $c(t)$, to the new value $p - \texttt{adjust}(c(t), p, \tau)$, where $\texttt{adjust}(c(t), p, \tau) = c(t) - ((\lfloor (c(t) - \tau)/p \rfloor * p) + \tau)$.*

Therefore, given the current local time t, the round time T, and the start time of the timer in the first round (τ), the value which should be subtracted from the new timer value is $\texttt{adjust}(t, T, \tau)$.

This need to explicitly incorporate into the model the adjustment for clock resets when the timers expire is not due to peculiarities with our way of modeling clock synchronization, but must be done for the system whenever we have a clock synchronization algorithm that can adjust the clocks in "jumps". Of course, if the clock functions are not only monotonic but also *continuous* (and not just piecewise continuous), this adjustment is not needed, because no such "jumps" can occur.

7 Correctness of the PALS Transformation

Given a typed machine ensemble $\mathcal{E}$ with associated environment constraint c_e, and values $\alpha_{min}, \alpha_{max}, \mu_{min}, \mu_{max}, T, \epsilon$, and a vector $\boldsymbol{c}$ of ϵ-drift clock functions, we have defined in Section 5 its asynchronous PALS transformation $\mathcal{A}(\mathcal{E}_{c_e}, \alpha_{min}, \alpha_{max}, \mu_{min}, \mu_{max}, T, \epsilon, \boldsymbol{c})$ as an object-oriented Real-Time Maude model. This section establishes a precise relationship between the synchronous composition $\mathcal{M}_{\mathcal{E}}$ of the ensemble $\mathcal{E}$ defined in Section 3 and the asynchronous model $\mathcal{A}(\mathcal{E}_{c_e}, \alpha_{min}, \alpha_{max}, \mu_{min}, \mu_{max}, T, \epsilon, \boldsymbol{c})$.

Notation. When the various parameters of the PALS transformation are implicit, we sometimes write $\mathcal{A}(\mathcal{E})$ for $\mathcal{A}(\mathcal{E}_{c_e}, \alpha_{min}, \alpha_{max}, \mu_{min}, \mu_{max}, T, \epsilon, \boldsymbol{c})$.

Each synchronous transition step in the synchronous composition $M_{\mathcal{E}}$ corresponds to multiple rewrite steps in $\mathcal{A}(\mathcal{E})$. The key idea is to define "bigger" transition steps that consist of multiple rewrite steps in $\mathcal{A}(\mathcal{E})$, so that each of these bigger transitions corresponds to a single transition step in $M_{\mathcal{E}}$.

Definition 6. *A state $\texttt{\{}C\,\texttt{;}\,t\texttt{\}}$ in $\mathcal{A}(\mathcal{E})$ is called* stable *iff*

- *all input buffers in C are full,*
- *all output buffers are empty (`none`), and*
- *there are no messages "in transit" in C.*

All other states in $\mathcal{A}(\mathcal{E})$ are called unstable.

Intuitively, a stable state corresponds to a state $(\boldsymbol{s}, \boldsymbol{i})$ in $M_{\mathcal{E}}$, and a sequence of rewrite steps between two stable states, reachable from some initial state, corresponds to a transition in $M_{\mathcal{E}}$. However, due to time ticks, there could be a rewrite sequence from one stable state to another (very similar) stable state that does not correspond to a transition in $M_{\mathcal{E}}$.

7.1 The Transition System of Stable Configurations

Definition 7. *A* behavior *in $\mathcal{A}(\mathcal{E})$ is a sequence*

$$\{C_0 \,;\, t_0\} \longrightarrow \{C_1 \,;\, t_1\} \longrightarrow \{C_2 \,;\, t_2\} \longrightarrow \cdots$$

of one-step rewrites in $\mathcal{A}(\mathcal{E})$, where $\{C_0 \,;\, t_0\}$ is an initial state of the form described in Section 5.6.

Definition 8. *For $\{C_i \,;\, t_i\}$ a state in $\mathcal{A}(\mathcal{E})$ reachable from some initial state, a* path *in $\mathcal{A}(\mathcal{E})$ is an infinite or nonextensible finite sequence*

$$\{C_i \,;\, t_i\} \longrightarrow \{C_{i+1} \,;\, t_{i+1}\} \longrightarrow \{C_{i+2} \,;\, t_{i+2}\} \longrightarrow \cdots$$

of one-step rewrites in $\mathcal{A}(\mathcal{E})$. Furthermore, the above path is called time-diverging *if and only if for each time value $t \in \mathbb{R}_{\geq 0}$, there exists a $q \in \mathbb{N}$ such that $t_q \geq t$. We denote by $Paths(\mathcal{A}(\mathcal{E}))_{\{C;t\}}$ the set of paths in $\mathcal{A}(\mathcal{E})$ starting in $\{C \,;\, t\}$, and denote by $TDPaths(\mathcal{A}(\mathcal{E}))_{\{C;t\}}$ the set of time-diverging paths.*

The following theorem, whose proof is given in Appendix A, shows that it is impossible to reach a deadlock state in $\mathcal{A}(\mathcal{E})$, and that Zeno behaviors are not forced by the specification of $\mathcal{A}(\mathcal{E})$ due to some design error, but are instead always avoidable.

Theorem 1. *Let $\{C_0; t_0\}$ be an initial stable state in $\mathcal{A}(\mathcal{E})$. Then, any finite rewrite sequence*

$$\{C_0; t_0\} \longrightarrow \{C_1; t_1\} \longrightarrow \{C_2; t_2\} \longrightarrow \cdots \longrightarrow \{C_n; t_n\}$$

in $\mathcal{A}(\mathcal{E})$ can be extended into a time-diverging path in $TDPaths(\mathcal{A}(\mathcal{E}))_{\{C_0;t_0\}}$.

Definition 9. *Let $Stable(\mathcal{A}(\mathcal{E}))$ denote both the set of states and the transition system whose states are the stable states of $\mathcal{A}(\mathcal{E})$, and where "big step"* stable *transitions, denoted*

$$\{C \,;\, t\} \longrightarrow_{st} \{C' \,;\, t'\}$$

are defined as follows: $\{C; t\} \longrightarrow_{st} \{C'; t'\}$ iff there exists a behavior of $\mathcal{A}(\mathcal{E})$ of the form

$$\{C_0 \,;\, t_0\} \longrightarrow \{C_1 \,;\, t_1\} \longrightarrow \cdots \longrightarrow \{C_n \,;\, t_n\} \longrightarrow \{C_{n+1} \,;\, t_{n+1}\} \longrightarrow \cdots$$

and numbers k, k' with $k < k'$ such that

- *$C = C_k$ and $C' = C_{k'}$ are stable configurations, $t = t_k$ and $t' = t_{k'}$, and*

– $\{C_k\ ;\ t_k\} \longrightarrow \{C_{k+1}\ ;\ t_{k+1}\} \longrightarrow \cdots \longrightarrow \{C_{k'}\ ;\ t_{k'}\}$ *is a subsequence of rewrites in such a behavior such that*
  - *the sequence contains at least one application of an instantaneous rewrite rule, and*
  - *if* C_j *is* not *a stable state, for* $k < j < k'$*, then there is no* $j < l < k'$ *such that* C_l *is a stable state.*

  These two conditions imply that for some $k < j < k'$*, for all* $k \leq i < j$*, all states* $\{C_i; t_i\}$ *are stable; and for all* $j \leq l < k'$ *all states* $\{C_l; t_l\}$ *are unstable.*

The following theorem, proved in Appendix A, shows that stable states and stable transitions provide a high-level "big step" view of any time-diverging behaviors in $\mathcal{A}(\mathcal{E})$.

Theorem 2. *Let* $\mathcal{E}$ *be a synchronous machine ensemble, and let* $\{C_i; t_i\}$ *be a* stable *state reachable from an initial state according to the definition of initial states in Section 5.6. Then, any time-diverging path*

$$\pi : \quad \{C_i; t_i\} \longrightarrow \{C_{i+1}; t_{i+1}\} \longrightarrow \{C_{i+2}; t_{i+2}\} \longrightarrow \cdots$$

in $\mathit{TDPaths}(\mathcal{A}(\mathcal{E}))_{\{C_i;t_i\}}$ *can be composed into an infinite sequence*

$$\{C_i; t_i\} \longrightarrow_{st} \{C_{i+k_1}; t_{i+k_1}\} \longrightarrow_{st} \{C_{i+k_2}; t_{i+k_2}\} \longrightarrow_{st} \cdots$$

of stable transitions.

That is, there is a strictly monotonic function $\gamma_\pi : \mathbb{N} \to \mathbb{N}$ *with* $\gamma_\pi(0) = 0$ *such that for each* $j \geq 0$*, the rewrite sequence* $\pi(\gamma_\pi(j)) \longrightarrow \pi(\gamma_\pi(j) + 1) \longrightarrow \cdots \longrightarrow \pi(\gamma_\pi(j + 1))$ *corresponds to a stable transition* $\pi(\gamma_\pi(j)) \longrightarrow_{st} \pi(\gamma_\pi(j + 1))$.

7.2 Relating the Synchronous and the Asynchronous Models

In this section we prove that $\mathit{Stable}(\mathcal{A}(\mathcal{E}))$ and $\mathcal{E}_{c_e}$ are bisimilar by first relating stable states to states in the synchronous ensemble composition, and then by showing that each synchronous transition has a corresponding stable transition, and vice versa. This bisimilarity is then lifted to the level of Kripke structures to show that $\mathit{Stable}(\mathcal{A}(\mathcal{E}))$ and $\mathcal{E}_{c_e}$ satisfy the same temporal logic properties. We further expand this important result to prove similar equivalences for satisfaction of temporal logic properties in two increasingly more general systems: (i) one where "stuttering" tick transitions between stable states are also allowed; and (ii) the entire asynchronous system $\mathcal{A}(\mathcal{E})$.

The relation between a stable state and the corresponding state in the synchronous composition is fairly obvious:

Definition 10. *Let*

$$\mathit{sync} : \mathit{Stable}(\mathcal{A}(\mathcal{E})) \rightarrow S^{\mathcal{E}} \times D_i^{\mathcal{E}}$$

be a function that maps each stable state of the asynchronous model to the corresponding state of the synchronous system as follows:

– *The local state of each object j, given in the object's* `state` *attribute, determines the local state in M_j. This is well defined, since in a* stable *state, the* `state` *attribute does not have a "delayed" value.*
– *The messages in the input buffers determine the state of the environment input and feedback wires using the functions $fout_j$, $j \in J$ defined in Definition 3 of Section 3.*

More precisely, $sync(\{C; t\})$ is defined to be the tuple

$$(((s_1, \ldots, s_j), ((d^1_{o_1}, \ldots, d^1_{o_{m_1}}), \ldots, (d^j_{o_1}, \ldots, d^j_{o_{m_j}}))),\ (d^e_{o_1}, \ldots, d^e_{o_{m_e}}))$$

where:

– *the* `state` *attribute of the object i has the value s_i in C,*
– *d^i_k (for $i \neq e$) equals $*$ iff for all "connections" of the form k* `-->` *$o.p$ (for the given k) in the* `localWiring` *attribute of object i in C, the object o is the identifier of the environment object of class* `Env`*, and*
– *otherwise, d^i_k, for $i \in J \cup \{e\}$, equals the value d in a message*

    `to` l `from` i `(`p`,` d`)`

  in the `inBuffer` *of object l in C for some l, p for which the* `localWiring` *attribute of object i contains a connection k* `-->`*l $.p$.*

The requirement that the messages in the `inBuffer`*s in the initial states are "wiring consistent" (see Section 5.6) ensures that sync is well-defined for all reachable stable states.*

Example 2. Let t_0 be the initial state given in Section 5.6. Then $sync(t_0)$ is the machine ensemble in Fig. 2, where the internal state of machine M_i is s_i, where the value in the feedback wire from M_1 to M_3 is $d^1_{o_1}$, and so on. That is, $sync(t_o)$ is the tuple

$$(((s_1, s_2, s_3), (d^1_{o_1}, *, (d^3_{o_1}, *, d^3_{o_3}))),\ (d^e_{o_1}, d^e_{o_2})).$$

Theorem 3. *The function sync defines a bisimulation between the transition systems $Stable(\mathcal{A}(\mathcal{E}))$ and $\mathcal{E}_{c_e} = (S^{\mathcal{E}} \times D^{\mathcal{E}}_i, \longrightarrow_{\mathcal{E}})$.*

Proof. We need to prove:

1. If $(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{\mathcal{E}} (\boldsymbol{s'}, \boldsymbol{i'})$, then, for each stable state c such that $sync(c) = (\boldsymbol{s}, \boldsymbol{i})$, there must exist a transition $c \longrightarrow_{st} c'$ of stable states c, c' such that $sync(c') = (\boldsymbol{s'}, \boldsymbol{i'})$.
2. If $c \longrightarrow_{st} c'$, then it must be the case that $sync(c) \longrightarrow_{\mathcal{E}} sync(c')$.

These two properties are proved in Appendix A as, respectively, Lemmas 9 and 8. □

The above bisimulation has a very important consequence at the level of temporal logic properties. Suppose a transition system $(A, \rightarrow)$, and a set AP of atomic propositions that are meant to describe basic properties of some states

in A. We can describe when $p \in AP$ holds in state $a \in A$, written $a \models p$, by the defining equivalence $a \models p \Leftrightarrow p \in L(a)$, where $L : A \longrightarrow \mathcal{P}(AP)$ is a *labeling function* specifying which atomic propositions hold in which states. The triple $(A, \rightarrow, L)$ is called a *Kripke structure*. Given a Kripke structure and a state a chosen as the desired initial state, and given a temporal logic formula φ with atomic propositions in AP, the semantics of temporal logic then defines the satisfaction relation $(A, \rightarrow, L), a \models \varphi$, settling whether φ holds in $(A, \rightarrow, L)$ from the initial state a. We refer the reader to [7] for a detailed description of the syntax and semantics of the temporal logic CTL^* which includes the logics CTL and LTL as special cases. In essence, a CTL^* formula is built up from atomic propositions in AP by the Boolean connectives $\vee$ and $\neg$, the "next" and "until" temporal operators[7] $\bigcirc$ and $\mathcal{U}$, and the universal path quantifier $\forall$. If the number of states reachable from an initial state a is finite, then the satisfaction relation $(A, \rightarrow, L), a \models \varphi$ can be computed by a CTL^* model checking algorithm (see again [7]). In our case, the satisfaction of atomic propositions AP in an ensemble $\mathcal{E}$ can be specified by a labeling function $L : S^{\mathcal{E}} \times D_i^{\mathcal{E}} \longrightarrow \mathcal{P}(AP)$, giving rise to a Kripke structure $(\mathcal{E}_{c_e}, L)$ as already explained in Section 3.4. Likewise, we can define "the same" atomic predicates on the corresponding stable states of $Stable(\mathcal{A}(\mathcal{E}))$ just by using as our labeling the composed function $sync; L : Stable(\mathcal{A}(\mathcal{E})) \longrightarrow \mathcal{P}(AP)$. In this way, we obtain the two closely-related Kripke structures $(\mathcal{E}_{c_e}, L)$ and $(Stable(\mathcal{A}(\mathcal{E})), sync; L)$. The importance of Theorem 3 for formal verification purposes is that it has as an immediate corollary the following result, reducing the verification of $CTL^*(AP)$ properties in $(Stable(\mathcal{A}(\mathcal{E})), sync; L)$ to the much simpler verification of such properties on $(\mathcal{E}_{c_e}, L)$.

Theorem 4. *For any formula $\phi \in CTL^*(AP)$ and for any stable initial configuration $\{C_0; t_0\}$ of the form described in Section 5.6, we have*

$$(Stable(\mathcal{A}(\mathcal{E})), (sync; L)), \{C_0 \,;\, t_0\} \models \phi$$
$$\Updownarrow$$
$$(\mathcal{E}_{c_e}, L), sync(\{C_0 \,;\, t_0\}) \models \phi.$$

Proof. The function *sync* defines not only a bisimulation between transition systems, but also one between the Kripke structures $(Stable(\mathcal{A}(\mathcal{E})), sync; L)$ and $(\mathcal{E}_{c_e}, L)$, since if $sync(\{C \,;\, t\}) = s$, then $\{C \,;\, t\}$ and s satisfy the same atomic propositions in their respective Kripke structures, since $(sync; L)(\{C \,;\, t\}) = L(sync(\{C \,;\, t\})) = L(s)$. It is well-known that bisimilar structures satisfy the same CTL^* formulas (see, e.g., [7]). □

We now extend the above theorem on preservation of temporal logic properties to increasingly more general settings in two steps: (i) we first allow "stuttering" between stable states by application of "tick" rules; and (ii) we then consider the fully general extension to the asynchronous system $\mathcal{A}(\mathcal{E})$.

[7] Other temporal operators such as $\diamond$, $\Box$, $\leadsto$, $\mathcal{R}$, and so on, can be defined in terms of these and the Boolean connectives; likewise, $\exists$ can be defined in terms of $\forall$.

The first extension is based on considering the Kripke structure

$$(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st} \cup \longrightarrow_{tick}, (sync; L))$$

where, given two stable states c and c', there is a transition $c \longrightarrow_{tick} c'$ iff there is a one-step tick rewrite $c \longrightarrow c'$ in $\mathcal{A}(\mathcal{E})$. We can now relate the Kripke structures $(\mathcal{E}_{c_e}, L)$ and $(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st} \cup \longrightarrow_{tick}, (sync; L))$ by a so-called *stuttering bisimulation* [7, 4]. There is, however, a slight technical point to iron out: due to the presence of dense time, it is possible to consider "Zeno behaviors" in which we have an infinite sequence of "tick" transitions $c_0 \longrightarrow_{tick} c_1 \longrightarrow_{tick} c_2 \ldots$ whose ticking intervals converge to 0. Such behaviors are nonsensical and should be ruled out by requiring that the behaviors of $(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st} \cup \longrightarrow_{tick}, (sync; L))$ on which we evaluate our temporal logic formulas are time diverging. In this way, a stable state can only stutter with ticks for a finite number of steps before a stable transition step is taken. Of course, since now the notions of a step in our synchronous system and in our extended model of stable states do not agree due to stuttering, the "next" operator $\bigcirc$ should be ruled out from formulas; that is, we should consider formulas in the fragment $CTL^* \setminus \{\bigcirc\}(AP)$, where the $\bigcirc$ operator is excluded.

Theorem 5. *For any formula $\phi \in CTL^* \setminus \{\bigcirc\}(AP)$ and for any stable initial configuration $\{C_0; t_0\}$ of the form described in Section 5.6, we have*

$$(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st} \cup \longrightarrow_{tick}, (sync; L)), \{C_0\,;\, t_0\} \models \phi$$
$$\Updownarrow$$
$$(\mathcal{E}_{c_e}, L), sync(\{C_0\,;\, t_0\}) \models \phi.$$

where the semantics of $CTL^* \setminus \{\bigcirc\}(AP)$ in $(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st} \cup \longrightarrow_{tick}, (sync; L))$ is restricted to time diverging paths as explained in [22].

Proof. The function *sync* defines a stuttering bisimulation between the Kripke structures $(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st} \cup \longrightarrow_{tick}, (sync; L))$ and $(\mathcal{E}_{c_e}, L)$. Specifically, we "match" a time-divergent computation in $(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st} \cup \longrightarrow_{tick}, (sync; L))$ with one in $(\mathcal{E}_{c_e}, L)$ by grouping together all stable states connected by tick steps $c_n \longrightarrow_{tick} c_{n+1}$, which all correspond to the single state $sync(c_n) = sync(c_{n+1})$ in the synchronous model (and hence $L(sync(c_n)) = L(sync(c_{n+1}))$). Since the time-divergent behaviors of $(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st} \cup \longrightarrow_{tick}, (sync; L))$ are just like the ordinary behaviors of $(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st}, (sync; L))$, except for the possible insertion of a finite number of "tick" stuttering steps before each stable transition, it is straightforward to adapt the bisimulation proof of Theorem 3 to show that for each time-divergent behavior of $(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st} \cup \longrightarrow_{tick}, (sync; L))$ we have a matching behavior of $(S^{\mathcal{E}} \times D_i^{\mathcal{E}}, \longrightarrow_{\mathcal{E}}, L)$ *and conversely.* That is, that *sync* is a stuttering bisimulation as claimed. One can then use the well-known result (see, e.g., [4, 17, 19]) ensuring that satisfaction of formulas in $CTL^* \setminus \{\bigcirc\}(AP)$ is invariant under stuttering bisimulations. □

In light of Theorem 2, the Kripke structure $(Stable(\mathcal{A}(\mathcal{E})), \longrightarrow_{st} \cup \longrightarrow_{tick}, (sync; L))$ provides the following high-level view of any time-diverging behavior

in $\mathcal{A}(\mathcal{E})$, where in Fig. 7.2 below stable transitions are marked by (curved) thicker arrows $\Longrightarrow$, and all the transitions filling the gaps between stable states are tick transitions. Note that there is a fair degree of choice on when to take a stable transition. Theorem 2 considered the case in which there are no "ticking gaps," but a finite number of ticks between stable states can be interleaved without any problems.

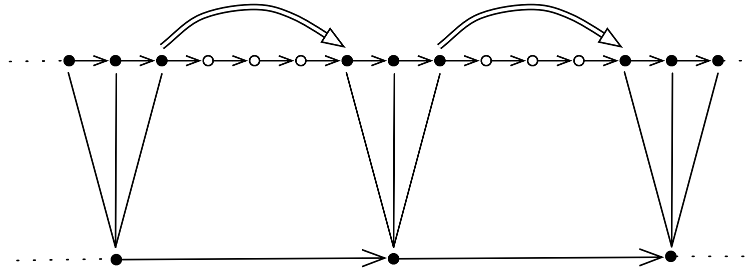

Fig. 4. The top sequence represents a path in $\mathcal{A}(\mathcal{E})$, where the black circles represent stable states, and the thick arrows represent stable transitions. The bottom sequence shows the corresponding in $\mathcal{E}_{c_e}$. The vertical lines denote the *sync* function, mapping stable states in the asynchronous system to states in the synchronous system.

We are now ready to state and prove the fullest semantic equivalence between satisfaction of temporal logic properties in $\mathcal{E}_{c_e}$ and in $\mathcal{A}(\mathcal{E})$. The practical importance of this semantic equivalence resides in the fact that, due to the great amount of concurrency in $\mathcal{A}(\mathcal{E})$, the increase in the number of states when passing from $\mathcal{E}_{c_e}$ to $\mathcal{A}(\mathcal{E})$ is typically exponential, which easily makes the model checking of $\mathcal{A}(\mathcal{E})$ unfeasible, whereas model checking the much simpler model $\mathcal{E}_{c_e}$ may in fact be feasible. Of course, $\mathcal{A}(\mathcal{E})$ has many "unstable" states that do not correspond to any states in $\mathcal{E}_{c_e}$. Therefore, a temporal logic property $\varphi \in CTL^*(AP)$ of $\mathcal{E}_{c_e}$, when evaluated in $\mathcal{A}(\mathcal{E})$ has somehow to be restricted to the stable states in order to be meaningful. This is accomplished by a related formula φ_{stable} as explained below. The Kripke structure associated to $\mathcal{A}(\mathcal{E})$ is denoted $(\mathcal{A}(\mathcal{E}), L') = (\mathcal{T}_{\mathcal{A}(\mathcal{E})_{\texttt{GlobalSystem}}}, \longrightarrow^1_{\mathcal{A}(\mathcal{E})}, L')$, where $\mathcal{T}_{\mathcal{A}(\mathcal{E})_{\texttt{GlobalSystem}}}$ is the set of E-equivalence classes of ground terms of sort `GlobalSystem` for E the equations of the theory $\mathcal{A}(\mathcal{E})$, $\longrightarrow^1_{\mathcal{A}(\mathcal{E})}$ is the one-step rewrite relation between such equivalence classes, and L' is a labeling function satisfying the requirements explained in the theorem below, whose proof is given in Appendix A. This theorem states that *any* CTL^* property φ about the Kripke structure $(\mathcal{E}_{c_e}, L)$ associated to the ensemble $\mathcal{E}$ has a semantically equivalent property φ_{stable} at the level of the Kripke structure $(\mathcal{A}(\mathcal{E}), L')$ associated to the asynchronous system $\mathcal{A}(\mathcal{E})$. Of course, because of stuttering, φ_{stable} does not contain the next operator $\bigcirc$; however, φ can make use of the next operator, giving us full freedom to express

and verify CTL^* properties at the level of the synchronous system specified by $\mathcal{E}$.

Theorem 6. *Given a formula* $\varphi \in CTL^*(AP)$*, and assuming that a new state predicate* $stable \notin AP$ *characterizing stable states has been defined, then there is a formula* $\varphi_{stable} \in CTL^* \setminus \{\bigcirc\}(AP \cup \{stable\})$ *(qualifying* φ *such that it is restricted to stable states) defined recursively as follows:*

$$\begin{aligned}
a_{stable} &= a, \text{ for } a \in AP \\
(\neg\,\varphi)_{stable} &= \neg\,(\varphi_{stable}) \\
(\varphi_1 \wedge \varphi_2)_{stable} &= \varphi_{1_{stable}} \wedge \varphi_{2_{stable}} \\
(\varphi_1\ U\ \varphi_2)_{stable} &= (stable \rightarrow \varphi_{1_{stable}})\ U\ (stable \wedge \varphi_{2_{stable}}) \\
(\bigcirc\,\varphi)_{stable} &= stable\ U\ (\neg stable \wedge (\neg stable\ U\ (stable \wedge \varphi_{stable}))) \\
(\forall\,\varphi)_{stable} &= \forall\,\varphi_{stable}
\end{aligned}$$

such that for each stable state s *in* $\mathcal{A}(\mathcal{E})$ *reachable from initial states defined in Section 5.6, we have*

$$(\mathcal{A}(\mathcal{E}), L'), s \models \varphi_{stable} \quad \Longleftrightarrow \quad (\mathcal{E}_{c_e}, L), sync(s) \models \varphi,$$

where $CTL^*\setminus\{\bigcirc\}(AP\cup\{stable\})$ *formulas are interpreted in* $(\mathcal{A}(\mathcal{E}), L')$ *under the* time-diverging path semantics*, and where* $L' : \mathcal{T}_{\mathcal{A}(\mathcal{E})_{\texttt{GlobalSystem}}} \rightarrow \mathcal{P}(AP \cup \{stable\})$ *is a labeling function satisfying* $L'(s) = L(sync(s)) \cup \{stable\}$ *when* s *is a stable state, and* $stable \notin L'(s)$ *otherwise.*

8 Optimality Results

This section shows that the period $T = 2\epsilon + \mu_{max} + \max(\alpha_{max}, 2\epsilon - \mu_{min})$ is the smallest possible for PALS.

Proposition 1. *Assume that each object reads its input buffer at its* local *time* t_0*, and at that time starts performing a transition and generating new messages. To ensure that all such generated messages are read by all other objects at or before their* local *times* $t_0 + T$*, we must have*

$$T \geq 2\epsilon + \mu_{max} + \alpha_{max}.$$

Proof. Assume for a proof by contradiction that T is *strictly smaller* than $2\epsilon + \mu_{max} + \alpha_{max}$. Then, $T = 2\epsilon + \mu_{max} + \alpha_{max} - \Delta$ for some $\Delta > 0$. Furthermore, let k be a number $k \geq 3$ such that $\Delta < k \cdot \epsilon$.

Now, assume two objects with local clocks c_1 and c_2 such that $c_1(t_0+\epsilon-\frac{\Delta}{k}) = t_0$ and $c_2(t_0+T-\epsilon+\frac{\Delta}{k}) = t_0+T$. That is, at *global* time $t_0+\epsilon-\frac{\Delta}{k}$, the object 1 generates messages that should arrive no later than at *global* time $t_0+T-\epsilon+\frac{\Delta}{k}$, when object 2 reads messages for the next round. However, it is easy to see that, in the worst case (longest execution time and network delay), the messages from object 1 arrive at *global* time $t_0+\epsilon-\frac{\Delta}{k}+\mu_{max}+\alpha_{max}$, which is strictly later than global time $t_0 + T - \epsilon + \frac{\Delta}{k} = t_0 + \epsilon + \mu_{max} + \alpha_{max} - \frac{(k-1)\Delta}{k}$, since $\frac{(k-1)\,\Delta}{k} > \frac{\Delta}{k}$ for $k \geq 3$, so that object 2 misses the messages sent from object 1. □

Proposition 1 proves optimality of T when $\alpha_{max} \geq 2\epsilon - \mu_{min}$. For the converse, and highly unlikely, case where $2\epsilon - \mu_{min} > \alpha_{max}$, it is harder to claim a "general" optimality result. PALS uses a backoff timer to avoid that messages arrive too early. One could imagine variations of PALS where there were no such backoff timers are used, but where messages are instead equipped with, e.g., sequence numbers denoting the round in which they were generated. In such cases, a backoff timer would not be needed, and $T \geq 2\epsilon + \mu_{max} + \alpha_{max}$ might suffice as a smallest period.

However, if we want to ensure that each message arrives in the right round by using backoff timers, then the backoff timers must be set to at least $2\epsilon - \mu_{min}$:

Proposition 2. *To ensure that a message generated in round i (i.e., at* local *time $i \cdot T$) does not arrive too early (i.e., is read by another object at that object's local time $i \cdot T$), the message must not be sent before* local *time $i \cdot T + 2\epsilon - \mu_{min}$.*

Proof. Assume that object 1 sends a message to object 2 *before* local time $i \cdot T + 2\epsilon - \mu_{min}$. That is, it sends the messages at its *local* time $i \cdot T + 2\epsilon - \mu_{min} - \Delta$ for some $\Delta > 0$. Again, we let $k \geq 3$ be some number such that $\Delta < k \cdot \epsilon$.

Furthermore, suppose that $c_1(i \cdot T + 2\epsilon - \epsilon + \frac{\Delta}{k} - \mu_{min} - \Delta) = i \cdot T + 2\epsilon - \mu_{min} - \Delta$. That is, the messages from object 1 are sent at *global* time $i \cdot T + 2\epsilon - \epsilon + \frac{\Delta}{k} - \mu_{min} - \Delta$. With the smallest possible network delay, these messages may arrive at *global* time $i \cdot T + \epsilon - \frac{(k-1)\Delta}{k}$, whereas it could be the case that $c_2(i \cdot T + \epsilon - \frac{\Delta}{k}) = i \cdot T$, and, therefore, object 2 would read the messages from object 1 one round too early. □

The optimality of the period follows immediately:

Proposition 3. *If each message for round $i + 1$ is sent no earlier than at* local *time $i \cdot T + 2\epsilon - \mu_{min}$, then we must have*

$$T \geq 4\epsilon + \mu_{max} - \mu_{min}$$

to ensure that each object has received these messages at its local *time $i \cdot T + T$.*

Proof. Assume for a counterexample that $T = 4\epsilon + \mu_{max} - \mu_{min} - \Delta$ for $\Delta > 0$ with $\Delta < k \cdot \epsilon$ for some $k \geq 3$.

Let us assume two objects with local clocks c_1 and c_2, where $c_1(i \cdot T + 2\epsilon - \mu_{min} + (\epsilon - \frac{\Delta}{k})) = i \cdot T + 2\epsilon - \mu_{min}$. That is, object 1 does not send its messages for round $i + 1$ earlier than at *global* time $i \cdot T + 2\epsilon - \mu_{min} + (\epsilon - \frac{\Delta}{k})$. In the worst case, these messages arrive at *global* time $i \cdot T + 3\epsilon - \mu_{min} + \mu_{max} - \frac{\Delta}{k}$. However, it could well be the case that $c_2(i \cdot T + T - \epsilon + \frac{\Delta}{k}) = i \cdot T + T$. That is, the messages arriving at time $i \cdot T + 3\epsilon - \mu_{min} + \mu_{max} - \frac{\Delta}{k}$ arrive later than the global time $i \cdot T + T - \epsilon + \frac{\Delta}{k} = i \cdot T + 3\epsilon + \mu_{max} - \mu_{min} - \frac{(k-1)\Delta}{k}$ when object 2 reads its messages for round $i + 1$. □

9 An Avionics Case Study

To illustrate the benefits of PALS for model checking, we have defined in Maude and Real-Time Maude, respectively, a synchronous version and a simplified asynchronous PALS version of an *active standby* avionics system for deciding which of two computer systems is active in an aircraft. The active standby system is in essence a synchronous design, but must be realized as an asynchronous distributed system for fault tolerance reasons. Our models are based on an AADL [30] model for active standby developed by Abdullah Al-Nayeem at UIUC of a similar active standby specification developed by Steve Miller and Darren Cofer at Rockwell-Collins. The active standby system is extensively discussed in [20].

As explained in Section 9.1, the active standby system consists of three components. We have defined in Maude the synchronous composition of these three components, and have defined in Real-Time Maude a much simplified PALS-based asynchronous model of the active standby system. The simplifications in the PALS-based model include: (i) perfect and perfectly synchronized clocks, (ii) discrete time, (iii) the execution time of a transition and the minimum network delay are both 0, and (iv) the maximum network delay is a parameter of the system. As explained in Section 9.2, the synchronous model has 185 reachable states and model checks in less than a quarter of a second, whereas even the much simplified asynchronous model, with *maximum* messaging delay zero, has more than 3 million reachable states and takes more than half an hour to model check. If the messaging delay can be either 0 or 1, then the simple asynchronous system has a huge number of reachable states exceeding the memory capacity (8 GB RAM) of the server machine on which we performed the model checking experiments.

Since the active standby model is quite small, and even a very simple asynchronous version of it cannot be model checked, this example illustrates the practical impossibility of *directly* model checking DES systems, except in extremely simplified cases. The great advantage of PALS and of Theorem 6 is that they provide an *indirect* method for formally verifying DES systems of the general style described in this paper by model checking their synchronous designs.

9.1 The Active Standby System

The *active standby* system is a simplified example of a fault-tolerant avionics system. 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 (e.g., an explosion) 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*. While one side is active, the other side remains passive. The two sides receive inputs through communication channels. Each side could fail, but it is assumed that both sides cannot fail at the same

time. 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. Each side is dependent on other system components. In this example, the full functionality of each side is dependent on these two sides' perception of the availability of these system components. Only a fully functional/available side should be active, while the other side is alive but not fully functional.

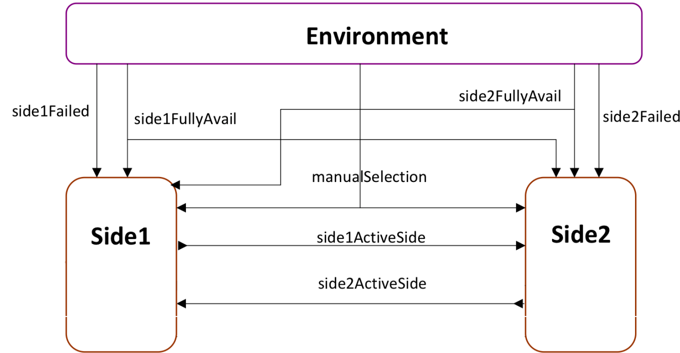

Fig. 5. The architecture of the active standby system.

As already mentioned, our models of active standby are based on a model, defined using the avionics standard AADL [30], developed by Abdullah Al-Nayeem. The architecture of the system is shown in Figure 5. The system consists of three components: Side1, Side2, and Environment. Side1 and Side2 encapsulate the behaviors of the two sides. The Environment component can be considered as an abstract representation of other non-specified components that interact with Side1 and Side2. The design of the Active Standby system is globally synchronous; i.e., Side1, Side2, and Environment all have the same period and dispatch at the same time. Each time Environment dispatches, it sends 5 Boolean values, one through each of its out ports shown in Figure 5. These values are nondeterministically generated, with the following constraints:

1. two sides cannot fail at the same time, and
2. a failed side cannot be fully available.

Therefore, in each round, the environment can nondeterministically generate any one of 16 different 5-tuples of Boolean values. It is also worth remarking that the connections between the two sides are "delayed" connections; a message sent in one round is read by the other side in the next round.

Important properties that the Active Standby system should satisfy include:

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.

9.2 Synchronous and Asynchronous Models of Active Standby in Maude

To illustrate the usefulness of the PALS methodology, we have modeled both the synchronous and a simplified PALS-based asynchronous version of the active standby system in, respectively, Maude and Real-Time Maude. In this section we just give a brief overview of these executable models; they are explained in more detail in Appendix B, where the entire executable models are also given.

The Synchronous Model. We have defined the synchronous system in a object-oriented style, where the two sides and the environment are modeled as objects; furthermore, each *output port* is modeled by an object that contains the value sent from the source. The transition relation of each side is a function.

The synchronous model has only one rewrite rule:

```
subsort EnvOutput < EnvOutputs .
op _;_ : EnvOutputs EnvOutputs -> EnvOutputs [ctor assoc comm] .

var ENVOUTPUT : EnvOutput .   var RESTOUTPUT : EnvOutputs .
var : Configuration .

crl [step] :
     {SYSTEM} => {genOutput(performTrans(envoutput(ENVOUTPUT, SYSTEM)))}
     if ENVOUTPUT ; RESTOUTPUT := possibleEnvOutputs .
```

The constant `possibleEnvOutputs` denotes the set of all 16 possible environment outputs, and is modeled as a set with an associative and commutative set union operator `_;_`. The variable `ENVOUTPUT` can therefore match any one of these 16 environment outputs. The operator `{_}` encloses the entire state. Therefore, in each round, this rule first generates the environment output nondeterministically, these environment values are then entered into the appropriate output ports (`envoutput`), and the transition function is applied to each side object (and all output ports) (`performTrans`) by just applying the transition function to each single side object, and, finally, the outputs from the side objects are entered into the corresponding output ports (`genOutput`; this must obviously be done after the transitions have been performed). The above functions are all declared to be partial functions of sort `Configuration` to ensure that they are indeed applied in the above stated order.

The Asynchronous Model. As already mentioned, we have modeled in Real-Time Maude a much simplified PALS-based distributed asynchronous version of the active standby system. We have made the following simplifying assumptions:

– The time domain is discrete.
– The clocks all advance at the same rate and are perfectly synchronized.
– The time to perform a transition, including reading from the input buffer and writing to the output buffer, is zero.
– The minimum message delay is zero, and the maximum messaging delay is a parameter constant `maxMsgDelay`.

The behavior of the asynchronous system can then be summarized as follows:

– The length of each PALS period is `maxMsgDelay` + 2.
– When a new round starts, an object reads the incoming messages from its input buffer, performs the transition, thereby changing its internal state and generating output messages, which are placed in the object's output buffer.
– One time unit after the start of the PALS period, each object sends its output messages from the output buffer into the network in *one* step.
– When an object receives a message, the message is stored in the object's input buffer.

In our model, we define "PALS wrappers" as object instances of the following class:

```
class PalsWrapper | roundTimer : Time,  inputBuffer : Configuration,
                    outputBuffer : Configuration,
                    outputBackoffTimer : TimeInf,  machine : Object .
```

The `machine` attribute denotes the object modeling the actual component, and the other attributes are as in Section 4.

The rewrite rules in this model are straightforward. The following rule reads an incoming message and puts it into the input buffer:[8]

```
rl [readMsg] :
   dly(msg D from port P of O' to O), T)
   < O : PalsWrapper | inputBuffer : MSGS >
  =>
   < O : PalsWrapper | inputBuffer : MSGS (msg D from port P of O' to O) > .
```

When the round timer expires, messages are read from the input buffer, the transition is performed and output is put into output buffer, and the output timer is set to 1:

```
crl [executeTransitionSide1] :
    < side1 : PalsWrapper | roundTimer : 0,
                            inputBuffer : MSGS,
```

[8] The second argument of the `dly` operator shows the maximum remaining delay of the message.

```
                           machine :
                             (< side1 : Side1 | state : L1, prevS2AS : N1,
                                                prevManualSwitch : B1,
                                                nexts1as : data(0) >) >
   =>
    < side1 : PalsWrapper | roundTimer : palsRound,
                           inputBuffer : none,
                           machine :
                             (< side1 : Side1 | state : L2, prevS2AS : N2,
                                                prevManualSwitch : B2,
                                                nexts1as : data(0) >),
                           outputBuffer :
                            dly(msg D2 from port s1AS of side1 to side2,
                                maxMsgDelay),
                           outputBackoffTimer : 1 >
   if < side1 : Side1 | state : L2, prevS2AS : N2, prevManualSwitch : B2,
                       nexts1as : D2 >
      C:Configuration
       := performTrans(< side1 : Side1 | state : L1, prevS2AS : N1,
                                         prevManualSwitch : B1,
                                         nexts1as : data(0) >
                      changeForm(MSGS)) .
```

The function `performTrans` is the transition function for the components defined for the synchronous model (and hence `changeForm` is needed to transform messages into machine inputs in corresponding “wires”). The rule for `side2` is entirely similar. The rule for the environment is also straightforward:

```
crl [envRound] :
    < e : PalsWrapper | roundTimer : 0 >
   =>
    < e : PalsWrapper |
           roundTimer : palsRound,
           outputBuffer :
             (dly(msg data(B1) from port s1F of e to side1, maxMsgDelay)
              dly(msg data(B2) from port s2F of e to side2, maxMsgDelay)
              dly(msg data(B3) from port mS of e to side1, maxMsgDelay)
              dly(msg data(B3) from port mS of e to side2, maxMsgDelay)
              dly(msg data(B4) from port s1FA of e to side1, maxMsgDelay)
              dly(msg data(B4) from port s1FA of e to side2, maxMsgDelay)
              dly(msg data(B5) from port s2FA of e to side1, maxMsgDelay)
              dly(msg data(B5) from port s2FA of e to side2, maxMsgDelay)),
          outputTimer : 1 >
   if env(B1, B2, B3, B4, B5) ; ENVOUTPUTS := possibleEnvOutputs .
```

In the following rule, the messages in the output buffer are sent into the network in one step:

```
  rl [send] :
     < O : PalsWrapper | outputTimer : 0, outputBuffer : MSGS >
```

```
=>
  < O : PalsWrapper | outputTimer : INF, outputBuffer : none >  MSGS .
```

Finally, the tick rule advances time by one time unit in each tick step:

```
crl [tick] : {CONF} => {delta(CONF, 1)} in time 1 if 1 <= mte(CONF) .
```

9.3 Comparing the Performance of Model Checking the Synchronous and Asynchronous Models

We have compared the number of reachable states in our two models, as well as the execution times times for model checking analysis. We have chosen an invariant to compare the model checking performance in the synchronous and asynchronous models. Model checking an invariant requires exhaustive search of all reachable states and is therefore not subject to peculiarities in the search strategy that can make the model checking performance of other LTL properties in two different models less predictable, so that performance comparisons become less reliable. The invariant we analyze is that when a side is failed, it will only transmit the value 0.

In the *synchronous model*, the number of states reachable from the initial state is *185*, and both reachability analysis and LTL model checking take less than a second on a 1.86 GHz server with 8 GB RAM.

We model check the invariant on the *asynchronous model* first with *no messaging delay*. With instantaneous message transmission, we could model check the system in 33 minutes, and Maude shows that there are then *3,047,832 reachable states*. If we restrict the environment to 12 instead of 16 possible different outputs (by not allowing side 1 to fail), then Maude shows that there are *1,041,376 reachable states*, and it takes about *190 seconds* to search the entire state space. If we further restrict the environment so that *no side can fail*, then we get *243,360 reachable states* that can be analyzed in *30 seconds*.

We have also analyzed the asynchronous model for *maximal messaging delay 1*. That is, each message may take either zero or one time units to arrive. In this case, exhaustive state space exploration was *aborted* by the operating system after two hours, most likely due to the execution using up too much memory. Restricting to 12 environment possibilities showed that *1,496,032 states* were reachable from the given initial state. The analysis took *420 seconds* of cpu time. With only eight different environment possibilities, the numbers were *349,856 reachable states* and *52 seconds*.

These numbers are summarized in the following table (where the command executed is the search command that searches for a state violating the invariant described above):

Model	Max.msg.dly	8 env. possibilities		12 env. poss.		16 env. poss.	
		# states	ex.time	# states	ex.time	# states	ex.time
Synchr.	n/a	47	0.04 sec.	107	0.1 sec.	185	0.2 sec.
Asynchr.	0	243,360	30 sec.	1,041,376	190 sec.	3,047,832	2000 sec.
Asynchr.	1	349,856	52 sec.	1,496,032	420 sec.	aborted	

To conclude, whereas the synchronous version can be model checked in less than one second, only the simplest possible distributed asynchronous version can be feasibly model checked.

It is worth remarking that the system is not particularly large: 10 messages are sent in each round; the number of internal states of each machine is bounded by $6 \cdot 3 \cdot 2 = 36$; the data in the messages are either Boolean values or a natural number between 0 and 2; time is discrete; executions and message transmissions are instantaneous (when `maxMsgDelay` is 0; otherwise message delays are either 0 or 1); and there are no clock skews. Furthermore, there is nothing special about our model that causes the state space explosion; indeed, the multiple messages generated by an object are all generated in one step, and are also sent into the network in one step. The main factor contributing to the state space explosion is the great number of interleavings caused by the intrinsic concurrency of the asynchronous system, since there are of course 10! different orders in which messages in one round can be received. Although in some cases this combinatorial explosion can be partially tamed by model checking techniques such as partial order reduction and (for systems exhibiting a good degree of symmetry) symmetry reduction (see, e.g., [7]), the great advantage of PALS and of Theorem 6 is that they offer the possibility of avoiding such an explosion altogether, by reducing the (generally unfeasible) model checking of asynchronous DES systems of the style described in this work to that of their much simpler synchronous designs.

9.4 Model Checking the Requirements R1–R5

This section gives a brief summary of our model checking analyses of the *synchronous* model w.r.t. the requirements R1–R5. Theorem 6 then gives the corresponding property that is then indirectly analyzed in the *asynchrnous* model.

A detailed discussion of the model checking analyses, including the formal definition of the state predicates in the folrmulas below, can be found in Appendix B.

Requirement R1: *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).*

By "side i being active" we assume that what is meant is that that side i sends its number to the other side through its side*i*activeSide port; this value is given in the next*i*as attribute of the side*i* object. The parameterized atomic proposition `side_active` can therefore be defined as follows:

```
op side_active : Nat -> Prop [ctor] .
eq {CONF  < side1 : Side1 | nexts1as : data(N) >} |= side 1 active = (N == 1) .
eq {CONF  < side2 : Side2 | nexts2as : data(N) >} |= side 2 active = (N == 2) .
```

We can then define what it means that both sides agree on which side is active:

```
op agreeOnActiveSide : -> Prop [ctor] .
eq {CONF
    < side1 : Side1 | nexts1as : data(N1) >
    < side2 : Side2 | nexts2as : data(N2) >} |= agreeOnActiveSide
  = N1 == N2  and N1 =/= 0 .
```

Likewise, we can define state predicates `side 1 availChanged` (the full availability of side 1 has just changed), `side 2 availChanged`, `changeInAvailability` (the full availability at least one of the sides has just changed), `manSelectPressed` (the pilot has just toggled the manual switch), and `side` i `failed`. We can combine these state predicates into formulas stating, respectively, that neither side has failed and that there is no change in the assumptions:[9]

```
ops  neitherSideFailed noChangeAssumption : -> Formula .

eq neitherSideFailed = (~ side 1 failed) /\ (~ side 2 failed) .
eq noChangeAssumption
 = ~ changeInAvailability /\ ~ manSelectPressed /\ neitherSideFailed .
```

We are now ready to define formally Requirement R1. However, as explained above, it is the *passive* side that monitors the manual selection and fully available values from the environment. When the passive (or standby) side realizes that the active side should change, it notifies the currently active side. This notification will arrive in the *next* iteration, so there is a round in which each side thinks that it is active. The best we can hope for is that they agree either in this round or in the next; furthermore, if one side fails in the next round, then we may still not have an agreement, so the following is the best we can hope for:

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

Indeed, model checking this property returns `true` (in about 0.8 seconds), so the property holds:

```
Maude> (red modelCheck(init, R1) .)
rewrites: 102954 in 829ms cpu (837ms real) (124097 rewrites/second)

result Bool : true
```

Requirement R2: *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).*

[9] Remember that ~ denotes negation of Maude's LTL formulas.

This property obviously does not hold as stated; the standby side monitors full availability, and hence the change of active side might be delayed by one round. Therefore the fomula `R2a` is the best we can hope for *side 1*:

```
op R2a : -> Formula .
eq R2a
 = [] ((noChangeAssumption /\ side 1 fullyAvailable /\ ~ side 2 fullyAvailable)
       -> (~ side 2 active \/ O (noChangeAssumption -> ~ side 2 active))) .
```

Model checking shows (again in 0.8 seconds) that `R2a` holds in our model:

```
Maude> (red modelCheck(init, R2a) .)
rewrites: 101703 in 812ms cpu (814ms real) (125160 rewrites/second)

result Bool : true
```

We have model checked similar formulas for side 2, but the property does not hold. The counterexamples provided by Maude's model checker allowed us to analyze the failures of the property for side 2 (see Appendix B); it may take as much as four steps to reach the desired state after side 1 is no longer fully available. Therefore, the best we can hope for side 2 is:

```
op R2b : -> Formula .
eq R2b
 = [] ((noChangeAssumption /\ side 2 fullyAvailable /\ ~ side 1 fullyAvailable)
       -> (~ side 1 active \/
           O (noChangeAssumption -> (~ side 1 active \/
              O (noChangeAssumption -> (~ side 1 active \/
                 O (noChangeAssumption -> (~ side 1 active \/
                    O (noChangeAssumption -> ~ side 1 active)))))))) .
```

Model checking this property returns `true` (in 0.8 seconds). The reason for the difference in sides seems to be due to the fact that the sides are asymmetric in their failure recovery. After a failure, there is bias towards side 1 being the active side.

Requirement R3: *The pilot can always change the active side (except if a side is failed or the availability of a side has changed).*

It is unclear what is meant by *"The pilot can always change the active side."* It is obvious that the pilot can always request the switch; however, the request may be ignored, because it contradicts the requirement that if one side is fully available and the other one is not, then the fully available side should be the active side.

Given that it is easy to see that the environment always can generate a *manual selection* request, we interpret requirement R3 as follows:

> If both sides agree on the active side, both sides are fully available, and then the manual selection is activated (and there is still no lack of availability), then the active side should change either immediately, or, at latest, in the next round (unless there are failures or lack of availability).

This interpretation can be formalized as the following LTL formula `R3a`:

```
op R3a : -> Formula .
eq R3a =
   [] ((side 1 fullyAvailable /\ side 2 fullyAvailable /\ agreeOnActiveSide)
       ->
       (  (side 1 active
           -> O ((manSelectPressed
                  /\ side 1 fullyAvailable /\ side 2 fullyAvailable)
                 -> (side 2 active
                     \/ O (noChangeAssumption -> side 2 active))))
       /\ (side 2 active
           -> O ((manSelectPressed
                  /\ side 1 fullyAvailable /\ side 2 fullyAvailable)
                 -> (side 1 active
                     \/ O (noChangeAssumption -> side 1 active)))))) .
```

Model checking this formula returns a counterexample, in which both sides *continue* to agree that side 1 is the active side, even though there are no failures when the pilot presses the button. (Adding more next-state disjunctions does not help.) Briefly stated, the source of the problem is the following:

- In most circumstances, if the system gets a manual selection request, this request will be "recorded," and the following consecutive manual selection requests will be ignored.
- When some component is not fully available, the system cannot always obey the pilot's desire to switch the active side. *However, even if the system cannot grant the manual switch request, it remembers that the manual switch was requested.*

The path provided by Real-Time Maude's model checker as a counterexample to the validity of the above LTL property shows that the pilot makes a manual switch request when a side is not fully available (and hence the switch of active sides does not take place). In the next round, all components are OK, and the pilot again requests a switch of active sides. But, this last request is ignored, since the system registered that the pilot pressed the manual selection in the previous round. All following consecutive manual requests will also be ignored.

It seems that the following property `R3g` is the strongest one that holds (except in the initialization phase) in our specification. 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 fully available and receive a manual switch request in the third round, then the active side will switch *instantaneously*:

```
op R3g : -> Formula .
eq R3g = [] ( (~ manSelectPressed /\ agreeOnActiveSide
              /\ side 1 fullyAvailable /\ side 2 fullyAvailable
              /\ (O noChangeAssumption))
             -> (   (side 1 active
                       -> O O ( (manSelectPressed /\ side 1 fullyAvailable
```

```
                              /\ side 2 fullyAvailable)
                            -> (side 2 active)))
               /\ (side 2 active
                    -> O O ( (manSelectPressed /\ side 1 fullyAvailable
                              /\ side 2 fullyAvailable)
                            -> (side 1 active))))) .
```

This property does hold in the initialization phase, so we start the model checking of the above property in the second state:

```
Maude> (red modelCheck(init, O R3g) .)
rewrites: 102216 in 834ms cpu (840ms real) (122521 rewrites/second)

result Bool : true
```

Requirement R4: *If a side is failed the other side should become active.*

As seen in Fig. 6, only the failed side gets the signal about its failure. A failed side signals the failure to the other side by sending a '0' value to the other side. Since this communication has a one-step delay, the best we can hope for is that the other side becomes active in the *next* state:

```
op R4 : -> Formula .
eq R4 = [] (((side 1 failed /\ ~ side 2 failed)
             -> O (~ side 2 failed -> side 2 active))
           /\ ((side 2 failed /\ ~ side 1 failed)
               -> O (~ side 1 failed -> side 1 active))) .
```

This property holds in our model:

```
Maude> (red modelCheck(init, R4) .)
rewrites: 101597 in 825ms cpu (831ms real) (123055 rewrites/second)

result Bool : true
```

Requirement R5: *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:

```
op R5side1 : -> Formula .
eq R5side1
 = [] (((side 1 active /\ side 1 fullyAvailable /\ ~ manSelectPressed)
        -> (side 1 active W (~ side 1 fullyAvailable \/ manSelectPressed)))
      /\ ((side 1 active /\ ~ side 1 fullyAvailable /\ ~ side 2 fullyAvailable
           /\ ~ manSelectPressed /\ ~ side 1 failed)
          -> (side 1 active W
              (side 1 fullyAvailable \/ side 2 fullyAvailable
               \/ manSelectPressed \/ side 1 failed)))) .
```

This formula also model checks successfully:

```
Maude> (red modelCheck(init, R5side1) .)
rewrites: 101702 in 828ms cpu (831ms real) (122803 rewrites/second)

result Bool : true
```

Side 2 is trickier, 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. 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 : -> Prop [ctor] .
eq {CONF < side2 : Side2 | state : side2Active >} |= s2InStateSide2Active
 = true .

op R5side2X : -> Formula .
eq R5side2X
 = [] (((s2InStateSide2Active /\ side 2 fullyAvailable
         /\ ~ manSelectPressed /\ ~ side 1 failed)
        -> (s2InStateSide2Active W
            (~ side 2 fullyAvailable \/ manSelectPressed \/ side 1 failed)))
       /\ ((s2InStateSide2Active /\ ~ side 2 fullyAvailable
            /\ ~ side 1 fullyAvailable /\ ~ manSelectPressed
            /\ ~ side 2 failed /\ ~ side 1 failed)
           -> (s2InStateSide2Active W
               (side 2 fullyAvailable \/ side 1 fullyAvailable
                \/ manSelectPressed \/ side 2 failed \/ side 1 failed)))
       /\ ((side 2 active /\ ~ manSelectPressed /\ ~ side 2 failed
            /\ side 1 failed)
           -> (side 2 active W
               (manSelectPressed \/ side 2 failed \/ ~ side 1 failed)))) .
```

This property also model checks successfully in less than a second

```
Maude> (red modelCheck(init, R5side2X) .)
rewrites: 102073 in 837ms cpu (845ms real) (121841 rewrites/second)
```

10 Related Work

We first explain how this work is related to other work on PALS involving our colleagues at Rockwell-Collins and at UIUC [20, 2, 31]. The PALS transformation itself and its optimal period, as well as the active standby example, are also presented in [20, 2, 31]. The main new contributions of the work presented here are the formal specification of PALS as a real-time rewrite theory parameterized by the input synchronous ensemble and the performance bounds, the proof of correctness of such a formal model, and the mathematical justification of the

method by which the verification of temporal logic properties of the DRTS thus obtained can be reduced to the verification of such properties on the typically much simpler synchronous model. We refer to [20, 2, 31] for additional discussion on PALS and its engineering applications.

Generally speaking, distributed computation models are classified into: (i) *synchronous* models, which operate in a lock-step fashion; and (ii) *asynchronous* ones, where there is no a priory bound on message delays. Our notion of an ensemble is an automata-theoretic version of a synchronous model quite similar to other models such as, for example, the notion of a synchronous system in [32], and the synchronous model of Mealy machines in [34], but with some differences. For example, ensembles allow non-deterministic machines, whereas the Mealy model in [34] assumes deterministic ones; and ensembles make explicit the notion of an external environment, which is important for embedded system applications, whereas in the model in [32] an environment would typically be abstracted as another synchronous process.

The PALS pattern can then be understood as part of a broader body of work on so-called *synchronizers*, which allow synchronous systems to be *simulated* by asynchronous ones. Very general synchronizers such as those in [3] place no a priori bounds in message delays, so that *physical time* in the original synchronous system is simulated by *logical time*[10] in its asynchronous counterpart. More recent work has developed synchronizers for the Asynchronous Bounded Delay (ABD) Network model [6, 33], in which a bound can be given for the delay of any message transmission from any process to any other process. PALS can then be understood as a synchronizer that also assumes the ABD model (plus clock synchronization) as its infrastructure and furthermore provides *real-time guarantees* needed for embedded systems applications. The main differences between the synchronizers in [6, 33] and PALS can be summarized as follows:

1. The work on PALS is motivated by the fact that clock synchronization is routinely used in distributed embedded systems. PALS therefore assumes that a clock synchronization algorithm with a skew bound ϵ is running in the underlying infrastructure. Instead, the synchronizers in [6, 33, 32] provide a clock synchronization algorithm as part of the synchronizer itself.
2. As a consequence of (1), in systems using the synchronizers in [6, 33], the nodes are not closely synchronized in physical time, so that, at the same global physical time, one node could be in its nth round while another node is in its kth round, for any $n, k \in \mathbb{N}$. There may therefore be no global physical time at which all nodes are in the same round. The fact that a state s in the synchronous system may not have a corresponding "stable" state s' in the asynchronous execution makes it impossible to relate the temporal logic properties of the synchronous system and its asynchronous counterpart *in physical time*, as we have done for PALS, although it would still be possible to relate them with a notion of logical time à la Lamport. This lack of "physical time synchronization" is of course unsatisfactory for

[10] That is, an assignment of logical clocks to processes in the style of [15], whose values need not reflect physical time.

safety-critical and performance-critical distributed embedded systems, such as avionics systems and motor vehicles, that have to satisfy hard real-time requirements. In contrast, in PALS, at any moment in (global) physical time, each node is either in round i or in round $i+1$, and in each round there are "stable" states in which all components are in the same round.

3. There is also a period optimality result in [33] which has a similar counterpart in the PALS's optimal period. However, as explained in (2), the meaning of those results is different, with optimality in PALS ensuring synchrony in physical time while this cannot be ensured by the synchronizer in [33].

The work by Tripakis et al. [34] provides formal models of synchrony and asynchrony which can be related to those of PALS. As pointed out above, their synchronous model is one of interconnected deterministic Mealy machines. As in the case of PALS, the work in [34] also deals with the problem of mapping a synchronous architecture consisting of a synchronous interconnection of state machines to an asynchronous architecture. In their case, this is a loosely timed triggered architecture, where processes communicate asynchronously and have local clocks that can advance at different rates and where no clock synchronization is assumed. This mapping is achieved through an intermediate translation into a Kahn-like dataflow network with bounded buffers. The main result in [34] is the preservation of streams and therefore the correctness of the asynchronous architecture's implementation of the original synchronous system. In some sense their result shows the robustness of their mapping, since correctness is achieved in spite of unpredictable communication delays and possibly different clock rates in the different processes. The main differences with this work, and with the PALS idea more generally, is that, due to the quite minimal assumptions made on their asynchronous dataflow model, it does not seem possible to give hard real time bounds for the behavior of the asynchronous system realization; and it seems also problematic to deal with the freshness of environment data coming from sensors that must be responded to within specific time bounds. Because our concern is with systems, such as avionics ones, whose distributed implementation must satisfy hard real-time constraints just as stringent as those of the original synchronous systems they implement, the model in [34], while very useful and flexible for correctness purposes, does not seem to fully meet the real-time needs of such systems.

PALS is also closely related, with some important differences, to *time-triggered systems* in the sense of J. Rushby [27]. More specifically, our PALS model gives a detailed formal specification of the middleware to achieve a somewhat different notion of a time-triggered system, for which we prove correctness and time optimality results. In contrast, Rushby's model (as corrected by L. Pike for some minor inconsistencies [24, 25]) is more abstract, and does not specify a detailed middleware. Indeed, Theorem 1 in [27] (similar to our Theorem 3) says that states of the synchronous system are identical to what we call stable states of the asynchronous system.

One important difference between the work of Rushby (and Pike) and ours on PALS comes from the somewhat different definitions of the respective syn-

chronous models, which have significant repercussions in the behaviors of the corresponding asynchronous models. In the synchronous model of [27, 25], inspired by Lynch's synchronous model [16], each round has two phases: in the first phase, processors send messages based (only) on their current state (this phase is formalized by a function $msg_p : states_p \times out_nbrs_p \rightarrow msgs$, for each processor p, in [27]); in the second phase, each node reads the incoming messages and updates its state accordingly (formalized by a function $trans_p : states_p \times inputs_p \rightarrow states_p$ in [27]). In contrast, our synchronous model only has one "phase" in each round: read incoming messages and update the state and generate new messages (formalized by the relation $\delta_M \subseteq ((D_i \times S) \times (S \times D_o))$. This seemingly innocuous difference carries over to the asynchronous timed models. In [27, 25], after reading incoming messages, the system executes the transition, and only after this execution phase is finished (and hence the new state is computed), are the new messages created, which are then sent into the network *after an additional "backoff" delay* corresponding to our dly_{out} value. In PALS, the transition execution time is "included" in the backoff delay in sending the newly generated messages into the network. For example, if the execution time α is greater than dly_{out}, there is no (additional) backoff delay until the messages are sent into the network. Therefore, the smallest possible period of the asynchronous system of Rushby and Pike is typically significantly larger than the optimal PALS period.

Another important difference between [27, 25] and our work is that, to the best of our knowledge, our work is the first to systematically study the equivalence of temporal logic properties between the synchronous system and the *entire* asynchronous one, including non-stable states. In particular, we show that if the synchronous system is finite-state, verification of properties in the (typically infinite-state) real-time asynchronous system can be achieved by finite-state model checking of the synchronous one. This does not seem possible in Rushby's formalization as given, since it includes a round counter that makes the synchronous system itself infinite-state. Furthermore, Rushby's synchronous systems are *deterministic*—that is, are given by output and transition *functions* msg_p and $trans_p$—whereas we deal with systems that can be nondeterministic both because of faults and because of inputs from an external environment. In fact, it is the nondeterminism allowed in our synchronous model that makes adding a round counter to indirectly express nondeterministic behaviors unnecessary.

Yet another body of related work is centered around the Globally Synchronous Locally Asynchronous (GALS) Architecture, e.g., [10, 11, 26]. The kinds of systems envisioned by the GALS approach are broader and more general than those that can be naturally modeled with PALS, in the sense that GALS systems may be widely distributed and it may not be possible to enforce or assume that all message communication delays are bounded, although such delays may be bounded within a synchronous subdomain. For example, [10] consider in detail the formal verification of a GALS case study: a ground-plane communication system where the ground and the plane can exchange files using a TFTP protocol that executes asynchronously over unreliable UDP channels. Their model of this

system encapsulates synchronous subdomains as automata transition functions within different processes of the LOTOS process calculus. Within the GALS framework, several research efforts, including [11, 26], have studied the problem of correctly simulating a synchronous model as an asynchronous GALS model. In some sense, their solutions have some similarities with the approach in [34]. For example, the work in [26] uses a concurrent transition system formalism to define both synchronous and asynchronous compositions of synchronous systems linked by FIFO channels, and studies conditions under which a synchronous system can thus be correctly simulated as an asynchronous GALS system. Likewise, the work in [11] uses FIFO channels to connect several synchronous hardware circuits into a GALS system that correctly simulates the lockstep synchronous behavior of the bigger circuit obtained by composing the subcircuits. The main difference between these approaches and the PALS pattern is that no hard real-time guarantees can be given for such GALS implementations.

The ABD Network model used by the synchronizers in [6, 33] and by PALS is a very useful abstraction. However, this abstraction places stringent demands on an actual network design that must guarantee such bounded time delivery of messages in the physical world under some stringent model of possible failures. Furthermore, such a network must ensure that clock skew is always bounded even in the presence of the failures assumed by the model. Therefore, an actual implementation of the ABD model must of necessity deal with issues such as: (i) fault tolerance and consistency of message transmission, by replicating network components and by providing appropriate middleware; and (ii) fault-tolerant clock synchronization. These real-time and fault-tolerance requirements have stimulated the development of various network architectures such as, e.g., [5, 14, 1, 12, 21, 9]. In general, such network architectures are classified as either *time-triggered*, in which all activities are triggered by clock pulses, and *event-triggered*, where events in the environment or in the processors can trigger system activities [14]. We refer to the excellent survey by J. Rushby [28] for a detailed discussion of several of these network architectures, specifically [14, 12, 21, 9], some of which are realized in commercial products used in actual airplane or automotive systems, and have in some cases become standards, such as the ARINC standard based on [12]. The good news from the PALS point of view is that the ABD network assumptions made by PALS can indeed be met by real network systems under stringent fault models. Another piece of good news is that, as surveyed by J. Rushby in [29], some of these network architectures, including advanced versions of [14], have been partially formally verified, further increasing the confidence on their correct realization of the ABD network abstraction.

11 Conclusions and Future Work

This work has presented a formal specification of the PALS architectural pattern for obtaining correct-by-construction distributed real-time systems from their synchronous designs under given performance assumptions on the underlying infrastructure. Using the PALS formal model we have given proofs of correctness

of PALS, and of optimality of the PALS period; and we have based on such proofs a method to verify temporal logic properties of the DTRS so obtained by verifying such properties on its much simpler synchronous design. We have also illustrated this method's usefulness by means of an avionics case study. We believe that PALS, as a formalized architectural pattern that greatly reduces system complexity, can substantially increase system quality and can greatly reduce the cost of design, verification, and implementation of distributed real-time systems; and also the cost of certifying highly critical systems of this kind.

Several future developments would be highly desirable. First, both the synchronous composition of a machine ensemble and the PALS transformation itself should be automated within Maude and Real-Time Maude as parameterized specification transformations. A first prototype of a parameterized specification transformation in Maude for the synchronous composition of a machine ensemble is reported in [13], but this should be made more flexible to support, for example, object-based ensemble specifications. Second, since the formal executable specifications of the wrappers used to build PALS as a collection of wrapped abstract machines communicating through message passing are *executable*, they can be used as a basis on which correct-by-construction PALS implementations could be developed by code generation from synchronous implementations. Such code generation schemes should be formally verified, based on a rewriting logic semantics of the programming language of the target code.

References

1. T. F. Abdelzaher, A. Shaikh, F. Jahanian, and K. G. Shin. Rtcast: lightweight multicast for real-time process groups. In *IEEE Real Time Technology and Applications Symposium*, pages 250–259. IEEE Computer Society, 1996.
2. A. Al-Nayeem, M. Sun, X. Qiu, L. Sha, S. P. Miller, and D. D. Cofer. A formal architecture pattern for real-time distributed systems. In *Proc. IEEE Real Time Systems Symposium*. IEEE, 2009. To appear.
3. B. Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.
4. M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.
5. B. Chen, S. Kamat, and W. Zhao. Fault-tolerant real-time communication in fddi-based networks. In *IEEE Real-Time Systems Symposium*, pages 141–151, 1995.
6. C.-T. Chou, I. Cidon, I.S. Gopal, and S. Zaks. Synchronizing asinchronous bounded delay networks. *IEEE Trans. Commun.*, 38(2):144–147, 1990.
7. E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
9. J. Berwanger et al. FlexRay–the communication system for advanced automotive control systems. In *SAE 2001 World Congress*, pages 2001–01–0676. Society of Automotive Engineers, 2001.

10. H. Garavel and D. Thivolle. Verification of gals systems by combining synchronous languages and process calculi. In *16th International SPIN Workshop*, volume 5578 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2009.
11. A. Girault and C. Ménier. Automatic production of globally asynchronous locally synchronous systems. In *Embedded Software, Second International Conference, EMSOFT*, volume 2491 of *Lecture Notes in Computer Science*, pages 266–281. Springer, 2002.
12. K. Hoyme and K. Driscoll. SAFEbus^{TM}. In *11th AIAA/IEEE Digital Avionics Systems Conference*, pages 68–73, 1992.
13. M. Katelman and J. Meseguer. Using the PALS architecture to verify a distributed topology control protocol for wireless multi-hop networks in the presence of node failures. To appear in Proc. RTRTS 2010.
14. H. Kopetz and G. Grünsteidl. Ttp - a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, 1994.
15. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
16. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
17. P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, 2001.
18. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
19. J. Meseguer, M. Palomino, and N. Martí-Oliet. Algebraic simulations. *J. Log. Algebr. Program.*, 79(2):103–143, 2010.
20. S.P. Miller, D. Cofer, L. Sha, J. Meseguer, and A. Al-Nayeem. Implementing logical synchrony in integrated modular avionics. In *Proc. 28th Digital Avionics Systems Conference*. IEEE, 2009.
21. P.S. Miner. Analysis of the SPIDER fault-tolerant protocols. In *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, 2000.
22. P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):5–27, 2007.
23. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.
24. L. Pike. A note on inconsistent axioms in Rushby's "Systematic formal verification for fault-tolerant time-triggered algorithms'. *IEEE Trans. Software Eng.*, 32(5):347–348, 2006.
25. L. Pike. Modeling time-triggered protocols and verifying their real-time schedules. In *FMCAD*, pages 231–238. IEEE Computer Society, 2007.
26. D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. *Fundam. Inform.*, 78(1):131–159, 2007.
27. J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Trans. Software Eng.*, 25(5):651–660, 1999.
28. J. M. Rushby. Bus architectures for safety-critical embedded systems. In *Embedded Software, First International Workshop, EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2001.
29. J. M. Rushby. An overview of formal verification for the time-triggered architecture. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002*, volume 2469 of *Lecture Notes in Computer Science*, pages 83–106. Springer, 2002.
30. SEA. Architecture Analysis & Design Language (AADL). AS5506, Version 1.0, SAE Aerospace, November 2004.

31. L. Sha, A. Al-Nayeem, M. Sun, J. Meseguer, and P. C. Ölveczky. PALS: Physically asynchronous logically synchronous systems. Technical report, University of Illinois at Urbana-Champaign, 2009. Available at `http://hdl.handle.net/2142/11897`.
32. G. Tel. *Introduction to Distributed Algorithms.* Cambridge U.P., 1994.
33. G. Tel, E. Korach, and S. Zaks. Synchronizing ABD networks. *IEEE Trans. Networking*, 2(1):66–69, 1994.
34. S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. DiNatale. Implementing synchronous models on loosely time triggered architectures. *IEEE Trans. on Computers*, 1, 2008.

A Proofs of Lemmas and Theorems

This appendix presents some lemmas used to prove some key theorems, as well as proofs of those theorems whose proofs have been omitted from the body of the paper.

Lemma 1. *Let a timer in an object have value q at time t_0. Then, the timer expires at some global time in the interval $(q + c(t_0) - \epsilon,\ q + c(t_0) + \epsilon]$ for c the local ϵ-drift clock function of the object.*

Proof. Some helpful lemmas, such as that the effect of two applications of the function `delta` equals one `delta` application for the sum of the time advances, etc., are proved below.

At global time t, the timer has the value q `monus` $(c(t) - c(t_0))$ if everything is done using *one* application of the `delta` function, which we will argue below in Lemma 2 can be assumed. The timer will expire the first time that the above value reaches 0. This happens at the first global time t_1 when $c(t_1) - c(t_0) \geq q$, which is the same as $c(t_1) \geq q + c(t_0)$. This may happen at time t_1 either when

1. $c(t_1) = q + c(t_0)$, or when
2. $c(t_1) > q + c(t_0)$ and there is no $t' < t_1$ such that the timer expires at time t'.

Let us consider case (1) first. By definition of drift functions, t_1 is in the global time interval $(c(t_1) - \epsilon,\ c(t_1) + \epsilon)$, which in case (1) equals the interval $(q + c(t_0) - \epsilon,\ q + c(t_0) + \epsilon)$, which is inside the interval in the theorem.

Let us now consider case (2). Now, $c(t_1) > q + c(t_0)$, and hence we have $c(t_1) - \epsilon > q + c(t_0) - \epsilon$. In addition, due to the assumption of ϵ-drift functions, we have $t_1 > c(t_1) - \epsilon$. Together, these last inequalities give $t_1 > c(t_1) - \epsilon > q + c(t_0) - \epsilon$, which proves the lower bound in the interval in the theorem. Furthermore, t_1 cannot be greater than $q + c(t_0) + \epsilon$, because at time $q + c(t_0) + \epsilon$, the clock must be at least $q + c(t_0)$ and the timer would expire then. This proves the upper bound. □

In the above proof, we reasoned about the expiration of a timer given that the tick rule and the `delta` function are only applied once. We notice that the timer value is only changed by either the tick rule, or when the timer expires. The following shows that we can contract multiple tick steps into one for the sake of reasoning about timers:

Lemma 2. *For any object in state o, we have*

$$\texttt{delta}(\texttt{delta}(o, t_0, \Delta), t_0 + \Delta, \Delta') = \texttt{delta}(o, t_0, \Delta + \Delta')$$

for any time values t_0, Δ, Δ'*.*

Proof. The proof of this lemma follows directly from the equations defining the semantics of the `delta` function in Section 5.5. Mathematically, this just follows from the general fact that those equations imply that `delta` is in essence an *action* of the additive monoid of time $(\mathbb{R}_{\geq 0}, +, 0)$ over the configurations of objects and messages. The details are left to the reader. □

Lemma 3. *The* `roundTimer` *for each object expires somewhere in the global time interval* $(i \cdot T - \epsilon,\ i \cdot T + \epsilon]$ *for all* $i \in \mathbb{N}$*, from any initial state of the form assumed in Section 5.6.*

Proof. In the initial state, the value of `roundTimer` for object j is $T - c_j(T - \epsilon)$, and the local clock is $c_j(T - \epsilon)$. It follows directly from Lemma 1 that the timer expires somewhere in the global time interval $(T-\epsilon,\ T+\epsilon]$. In addition, it follows trivially that the local clock is greater than or equal to T.

Assume that at the time when the `roundTimer` first expires, the *local* clock is $T+\Delta$ for some $2\epsilon > \Delta \ \geq 0$ (where Δ will be strictly greater than 0 only if the timer expired in a "clock jump"). In the rules `applyTrans`, the `roundTimer` is reset to $T-((T+\Delta)-\lfloor\frac{T+\Delta}{T}\rfloor \cdot T)$, which equals $T-\Delta$, since T is greater than 2ϵ according to our constraints. The sum of the local clock and the newly set timer is therefore $2 \cdot T$, and the timer will therefore expire the next time sometime in the global time interval $(2T - \epsilon,\ 2T + \epsilon]$ according to Lemma 1. This reasoning can be replicated for any round i. □

Lemma 4. *The* `outputBackoffTimer` *of each object in states reachable from the initial states of the given form expires somewhere in the global time interval* $((i \cdot T) + (2\epsilon \ \texttt{monus}\ \mu_{min}) - \epsilon,\ (i \cdot T) + (2\epsilon \ \texttt{monus}\ \mu_{min}) + \epsilon]$ *for each* $i \in \mathbb{N}$ *with* $i \geq 1$*.*

Proof. In the initial state, the `outputBackoffTimer`s are turned off. This timer is set in the rule `applyTrans` and, for the environment, rule `consumeInputAndGenerateOutput`, when the `roundTimer` expires. As shown in the proof of Lemma 3, the local clock is $T + \Delta$ the first time this happens (for Δ as described in the above proof of Lemma 3). In these rules, the `outputBackoffTimer` is set to $(2\epsilon \ \texttt{monus}\ \mu_{min}) \ \texttt{monus}\ \Delta$. We need to consider three cases: (1) $2\epsilon \leq \mu_{min}$, (2) $2\epsilon > \mu_{min}$ and $(2\epsilon - \mu_{min}) < \Delta$, and (3) $2\epsilon > \mu_{min}$ and $(2\epsilon - \mu_{min}) \geq \Delta$. In case (1), the `outputBackoffTimer` expires when it is set, which according to Lemma 3 takes place in the global time interval $(i{\cdot}T-\epsilon, i{\cdot}T+\epsilon]$, which equals the time interval in Lemma 4 when $2\epsilon \leq \mu_{min}$. In case (2), the `outputBackoffTimer` is also set to 0, but the desired time interval in Lemma 4 is now $((i \cdot T) + (2\epsilon - \mu_{min}) - \epsilon,\ (i \cdot T) + (2\epsilon - \mu_{min}) + \epsilon]$. According to Lemma 1, the timer can expire no earlier than at time $T+\Delta-\epsilon$, which is later than or equal to the lower bound $T + (2\epsilon - \mu_{min}) - \epsilon$ in Lemma 4, since case (2) assumes $(2\epsilon - \mu_{min}) < \Delta$. The

upper bound $T+(2\epsilon-\mu_{min})+\epsilon$ follows from Lemma 3, since the timer expires at latest at time $T+\epsilon$, and we have assumed that in case (2) that $(2\epsilon-\mu_{min})>0$. Finally, for case (3), the *local* clock of the object under consideration is $T+\Delta$, and the `outputBackoffTimer` is set to $(2\epsilon-\mu_{min})-\Delta \geq 0$. It then follows from Lemma 1 that this local `outputBackoffTimer` expires for the first time somewhere in the global time interval $(T+(2\epsilon-\mu_{min})-\epsilon,\ T+(2\epsilon-\mu_{min})+\epsilon]$. Again, this reasoning can be replicated for any round i. □

Remark. The above lemmas should be read as safety and not as liveness guarantees. That is, if time advances at all that far, then the timers expire in the given intervals.

Lemma 5. *Messages are sent from the output buffers in the global time interval* $(iT-\epsilon+\max(2\epsilon-\mu_{min},\alpha_{min}),\ iT+\epsilon+\max(2\epsilon-\mu_{min},\alpha_{max})]$ *during round each round* $i \geq 1$.

Proof. Messages are only generated into the output buffers when the `roundTimer`s expire. Then, the `outputBackoffTimer`s are set, and expire within the time intervals given in Lemma 4. At the same time (that is, when the `roundTimer` expires), the execution delay is set to somewhere between α_{min} and α_{max}. Messages are only sent when the output backoff timer has expired and when the execution delay has elapsed. Furthermore, we see that the backoff timer is only turned off when it has expired (that is, when it has value 0). From Lemma 4, the backoff timer expires *strictly later* than global time $iT-\epsilon+(2\epsilon\ \texttt{monus}\ \mu_{min})$; furthermore, since the `roundTimer` expires strictly later than global time $iT-\epsilon$, the execution delay ends strictly later than at global time $iT-\epsilon+\alpha_{min}$, together giving the lower bound in the lemma, since $\max(2\epsilon\ \texttt{monus}\ \mu_{min},\alpha_{max}) = \max(2\epsilon-\mu_{min},\alpha_{max})$ since $\alpha_{max}\geq 0$.

As for the upper bound, the `roundTimer` expires at latest at time $iT+\epsilon$, and hence the messages are ready to be sent at latest at global time $iT+\epsilon+\alpha_{max}$. Likewise, the latest time the backoff expires is at time $iT+\epsilon+(2\epsilon\ \texttt{monus}\ \mu_{min})$, together giving the upper bound. □

Lemma 6. *The messages sent into the configuration in round i will be received at times within the global time interval* $(i\cdot T+\epsilon,\ (i+1)\cdot T-\epsilon]$.

Proof. As seen in Lemma 5, each message is sent out in round i somewhere in the global time interval $(iT-\epsilon+\max(2\epsilon-\mu_{min}),\ iT+\epsilon+\max(2\epsilon-\mu_{min},\alpha_{max})]$, and is given a delay between μ_{min} and μ_{max}. We see that the remaining delay of a message decreases by the same amount that global time advances, and that $\texttt{mte}(m)=0$ for an undelayed message (which by the identity attribute of the message delay operator is the same as a message with delay 0) implies that the message must be received when its delay reaches 0 "for the first time." Therefore, each of these messages is created in the interval $(iT-\epsilon+\max(2\epsilon-\mu_{min},\alpha_{min}),\ iT+\epsilon+\max(2\epsilon-\mu_{min},\alpha_{max})]$ and is received at some time in the global interval $(iT-\epsilon+\max(2\epsilon-\mu_{min},\alpha_{min})+\mu_{min},\ iT+\epsilon+\max(2\epsilon-\mu_{min},\alpha_{max})+\mu_{max}]$.

To prove the lemma, we must therefore show

1. $(i \cdot T) + \epsilon \leq iT - \epsilon + \max(2\epsilon - \mu_{min}, \alpha_{min}) + \mu_{min}$ and
2. $iT + \epsilon + \max(2\epsilon - \mu_{min}, \alpha_{max}) + \mu_{max} \leq ((i+1) \cdot T) - \epsilon$.

Requirement (1) follows by arithmetic. The upper time limit requirement (2) reduces to proving $\epsilon + \max(2\epsilon - \mu_{min}, \alpha_{max}) + \mu_{max} \leq T - \epsilon$, which follows from the global requirement that $T \geq \mu_{max} + 2\epsilon + \max(2\epsilon - \mu_{min}, \alpha_{max})$. □

These theorems, together with a trivial inspection of the rules and the well-known timed behavior, together prove the time line described in Section 4.3:

Lemma 7. *For all time diverging paths from the given initial states in $\mathcal{A}(\mathcal{E})$, the behaviors have the following time line for all $i \in \mathbb{N}$:*

- *at times in the global time interval $(iT - \epsilon,\ iT + \epsilon]$ the* `roundTimer`*s expire, all input buffers are read, and transitions corresponding to transitions in the synchronous system are applied. The resulting states and output messages (stored in the output buffers) are undelayed no later than at global time $iT + \epsilon + \alpha$, and the backoff timer expires at latest at time $iT + \epsilon + (2\epsilon$* `monus` *$\mu_{min})$, ensuring that all these messages are sent to the global configuration in the global time interval $(iT - \epsilon + \max(2\epsilon - \mu_{min}, \alpha_{min}),\ iT + \epsilon + \max(2\epsilon - \mu_{min}, \alpha_{max})]$; furthermore, also the messages generated nondeterministically by the environment are sent into the global configuration in this interval;*
- *these messages are received at times within the global time interval $(i \cdot T + \epsilon,\ (i+1) \cdot T - \epsilon]$, ensuring that all messages are received and stored in the respective input buffers before or at global time $(i+1) \cdot T - \epsilon$;*
- *a new round therefore begins at times in the global time interval $((i+1) \cdot T - \epsilon,\ (i+1) \cdot T + \epsilon]$: the* `roundTimer`*s expire, all input buffers are read, and the transitions corresponding to those in the synchronous system are applied, and so on.*

The above lemma defines the behaviors from the given initial states, and are crucial in the proof of the following:

Lemma 8. *Let $\{C\,;\,t\} \longrightarrow_{st} \{C'\,;\,t'\}$ be a transition in $Stable(\mathcal{A}(\mathcal{E}))$ for a machine ensemble $\mathcal{E}$. Then there is a transition $(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{\mathcal{E}} (\boldsymbol{s'}, \boldsymbol{i'})$ such that $sync(\{C\,;\,t\}) = (\boldsymbol{s}, \boldsymbol{i})$ and $sync(\{C'\,;\,t'\}) = (\boldsymbol{s'}, \boldsymbol{i'})$.*

Proof. (Sketch) Since $\{C;t\}$ is the source of a stable transition, it is reachable from an initial state in $\mathcal{A}(\mathcal{E})$; therefore, its local clocks and timers have appropriate values. Furthermore, since C is a stable configuration, all its input buffers are full, all the output buffers are empty, and there are no (delayed or undelayed) messages in C outside of these buffers. In addition, the backoff timers are turned off. It is easy to see by inspecting the rules that only the tick rule, the rule `applyTrans`, or the rule `consumeInputAndGenerateOutput` can be applied. Repeated applications of the tick rule leave us in stable states, until eventually the `roundTimer`s start expiring. At these times, the rules `applyTrans` and `consumeInputAndGenerateOutput` generate new messages and delayed states,

and by the above time line, all these messages will reach the input buffers before the `roundTimers` expire, hence we have reached a new stable state C' when all input buffers are full, but before transitions for the "next" round are applied. The fact that the transitions are total relations ensure that the `applyTrans` rules can always be applied when the `roundTimers` expire and the input buffers are full; likewise, since the environment constraint is assumed to be satisfiable, the rule `consumeInputAndGenerateOutput` is enabled when the timer expires.

The above reasoning shows that from a stable state $\{C\,;\,t\}$ we can always reach another stable state $\{C'\,;\,t'\}$ by a transition $\{C\,;\,t\} \longrightarrow_{st} \{C'\,;\,t'\}$.

Assume that $sync(\{C;t\}) = ((\boldsymbol{s}, (\boldsymbol{d_1}, \ldots, \boldsymbol{d_{|J|}})), \boldsymbol{e})$. It is easy to see, and is argued above, that in the rewrite path from a stable state to the next, each object must have executed rule `applyTrans` exactly once, and that the environment object has executed the rule `consumeInputAndGenerateOutput` exactly once. Furthermore, when `applyTrans` is applied, the messages in the object's input buffer are the same as in C.

Consider an application of the rule `applyTrans` for object l. Now, $in_l(\boldsymbol{e}, (\boldsymbol{d_1}, \ldots, \boldsymbol{d_{|J|}}))$ equals $vec_l(B_l)$ for the input buffer B_l of object l in C. Let s'_l be the new state and let $\boldsymbol{d''_l}$ be the generated output in `applyTrans`. That is,

$$((in_l(\boldsymbol{e}, (\boldsymbol{d_1}, \ldots, \boldsymbol{d_{|J|}})), s_l), (s'_l, \boldsymbol{d''_l})) \in \delta_{M_l}.$$

Likewise, let the environment generate the new messages $\boldsymbol{e'}$. In the rewrite path to the stable state C', these messages arrive at their respective input buffers. It is then follows from the definition of $sync$ that $sync(\{C';t'\}) = ((\boldsymbol{s'}, (\boldsymbol{d'_1}, \ldots, \boldsymbol{d'_{|J|}})), \boldsymbol{e'})$, where $\boldsymbol{d'_i} = fout_i(\boldsymbol{d''_i})$.

Now, we must show that each such stable transition $\{C\,;\,t\} \longrightarrow_{st} \{C'\,;\,t'\}$ corresponds to a transition $sync(\{C\,;\,t\}) \longrightarrow_{\mathcal{E}} sync(\{C'\,;\,t'\})$ in $\mathcal{E}$. Then, we must prove that

$$((\boldsymbol{s}, (\boldsymbol{d_1}, \ldots, \boldsymbol{d_{|J|}})), \boldsymbol{e}) \longrightarrow_{\mathcal{E}} ((\boldsymbol{s'}, (\boldsymbol{d'_1}, \ldots, \boldsymbol{d'_{|J|}})), \boldsymbol{e'}).$$

That is,

1. there exists an output $\boldsymbol{o}$ to the environment such that $((\boldsymbol{e}, (\boldsymbol{s}, (\boldsymbol{d^1}, \ldots, \boldsymbol{d^{|J|}}))), ((\boldsymbol{s'}, (\boldsymbol{d'_1}, \ldots, \boldsymbol{d'_{|J|}})), \boldsymbol{o})) \in \delta_{\mathcal{E}}$, and
2. $c_e(\boldsymbol{e'})$.

The second requirement is immediate, as the rule `consumeInputAndGenerateOutput` only generates input from the environment that satisfies c_e.

The first requirement follows directly from the definition of $\delta_{\mathcal{E}}$, where $\boldsymbol{o}$ is given in the obvious way by $o_i = (d''_l)_k$ if $src(e,i) = (l,k)$. □

Lemma 9. *Let* $((\boldsymbol{s}, (\boldsymbol{d_1}, \ldots, \boldsymbol{d_{|J|}})),\ \boldsymbol{e}) \longrightarrow_{\mathcal{E}} ((\boldsymbol{s'}, (\boldsymbol{d'_1}, \ldots, \boldsymbol{d'_{|J|}})),\ \boldsymbol{e'})$ *be a transition in* $M_{\mathcal{E}}$ *for a machine ensemble* $\mathcal{E}$*. Then, for all reachable* $\{C\,;\,t\}$ *such that* $sync(\{C\,;\,t\}) = ((\boldsymbol{s}, (\boldsymbol{d_1}, \ldots, \boldsymbol{d_{|J|}})),\ \boldsymbol{e})$*, there exists a* $\{C'\,;\,t'\}$ *such that* $sync(\{C'\,;\,t'\}) = ((\boldsymbol{s'}, (\boldsymbol{d'_1}, \ldots, \boldsymbol{d'_{|J|}})),\ \boldsymbol{e'})$ *and* $\{C\,;\,t\} \longrightarrow_{st} \{C'\,;\,t'\}$*.*

Proof. (Sketch) The new environment output $\boldsymbol{e'}$ can be generated by the environment object as long as it as valid output. For any stable state C whose feedback wire states and internal states define $(\boldsymbol{s}, (\boldsymbol{d_1}, \ldots, \boldsymbol{d_{|J|}}))$, the time-line reasoning then implies that these messages will all be read in the correct time interval, and the next feedback states will be generated. The remaining details are left to the reader. □

Finally, we below restate Theorems 1, 2, and 6, and present their respective proofs.

Theorem 1. *Let $\{C_0; t_0\}$ be an initial stable state in $\mathcal{A}(\mathcal{E})$. Then, any finite rewrite sequence*

$$\{C_0; t_0\} \longrightarrow \{C_1; t_1\} \longrightarrow \{C_2; t_2\} \longrightarrow \cdots \longrightarrow \{C_n; t_n\}$$

in $\mathcal{A}(\mathcal{E})$ can be extended into a time-diverging path in $TDPaths(\mathcal{A}(\mathcal{E}))_{\{C_0;t_0\}}$.

Proof. The tick rules applies to all configurations and can advance time as much as allowed by the function `mte`. The function `mte` is defined to be the smallest time until some timer expires or a message becomes ripe. Lemmas 3 and 4 state that the timers always expire somewhere in the respective global time intervals $(i \cdot T - \epsilon, i \cdot T + \epsilon]$ and $((i \cdot T) + (2\epsilon \texttt{ monus } \mu_{min}) - \epsilon, (i \cdot T) + (2\epsilon \texttt{ monus } \mu_{min}) + \epsilon]$ for each $i \in \mathbb{N}$ with $i \geq 1$. Since $T > 0$, timers cannot force Zeno behaviors if they are reset whenever they expire.

Consider the expiration of a `roundTimer`. The appropriate rule `applyTrans` applies to such an object if (i) its local state is not delayed, (ii) the input buffer is full, and (iii) there is a transition in the corresponding synchronous machine from the current state and input. Condition (iii) follows from (i), (ii), and the fact that the transition relation for each synchronous machine is a total relation. Condition (i) is also fairly trivial, since the delay on the state component is always set to a value less than or equal to α_{max}, and is only set in the `applyTrans` rules. By the equation for `delta`, the state delay is reduced according to the elapsed time, and when the delay is `0`, then we get an undelayed state, since the state delay operator is declared to have right identity `0`. Since the `roundTimer` expires at times $(i \cdot T - \epsilon, i \cdot T + \epsilon]$, then the state component will be "undelayed" at latest at time $i \cdot T + \epsilon + \alpha_{max}$, which is before the next time interval $((i+1) \cdot T - \epsilon, (i+1) \cdot T + \epsilon]$ (because $T \geq 2\epsilon + \alpha_{max}$) when the `applyTrans` rule can be applied in the next round. We must finally show (ii), that the input buffers are full whenever the timers expire. This can be proved by mutual induction, also taking the application of the sending rules into account. More informally, in the initial state, the input buffers are full, and messages do not disappear from the input buffer, except when rule `applyTrans` is applied. Then all the output messages are generated, and, as argued below, will be sent out into the configuration in due time. By Lemma 7, messages generated in round i will be received after the `roundTimer` for the round has expired, and are hence saved until the `roundTimer` expires for the next round, at which time the input buffers are therefore full.

The rules `applyTrans` do not apply to environment objects; instead, the rule `consumeInputAndGenerateOutput` is always applicable when the environment object's `roundTimer` expires.

Now, consider another possible source for deadlock: the "timer" on the *outgoing* messages. This causes no problems, since when it becomes zero, then this timer is removed in rule `transitionFinished` if the backoff timer is still turned on, and in (multiple applications of) rule `outputMsg2` when the backoff timer is turned off.

A third possible source for a time block is the output backoff timer. Again, such a timer does not cause any problems, since it is turned off in (repeated applications of) rule `outputMsg1` if the output messages have been generated, and in rule `turnOffOutTimer` if the output is still delayed.

Finally, time advance is blocked when a ripe message is in the outermost level of the configuration; that is, traveling between two nodes. As already explained, this causes no problems, since the rule `receiveMsg` is always applicable when there is a message in the system. □

Theorem 2. *Let $\mathcal{E}$ be a synchronous machine ensemble, and let $\{C_i; t_i\}$ be a* stable *state reachable from an initial state according to the definition of initial states in Section 5.6. Then, any time-diverging path*

$$\pi : \quad \{C_i; t_i\} \longrightarrow \{C_{i+1}; t_{i+1}\} \longrightarrow \{C_{i+2}; t_{i+2}\} \longrightarrow \cdots$$

in $TDPaths(\mathcal{A}(\mathcal{E}))_{\{C_i;t_i\}}$ can be composed into an infinite sequence

$$\{C_i; t_i\} \longrightarrow_{st} \{C_{i+k_1}; t_{i+k_1}\} \longrightarrow_{st} \{C_{i+k_2}; t_{i+k_2}\} \longrightarrow_{st} \cdots$$

of stable transitions.

That is, there is a strictly monotonic function $\gamma_\pi : \mathbb{N} \to \mathbb{N}$ with $\gamma_\pi(0) = 0$ such that for each $j \geq 0$, the rewrite sequence $\pi(\gamma_\pi(j)) \longrightarrow \pi(\gamma_\pi(j) + 1) \longrightarrow \cdots \longrightarrow \pi(\gamma_\pi(j + 1))$ corresponds to a stable transition $\pi(\gamma_\pi(j)) \longrightarrow_{st} \pi(\gamma_\pi(j + 1))$.

Proof. The theorem follows directly from Lemma 7, since "all messages are received and stored in the respective input buffer," in addition to the fact that messages were sent from the output buffer earlier in the period (the first item in Lemma 7), characterizes the stable states. The function γ_π is then defined as follows:

- $\gamma_\pi(0) = 0$,
- $\gamma_\pi(1) = k$, for the k in the path π of the above form such that $\{C_i; t_i\} \longrightarrow_{st} \{C_{i+k}; t_{i+k}\}$ is a stable transition, and
- for all $j \geq 1$, $\gamma_\pi(j + 1) = \gamma_\pi(1) + \gamma_{\pi^{\gamma_\pi(1)}}(j)$.

Theorem 6. *Given a formula $\varphi \in CTL^*(AP)$, and assuming that a new state predicate stable $\notin AP$ characterizing stable states has been defined, then there*

is a formula $\varphi_{stable} \in CTL^* \setminus \{\bigcirc\}(AP \cup \{stable\})$ *(qualifying* φ *such that it is restricted to stable states) defined recursively as follows:*

$$\begin{aligned} a_{stable} &= a, \text{ for } a \in AP \\ (\neg\,\varphi)_{stable} &= \neg\,(\varphi_{stable}) \\ (\varphi_1 \wedge \varphi_2)_{stable} &= \varphi_{1_{stable}} \wedge \varphi_{2_{stable}} \\ (\varphi_1\, U\, \varphi_2)_{stable} &= (stable \rightarrow \varphi_{1_{stable}})\; U\; (stable \wedge \varphi_{2_{stable}}) \\ (\bigcirc\,\varphi)_{stable} &= stable\, U\, (\neg stable \wedge (\neg stable\, U\, (stable \wedge \varphi_{stable}))) \\ (\forall\,\varphi)_{stable} &= \forall\,\varphi_{stable} \end{aligned}$$

such that for each stable state s *in* $\mathcal{A}(\mathcal{E})$ *reachable from initial states defined in Section 5.6, we have*

$$(\mathcal{A}(\mathcal{E}), L'), s \models \varphi_{stable} \quad \Longleftrightarrow \quad (\mathcal{E}_{c_e}, L), sync(s) \models \varphi,$$

where $CTL^* \setminus \{\bigcirc\}(AP \cup \{stable\})$ *formulas are interpreted in* $(\mathcal{A}(\mathcal{E}), L')$ *under the* time-diverging path semantics*, and where* $L' : \mathcal{T}_{\mathcal{A}(\mathcal{E})_{\texttt{GlobalSystem}}} \rightarrow \mathcal{P}(AP \cup \{stable\})$ *is a labeling function satisfying* $L'(s) = L(sync(s)) \cup \{stable\}$ *when* s *is a stable state, and* $stable \notin L'(s)$ *otherwise.*

Proof. We prove simultaneously that for all reachable *stable* states $s \in \mathcal{T}_{\mathcal{A}(\mathcal{E})_{\texttt{GlobalSystem}}}$ and for all *time-diverging* paths $\pi \in TDPaths(\mathcal{A}(\mathcal{E}))_s$,

$$(\mathcal{A}(\mathcal{E}), L'), s \models \varphi_{stable} \quad \Longleftrightarrow \quad (\mathcal{E}_{c_e}, L), sync(s) \models \varphi$$

holds for all *state formulas* φ and

$$(\mathcal{A}(\mathcal{E}), L'), \pi \models \varphi_{stable} \quad \Longleftrightarrow \quad (\mathcal{E}_{c_e}, L), sync(\pi) \models \varphi$$

holds for all *path formulas* φ, by induction on the structure of φ, where the projection functions $sync : TDPaths(\mathcal{A}(\mathcal{E}))_s \rightarrow Paths(\mathcal{E}_{c_e})_{sync(s)}$ relating infinite paths in $\mathcal{A}(\mathcal{E})$ and $\mathcal{E}_{c_e}$ are defined in the obvious way, by $sync(\pi)(i) = sync(\pi(\gamma_\pi(i)))$ for γ_π the function in Theorem 2. The fact that we only consider time-diverging paths (remember from Theorem 1 that any finite computation in $\mathcal{A}(\mathcal{E})$, starting in a suitable initial state, can be extended into an infinite time-diverging path) ensures that an infinite path in $TDPaths(\mathcal{A}(\mathcal{E}))_s$ indeed maps to an infinite path in $Paths(\mathcal{E}_{c_e})_{sync(s)}$.

Notation: In this proof, we write $s \models \phi$ for $(\mathcal{A}(\mathcal{E}), L'), s \models \phi$ and $sync(s) \models \varphi$ for $(\mathcal{E}_{c_e}, L), sync(s) \models \varphi$ when the context is obvious. Likewise, we write $\pi \models \phi$ for $(\mathcal{A}(\mathcal{E}), L'), \pi \models \phi$ and $sync(\pi) \models \varphi$ for $(\mathcal{E}_{c_e}, L), sync(\pi) \models \varphi$.

We first prove the equivalence for all stable states:

- $\varphi = a$ for $a \in AP$: We must prove $s \models a \Longleftrightarrow sync(s) \models a$, which is immediate, since s is stable, and therefore $a \in L'(s) \Longleftrightarrow a \in L(sync(s))$.
- $\varphi = \neg\varphi'$: Must prove $s \models \neg\varphi' \Longleftrightarrow sync(s) \models \neg\varphi'$. The induction hypothesis gives $s \models \varphi'_{stable} \Longleftrightarrow sync(s) \models \varphi'$, from which the desired conclusion follows since

$$s \models \neg\varphi'_{stable} \Longleftrightarrow s \not\models \varphi'_{stable} \overset{I.H.}{\Longleftrightarrow} sync(s) \not\models \varphi' \Longleftrightarrow sync(s) \models \neg\varphi'.$$

- $\varphi = \varphi_1 \wedge \varphi_2$: Follows directly from the induction hypotheses (for both φ_1 and φ_2) and the definition of the satisfaction relation.
- $\varphi = \forall \varphi'$: We must prove that $s \models \forall \varphi'_{stable} \iff sync(s) \models \forall \varphi'$. That is, we must prove that $\pi \models \varphi'_{stable}$ holds for all $\pi \in TDPaths(\mathcal{A}(\mathcal{E}))_s$ if and only if $\rho \models \varphi$ for all paths $\rho \in Paths(\mathcal{E}_{c_e})_{sync(s)}$. From the induction hypothesis, we can assume that, for all $\pi \in TDPaths(\mathcal{A}(\mathcal{E}))_s$, $\pi \models \varphi'_{stable}$ if and only if $sync(\pi) \models \varphi'$. The desired conclusion follows if *sync* is a *surjective* function from $TDPaths(\mathcal{A}(\mathcal{E}))_s$ to $Paths(\mathcal{E}_{c_e})_{sync(s)}$, since then all paths in $Paths(\mathcal{E}_{c_e})_{sync(s)}$ are of the form $sync(\pi)$ for some $\pi \in TDPaths(\mathcal{A}(\mathcal{E}))_s$. That *sync* is indeed a surjective function from $TDPaths(\mathcal{A}(\mathcal{E}))_s$ to $Paths(\mathcal{E}_{c_e})_{sync(s)}$ for a reachable stable states s follows from Theorem 3, and from the fact that each stable transition is a sequence of rewrite steps.

We now prove the equivalence for all paths from stable states:

- $\varphi = \neg\varphi'$: We must prove $\pi \models \neg\varphi'_{stable} \iff sync(\pi) \models \neg\varphi'$, given the induction hypothesis $\pi \models \varphi'_{stable} \iff sync(\pi) \models \varphi'$. The desired conclusion follows trivially:

$$\pi \models \neg\varphi'_{stable} \iff \pi \not\models \varphi'_{stable} \overset{I.H.}{\iff} sync(\pi) \not\models \varphi' \iff sync(\pi) \models \neg\varphi'.$$

- $\varphi = \varphi' \wedge \varphi''$: Equally straight-forward.
- $\varphi = \varphi' \; U \; \varphi''$: We must prove that $\pi \models (stable \rightarrow \varphi'_{stable}) \; U \; (stable \wedge \varphi''_{stable})$ if and only if $sync(\pi) \models \varphi' \; U \; \varphi''$. As induction hypotheses we can assume $\pi' \models \varphi'_{stable}$ if and only if $sync(\pi') \models \varphi'$ (and likewise for φ'') for all paths π' starting in a reachable stable state.

  An important consequence of the induction hypotheses is that if $\pi(k)$ and $\pi(k')$ are neighboring stable states in π; that is, if there is no unstable state between $\pi(k')$ and $\pi(k)$ in π, then $\pi^k \models \varphi'_{stable} \iff \pi^{k'} \models \varphi'_{stable}$ (and the same for φ''_{stable}). This is because the paths $sync(\pi^k)$ and $sync(\pi^{k'})$ are the same (this is immediate), and both start in reachable stable states $\pi(k)$ and $\pi(k')$. We therefore have

$$\pi^{k'} \models \varphi'_{stable} \overset{I.H.}{\iff} sync(\pi^{k'}) \models \varphi' \iff sync(\pi^k) \models \varphi' \overset{I.H.}{\iff} \pi^k \models \varphi'_{stable}.$$

  Assume that $\pi \models (stable \rightarrow \varphi'_{stable}) \; U \; (stable \wedge \varphi''_{stable})$ holds. Then there is a *smallest* k such that $\pi^k \models (stable \wedge \varphi''_{stable})$ holds, and for $i < k$, $\pi^i \models (stable \rightarrow \varphi'_{stable})$. Since *stable* only holds for stable states, the assumption that k is smallest, together with the above property for neighboring stable states, mean that either $k = 0$, or $\pi(k-1)$ is unstable. Furthermore, there is a j such that $\gamma_\pi(j) = k$, and hence $sync(\pi^k) = (sync(\pi))^j$. Since $\pi(k)$ is a stable state, π^k is a path starting in a reachable stable state, and it therefore follows from the induction hypothesis that $\pi^k \models \varphi''_{stable}$ if and only if $(sync(\pi))^j \models \varphi''$. Hence $(sync(\pi))^j$ satisfies φ''. Furthermore, since we assume that $\pi \models (stable \rightarrow \varphi'_{stable}) \; U \; (stable \wedge \varphi''_{stable})$, we have $\pi^i \models (stable \rightarrow \varphi'_{stable})$ for each $0 \leq i < k$. In particular, let $0 = i_0 < \cdots < i_m < k$ be the indices such that $\gamma_\pi(i_l) = l$. We then have $\pi^{\gamma_\pi(i_l)} \models (stable \rightarrow$

φ'_{stable}) for each i_l. All these $\pi(\gamma_\pi(i_l))$ states are stable states, and hence we have $\pi^{\gamma_\pi(i_l)} \models \varphi'_{stable}$ for each i_l. By the induction hypothesis, it follows that $sync(\pi^{i_l}) \models \varphi'$ for each of these i_l. Since by definition $(sync(\pi))^l = sync(\pi^{\gamma_\pi(l)})$, we have $(sync(\pi))^l \models \varphi'$ for $0 \leq l < j$, and therefore, we have that $sync(\pi) \models \varphi' \, U \, \varphi''$.

Conversely, let us prove that $sync(\pi) \models \varphi' \, U \, \varphi''$ implies $\pi \models (stable \rightarrow \varphi'_{stable}) \, U \, (stable \wedge \varphi''_{stable})$. Therefore, there is a k such that $(sync(\pi))^k$ satisfies φ'', and for all $j < k$, $(sync(\pi))^j$ satisfies φ'. Then, $\pi(\gamma_\pi(k))$ is stable and satisfies φ''_{stable} by the induction hypothesis; therefore $\pi^{\gamma_\pi(k)} \models stable \wedge \varphi''_{stable}$. Furthermore, all $\pi^0, \pi^{\gamma_\pi(1)}, \ldots, \pi^{\gamma_\pi(k-1)}$ satisfy φ'_{stable} by the induction hypothesis, and therefore the implication $stable \rightarrow \varphi'_{stable}$. For all the other paths π^l for $l < \gamma_\pi(k)$; if they start in an unstable state, $(stable \rightarrow \varphi'_{stable})$ obviously holds; otherwise, if they start in a "non-γ_π" stable state, it follows from the above facts about paths starting in neighboring stable states that they satisfy the same property as the other stable states surrounding them, including the "corresponding" γ_π-state. Therefore, all the "non-γ_π"-starting paths $\pi^{k'}$, for $k' < \gamma_\pi(k)$, starting from a stable state satisfy φ'_{stable}. Hence, we have that $\pi \models (stable \rightarrow \varphi'_{stable}) \, U \, (stable \wedge \varphi''_{stable})$.

– $\varphi = \bigcirc \varphi'$: We must prove that $\pi \models stable \, U \, (\neg stable \wedge (\neg stable \, U \, (stable \wedge \varphi'_{stable})))$ if and only if $sync(\pi) \models \bigcirc \varphi'$, assuming the induction hypotheses that $\pi' \models \varphi'_{stable}$ if and only if $sync(\pi') \models \varphi'$ for all paths π' starting in a reachable stable state.

We first prove the property

$$sync(\pi^{\gamma_\pi(1)}) = (sync(\pi))^1$$

by proving that for all $i \geq 0$,

$$sync(\pi^{\gamma_\pi(1)})(i) = (sync(\pi))^1(i).$$

The left-hand side $sync(\pi^{\gamma_\pi(1)})(i)$ equals $sync(\pi^{\gamma_\pi(1)}(\gamma_{\pi^{\gamma_\pi(1)}}(i)))$ (by the definition of $sync$ on paths), which equals $sync(\pi(\gamma_\pi(1) + \gamma_{\pi^{\gamma_\pi(1)}}(i)))$ by the property $\pi^j(k) = \pi(j + k)$ of paths, which again equals $sync(\pi(\gamma_\pi(i + 1)))$ by the definition of γ_π in the proof of Theorem 2. The right-hand side $(sync(\pi))^1(i)$ equals $sync(\pi)(i + 1)$ by properties of paths, which equals the desired $sync(\pi(\gamma_\pi(i + 1)))$ by the definition of $sync$.

For the main property, that $\pi \models stable \, U \, (\neg stable \wedge (\neg stable \, U \, (stable \wedge \varphi'_{stable})))$ if and only if $sync(\pi) \models \bigcirc \varphi'$, consider the path π. Since it starts in a stable state, *stable* holds initially. By Theorem 2 and the definition of stable transitions, $\pi(0)$ is followed by zero or more stable states, which are then followed by one or more *unstable* states (where $\neg stable$ holds), which are again followed by the stable state $\pi(\gamma_\pi(1))$. Therefore, $\pi \models stable U (\neg stable \wedge (\neg stable \, U \, (stable \wedge \varphi'_{stable})))$ if and only if $\pi^{\gamma_\pi(1)} \models \varphi'_{stable}$. By the induction hypotheses, since $\pi(\gamma_\pi(1))$ is a reachable *stable* state, $\pi^{\gamma_\pi(1)} \models \varphi'_{stable}$ if and only if $sync(\pi^{\gamma_\pi(1)}) \models \varphi'$. We have proved above that $sync(\pi^{\gamma_\pi(1)}) = (sync(\pi))^1$. Therefore, $\pi \models stable \, U \, (\neg stable \wedge (\neg stable \, U \, (stable \wedge \varphi'_{stable})))$ if and only if $sync(\pi^{\gamma_\pi(1)}) \models \varphi'$ if and only if $(sync(\pi))^1 \models \varphi'$, which by the definition of the next operator holds if and only if $sync(\pi) \models \bigcirc \varphi'$.

– $\varphi = \forall\, \varphi'$: We must show that $\pi \models \forall\, \varphi'_{stable} \iff sync(\pi) \models \forall\, \varphi'$. By the definition of the satisfaction relation for CTL^*, this amounts to proving $\pi(0) \models \forall \varphi'_{stable} \iff sync(\pi)(0) \models \forall \varphi'$. Since $sync(\pi)(0) = sync(\pi(\gamma_\pi(0)))$, and $\gamma_\pi(0)$ always equals 0, this amounts to proving $\pi(0) \models \forall\, \varphi'_{stable} \iff sync(\pi(0)) \models \forall\, \varphi'$, which was already done in the above inductive proof of state formulas, since $\pi(0)$ is a reachable stable state. □

B The Formal Models and Verification of the Active Standby System

This appendix presents both the synchronous and the asynchronous formal model of the active standby example discussed in Section 9, as well as our verification of the satisfaction of (a proper formalization of) requirements R1–R5 by the synchronous model. In particular, Section B.1 introduces the active standby system in more depth than in Section 9. Sections B.2 and B.4 explain and present the entire executable specifications of, respectively, the Maude model of the synchronous version and the Real-Time Maude model of the asynchronous version of the active standby example discussed in Section 9. These models are also available at `http://www.ifi.uio.no/RealTimeMaude/PALS/`. Finally, Section B.6 reports on our verification of requirements R1–R5 for the active standby system.

B.1 The Active Standby System

This section recapitulates the active standby system described in [20]. As mentioned in Section 9, in the active standby system we have two physically separate computation platforms, located on each side of the aircraft. At any time, the computer in one side should be the *active* computer guiding the aircraft, and the computer on the opposite side should be in *standby* mode. The logic on each side decides which side is active, and is driven by its own clock and therefore executes asynchronously with respect to the other side. A simple synchronous model assumes that all components and channels are driven by the same clock, and that there is a one-step transmission delay between the two sides.

Figure 6 shows an overview of the architecture of the system. Each side is able to sense the status of a number of separate aircraft systems and can decide which sides are *fully available*. Furthermore, each side can fail at any time, which can be reliably detected. The pilot also has available a "manual selection" switch to change the currently active side. We assume that there is an environment that provide these data through ports, as shown in Fig. 6. Furthermore, each side sends in each round a signal to the other side, with value either 1 (side 1 is active), 2 (side 2 is active), or 0 (the side has failed).

We will not go here into the details of the logic of the active standby system, and refer to the formal model in Section B.3 for details. However, to understand our model checking effort and its results, we outline some aspects of the system:

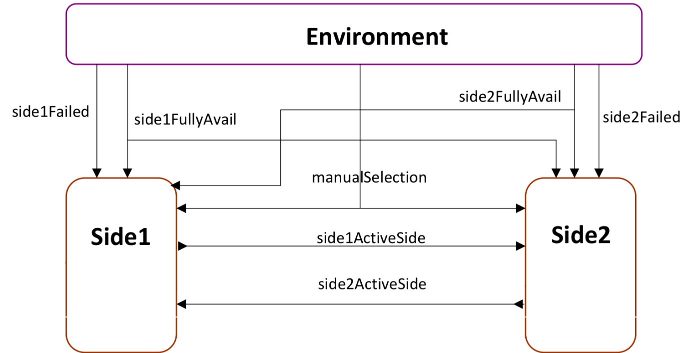

Fig. 6. The architecture of the active standby system.

1. An active side should focus most of its efforts on guiding the aircraft and not on deciding the active side; therefore, the non-active (standby) side monitors the *fully available data* and the pilot's *manual selection* switch, and is hence responsible for changing sides.
2. A failed side should obviously not be the active side.
3. If one side is fully available and the other side is not, then the fully available side should be the active side.
4. There is bias towards selecting side 1 as the active side after failures.

For example, if side 1 is the active side, and side 2 is fully available while the active side 1 suddenly becomes not fully available, then this is detected by the standby side 2. Side 2 then becomes the active side and sends '2' to the active side 1. When side 1 reads the '2', it realizes that side 2 has become active. Since there a one-step communication delay between the sides, we will encounter states in which each side claims to be the active side. If the non-active side is in some failure recovery mode, it may take even longer time before the active side is changed.

The environment generates its values nondeterministically in each round, with the following obvious constraints:

- Both sides cannot fail at the same time.
- If a side is failed it is not fully available.

Therefore, the environment has 16 different choices for the Boolean 5-tuple it generates in each round.

In [20], the authors list the following important properties that the active standby system should satisfy include:

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.

B.2 Formal Model of the Synchronous Version of Active Standby

We model the active standby system in an object-oriented way, where the two sides and the environment are modeled by objects.

Side 1 is modeled as an object of the following class:

```
class Side1 | state : Location, prevS2AS : Nat, prevManualSwitch : Bool,
              nexts1as : Data .

sort Location .   --- local states
ops initState side1Failed side2Failed side1Wait side2Wait side1Active
    side2Active : -> Location [ctor] .
```

The first three attributes correspond to the state variables of side 1 in the AADL specification; `state` denotes the local "state" of the object, `prevS2AS` denotes the previous value received from the `side2ActiveSide` connection from side 2, and `prevManualSwitch` is `true` iff the pilot did a manual selection in the previous round. The attribute `nexts1as` denotes the next value that the side 1 object should send to side 2 along its `side1ActiveSide` connection; that is, the value of `side1ActiveSide` to be sent at the end of an iteration. This next value is set when a transition is executed, but at that time the actual link cannot be set to that new value, to ensure that side 2 reads the *previous* value in its *current* round when side 2 performs the transition.

The definition of the class for the side 2 object is similar:

```
class Side2 | state : Location, prevS1AS : Nat, prevManualSwitch : Bool,
              nexts2as : Data .
```

The environment object does not have any attributes:

```
class Environment .
```

The communication links are modeled as simple objects characterized by the output port of the sender and the current value it holds; this value has the form `data(`n`)`, for $n \in \{0, 1, 2\}$, or `data(`b`)`, for b a Boolean value:

```
op `[port_from_has`value_`] : Oid Oid Data -> Object [ctor] .

sort Data .   --- Data in wires/channels
op data : Bool -> Data [ctor] .
op data : Nat -> Data [ctor] .

ops side1 side2 e : -> Oid [ctor] .                    --- component names
ops s1F s2F mS s1FA s2FA s1AS s2AS : -> Oid [ctor] .   --- port names
```

The function `performTrans` defines the transition function for each side. It takes the entire configuration as argument, and changes the internal state of the objects modeling the sides, but leaves the port objects unchanged:

```
 op performTrans : Configuration ~> Configuration .
```

For example, the following equation defines the transition function for side 1 when side 1 is in local state `side1Failed`. In this case, the value sent from the environment in this iteration through its `side1Failed` port (abbreviated to `s1F` in our model) is `false`, the value sent at the end of the previous iteration from side 2 through its `side2ActiveSide` port (abbreviated to `s2AS`) is `I =/= 0`, and the value sent by the environment through its `manualSelection` (`mS`) port is any Boolean value `B1`. The values of the inputs from the remaining two ports do not matter in this case. The transition takes the side 1 object to its local state `side1Wait`, sets its `prevManualSwitch` attribute to the received value `B1`, and sets its `prevS2AS` attribute to the received `s2AS` value `I`. At the end of the round, side 1 should send the value `1`[11], through its `side1ActiveSide` (`s1AS`) port. As already mentioned, the function `performTrans` does not set the output value explicitly, but only sets its internal attribute `nexts1as` to `1`. The function `performTrans` is then applied recursively to the rest of configuration, to execute the transition for the side 2 object as well, if that has not already been done.

```
vars CONF REST : Configuration .

ceq performTrans(< side1 : Side1 | state : side1Failed >
                 [port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
 =
   < side1 : Side1 | state : side1Wait, prevManualSwitch : B1,
                     prevS2AS : I, nexts1as : data(1) >
   performTrans([port s1F from e has value data(false)]
                [port s2AS from side2 has value data(I)]
                [port mS from e has value data(B1)]
                REST)
  if I =/= 0 .
```

[11] This is one of the few places where the sides are asymmetric; after a failure recovery, side 1 will be established as the active side.

The following equation models the standby side monitoring the system and deciding that the active side must change, either because the pilot pressed the manual selection switch to change active sides, or because the current active side is not fully available while the standby side is fully available. The following equation models the transition taken when the side 2 object is in state `side1Active`. In this case, side 2 should become active. This is achieved by side 2 going to state `side2Active` and sending a '2' message to side 1:

```
ceq performTrans(< side2 : Side2 | state : side1Active, prevS1AS : I2,
                                   prevManualSwitch : B2 >
                 [port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 [port s2FA from e has value data(true)]
                 [port s1FA from e has value data(B3)]
                 REST)
  =
    < side2 : Side2 | state : side2Active, prevManualSwitch : B1,
                      prevS1AS : I,  nexts2as : data(2) >
    performTrans([port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 [port s2FA from e has value data(true)]
                 [port s1FA from e has value data(B3)]
                 REST)
   if (I =/= 0) and (((not B2) and B1) or (not B3)) .
```

The following equation then models active side 1 receiving a '2' from side 2 and changing state to `side2Active`:

```
ceq performTrans(< side1 : Side1 | state : side1Active, prevS2AS : I >
                 [port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(2)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side1 : Side1 | state : side2Active, prevManualSwitch : B1,
                      prevS2AS : 2,  nexts1as : data(2) >
    performTrans([port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(2)]
                 [port mS from e has value data(B1)]
                 REST)
   if I =/= 2 .
```

Our definition of the transition function follows the definition of the transitions in the AADL model that was the starting point for our modeling and analysis effort. The 19 equations of the same style that define the transitions for side 1 are given in Section B.3; and so are the very similar 19 equations defining the transitions for side 2.

Finally, after the transition function has already been applied to both objects, the transition function becomes the identity function on the remaining configuration:

```
eq performTrans(CONF) = CONF [owise] .
```

We enclose the global state with an operator `{_}`:

```
op `{_`} : Configuration -> GlobalState [ctor] .
```

Our model of the synchronous system has only one rule, which models an iteration (or a synchronous step) of the composed system:

```
var ENVOUTPUT : EnvOutput .    var RESTOUTPUT : EnvOutputs .
var SYSTEM : Configuration .

crl [step] :
    {SYSTEM} => {genOutput(performTrans(envoutput(ENVOUTPUT, SYSTEM)))}
    if ENVOUTPUT ; RESTOUTPUT := possibleEnvOutputs .
```

In the matching condition, the variable `ENVOUTPUT` is assigned nondeterministically to any of the 16 possible 5-tuple of Boolean values that the environment can send (see below); the `envoutput` function inserts these values into the appropriate output ports objects; `performTrans` then performs the transitions in the two sides as explained above; and, finally, `genOutput` puts the output from the two sides (stored in the attributes `nexts1as` and `nexts2as`) into the appropriate port objects. The use of variables of sort `Configuration` in all equations defining these functions, in combination with the fact that these functions are declared to be *partial* functions on the sort `Configuration`, ensures that the various functions are applied in the order described above.

We define a sort `Envoutput` of 5-tuples of Boolean values, and a sort of `Envoutputs` of sets of such 5-tuples as follows:

```
sorts EnvOutput EnvOutputs .
op env : Bool Bool Bool Bool Bool -> EnvOutput [ctor] .
--- usage: env(s1F, s2F, ms, s1FA, s2FA)

subsort EnvOutput < EnvOutputs .
op _;_ : EnvOutputs EnvOutputs -> EnvOutputs [ctor assoc comm] .
```

The constant `possibleEnvOutputs` denoting the set of 5-tuples that the environment can generate in each iteration is defined as follows. Since at most one side can fail at any time, and that the first parameter of `env` states whether side 1 has failed, and the second whether side 2 has failed, at most one of the first two Booleans can be `true`. Furthermore, if one side has failed, then that side cannot be fully available. Therefore, we have the following set of possible environment outputs:

```
eq possibleEnvOutputs =
      --- side 1 fails:
      env(true, false, true, false, true) ;
      env(true, false, false, false, true) ;
      env(true, false, true, false, false) ;
      env(true, false, false, false, false) ;
      --- side 2 fails:
      env(false, true,  true, true, false) ;
      env(false, true, false, true, false) ;
      env(false, true, true, false, false) ;
      env(false, true, false, false, false) ;
      --- no side fails:
      env(false, false, true, true, true) ;
      env(false, false, true, true, false) ;
      env(false, false, true, false, true) ;
      env(false, false, true, false, false) ;
      env(false, false, false, true, true) ;
      env(false, false, false, true, false) ;
      env(false, false, false, false, true) ;
      env(false, false, false, false, false) .
```

The final two functions, inserting values from, respectively, the environment and the two sides into the appropriate port objects, are straight-forward:

```
vars D1 D2 D3 D4 D5 : Data .     vars B1 B2 B3 B4 B5 : Bool .

op genOutput : Configuration ~> Configuration .
op envoutput : EnvOutput Configuration ~> Configuration .

eq envoutput(env(B1, B2, B3, B4, B5),
       [port s1F from e has value D1]   [port s2F from e has value D2]
       [port mS from e has value D3]    [port s1FA from e has value D4]
       [port s2FA from e has value D5] REST)
  =
   [port s1F from e has value data(B1)]
   [port s2F from e has value data(B2)]
   [port mS from e has value data(B3)]
   [port s1FA from e has value data(B4)]
   [port s2FA from e has value data(B5)]  REST .

eq genOutput(< side1 : Side1 | nexts1as : D1 >
             [port s1AS from side1 has value D2]
             < side2 : Side2 | nexts2as : D3 >
             [port s2AS from side2 has value D4]
             CONF)
   =
    < side1 : Side1 | >      [port s1AS from side1 has value D1]
    < side2 : Side2 | >      [port s2AS from side2 has value D3]
    CONF .
```

Finally, the following equation defines an initial state `init`:

```
op init : -> GlobalState .
eq init =
   {< side1 : Side1 | state : initState, prevManualSwitch : false,
                      prevS2AS : 1, nexts1as : data(1) >
     [port s1AS from side1 has value data(1)]
     < side2 : Side2 | state : initState, prevManualSwitch : false,
                       prevS1AS : 1, nexts2as : data(1) >
     [port s2AS from side2 has value data(1)]
     < e : Environment | none >
     [port s1F from e has value data(false)]
     [port s2F from e has value data(false)]
     [port mS from e has value data(false)]
     [port s1FA from e has value data(true)]
     [port s2FA from e has value data(true)]} .
```

B.3 The Executable Model of the Synchronous System

The entire specification is given as follows:

```
(omod COMMON-ACTIVE-STANDBY is
  protecting NAT .

  vars CONF REST : Configuration .
  vars B1 B2 B3 B4 B5 B6 : Bool .
  vars I I2 : Nat .

  *** We first treat the output from the environment, which outputs 5 Booleans
  *** through the ports s1F, s2F, mS, s1FA, s2FA,
  --- satisfying [not s1F and s2F and not s2FA]      *** side 2 may fail
  ---            or [not s2F and s1F and not s1FA]   *** side 1 may fail
  ---            or [not s1F and not s2F]            *** no side fails

  --- Possible environment output:
  sort EnvOutput .
  op env : Bool Bool Bool Bool Bool -> EnvOutput [ctor] .
  --- usage: env(s1F, s2F, ms, s1FA, s2FA)

  sort EnvOutputs .
  subsort EnvOutput < EnvOutputs .
  op _;_ : EnvOutputs EnvOutputs -> EnvOutputs [ctor assoc comm] .

  --- All possible outputs from the environment:
  op possibleEnvOutputs : -> EnvOutputs .
  eq possibleEnvOutputs =
       --- side 1 fails:
       env(true, false, true, false, true) ;
       env(true, false, false, false, true) ;
       env(true, false, true, false, false) ;
       env(true, false, false, false, false) ;
       --- side 2 fails:
       env(false, true,  true, true, false) ;
       env(false, true, false, true, false) ;
       env(false, true, true, false, false) ;
```

```
     env(false, true, false, false, false) ;
     --- no side fails:
     env(false, false, true, true, true) ;
     env(false, false, true, true, false) ;
     env(false, false, true, false, true) ;
     env(false, false, true, false, false) ;
     env(false, false, false, true, true) ;
     env(false, false, false, true, false) ;
     env(false, false, false, false, true) ;
     env(false, false, false, false, false) .

--- Wires/channels are characterized by output source and port names:
op `[port_from_has`value_`] : Oid Oid Data -> Object [ctor] .

sort Data .   --- Data in wires/channels
op data : Bool -> Data [ctor] .
op data : Nat -> Data [ctor] .

ops side1 side2 e : -> Oid [ctor] .                    --- component names
ops s1F s2F mS s1FA s2FA s1AS s2AS : -> Oid [ctor] .   --- port names


*** Defining (deterministic) transitions:
op performTrans : Configuration ~> Configuration .

*** Object for the environment:
class Environment .    --- empty class

*** Object for Side 1:
class Side1 | state : Location, prevS2AS : Nat, prevManualSwitch : Bool,
              nexts1as : Data . *** next output produced by the machine

sort Location .   --- local states
ops initState side1Failed side2Failed side1Wait side2Wait
    side1Active side2Active : -> Location [ctor] .


*** The transitions of machine Side 1:

--- initState:
eq performTrans(< side1 : Side1 | state : initState >
                [port s1F from e has value data(true)]
                REST)
=
   < side1 : Side1 | state : side1Failed, prevS2AS : 0, prevManualSwitch : false,
                     nexts1as : data(0) >
   performTrans([port s1F from e has value data(true)]  REST) .

eq performTrans(< side1 : Side1 | state : initState >
                [port s1F from e has value data(false)]
                REST)
=
   < side1 : Side1 | state : side2Failed, prevS2AS : 0, prevManualSwitch : false,
                     nexts1as : data(1) >
```

```
    performTrans([port s1F from e has value data(false)]  REST) .


 --- state side1Failed:
 eq performTrans(< side1 : Side1 | state : side1Failed >
                 [port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(0)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side1 : Side1 | state : side2Failed, prevManualSwitch : B1, prevS2AS : 0,
                      nexts1as : data(1) >
    performTrans([port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(0)]
                 [port mS from e has value data(B1)]
                 REST) .

ceq performTrans(< side1 : Side1 | state : side1Failed >
                 [port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side1 : Side1 | state : side1Wait, prevManualSwitch : B1, prevS2AS : I,
                      nexts1as : data(1) >
    performTrans([port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
   if I =/= 0 .

 eq performTrans(< side1 : Side1 | state : side1Failed >
                 [port s1F from e has value data(true)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side1 : Side1 | prevManualSwitch : B1, prevS2AS : I,
                      nexts1as : data(0) >
    performTrans([port s1F from e has value data(true)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST) .

 --- state side2Failed:
 eq performTrans(< side1 : Side1 | state : side2Failed >
                 [port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(0)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side1 : Side1 | prevManualSwitch : B1, prevS2AS : 0,
                      nexts1as : data(1) >
    performTrans([port s1F from e has value data(false)]
```

```
                 [port s2AS from side2 has value data(0)]
                 [port mS from e has value data(B1)]
                 REST) .

 ceq performTrans(< side1 : Side1 | state : side2Failed >
                 [port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
   =
     < side1 : Side1 | state : side1Wait, prevManualSwitch : B1, prevS2AS : I,
                      nexts1as : data(1) >
     performTrans([port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
    if I =/= 0 .

  eq performTrans(< side1 : Side1 | state : side2Failed >
                 [port s1F from e has value data(true)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
   =
     < side1 : Side1 | state : side1Failed, prevManualSwitch : B1, prevS2AS : I,
                      nexts1as : data(0) >
     performTrans([port s1F from e has value data(true)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST) .

  --- state side1Wait:
 ceq performTrans(< side1 : Side1 | state : side1Wait >
                 [port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
   =
     < side1 : Side1 | state : side1Active, prevManualSwitch : B1, prevS2AS : I,
                      nexts1as : data(1) >
     performTrans([port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
    if I =/= 0 .

  eq performTrans(< side1 : Side1 | state : side1Wait >
                 [port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(0)]
                 [port mS from e has value data(B1)]
                 REST)
   =
     < side1 : Side1 | state : side2Failed, prevManualSwitch : B1, prevS2AS : 0,
                      nexts1as : data(1) >
```

```
   performTrans([port s1F from e has value data(false)]
                [port s2AS from side2 has value data(0)]
                [port mS from e has value data(B1)]
                REST) .

 eq performTrans(< side1 : Side1 | state : side1Wait >
                [port s1F from e has value data(true)]
                [port s2AS from side2 has value data(I)]
                [port mS from e has value data(B1)]
                REST)
  =
   < side1 : Side1 | state : side1Failed, prevManualSwitch : B1, prevS2AS : I,
                     nexts1as : data(0) >
   performTrans([port s1F from e has value data(true)]
                [port s2AS from side2 has value data(I)]
                [port mS from e has value data(B1)]
                REST) .

 --- state side1Active:
ceq performTrans(< side1 : Side1 | state : side1Active, prevS2AS : I >
                [port s1F from e has value data(false)]
                [port s2AS from side2 has value data(2)]
                [port mS from e has value data(B1)]
                REST)
  =
   < side1 : Side1 | state : side2Active, prevManualSwitch : B1, prevS2AS : 2,
                     nexts1as : data(2) >
   performTrans([port s1F from e has value data(false)]
                [port s2AS from side2 has value data(2)]
                [port mS from e has value data(B1)]
                REST)
  if I =/= 2 .

 eq performTrans(< side1 : Side1 | state : side1Active >
                [port s1F from e has value data(true)]
                [port s2AS from side2 has value data(I)]
                [port mS from e has value data(B1)]
                REST)
  =
   < side1 : Side1 | state : side1Failed, prevManualSwitch : B1, prevS2AS : I,
                     nexts1as : data(0) >
   performTrans([port s1F from e has value data(true)]
                [port s2AS from side2 has value data(I)]
                [port mS from e has value data(B1)]
                REST) .

 eq performTrans(< side1 : Side1 | state : side1Active >
                [port s1F from e has value data(false)]
                [port s2AS from side2 has value data(0)]
                [port mS from e has value data(B1)]
                REST)
  =
   < side1 : Side1 | state : side2Failed, prevManualSwitch : B1, prevS2AS : 0,
                     nexts1as : data(1) >
```

```
    performTrans([port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(0)]
                 [port mS from e has value data(B1)]
                 REST) .

ceq performTrans(< side1 : Side1 | state : side1Active, prevS2AS : I2 >
                 [port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side1 : Side1 | state : side1Active, prevManualSwitch : B1, prevS2AS : I,
                      nextsias : data(1) >
    performTrans([port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)  if (I2 == 2 or I =/= 2) and I =/= 0 .

 --- state side2Active:
ceq performTrans(< side1 : Side1 | state : side2Active, prevS2AS : I2,
                                   prevManualSwitch : B2 >
                 [port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 [port s1FA from e has value data(true)]
                 [port s2FA from e has value data(B3)]
                 REST)
  =
    < side1 : Side1 | state : side1Active, prevManualSwitch : B1, prevS2AS : I,
                      nextsias : data(1) >
    performTrans([port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 [port s1FA from e has value data(true)]
                 [port s2FA from e has value data(B3)]
                 REST)  if (((not B2) and B1) or (not B3))  and I =/= 0 .

 eq performTrans(< side1 : Side1 | state : side2Active >
                 [port s1F from e has value data(true)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side1 : Side1 | state : side1Failed, prevManualSwitch : B1, prevS2AS : I,
                      nextsias : data(0) >
    performTrans([port s1F from e has value data(true)]
                 [port s2AS from side2 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST) .

 eq performTrans(< side1 : Side1 | state : side2Active >
                 [port s1F from e has value data(false)]
                 [port s2AS from side2 has value data(0)]
                 [port mS from e has value data(B1)]
```

```
                REST)
 =
   < side1 : Side1 | state : side2Failed, prevManualSwitch : B1, prevS2AS : 0,
                     nexts1as : data(1) >
   performTrans([port s1F from e has value data(false)]
                [port s2AS from side2 has value data(0)]
                [port mS from e has value data(B1)]
                REST) .

ceq performTrans(< side1 : Side1 | state : side2Active, prevS2AS : I2,
                                   prevManualSwitch : B2 >
                [port s1F from e has value data(false)]
                [port s2AS from side2 has value data(I)]
                [port mS from e has value data(B1)]
                [port s1FA from e has value data(B4)]
                [port s2FA from e has value data(B3)]
                REST)
 =
   < side1 : Side1 | prevManualSwitch : B1, prevS2AS : I,
                     nexts1as : data(2) >
   performTrans([port s1F from e has value data(false)]
                [port s2AS from side2 has value data(I)]
                [port mS from e has value data(B1)]
                [port s1FA from e has value data(B4)]
                [port s2FA from e has value data(B3)]
                REST)  if ((not B4) or (B3 and (B2 or not B1)))  and I =/= 0 .

 *** Side 2:

 class Side2 | state : Location, prevS1AS : Nat, prevManualSwitch : Bool,
              nexts2as : Data . *** next output to be generated by side2

 *** The transitions of Side 2:

 --- initState:
 eq performTrans(< side2 : Side2 | state : initState >
                [port s2F from e has value data(true)]
                REST)
 =
   < side2 : Side2 | state : side2Failed, prevS1AS : 0, prevManualSwitch : false,
                     nexts2as : data(0) >
   performTrans([port s2F from e has value data(true)]  REST) .

 eq performTrans(< side2 : Side2 | state : initState >
                [port s2F from e has value data(false)]
                REST)
 =
   < side2 : Side2 | state : side1Failed, prevS1AS : 0, prevManualSwitch : false,
                     nexts2as : data(2) >
   performTrans([port s2F from e has value data(false)]  REST) .


 --- state side2Failed:
 eq performTrans(< side2 : Side2 | state : side2Failed >
```

```
                [port s2F from e has value data(false)]
                [port s1AS from side1 has value data(0)]
                [port mS from e has value data(B1)]
                REST)
  =
    < side2 : Side2 | state : side1Failed, prevManualSwitch : B1, prevS1AS : 0,
                     nexts2as : data(2) >
    performTrans([port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(0)]
                 [port mS from e has value data(B1)]
                 REST) .

ceq performTrans(< side2 : Side2 | state : side2Failed >
                 [port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side2 : Side2 | state : side2Wait, prevManualSwitch : B1, prevS1AS : I,
                     nexts2as : data(1) >
    performTrans([port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
   if I =/= 0 .

 eq performTrans(< side2 : Side2 | state : side2Failed >
                 [port s2F from e has value data(true)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side2 : Side2 | prevManualSwitch : B1, prevS1AS : I,
                     nexts2as : data(0) >
    performTrans([port s2F from e has value data(true)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST) .

 --- state side1Failed:

 eq performTrans(< side2 : Side2 | state : side1Failed >
                 [port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(0)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side2 : Side2 | prevManualSwitch : B1, prevS1AS : 0,
                     nexts2as : data(2) >
    performTrans([port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(0)]
                 [port mS from e has value data(B1)]
                 REST) .
```

```
ceq performTrans(< side2 : Side2 | state : side1Failed >
                 [port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side2 : Side2 | state : side2Wait, prevManualSwitch : B1, prevS1AS : I,
                      nexts2as : data(1) >
    performTrans([port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
   if I =/= 0 .

 eq performTrans(< side2 : Side2 | state : side1Failed >
                 [port s2F from e has value data(true)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side2 : Side2 | state : side2Failed, prevManualSwitch : B1, prevS1AS : I,
                      nexts2as : data(0) >
    performTrans([port s2F from e has value data(true)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST) .

 --- state side2Wait:
ceq performTrans(< side2 : Side2 | state : side2Wait >
                 [port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side2 : Side2 | state : side1Active, prevManualSwitch : B1, prevS1AS : I,
                      nexts2as : data(1) >
    performTrans([port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
   if I =/= 0 .

 eq performTrans(< side2 : Side2 | state : side2Wait >
                 [port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(0)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side2 : Side2 | state : side1Failed, prevManualSwitch : B1, prevS1AS : 0,
                      nexts2as : data(2) >
    performTrans([port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(0)]
                 [port mS from e has value data(B1)]
                 REST) .
```

```
 eq performTrans(< side2 : Side2 | state : side2Wait >
                [port s2F from e has value data(true)]
                [port s1AS from side1 has value data(I)]
                [port mS from e has value data(B1)]
                REST)
  =
    < side2 : Side2 | state : side2Failed, prevManualSwitch : B1, prevS1AS : I,
                     nexts2as : data(0) >
    performTrans([port s2F from e has value data(true)]
                [port s1AS from side1 has value data(I)]
                [port mS from e has value data(B1)]
                REST) .

 --- state side1Active:
 eq performTrans(< side2 : Side2 | state : side1Active, prevS1AS : I >
                [port s2F from e has value data(true)]
                [port s1AS from side1 has value data(I)]
                [port mS from e has value data(B1)]
                REST)
  =
    < side2 : Side2 | state : side2Failed, prevManualSwitch : B1,
                     nexts2as : data(0) >
    performTrans([port s2F from e has value data(true)]
                [port s1AS from side1 has value data(I)]
                [port mS from e has value data(B1)]
                REST) .

ceq performTrans(< side2 : Side2 | state : side1Active, prevS1AS : I2,
                                  prevManualSwitch : B2 >
                [port s2F from e has value data(false)]
                [port s1AS from side1 has value data(I)]
                [port mS from e has value data(B1)]
                [port s2FA from e has value data(true)]
                [port s1FA from e has value data(B3)]
                REST)
  =
    < side2 : Side2 | state : side2Active, prevManualSwitch : B1, prevS1AS : I,
                     nexts2as : data(2) >
    performTrans([port s2F from e has value data(false)]
                [port s1AS from side1 has value data(I)]
                [port mS from e has value data(B1)]
                [port s2FA from e has value data(true)]
                [port s1FA from e has value data(B3)]
                REST)
   if (I =/= 0) and (((not B2) and B1) or (not B3)) .

 eq performTrans(< side2 : Side2 | state : side1Active >
                [port s2F from e has value data(false)]
                [port s1AS from side1 has value data(0)]
                [port mS from e has value data(B1)]
                REST)
  =
    < side2 : Side2 | state : side1Failed, prevManualSwitch : B1, prevS1AS : 0,
```

```
                    nexts2as : data(2) >
    performTrans([port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(0)]
                 [port mS from e has value data(B1)]
                 REST) .

ceq performTrans(< side2 : Side2 | state : side1Active, prevS1AS : I2,
                                   prevManualSwitch : B2 >
                 [port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 [port s1FA from e has value data(B4)]
                 [port s2FA from e has value data(B3)]
                 REST)
  =
    < side2 : Side2 | prevManualSwitch : B1, prevS1AS : I,
                      nexts2as : data(1) >
    performTrans([port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 [port s1FA from e has value data(B4)]
                 [port s2FA from e has value data(B3)]
                 REST)  if I =/= 0 and ((not B3) or (B4 and (B2 or (not B1)))) .

 --- state side2Active:
ceq performTrans(< side2 : Side2 | state : side2Active, prevS1AS : I2 >
                 [port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side2 : Side2 | prevManualSwitch : B1, prevS1AS : 2,
                      nexts2as : data(2) >
    performTrans([port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST) if I =/= 0 and (I2 == 1 or I =/= 1) .

 eq performTrans(< side2 : Side2 | state : side2Active >
                 [port s2F from e has value data(true)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side2 : Side2 | state : side2Failed, prevManualSwitch : B1, prevS1AS : I,
                      nexts2as : data(0) >
    performTrans([port s2F from e has value data(true)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST) .

 eq performTrans(< side2 : Side2 | state : side2Active >
                 [port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(0)]
```

```
                  [port mS from e has value data(B1)]
                  REST)
   =
     < side2 : Side2 | state : side1Failed, prevManualSwitch : B1, prevS1AS : 0,
                       nexts2as : data(2) >
     performTrans([port s2F from e has value data(false)]
                  [port s1AS from side1 has value data(0)]
                  [port mS from e has value data(B1)]
                  REST) .

 ceq performTrans(< side2 : Side2 | state : side2Active, prevS1AS : I2 >
                  [port s2F from e has value data(false)]
                  [port s1AS from side1 has value data(1)]
                  [port mS from e has value data(B1)]
                  REST)
   =
     < side2 : Side2 | state : side1Active, prevManualSwitch : B1, prevS1AS : 1,
                       nexts2as : data(1) >
     performTrans([port s2F from e has value data(false)]
                  [port s1AS from side1 has value data(1)]
                  [port mS from e has value data(B1)]
                  REST)  if I2 =/= 1 .

  *** Finally, only the env object and the wires should be inside the scope of
  *** performTrans:
  eq performTrans(CONF) = CONF [owise] .

endom)

*** Defining synchronous executions:

(omod SYNCHRONOUS-ACTIVE-STANDBY is
  including COMMON-ACTIVE-STANDBY .

  var ENVOUTPUT : EnvOutput .
  var RESTOUTPUT : EnvOutputs .
  vars SYSTEM CONF REST : Configuration .
  vars D1 D2 D3 D4 D5 : Data .
  vars B1 B2 B3 B4 B5 : Bool .

  --- Enclose the global state:
  sort GlobalState .
  op `{_`} : Configuration -> GlobalState [ctor] .

  --- The synchronous step function should be the following:
  crl [step] :
      {SYSTEM} => {genOutput(performTrans(envoutput(ENVOUTPUT, SYSTEM)))}
      if ENVOUTPUT ; RESTOUTPUT := possibleEnvOutputs .

  *** Defining  a function that writes the output to the wires for the next round,
  *** and one function that takes an output from the environment and inserts
  *** them into the appropriate wires.
  op genOutput : Configuration ~> Configuration .
  op envoutput : EnvOutput Configuration ~> Configuration .
```

```
  eq envoutput(env(B1, B2, B3, B4, B5),
               [port s1F from e has value D1]
               [port s2F from e has value D2]
               [port mS from e has value D3]
               [port s1FA from e has value D4]
               [port s2FA from e has value D5]
               REST)
    =
               [port s1F from e has value data(B1)]
               [port s2F from e has value data(B2)]
               [port mS from e has value data(B3)]
               [port s1FA from e has value data(B4)]
               [port s2FA from e has value data(B5)]
               REST .

  eq genOutput(< side1 : Side1 | nexts1as : D1 >
               [port s1AS from side1 has value D2]
               < side2 : Side2 | nexts2as : D3 >
               [port s2AS from side2 has value D4]
               CONF)
   =
               < side1 : Side1 | >
               [port s1AS from side1 has value D1]
               < side2 : Side2 | >
               [port s2AS from side2 has value D3]
               CONF .

  *** Initial state, everything works well and side 1 is the starter:
  op init : -> GlobalState .
  eq init =
      {< side1 : Side1 | state : initState, prevManualSwitch : false, prevS2AS : 1,
                         nexts1as : data(1) >
       [port s1AS from side1 has value data(1)]
       < side2 : Side2 | state : initState, prevManualSwitch : false, prevS1AS : 1,
                         nexts2as : data(1) >
       [port s2AS from side2 has value data(1)]
       < e : Environment | none >
       [port s1F from e has value data(false)]
       [port s2F from e has value data(false)]
       [port mS from e has value data(false)]
       [port s1FA from e has value data(true, true)]
       [port s2FA from e has value data(true, true)]} .
endom)
```

B.4 The Asynchronous Model of Active Standby

As already mentioned, our Real-Time Maude model of the asynchronous version of the active standby system is just an adaptation of the PALS transformation of the synchronous version to the simpler setting in which:

- the minimum message delay and the execution time of a transition are both zero,

- the time domain is discrete,
- there is no clock skew,
- and where aspects such as the wiring diagram is not explicitly represented.

In addition, all messages from a given object in a given round are sent to the network in *one* step.

We define a "PALS wrappers" as object instances of the following class:

```
class PalsWrapper | roundTimer : Time,  inputBuffer : Configuration,
                    outputBuffer : Configuration,
                    outputBackoffTimer : TimeInf,  machine : Object .
```

The `machine` attribute denotes the object modeling the component in our specification of the synchronous model, and the other attributes are as in Section 4.

In the asynchronous system, messages have the form `msg` *data* `from port` *portId* `of` *senderObject* `to` *receiverObject*. To model nondeterministic messaging delays, such messages are enclosed by a "wrapper" `dly`, so that `dly(`*msg*, t_1`,`t_2`)` denotes the message *msg* that has *remaining minimum delay* t_1 and *remaining maximum delay* t_2[12]:

```
ops minMsgDelay maxMsgDelay : -> Time .
op palsRound : -> Time .                 eq palsRound =  maxMsgDelay + 2 .
--- one round of the system for minimum msg delay 0 and discrete time
op msg_from`port_of_to_ : Data Oid Oid Oid -> Msg [ctor] .
op dly : Msg Time Time -> Msg .
```

The rewrite rules in this model are immediate. The following rule reads an incoming message and puts it into the input buffer:

```
rl [readMsg] :
   dly(msg D from port P of O' to O), 0, T)
   < O : PalsWrapper | inputBuffer : MSGS >
  =>
   < O : PalsWrapper | inputBuffer : MSGS (msg D from port P of O' to O) > .
```

When the round timer expires, messages are read from the input buffer, the transition is performed and output is put into output buffer, and the output timer is set to 1:

```
crl [executeTransitionSide1] :
    < side1 : PalsWrapper | roundTimer : 0,
                            inputBuffer : MSGS,
                            machine :
                              (< side1 : Side1 | state : L1, prevS2AS : N1,
                                                 prevManualSwitch : B1,
                                                 nexts1as : data(0) >) >
   =>
```

[12] As mentioned, in the experiments reported in this paper, the minimum delay is always zero.

```
   < side1 : PalsWrapper | roundTimer : palsRound,
                           inputBuffer : none,
                           machine :
                             (< side1 : Side1 | state : L2, prevS2AS : N2,
                                                prevManualSwitch : B2,
                                                nexts1as : data(0) >),
                           outputBuffer :
                            dly(msg D2 from port s1AS of side1 to side2,
                                0, maxMsgDelay),
                           outputBackoffTimer : 1 >
  if < side1 : Side1 | state : L2, prevS2AS : N2, prevManualSwitch : B2,
                       nexts1as : D2 >
     C:Configuration
      := performTrans(< side1 : Side1 | state : L1, prevS2AS : N1,
                                        prevManualSwitch : B1,
                                        nexts1as : data(0) >
                      changeForm(MSGS)) .
```

The function `performTrans` is the transition function for the components defined for the synchronous model (and hence `changeForm` is needed to transform messages into "wires"). The rule for `side2` is entirely similar. The rule for the environment is also immediate:

```
crl [envRound] :
    < e : PalsWrapper | roundTimer : 0 >
   =>
    < e : PalsWrapper |
           roundTimer : palsRound,
           outputBuffer :
            (dly(msg data(B1) from port s1F of e to side1, maxMsgDelay)
             dly(msg data(B2) from port s2F of e to side2, maxMsgDelay)
             dly(msg data(B3) from port mS of e to side1, maxMsgDelay)
             dly(msg data(B3) from port mS of e to side2, maxMsgDelay)
             dly(msg data(B4) from port s1FA of e to side1, maxMsgDelay)
             dly(msg data(B4) from port s1FA of e to side2, maxMsgDelay)
             dly(msg data(B5) from port s2FA of e to side1, maxMsgDelay)
             dly(msg data(B5) from port s2FA of e to side2, maxMsgDelay)),
           outputTimer : 1 >
   if env(B1, B2, B3, B4, B5) ; ENVOUTPUTS := possibleEnvOutputs .
```

In the following rule, the messages in the output buffer are sent into the network in one step:

```
rl [send] :
   < O : PalsWrapper | outputTimer : 0, outputBuffer : MSGS >
  =>
   < O : PalsWrapper | outputTimer : INF, outputBuffer : none >  MSGS .
```

Finally, the tick rule advances time by one time unit in each tick step:

```
  crl [tick] : {CONF} => {timeEffect(CONF, 1)} in time 1 if 1 <= mte(CONF) .
```

The function `timeEffect` defines the effect of time elapse on a configuration by decreasing the timer and message delay values according to the elapsed time, and the function `mte` gives the least time until the next timer expires or until some message must be delivered:

```
vars NECF1 NECF2 : NEConfiguration .  vars T T1 T2 : Time .
var TI : TimeInf .

op timeEffect : Configuration Time -> Configuration [frozen (1)] .
eq timeEffect(NECF1 NECF2, T) = timeEffect(NECF1, T) timeEffect(NECF2, T) .
eq timeEffect(none, T) = none .

eq timeEffect(< O : PalsWrapper | palsTimer : T1, outputTimer : TI >, T)
 = < O : PalsWrapper | palsTimer : T1 monus T, outputTimer : TI monus T > .
eq timeEffect(dly(MSG, T1, T2), T) = dly(MSG, T1 monus T, T2 monus T) .

op mte : Configuration -> TimeInf [frozen (1)] .
eq mte(NECF1 NECF2) = min(mte(NECF1), mte(NECF2)) .
eq mte(none) = INF .

eq mte(< O : PalsWrapper | palsTimer : T, outputTimer : TI >) = min(T, TI) .
eq mte(dly(MSG, T1, T2)) = T2 .
```

Finally, the initial state is a state where the input buffers are full, and the round timer is zero:

```
op init : -> GlobalSystem .
eq init =
   {< side1 : PalsWrapper |
        palsTimer : 0,
        inputBuffer :
         ((msg data(false) from port s1F of e to side1)
          (msg data(true) from port s1FA of e to side1)
          (msg data(false) from port mS of e to side1)
          (msg data(true) from port s2FA of e to side1)
          (msg data(0) from port s2AS of side2 to side1)),
        machine :
          (< side1 : Side1 | state : initState, prevManualSwitch : false,
                             prevS2AS : 0, nexts1as : data(0) >),
        outputBuffer : none,  outputTimer : INF >
    < side2 : PalsWrapper |
        palsTimer : 0,
        inputBuffer :
         ((msg data(false) from port s2F of e to side2)
          (msg data(true) from port s1FA of e to side2)
          (msg data(false) from port mS of e to side2)
          (msg data(true) from port s2FA of e to side2)
          (msg data(0) from port s1AS of side1 to side2)),
        machine :
          (< side2 : Side2 | state : initState, prevManualSwitch : false,
```

```
                          prevS1AS : 1, nexts2as : data(1) >),
        outputBuffer : none,  outputTimer : INF  >
    < e : PalsWrapper | palsTimer : 0, inputBuffer : none,
                        machine : (< e : Environment | none >),
                        outputBuffer : none, outputTimer : INF  >} .
```

B.5 The Executable Model of the Asynchronous System

The entire formal specification of the asynchronous version of the active standby system is given as follows:

```
(tomod ASYNCHRONOUS-ACTIVE-STANDBY is
  including COMMON-ACTIVE-STANDBY .
  protecting NAT-TIME-DOMAIN-WITH-INF .

  vars B1 B2 B3 B4 B5 : Bool .
  var D D1 D2 : Data .
  vars P O O' : Oid .
  vars T T' T1 T2 : Time .
  var TI : TimeInf .
  vars MSGS REST CONF : Configuration .
  vars NECF1 NECF2 : NEConfiguration .
  vars L1 L2 : Location .
  vars N1 N2 : Nat .
  var ENVOUTPUTS : EnvOutputs .
  var MSG : Msg .


  class PalsWrapper | palsTimer : Time,            --- expires each PALS round
                      inputBuffer : Configuration, --- input buffer
                      outputBuffer : Configuration,
                      outputTimer : TimeInf,
                      machine : Object .           --- inner machine

  ops minMsgDelay maxMsgDelay : -> Time .

  op palsRound : -> Time .                    eq palsRound =  maxMsgDelay + 2 .
                                              *** 1 for outtimer + maxDelay + 1 for margin

   --- Messages have the following form:
  op msg_from`port_of_to_ : Data Oid Oid Oid -> Msg [ctor] .

  --- Message delay wrapper:
  op dly : Msg Time Time -> Msg .
  --- dly(msg, minRemainingDelay, maxRemainingDelay)

  --- The name of the wrapper is the same as of the inner machine.

  --- Rule for reading messages.
  rl [readMsg] :
     dly(msg D from port P of O' to O, 0, T)
     < O : PalsWrapper | inputBuffer : MSGS >
    =>
```

```
  < O : PalsWrapper | inputBuffer : MSGS (msg D from port P of O' to O) > .


--- Rule for sending out messages, already equipped with delays.
--- Notice that we send out ALL messages in one step.
rl [send] :
   < O : PalsWrapper | outputTimer : 0, outputBuffer : MSGS >
  =>
   < O : PalsWrapper | outputTimer : INF, outputBuffer : none >  MSGS .

--- Timer expires. Execute transition; change the messages in the input buffer
--- to the right form.
crl [executeTransitionSide1] :
    < side1 : PalsWrapper |
         palsTimer : 0,
         inputBuffer : MSGS,
         machine : (< side1 : Side1 | state : L1, prevS2AS : N1,
                                      prevManualSwitch : B1, nexts1as : data(0) >) >
   =>
    < side1 : PalsWrapper |
         palsTimer : palsRound,
         inputBuffer : none,
         machine : (< side1 : Side1 | state : L2, prevS2AS : N2, prevManualSwitch : B2,
                                      nexts1as : data(0) >),
         outputBuffer : dly(msg D2 from port s1AS of side1 to side2, minMsgDelay, maxMsgDelay),
         outputTimer : 1 >
   if < side1 : Side1 | state : L2, prevS2AS : N2, prevManualSwitch : B2, nexts1as : D2 >
      C:Configuration
       := performTrans(< side1 : Side1 | state : L1, prevS2AS : N1,
                                         prevManualSwitch : B1, nexts1as : data(0) >
                       changeForm(MSGS)) .

crl [executeTransitionSide2] :
    < side2 : PalsWrapper | palsTimer : 0,
                            inputBuffer : MSGS,
                            machine : (< side2 : Side2 | state : L1, prevS1AS : N1,
                                                         prevManualSwitch : B1,
                                                         nexts2as : data(0) >) >
   =>
    < side2 : PalsWrapper | palsTimer : palsRound,
                            inputBuffer : none,
                            machine : (< side2 : Side2 | state : L2, prevS1AS : N2,
                                                         prevManualSwitch : B2,
                                                         nexts2as : data(0) >),
                            outputBuffer : dly(msg D2 from port s2AS of side2 to side1,
                                                   minMsgDelay, maxMsgDelay),
                            outputTimer : 1 >
   if < side2 : Side2 | state : L2, prevS1AS : N2, prevManualSwitch : B2, nexts2as : D2 >
      C:Configuration
       := performTrans(< side2 : Side2 | state : L1, prevS1AS : N1, prevManualSwitch : B1,
                                         nexts2as : data(0) >
                       changeForm(MSGS)) .

op changeForm : Configuration ~> Configuration .
```

```
eq changeForm((msg D from port P of O to O') REST) =
     [port P from O has value D]
     changeForm(REST) .
eq changeForm(none) = none .

*** Environment:
crl [envRound] :
    < e : PalsWrapper | palsTimer : 0 >
   =>
    < e : PalsWrapper |
          palsTimer : palsRound,
          outputBuffer :
           (dly(msg data(B1) from port s1F of e to side1, minMsgDelay, maxMsgDelay)
            dly(msg data(B2) from port s2F of e to side2, minMsgDelay, maxMsgDelay)
            dly(msg data(B3) from port mS of e to side1, minMsgDelay, maxMsgDelay)
            dly(msg data(B3) from port mS of e to side2, minMsgDelay, maxMsgDelay)
            dly(msg data(B4) from port s1FA of e to side1, minMsgDelay, maxMsgDelay)
            dly(msg data(B4) from port s1FA of e to side2, minMsgDelay, maxMsgDelay)
            dly(msg data(B5) from port s2FA of e to side1, minMsgDelay, maxMsgDelay)
            dly(msg data(B5) from port s2FA of e to side2, minMsgDelay, maxMsgDelay)),
          outputTimer : 1 >
   if env(B1, B2, B3, B4, B5) ; ENVOUTPUTS := possibleEnvOutputs .

*** Time advance; deterministic for simplicity.
crl [tick] : CONF => timeEffect(CONF, 1) in time 1 if 1 <= mte(CONF) .

op timeEffect : Configuration Time -> Configuration [frozen (1)] .
eq timeEffect(NECF1 NECF2, T) = timeEffect(NECF1, T) timeEffect(NECF2, T) .
eq timeEffect(none, T) = none .

eq timeEffect(< O : PalsWrapper | palsTimer : T1, outputTimer : TI >, T)
 = < O : PalsWrapper | palsTimer : T1 monus T, outputTimer : TI monus T > .
eq timeEffect(dly(MSG, T1, T2), T) = dly(MSG, T1 monus T, T2 monus T) .

op mte : Configuration -> TimeInf [frozen (1)] .
eq mte(NECF1 NECF2) = min(mte(NECF1), mte(NECF2)) .
eq mte(none) = INF .

eq mte(< O : PalsWrapper | palsTimer : T, outputTimer : TI >) = min(T, TI) .
eq mte(dly(MSG, T1, T2)) = T2 .

op init : -> GlobalSystem .
eq init =
     {< side1 : PalsWrapper | palsTimer : 0,
                              inputBuffer :
                                ((msg data(false) from port s1F of e to side1)
                                 (msg data(true) from port s1FA of e to side1)
                                 (msg data(false) from port mS of e to side1)
                                 (msg data(true) from port s2FA of e to side1)
                                 (msg data(1) from port s2AS of side2 to side1)),
                              machine :
                                 (< side1 : Side1 | state : initState,
                                                    prevManualSwitch : false,
```

```
                                                  prevS2AS : 1, nextsias : data(1) >),
                              outputBuffer : none,
                              outputTimer : INF >
      < side2 : PalsWrapper | palsTimer : 0,
                              inputBuffer :
                                ((msg data(false) from port s2F of e to side2)
                                 (msg data(true) from port s1FA of e to side2)
                                 (msg data(false) from port mS of e to side2)
                                 (msg data(true) from port s2FA of e to side2)
                                 (msg data(0) from port s1AS of side1 to side2)),
                              machine :
                                  (< side2 : Side2 | state : initState,
                                                     prevManualSwitch : false,
                                                     prevS1AS : 0,
                                                     nexts2as : data(0) >),
                              outputBuffer : none,
                              outputTimer : INF  >
      < e : PalsWrapper | palsTimer : 0, inputBuffer : none,
                          machine : (< e : Environment | none >),
                          outputBuffer : none, outputTimer : INF  >} .

  *** The 'next1as' attributes are never changed and hence do not add to state space!

  eq minMsgDelay = 0 .
  eq maxMsgDelay = 1 .
endtom)
```

B.6 Model Checking R1–R5 for the Synchronous Model

This section explains explains our model checking analysis of the synchronous model w.r.t. the requirements R1–R5.

Since most of these requirements contain the clause "the availability of a side has not changed," we have, for convenience, modified our synchronous model slightly, so that the ports `s1FA` (side 1 fully available) and `s2FA` (side 2 fully available) carry both the *current* value sent, and also the *previous* value sent. (In the asynchronous version, the same effect can be achieved with each side object remembering the last such value received.) This minor modification is not really necessary, since the fact that availability has not changed can be expressed by the LTL formula

```
  (side1FullyAvail <-> O side1FullyAvailable) /\
  (side2FullyAvail <-> O side2FullyAvailable)
```

but then we would need to add more next operators to the entire temporal logic formulas.

The function `envoutput` inserting the environment output into the appropriate output ports is thus modified to:

```
eq envoutput(env(B1, B2, B3, B4, B5),
             [port s1F from e has value D1]
             [port s2F from e has value D2]
```

```
              [port mS from e has value D3]
              [port s1FA from e has value data(B6, B7)]
              [port s2FA from e has value data(B8, B9)]
              REST)
  =
              [port s1F from e has value data(B1)]
              [port s2F from e has value data(B2)]
              [port mS from e has value data(B3)]
              [port s1FA from e has value data(B4,B6)]
              [port s2FA from e has value data(B5,B8)]
              REST .
```

However, the search commands that were used to find the number of reachable states in a model and execution times for reachability analyses were performed in the *original* systems.

Requirement R1. Recall Requirement R1:

> 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).

By "side N being active" we assume that what is meant is that that side N sends its number to the other side through its `side`N`activeSide` port (as opposed to being in location `side`N`Active`). The parameterized atomic proposition `side_active` can therefore be defined as follows:

```
op side_active : Nat -> Prop [ctor] .
eq {CONF  < side1 : Side1 | nexts1as : data(N) >} |= side 1 active = (N == 1) .
eq {CONF  < side2 : Side2 | nexts2as : data(N) >} |= side 2 active = (N == 2) .
```

We can then define what it means that both sides agree on which side is active:

```
op agreeOnActiveSide : -> Prop [ctor] .
eq {CONF
    < side1 : Side1 | nexts1as : data(N1) >
    < side2 : Side2 | nexts2as : data(N2) >} |= agreeOnActiveSide
  = N1 == N2  and N1 =/= 0 .
```

Since the ports `s`N`FA` now contain both the *previous* and the *current* value sent, the proposition defining whether or not full availability has changed is given as follows:

```
op side_availChanged : Nat -> Prop [ctor] .
eq {CONF [port s1FA from e has value data(B, B2)]} |= side 1 availChanged
  = (B =/= B2) .
eq {CONF [port s2FA from e has value data(B, B2)]} |= side 2 availChanged
 = (B =/= B2) .

op changeInAvailability : -> Formula .
eq changeInAvailability = side 1 availChanged \/ side 2 availChanged .
```

State predicates stating whether a manual selection has taken place and whether a given side has failed are defined by considering the output from the environment:[13]

```
op manSelectPressed : -> Prop [ctor] .
eq {CONF [port mS from e has value data(B)]} |= manSelectPressed =  B .

op side_failed : Nat -> Prop [ctor] .
eq {CONF [port s1F from e has value data(B)]} |= side 1 failed = B .
eq {CONF [port s2F from e has value data(B)]} |= side 2 failed = B .

op neitherSideFailed : -> Formula .
eq neitherSideFailed = (~ side 1 failed) /\ (~ side 2 failed) .
```

We combine these into a formula for the assumptions of R1:

```
op noChangeAssumption : -> Formula .
eq noChangeAssumption
 = ~ changeInAvailability /\ ~ manSelectPressed /\ neitherSideFailed .
```

We are now ready to define formally Requirement R1. However, as explained above, in "stable" situations where each side is in local state `side`j`Active` for the same j, it is the passive side that monitors the manual selection and fully available values from the environment. When the passive (or standby) side realizes that the active side should change, it sends its own value to the active side. This value will arrive in the *next* iteration, so there is a round in which both sides are active. The best we can hope for is that they agree either in this round or in the next; furthermore, if one side fails in the next round, then we may still not have an agreement, so the following is the best we can hope for:

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

Indeed, model checking this property returns `true` (in about 0.8 seconds), so the property holds:

```
Maude> (red modelCheck(init, R1) .)
rewrites: 102954 in 829ms cpu (837ms real) (124097 rewrites/second)

result Bool : true
```

Requirement R2. Remember that Requirement R2 says:

[13] Remember that ~ denotes negation of Maude's LTL formulas.

> 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).

This property obviously does not hold as stated; we have explained that the passive side monitors full availability, and hence the change of active side might be delayed by one round. Therefore the fomula `R2a` is the best we can hope for *side 1*:

```
op side_fullyAvailable : Nat -> Prop [ctor] .
eq {CONF [port s1FA from e has value data(B,B2)]} |= side 1 fullyAvailable = B .
eq {CONF [port s2FA from e has value data(B,B2)]} |= side 2 fullyAvailable = B .

op R2a : -> Formula .
eq R2a
 = [] ((noChangeAssumption /\ side 1 fullyAvailable /\ ~ side 2 fullyAvailable)
       -> (~ side 2 active \/ O (noChangeAssumption -> ~ side 2 active))) .
```

Model checking shows (again in 0.8 seconds) that `R2a` holds in our model:

```
Maude> (red modelCheck(init, R2a) .)
rewrites: 101703 in 812ms cpu (814ms real) (125160 rewrites/second)

result Bool : true
```

We have model checked similar formulas for side 2, but the property does not hold. The counterexamples provided by Maude's model checker allowed us to analyze the failures of the property for side 2; it may take as much as four steps to reach the desired state after side 1 is no longer fully available:

1. Start with sides 1 and 2 in their local states `side1Wait` and `side1Failed`, respectively. (Throughout this behavior, the new and old value of `s1FA` are `false` and both values of `s2FA` are `true`, there is no manual selection and no failures.) In this state, side 1 sends a '1' indicating it is active. Furthermore, since side 2 is in state `side1Failed` (side 1 has just recovered from failure, but that information has not yet reached side 2), and not in state `side1Active`, side 2 does *not* monitor the fully available values.
2. In the next state (with the same output from the environment), side 1 goes to state `side1Active`, whereas side 2, having received the value '1' from side 1, goes to state `side2Wait`.
3. In the next state, side 2 finally goes from state `side2Wait` to state `side1Active`.
4. In the following state, side 2, now in state `side1Active` and therefore monitoring the full availability of both sides, realizes that a side change is necessary, and sends the value '2' to side 1.
5. Finally, in the *next* state, the '2' sent by side 2 in the previous step is read by side 1, which prompty changes state from `side1Active` to `side2Active` and sends the number '2' as output.

Hence, the best we can hope for side 1 becoming not fully available is that side 1 becomes inactive in *four* steps or less. This leads to the following requirement R2b for *side 2*:

```
op R2b : -> Formula .
eq R2b
 = [] ((noChangeAssumption /\ side 2 fullyAvailable /\ ~ side 1 fullyAvailable)
       -> (~ side 1 active \/
           O (noChangeAssumption -> (~ side 1 active \/
              O (noChangeAssumption -> (~ side 1 active \/
                 O (noChangeAssumption -> (~ side 1 active \/
                    O (noChangeAssumption -> ~ side 1 active)))))))) .
```

Model checking this property returns `true` (in 0.8 seconds). The reason for the difference in sides is probably due to the fact that the sides are asymmetric in their failure recovery. After a failure, there is bias towards side 1 being the active side. We therefore could end up in the states (`side1Wait`, `side1Failed`) after failure of side 1 whereas we may not end up in the symmetric state after side 2 is repaired.

Requirement R3. Requirement R3 states the following:

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

This is a problematic requirement. First of all, it is unclear what is meant by *"The pilot can always change the active side."* It is obvious that the pilot can always press the switch; it is equally obvious that there are many states in which such a request is just ignored, because it contradicts the requirement that if one component is fully available and the other one is not, then the fully available side should be the active side.

In addition, the property cannot be expressed as an LTL requirement, since the pilot never *has to* change the active side. (However, if we can formalize "pilot changes active side," then the property can be expressed as a *CTL* property $\forall\Box(\exists\Diamond$ "pilot changes active side").)

Given that it is trivial to see that the environment always can generate a *manual selection* event, we interpret requirement R3 as follows:

> If both sides agree on the active side, both sides are fully available (and therefore there are no failures), and then the manual selection is activated (and there is still no lack of availability), then the active side should change either immediately, or, at latest, in the next round (unless there is lack of availability).

This interpretation can be formalized as the following LTL formula R3a:

```
op R3a : -> Formula .
eq R3a = [] ((side 1 fullyAvailable
```

```
      /\ side 2 fullyAvailable /\ agreeOnActiveSide)
    -> (  (side 1 active
           -> O ((manSelectPressed /\ side 1 fullyAvailable
                  /\ side 2 fullyAvailable)
                 -> (side 2 active
                     \/ O (noChangeAssumption -> side 2 active))))
       /\  (side 2 active
           -> O ((manSelectPressed /\ side 1 fullyAvailable
                  /\ side 2 fullyAvailable)
                 -> (side 1 active
                     \/ O (noChangeAssumption -> side 1 active))))) .
```

However, model checking this formula returns a counterexample, in which both sides *continue* to agree that side 1 is the active side, even though there are no failures when the pilot presses the button. (Adding more next-state disjunctions will not help.) Briefly stated, the source of the problem is the following:

- In most circumstances, if the system gets a manual selection request, this request will be "rcorded," and the following consecutive manual selection requests will be ignored.
- When some component is not fully available, the system cannot take always obey the pilot's desire to switch the active side. *However, even if the system cannot treat the manual selection event, it remembers in the attribute* `prevManualSwitch` *that the manual switch was requested.*

The path provided by Real-Time Maude's model checker as a counterexample to the validity of the above LTL property shows that the pilot makes a manual switch request when a side is not fully available (and hence the switch of active sides is not effectuated). In the next round, all components are OK, and the pilot again requests a switch of active sides. But, this last request is ignored since the system registered that the pilot pressed the manual selection in the previous round. All following consecutive manual requests will also be ignored.

In more detail, assume that both sides are in local state `side1Active`. In this case, side 2 should observe the availability of the sides and the manual selections, and should initiate a change of active sides if such a change should be performed. Now, assume that side 2 becomes *not* fully available at the same time when the pilot requests the manual switch. Consider the definition of the transition for this case:

```
ceq performTrans(< side2 : Side2 | state : side1Active, prevS1AS : I2,
                                   prevManualSwitch : B2 >
                 [port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 [port s1FA from e has value data(B4)]
                 [port s2FA from e has value data(B3)]
                 REST)
  =
    < side2 : Side2 | prevManualSwitch : B1, prevS1AS : I,
```

```
                     nexts2as : data(1) >
     performTrans([port s2F from e has value data(false)]
                  [port s1AS from side1 has value data(I)]
                  [port mS from e has value data(B1)]
                  [port s1FA from e has value data(B4)]
                  [port s2FA from e has value data(B3)]
                  REST)
   if I =/= 0 and ((not B3) or (B4 and (B2 or (not B1)))) .
```

The pilot has requested a manual switch (i.e., `B1` is `true`), hence `prevManualSwitch` is set to `true`, and side 2 is not fully available (hence `B3` has value `false`, and the condition holds, since `I` equals 1). Therefore, in essence nothing changes, which is correct, since the system should *not* switch to side 2 when side 2 is not fully available. However, in the *next* round, all sides are fully available, and the pilot again wants to manually switch the active side. The above equation again applies, but this time because `prevManualSwitch` (`B2`) now equals `true`. And, of course, this equation does not change the active side.

Since the problem above was that a manual switch request happened when a component is not fully available, we have weakened the above LTL property by adding a conjunct `~ manSelectPressed` to the main premise, so that the manual switch request in a non-error state should *not* directly follow a manual switch request in a non-perfect state. However, model checking this weaker property also returned a counterexample, from which we could extract the following problem: Side 1 is active according to both sides, but side 2 is in local state `side2Wait`, as it has just recovered from a failure of side 2. However, there are no failures in the state (allowing side 2 to recover to state `side2Wait`). Also in the next state, every component is fully available, and the manual selection is pressed. Side 2 continues its recovery according to the equation

```
ceq performTrans(< side2 : Side2 | state : side2Wait >
                 [port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
  =
    < side2 : Side2 | state : side1Active, prevManualSwitch : B1,
                      prevS1AS : I,  nexts2as : data(1) >
    performTrans([port s2F from e has value data(false)]
                 [port s1AS from side1 has value data(I)]
                 [port mS from e has value data(B1)]
                 REST)
   if I =/= 0 .
```

This does what it is supposed to do: side 2 does not fail, and when it gets the '1' from side 1, it goes to state `side1Active`. However, it also records the fact that a manual selection request was received, but it does not act on it, leading to the problems described in the previous counterexample.

We have model checked several variants of Requirement 3, and it seems that the following property `R3g` is the strongest one that holds – except in the initialization phase – in our specification. 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*. (Note, however, that since we have defined the state predicate `side 1 active` to hold iff *side 1* sends a '1' signal to side 2 (and vice versa for `side 2 active`), the previously active side will only be aware of the switch in the *following* round.)

```
op R3g : -> Formula .
eq R3g = [] ( (~ manSelectPressed /\ agreeOnActiveSide
               /\ side 1 fullyAvailable /\ side 2 fullyAvailable
               /\ (O noChangeAssumption))
             -> (   (side 1 active
                      -> O O ( (manSelectPressed /\ side 1 fullyAvailable
                                /\ side 2 fullyAvailable)
                              -> (side 2 active)))
                 /\ (side 2 active
                      -> O O ( (manSelectPressed /\ side 1 fullyAvailable
                                /\ side 2 fullyAvailable)
                              -> (side 1 active))))) .
```

Earlier model checking revealed that this propoerty does hold in the initialization phase, so we start the model checking of the above property in the second state:

```
Maude> (red modelCheck(init, O R3g) .)
rewrites: 102216 in 834ms cpu (840ms real) (122521 rewrites/second)

result Bool : true
```

Requirement R4. Requirement R4 states:

> If a side is failed the other side should become active.

As seen in Fig. 6, only the failed side gets the signal about its failure. A failed side signals the failure to the other side by sending a '0' value to the other side. Since this communication has a one-step delay, the best we can hope for is that the other side becomes active in the *next* state:

```
op R4 : -> Formula .
eq R4 = [] (((side 1 failed /\ ~ side 2 failed)
             -> O (~ side 2 failed -> side 2 active))
           /\ ((side 2 failed /\ ~ side 1 failed)
               -> O (~ side 1 failed -> side 1 active))) .
```

This property holds in our model:

```
Maude> (red modelCheck(init, R4) .)
rewrites: 101597 in 825ms cpu (831ms real) (123055 rewrites/second)

result Bool : true
```

Requirement R5. Requirement R5 states:

> 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:

```
op R5side1 : -> Formula .
eq R5side1
 = [] (((side 1 active /\ side 1 fullyAvailable /\ ~ manSelectPressed)
        -> (side 1 active W (~ side 1 fullyAvailable \/ manSelectPressed)))
      /\ ((side 1 active /\ ~ side 1 fullyAvailable /\ ~ side 2 fullyAvailable
           /\ ~ manSelectPressed /\ ~ side 1 failed)
          -> (side 1 active W
              (side 1 fullyAvailable \/ side 2 fullyAvailable
               \/ manSelectPressed \/ side 1 failed)))) .
```

We have here ignored cases where either the property does not hold (such as when side 2 is fully available whereas side 1 is not) or cases that cannot happen (side 1 and side 2 both failed). This formula also model checks successfully:

```
Maude> (red modelCheck(init, R5side1) .)
rewrites: 101702 in 828ms cpu (831ms real) (122803 rewrites/second)

result Bool : true
```

Side 2 is trickier, 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. 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 : -> Prop [ctor] .
eq {CONF < side2 : Side2 | state : side2Active >} |= s2InStateSide2Active
 = true .

op R5side2X : -> Formula .
eq R5side2X
 = [] (((s2InStateSide2Active /\ side 2 fullyAvailable
         /\ ~ manSelectPressed /\ ~ side 1 failed)
        -> (s2InStateSide2Active W
            (~ side 2 fullyAvailable \/ manSelectPressed \/ side 1 failed)))
```

```
       /\ ((s2InStateSide2Active /\ ~ side 2 fullyAvailable
            /\ ~ side 1 fullyAvailable /\ ~ manSelectPressed
            /\ ~ side 2 failed /\ ~ side 1 failed)
           -> (s2InStateSide2Active W
               (side 2 fullyAvailable \/ side 1 fullyAvailable
                \/ manSelectPressed \/ side 2 failed \/ side 1 failed)))
       /\ ((side 2 active /\ ~ manSelectPressed /\ ~ side 2 failed
            /\ side 1 failed)
           -> (side 2 active W
               (manSelectPressed \/ side 2 failed \/ ~ side 1 failed)))) .
```

This property also model checks successfully in less than a second

```
Maude> (red modelCheck(init, R5side2X) .)
rewrites: 102073 in 837ms cpu (845ms real) (121841 rewrites/second)

result Bool : true
```