A Formal Architecture Pattern for Real-Time Distributed Systems

Abdullah Al-Nayeem, Mu Sun, Xiaokang Qiu, Lui Sha
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{*aalnaye2, musun, qiu2, lrs*}*@illinois.edu*

Steven P. Miller, Darren D. Cofer
Advanced Technology Center
Rockwell Collins Inc.
Cedar Rapids, IA 52498
{*spmiller, ddcofer*}*@rockwellcollins.com*

***Abstract*—Pattern solutions [1] for software and architectures have significantly reduced design, verification, and validation times by mapping challenging problems into a solved generic problem. In the paper, we present an architecture pattern for ensuring synchronous computation semantics using the PALS protocol [2]. We develop a modeling framework in AADL to automatically transform a synchronous design of a real-time distributed system into an asynchronous design satisfying the PALS protocol. We present a detailed example of how the PALS transformation works for a dual-redundant system. From the example, we also describe the general transformation in terms of intuitively defined AADL semantics. Furthermore, we develop a static analysis checker to find necessary conditions that must be satisfied in order for the PALS transformation to work correctly. The transformations and static checks that we have described are implemented in OSATE using the generated EMF metamodel API for model manipulation.**

***Keywords*-Architecture pattern; logical synchronization; GALS; formal verification; architecture description language**

I. Introduction

A set of generally applicable heuristics can go a long way in reducing the complexity and design time for any software or system. Normally, good heuristics that have stood the test of time become documented patterns for design, so that even amateur designers can bask in the expertise captured in the pattern rules.

In this paper, we focus on a pattern for simplifying the design complexity of hard real-time distributed computations. The main source of difficulties with designing and developing for distributed components is the asynchronous interactions between different nodes. For example, in the avionics community, networked systems are modeled using the *Globally-Asynchronous Locally-Synchronous (GALS)* paradigm [3]. In the GALS design, tasks are locally driven by the same clock. However, the clocks of the networked nodes cannot be perfectly synchronized. Their relative skew can be bounded, but cannot be absolutely eliminated. As a result, the interactions between different nodes become asynchronous. In order to design a system with verifiable distributed properties across these asynchronously executing nodes, a designer must understand all possible state interleaving among different nodes. Even with the help of formal analysis tools such as model checking, this is a nontrivial task. For example, correcting a race condition can easily lead to more unforeseen errors due to the difficulty of comprehending all possible asynchronous interactions.

The difficulties of the asynchronous system design approach can be contrasted with a globally synchronous design approach. In the globally synchronous model, different nodes have perfectly synchronized clocks without any skew. As a result, due to the absence of any non-deterministic asynchronous interactions, this design is conceptually easy to grasp, and designers are less likely to make errors related to race conditions. Of course, the globally synchronous design is not a realistic design for large systems. While system engineers may start the design phase with a globally synchronous model, the design must be modified for the asynchronous architecture. Such modifications are usually non-trivial [3]. Thus, it is desirable to have a systematic way of relating a specification in a synchronous model to a corresponding asynchronous model.

In this paper, we present a design pattern for a correctness preserving transformation of an ideal globally synchronous model into the logically equivalent asynchronous model. This transformation is applicable to hard real-time distributed systems, such as avionics, which are generally engineered with a fault-tolerant real-time network, where reliable delivery of messages in nondeterministic but bounded time is assured by the communication subsystem. The pattern also assumes the clock of each node is synchronized to the global time within a small error.

Our solution is based on our earlier effort of the protocol called *Physically-Asynchronous Logically-Synchronous (PALS)* [2]. The PALS protocol can achieve the logical synchronization between real-time distributed components. In [2], we have proven the logical equivalence between the PALS models and the ideal, globally synchronous models with perfectly synchronized clocks. However, such logical equivalence does not immediately produce correct asynchronous designs. In the absence of any generic analysis framework, each implementation must be checked through a costly validation process to provide guarantees for the system properties in the physically asynchronous model. By generalizing the transformation as a pattern, we develop a generic analysis framework that works for different instantiations. In this case, the structural configuration of the PALS pattern allows designers to directly fit their globally synchronous models in the defined structure without making any modification of the logic and immediately achieve correct asynchronous models. We present a set of design constraints that are necessary to prove the correctness of an instantiation of the PALS pattern.

We give a formal definition of this PALS pattern using SAE Architecture Analysis and Design Language (AADL) [4] based constructs. AADL is an industry-standard architecture description language with precise semantics to model applications and execution platforms. We illustrate our pattern solution using a dual-redundant controller system. We have developed a prototype tool for the PALS pattern instantiation for models in AADL. We have also developed a static checking tool to enforce the design constraints against any inadvertent error after the transformation of the globally synchronous AADL models.

The rest of the paper is organized as follows. In Section II, we briefly describe the original PALS protocol to cover necessary background information. In Section III, we give a concrete example to show an implementation of the PALS protocol. Section IV presents the structure of the general PALS pattern by generalizing the example from Section III. In Section V, we describe the necessary design rules for ensuring correctness of the PALS pattern. In Section VI, we present a prototype framework for PALS model transformations and static analysis. Finally, we conclude the paper with related work and future research directions.

II. The PALS Protocol Background

In this section, we briefly describe the PALS protocol underlying our PALS pattern. Much of this section is to provide intuition how the PALS protocol provides a logically equivalent behavior as the globally synchronous model in an asynchronous environment. In depth details of these rules and formal proof can be found in [2].

A. Application

In a real-time distributed system, each node participates in two types of computation: global computation and local computation. A global computation is a function of distributed system states and requires coordination between distributed nodes. On the other hand, a local computation depends on the local state of the node and does not require synchronization with other nodes. For example, the supervisory control of a flight control system is a real-time hybrid control that 1) periodically adjusts the setpoints of engine speeds and control surface angles, and 2) performs discrete control such as changing the primary controller in a dual redundant flight control system. The supervisory control is a global computation because during mode changes, the views, actions and state transitions of the distributed state machines must be consistent with each other. On the other hand, in a flight control system, servo control at each node is a local computation, which is solely a function of the local tracking errors with respect to its setpoint and does not synchronize with other nodes. Local computations may use a combination of local data and/or a subset of globally consistent views and commands provided by the supervisory control, but do not change the operations of the supervisory control.

If some global computation depends on a local computation, this local computation can be merged as part of the global computation. Using this process, designers can abstract away all local computations. Thus, the PALS pattern can be applied for the global computation using this abstraction.

B. PALS System Parameters

The PALS protocol assumes following system parameters and their bounds.

- *Maximum clock skew* is ϵ with respect to a global clock.
- *Network transmission delay,* μ is bounded, i.e., $0 < \mu_{min} \leq \mu \leq \mu_{max}$.
- *Response time of the global computation task,* α including real-time scheduling, computation, and I/O time: α is bounded, i.e., $0 < \alpha_{min} \leq \alpha \leq \alpha_{max}$.
- *Real-time network queuing (scheduling) delay,* q is bounded, i.e., $0 < q_{min} \leq q \leq q_{max}$.

C. PALS Protocol Rules

The PALS protocol can be informally specified with the following rules.

- *PALS clocks*: The global computation of a node requires coordination with other nodes and thus, requires at least the delay of end-to-end computation and network transmission. Hence the global computation at the PALS protocol must run at a period longer than this delay[1]. Logical clocks, called PALS clocks, which run at this period are defined for each of the N nodes to control the execution of the global computation[2]. For example, the discrete mode changes in the supervisory control can happen only at the time of PALS clock tick. These PALS clocks are not perfectly synchronized, because of the physical asynchrony between nodes. They have a jitter of ϵ with respect to a global PALS clock with no physical clock skew. In [2], we prove the lower bound of the PALS clock period, T given in Equation II.1. The PALS clock period is optimal and long enough to

[1]Local computations are not affected in the protocol and their scheduling effect is abstracted in α.

[2]In the current version of the PALS protocol, these PALS clocks have same period. We will demonstrate different values for the period of these PALS clocks in future versions.

ensure all inputs have arrived from other nodes.

$$T > 2\epsilon + MAX(2\epsilon - \mu_{min}, \alpha_{max} + q_{max}) + \mu_{max} \tag{II.1}$$

- *PALS causality or output hold rule*: Since PALS clocks are not perfectly synchronized, delivering the node output too early may result in the violation of the synchronization logic. Therefore, a node can send its output only a delay of greater than $(2\epsilon - \mu_{min})$ after a PALS clock tick.
- *Environment event input synchronizer*: Nodes receiving common, external inputs must have consistent views. An environment event synchronizer guarantees that these nodes process these asynchronous inputs consistently within the same PALS clock period.

D. Consequences

These rules and parameters of the PALS protocol are used to ensure the logical synchrony between distributed nodes. The two main aspects of ensuring the synchronization are as follows:

- One PALS clock period is defined based on the worst-case communication delay, computation time, queuing delay, and clock skew to ensure that messages arrive well before the next PALS clock tick. This prevents race conditions between message transmission and node computation before the next clock tick.
- Output from a node is delayed for a minimal time based on the maximal skew and minimal message delay. This prevents any violation of causality as a message arriving too early may appear as a message with zero logical delay.

This means that views of inputs are consistent in the PALS protocol across all nodes, as if they were driven by a single perfect system clock running at PALS clock period, i.e. a globally synchronous model. A formal proof of the logical equivalence between the PALS system and a globally synchronous system is given in [2].

III. Example Application

In this section, we use a dual-redundant controller system, which we call the *active standby system* to illustrate the design transformation under the PALS protocol. Although relatively small, this example is representative enough to show the design and verification complexity of any hard real-time distributed system. Miller, et al. [3] presented the limitations of existing methodologies for designing and verifying a similar system. Even for a small system as this one, non-deterministic interleaving of the execution on an asynchronous architecture can create deadlock and race-conditions. We demonstrate how the PALS design transformation can simplify the design and verification process by reducing the likelihood of making design errors related to race-conditions.

AADL [4] provides notations for decomposing a system into components and connectors and specifying how these elements are combined to form a configuration. We define the structural, as well as behavioral specifications of the active standby system using AADL-*v1* constructs.

A. System Description

The active standby system consists of two physically separated controllers: Side1 and Side2 (Figure 1). They are connected by a fault-tolerant real-time network. Each controller periodically receives sensor data from a separate sensor subsystem. Only one controller is desired to operate as the active controller and deliver the control output, while the other controller operates in the hot standby mode. Users can send commands to these controllers for changing the active controller. In this model, both controllers have to synchronize their computations. To illustrate the application of PALS, we will focus on the discrete global computation that performs the synchronization logic between these control sides under the following fault model:

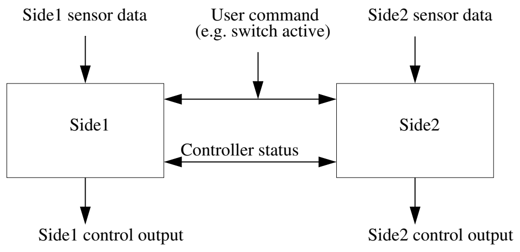


Figure 1. Active standby system.

- *Fail-stop*: Controllers are fail-stop and may recover.
- *No concurrent failure*: The probability of concurrent machine failures is negligible.
- *Full availability of a controller may change*. The full functionality/availability of a control side depends on the accuracy of the sensor subsystems. Each control side can sense the status of the sensors and determine if the sensors are fully available or not.

There are five design requirements. Although these requirements are necessary and realistic, but they are not meant to be the complete set of requirements.

1) *Both controllers should agree on which controller is active.*
2) *A controller that is not fully available should not be the active controller if the other controller is fully available.*
3) *If a controller is failed the other controller should become active.*
4) *The user can always change the active controller.*
5) *In other cases, the active controller should not change, unless its availability changes, or the user requests.*

Requirements 1-3 are necessary to guarantee both fault-tolerance and the consistent agreement between two controllers in a stable state. Requirement 4 allows the user to control the system in a stable state. Finally, requirement 5 is necessary to prevent unnecessary fluctuation of the status. Note that in the active standby configuration, these requirements may be violated if there is a failure or a change in the availability of the active controller. For example, when the active controller fails, it will take one step for the standby to detect the failure and to become active in next step.

B. Globally Synchronous Model

Figure 2 presents the software architecture of the globally synchronous model of the active standby system in AADL[3]. At the high-level, there are three AADL $system$ components: $Side1$, $Side2$, and $InputSynchronizer$.

InputSynchronizer is the environment event input synchronizer which is responsible for supplying asynchronous inputs to Side1 and Side2. Any system/device outside the operation of the active standby system is considered as environment. We use only one input synchronizer to abstract other components of the system. Since all components in the globally synchronous model operate synchronously at lockstep, this aggregation under a single component does not create any semantic difference.

In this model, we are only interested in the logic of deciding which controller is active and relevant asynchronous events, e.g. user command and failure. InputSynchronizer, therefore, supplies following data to Side1 and Side2:

- *switchActive, side1FullyAvailable, side2FullyAvailable*: switchActive is a boolean input used to model a switch command from the user. If $switchActive = true$, then the user commands to switch the active controller. side1FullyAvailable, side2FullyAvailable are two boolean inputs to model the full availability of Side1 and Side2 respectively. These two variables in our model essentially capture the overall statuses of different sensors dedicated for both Side1 and Side2. Without any loss of generality, we assume that the statuses of different sensors are available to both controllers.
- *side1Failed, side2Failed* are two boolean inputs used to inject failure to these controllers. For the sake of modeling, we use InputSynchronizer to generate these inputs non-deterministically.

In this model, both sides exchange their status through the outputs: *side1ActiveSide* and *side2ActiveSide*. They capture the active, standby or failure status of these controllers. For example,

- If $side1ActiveSide = 0$, then Side1 is failed. Side2 can then take appropriate action observing this value.
- If Side1 is active, Side1 outputs $side1ActiveSide = 1$.
- If Side2 is active, Side1 outputs $side1ActiveSide = 2$.

[3]The underlying hardware architecture can also be added using AADL components for processors, bus, memory, etc.

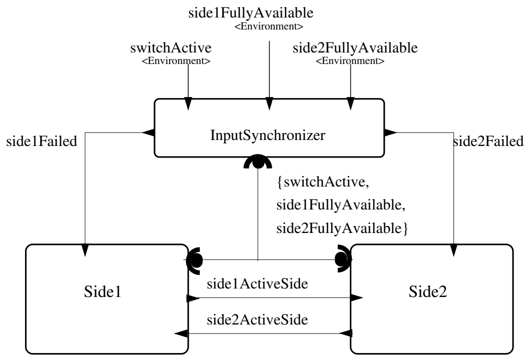


Figure 2. Active standby system in AADL.

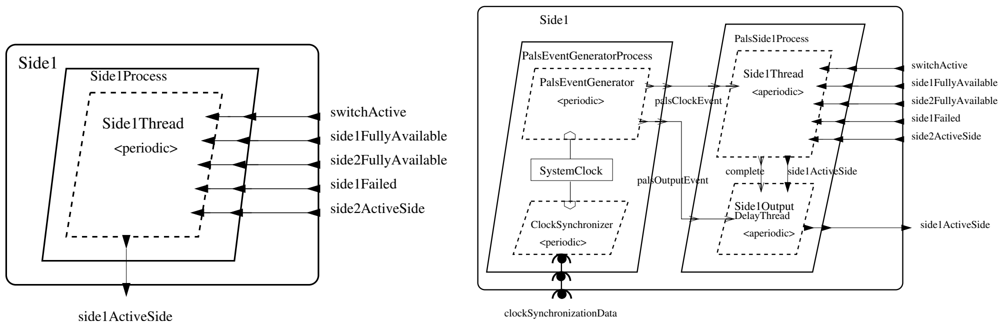
Thus, when both sides agree on which one is active (Requirement 1), $side1ActiveSide = side2ActiveSide$.

In the globally synchronous model, message transfer is guaranteed to be delivered at the next clock time. Therefore, there is no need for buffering any data. AADL *data port* can be used to capture this communication model.

1) Behavior of each system: In this model, there is a separate but perfectly synchronized clock for each system. We model the global computation of these systems using the AADL executable component, called $thread$. For example, *Side1Thread* is the global computation thread at Side1 in Figure 3(a). The global computation threads of Side1 and Side2 implement the desired system behavior based on the inputs generated by the thread in InputSynchronizer. We model the behavior of these threads using the state machine model. In the globally synchronous model, these distributed state machines perform a state transition periodically at lockstep. In the absence of non-deterministic interactions, these state machines can be easily defined and verified. The detail AADL model of these components and state machines can be found at *https://agora.cs.illinois.edu/download/attachments/9527/ActiveStandby.zip*. We have also automatically translated the AADL specification into Real-Time Maude [5] to verify the requirements. This automatic translation and verification with Real-Time Maude is an ongoing work and out of the scope for this paper.

C. Asynchronous PALS Model

In the PALS model, we would like to preserve the easily verifiable logic of the globally synchronous model. The PALS model has two parts, PALS event generation and global computation, which together implement the PALS rules. This separation of the PALS clock generation and the global computation ensures modularity in the design. Designers can also use this approach to implement the PALS clock generation as a part of the middleware development. Figure 3(b) illustrates of the PALS model of Side1.



(a) Globally synchronous model

(b) PALS model

Figure 3. Side1 in AADL

1) PALS event generation: According to the PALS protocol, the global computation at each node is performed at the PALS clock time. Also, the protocol requires that the output cannot be delivered before a certain period (PALS causality rule). These two rules can be enforced by two PALS events: the PALS clock event and the PALS output event. Before describing the global computation, let us first describe the generation of these events.

PALS models are applicable to systems with bounded clock skew. To guarantee the correctness of the PALS protocol, PALS clock events at different nodes must also be generated with ϵ-jitter of the ideal PALS clock. We achieve this synchronization using the combination of system clock synchronization and periodic generation of PALS events.

In our active standby model, both clock synchronization and PALS event generation are abstracted using an AADL $process$, named $PalsEventGeneratorProcess$ (Figure 3(b)). PalsEventGeneratorProcess consists of two periodic threads: $ClockSynchronizer$ and $PalsEventGenerator$.

ClockSynchronizer periodically resynchronizes the system clock with other nodes using a set of relevant data called, $clockSynchronizationData$. Since the system clock synchronization is not the focus of the paper, we model the ClockSynchronizer as a generic component. System clock synchronization is a well-established research topic in distributed systems [6]. ClockSynchronizer can be instantiated from any of these algorithms. Based on this clock synchronization, the PalsEventGenerator thread periodically generates two relevant events $palsClockEvent$ and $palsOutputEvent$ with expected jitter. palsClockEvent is the PALS clock event which is generated periodically with a period greater than the bound mentioned in Section II. On the other hand, palsOutputEvent is generated after $H = (2\epsilon - \mu_{min} - \Delta)$ to ensure that output is delivered according to the PALS causality rule. Δ will be explained shortly.

2) Global computation: In the globally synchronous model, we model the global computation of different components as a periodic thread, e.g. Side1Thread for Side1 in Figure 3(a). While we use the same synchronous global computation logic, the global computation threads are aperiodic in the PALS model and dispatched by the PALS clock event, palsClockEvent (Figure 3(b)).

In order to enforce the PALS causality rule, outputs of a global computation thread are delivered to an output delay thread. For example, the output of SideThread is sent to its output delay thread, *Side1OutputDelayThread*. The output delay thread is modeled as an AADL aperiodic $thread$ component. In AADL, each thread completion is signaled by a $Complete$ event. The $Complete$ event from the global computation thread, e.g. Side1Thread, is received at the $computationComplete$ event port of the output delay thread. Thus, the output delay thread can be dispatched by either the palsOutputEvent event from the PalsEventGenerator or the $Complete$ event from the global computation thread.

The output is delivered from the output delay thread only after both palsOutputEvent and computationComplete have arrived. For example, if palsOutputEvent arrives before the completion of the Side1Thread, i.e. the *Complete* event, then the output, side1ActiveSide is held. It is delivered after the *Complete* event arrives. We can model the behavior of the output delay thread using the following state machine model:

States: $S_{out} = s_0, s_1, s_2$
State Transitions:

- $s_0 - [palsOutputEvent?] \rightarrow s_1$
  Action: None
- $s_0 - [computationComplete?] \rightarrow s_2$
  Action: None
- $s_1 - [computationComplete?] \rightarrow s_0$
  Action: send output, e.g. side1ActiveSide
- $s_2 - [palsOutputEvent?] \rightarrow s_0$
  Action: send output, e.g. side1ActiveSide

3) Event propagation: Figure 4 shows the event propagation in the PALS model. PalsEventGenerator of each node generates two events, palsClockEvent and palsOutputEvent. In order to reduce the scheduling jitter of these events, we assume that PalsEventGenerator is implemented with the highest priority. There is an output delay thread for the global computation thread to enforce the PALS causality rule. Because of the scheduling delay of the output delay thread, PalsEventGenerator can send the palsOutputEvent event Δ time earlier and yet satisfy the PALS causality rule. Δ is a function of the minimum the queuing delay which includes the response time of the output delay thread.

In Figure 4, we mark the response times as the grey region. Once the Side1Thread completes its execution, its $Complete$ event propagates to the Side1OutputDelayThread which then sends output over the network. The PALS period, T guarantees that the output message will reach Side2 before its next palsClockEvent. In this way, we can have a logically equivalent PALS implementation of the globally synchronous model.

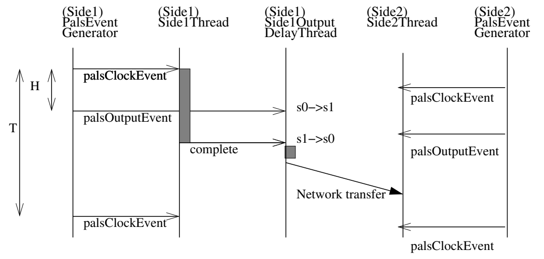


Figure 4. Sequence of events in the PALS model

IV. System Models

The example in the previous section illustrates the fault-tolerant design practice of replication. In the fault-tolerant design, components are normally replicated in a dual, triple or quadruple manner in which all components must be synchronized in terms of the states of operation. We have illustrated a concrete example of the PALS pattern for the case of dual-redundancy in the previous section. In this section, we consider the PALS pattern in a more general and precise fashion. It is desirable to show that the model transformation does indeed give the desired results of synchronous computation. We first define the semantics of various constructs similar to the ones used in AADL. After we define the semantics, we describe exactly what we mean by having synchronous computation, and show that the current model transformation in AADL provides synchronous semantics of execution for any distributed system satisfying the PALS preconditions.

A. System Model Definition

In this section, we describe the abstract model of a system. We try to keep a close mapping between our model constructs and AADL constructs.

A global configuration, G consists of a set of components, $Threads_G$, and a set of connections or links $Connections_G$; $G = (Threads_G, Connections_G)$. We will drop the subscript G when it is clear from context. Furthermore, let $Values$ be the set of data values used for communication.

A component $C \in Threads$ consists of a set of input and output ports, $Ports_C^{in}$ and $Ports_C^{out}$ respectively. Also, component C has a set of states $State_C$ and a transition function $trans_C : Values^{n_{in}} \times State_C \rightarrow State_C \times Values^{n_{out}}$; $C = (Ports_C^{in}, Ports_C^{out}, State_C, trans_C)$. n_{in} and n_{out} are the number of input and output ports for the component C. Again, we drop the subscript C, when it is clear from context.

For a global configuration G, we safely assume that all port names are unique, and we define the set of all input ports, the set of all output ports, and the set of all ports to be $Ports_G^{in} = \bigcup_{C \in G} Ports_C^{in}$, $Ports_G^{out} = \bigcup_{C \in G} Ports_C^{out}$, and $Ports_G = Ports_G^{out} \cup Ports_G^{in}$ respectively. Furthermore, we define a function $value : Ports_G \rightarrow Values$ which gives the actual data value on a port. For values of ports, they may change over time, so we intuitively use the operator $value(p)\ at\ time\ t$ to denote the value of data port p at the time t.

Finally, the connections or links $Connections$ is a relation from $Ports_G^{out}$ to $Ports_G^{in}$. Also, a reasonable configuration will not have multiple output ports connecting to the same input port (this is also enforced in AADL)[4], and every port is connected, so the connection relation $Connections$ can be fully captured by a function describing the source output port for every input port, $src : Ports_G^{in} \rightarrow Ports_G^{out}$.

We let $Time$ be the set representing the points in time. For purposes of this paper, time is represented by positive real numbers with 0 being system initialization time; $Time = \mathbb{R}^{\geq 0}$. We drive components in our systems by clocks. A clock is a monotonically nondecreasing function $c : Time \rightarrow \mathbb{N}$ mapping some global time to a set of clock ticks. Define $t_c : \mathbb{N} \rightarrow Time$ to be the function $t_c(i) = \inf\{x | c(x) = i\}$. We reasonably assume t_c always exists and that $c \circ t_c = 1_{\mathbb{N}}$ (i.e. c is surjective and the intervals of a clock tick are left closed and right open). Let $Clock$ be set of all valid clocks. We define a function $clock$ to associate every component to a clock, $clock : Threads \rightarrow Clock$. For a PALS system to be operate correctly, we assume that any two clocks in the system c and c' must be synchronized to extent that for all clock ticks i, $|t_c(i) - t_{c'}(i)| < 2\epsilon$. Also, clocks must advance time at some minimal rate $|t_c(i+1) - t_c(i)| > \tau$.

B. Synchronous Execution Semantics

Now, the question is, what is a definition of synchronous execution semantics that is general enough to capture the concept in the PALS pattern.

For all $i \in \mathbb{N}$, let c_i define a sequence of clock values such that $c_i < c_{i+1}$. A synchronous execution at the points c_i (called clock events) is defined as:

- All inputs and their source outputs show the same values on each clock tick c_i on different machines.

[4]This can happen in different modes. We are only considering only one mode of operation.

More precisely; let p be an input port in component C; and let $src(p)$ be its connected output port in component C':

$$value(p) \ at \ time \ t_{clock(C)}(c_i) \\ = value(src(p)) \ at \ time \ t_{clock(C')}(c_i) \qquad \text{(IV.1)}$$

- All system state and output at clock tick c_{i+1} can be obtained from the state transition function *trans* of the input and system state at clock tick c_i. More precisely, for component C, let V^{in} and V^{out} be the set of input and output values; and let $s \in State_C$ be the state of C:

$$(V^{out} \ at \ time \ t_c(c_{i+1}), s \ at \ time \ t_c(c_{i+1})) \\ = trans_C(V^{in} \ at \ time \ t_c(c_i), s \ at \ time \ t_c(c_i)) \qquad \text{(IV.2)}$$

C. The PALS Pattern Transformation

We have already seen an example of how to create a PALS architecture for the *active standby* system. We proceed to describe the actual PALS transformation in general. It may help to refer back to the example for a concrete instantiation.

The PALS transformation is a mapping $\mathcal{T}_{PALS} : \mathcal{G} \rightarrow \mathcal{G}$ where $\mathcal{G}$ is the set of all possible global systems. Given a global configuration for an ideal synchronous system G_{sync}, the PALS transformation takes this as input and gives an asynchronous system $\mathcal{T}_{PALS}(G_{sync})$ which should be logically equivalent. The details of the transformation for our AADL specification are as follows (for notation we liberally extend $\mathcal{T}_{PALS}$ to the subcomponents of G):

- For each component $C \in Threads_{sync}$, we transform C into two components in $\mathcal{T}_{PALS}(C)$, C_{comp} and C_{out}, such that $C_{comp} = C$ (a copy of all the computation of C) and C_{out} is the output delay thread that forwards outputs from C_{comp} after a holding delay. This is the same as the Side1OutputDelayThread described in Section III. Notice that C_{out} contains a copy of all of C_{comp}'s output ports for both input and output, so for each output port $p \in Ports^{out}_{C_{comp}}$, we use the notation $p^{fwd}_i \in Ports^{in}_{C_{out}}$ and $p^{fwd}_o \in Ports^{out}_{C_{out}}$ for the corresponding input and output ports of C_{out}. Finally, it is assumed that C_{comp} and C_{out} run on the same clock. That is $clock(C_{comp}) = clock(C_{out})$ and thus, completely synchronized.
- For each connection link $(p, p') \in Connections_{sync}$, we transform all direct connections to go through the output forwarding component. Thus, we have the links $(p, p^{fwd}_i), (p^{fwd}_o, p') \in \mathcal{T}_{PALS}(Connections_{sync})$.
- Thread C_{comp} is dispatched on the PALS clock ticks iK, called the palsClockEvent, and thread C_{out} receive an output event at some delay, $H = (2\epsilon - \mu_{min} - \Delta)$ after the palsClockEvent, called the palsOutputEvent; It also dispatches on a computationCompleteEvent generated by the corresponding computation thread. Δ is the function of minimum queuing delay, so that delivering the palsOutputEvent event Δ time earlier does not violate the PALS causality rule.

V. Design Rules

The PALS pattern shows a generic architecture for mapping a globally synchronous design into an asynchronous design. The design and verification of a global computation is separated from different local computations. But in an actual system, the global computation is integrated with these additional computations. Checking the correctness of the PALS pattern instantiation is important to prevent any inadvertent error resulting in the integrated model. A generic analysis for such purpose can be developed by establishing a set of correctness-enforcing design rules.

In this section, we describe a static analysis framework based on a set of structural constraints necessary to enforce the rules of the PALS protocol.

A. Structural Constraints

For any system, only a subset of components, performing global computations, require synchronous execution semantics. The other components can be treated as environment. The constraints are relevant to the global computation components.

We define a set of predicates to be used in the constraints:

- *SynchronousComputation(C)* returns true iff C is a global computation component, $C = C'_{comp}$, for some component $C'_{comp} \in Threads_{sync}$ input to the PALS transformation.
- *InputSynchronizer(C)* returns true iff *SynchronousComputation(C)=true* and it receives input from the environment.
- *OutputDelayComputation(C)* returns true iff C is an output delay component, $C = C'_{out}$, for some component $C' \in Threads_{sync}$ input to the PALS transformation.
- *Context(p)* returns the component C containing port p such that $p \in Ports^{in}_C \cup Ports^{out}_C$.
- *PALSClockDispatch(C)* returns true iff component C dispatches on the palsClockEvent, iK on $clock(C)$.
- *PALSOutputDispatch(C)* returns true iff component C dispatches on the palsOutputEvent, iK with delay H on $clock(C)$.
- *ComputationCompleteDispatch(C)* returns true iff component C dispatches on a computationComplete event.

Table I enumerates different structural constraints necessary for correct PALS implementation.

- *Equation V.1*: The global computations must be dispatched by the PALS clock events. Any component which is a SynchronousComputation must be dispatched by the event, palsClockEvent. According to the PALS rule, palsClockEvent in different nodes must

$$\forall_{C \in Threads}\ SynchronousComputation(C) \rightarrow PALSClockDispatch(C) \tag{V.1}$$

$$\forall_{C \in Threads}\ OutputDelayComputation(C) \rightarrow PALSOutputDispatch(C) \wedge ComputationCompleteDispatch(C) \tag{V.2}$$

$$\forall_{C \in Threads}\ SynchronousComputation(C) \wedge \neg InputSynchronizer(C) \rightarrow \forall_{p \in Ports^{in}_{C}}\ OutputDelayComputation(Context(src(p))) \tag{V.3}$$

$$\forall_{C \in Threads}\ SynchronousComputation(C) \rightarrow (\forall_{p \in Ports^{out}_{C}}\ \forall_{p' \in Ports^{in}}\ (p == src(p') \rightarrow OutputDelayComputation(Context(p'))) \tag{V.4}$$

Table I
STRUCTURAL CONSTRAINTS

be synchronized with ϵ-jitter with respect to the ideal PALS clock. Such correctness can be ensured by verifying the correctness of the clock synchronization algorithm.

- *Equation V.2*: The PALS pattern enforces the output hold rule or the PALS causality rule by using the output delay component, which is dispatched by the events palsOutputEvent and computationComplete.
- *Equation V.3*: Except for the InputSynchronizer, any SynchronousComputation component must interact with other SynchronousComputation components according to the PALS rules. This can be ensured by checking whether inputs to the SynchronousComputation component arrive from an OutputDelayComputation component.
- *Equation V.4*: The output of a SynchronousComputation component must go through the OutputDelayComputation component.

B. Necessity of Static Analysis Rules

We show that each of the static analysis rules presented earlier are necessary for a correct implementation of the PALS protocol. If any of the structural constraints are violated, counter examples will exist to show that the global computation does not satisfy globally synchronous model semantics.

The first two constraints specified, V.1 and V.2, although simple, are quite important since we must make sure that components are asynchronously dispatched on the correct events. The global computation component must be dispatched by the PALS clock event (palsClockEvent). Its output must be delivered after certain delay which is enforced by the computationComplete event (from the global computation component) and the PALS output event.

The last two constraints, V.3 and V.4, are dual constraints to require that all communication between two components involved in the global computation go through the output delay components. Recall that one of the conditions for the definition of synchronous execution presented earlier requires that the values seen at connected input and output ports are the same at PALS clock ticks. In Figure 5, we consider two components, $C1$ and $C2$, and $C1$'s output delay thread $C1_{out}$ at $palsClockEvent$ at clock tick iK. The two constraints are complementary constraints saying that no communication occurs directly between component $C1$ and $C2$. If communication can occur directly, and the clock skew is larger than the response time, the network queuing and transfer delay, the PALS causality rule will be violated. The output at $palsClockEvent_1$ for $C1$ will not match the input at $palsClockEvent_2$ for $C2$. That is for some port $p \in Ports^{in}_{C2}$, we have:

$$\begin{aligned} &value(p)\ at\ time\ t_{clock(C2)}(iK) \\ &\neq value(src(p))\ at\ time\ t_{clock(C1)}(iK) \end{aligned}$$

violating the synchronous execution constraint defined by Equation IV.1. The message line labeled "without $C1_{out}$" in Figure 5 shows how this error can occur with direct communication. The message line labeled "with $C1_{out}$" is the modified system in which the communication must go through the output delay component. Because of the bounds given for the PALS protocol in [2], values at inputs and outputs indeed match at each PALS clock tick.

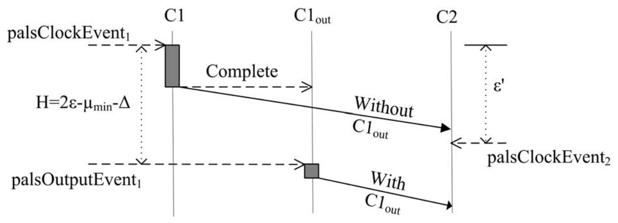


Figure 5. Counter example if Equation V.3 and V.4 are violated

VI. DESIGN TOOLS FOR PALS

We have developed a set of prototype design tools for the PALS pattern design process. The first tool is for automatic transformation of a globally synchronous design into a PALS design. The second tool is to perform static analysis of the PALS pattern instantiation. Both of these tools work on the AADL models and are developed using the Open Source AADL Tool Environment (OSATE) API. OSATE is the development environment for AADL-based model development and analysis [7]. It has been built as a set of plug-ins in the open source Eclipse platform. AADL models are defined in the Ecore meta-modeling notation of the Eclipse Modeling Framework (EMF) [8]. Both OSATE and Eclipse support libraries to manipulate the EMF models used in AADL. We use these libraries for both model transformation and static analysis.

A. Mechanical Transformation of a Globally Synchronous Model

The first prototype tool is developed as an OSATE plug-in, which transforms each global computation component of the globally synchronous model. It generates both the architecture and behavioral models for the PALS design as shown in Section IV. The plug-in receives an AADL model for each synchronous, global computation component as an input and transforms the model to generate the AADL instance of the PALS pattern. To assist the process, we define an AADL *property*, called *Pals::Computation*. Each synchronous global computation component must define this property with the value set to '*SynchronousComputation*'. The plug-in identifies the synchronous component based on this property and generates both the PALS event generator and the PALS computation process, e.g. PalsEventGeneratorProcess and PalsSide1Process of Figure 3(b). The AADL code generation of the PalsEventGeneratorProcess is kept as a generic component. The generated PalsEventGeneratorProcess must be extended with an explicit clock synchronization algorithm. In the generated model, PalsEventGeneratorProcess supplies palsClockEvent and palsOutputEvent to the global computation component and the output delay component respectively.

B. Static Checker of PALS Constraints

Now we describe the static analysis of instantiated AADL models after the transformation. The properties of the pattern must still hold, even though designers can integrate these models with other system components. The static analysis tool checks the constraints of Table I. Each predicate is validated based on the properties defined by the AADL model after transformation. Each component in the generated PALS design is distinguished by the property, *Pals::Computation*. Each port of the global computation component and the output delay component is also defined by the property, *Pals::EventPort*. The values for these properties are used as a basis for detecting any violation of the design constraints.

Since AADL models are designed hierarchically, the static checker starts its operation from the top-level system component. It traverses through the component hierarchy and identifies the relevant component based on the Pals::Computation property. For example, in order to validate the constraints V.1, V.3, and V.4, the tool identifies a global computation component and checks the port semantics to see whether any of these constraints is violated.

VII. Related Work

Architecture and design patterns [1] [9] encapsulate good software engineering practices. These patterns are generally useful for improving modularity, portability, and performance. The PALS pattern provides a simple method to reduce the design and verification complexity of real-time distributed systems by correctly transforming globally synchronous models into corresponding asynchronous models.

Modeling and verifying asynchronous systems have been an active area of research for many years. Designers traditionally use synchronous design tools/languages, such as Simulink, SCADE, Lustre, for modeling these systems. However, these tools/languages were originally intended for systems with a global clock. Several works [10] [11] have proposed a framework to simulate the asynchronous behavior of different asynchronous systems in a synchronous language. In their approach, nondeterministic asynchronous behavior is simulated by using auxiliary inputs and sporadic execution of processes. While these techniques are useful for verifying asynchronous systems, they are not sufficient due to combinatorial state interleaving by communication variations and complex interactions. In this respect, the PALS pattern can complement these tools and techniques. Synchronous design tools/languages can be used to design and verify the globally synchronous models. After the PALS transformation, the techniques of [10] and [11] can be used to model the physical asynchrony of the PALS design.

Tripakis et al. [12] also deals as ours with the problem of mapping a synchronous computation on an asynchronous architecture. In their approach, they consider a loosely timed-triggered architecture (LTTA) and the mapping is achieved through an intermediate finite FIFO platform (FFP) layer. Synchronization is achieved by an application node skipping (taking no action) until all its inputs have arrived and all its outputs can be written. In some sense, their result shows the robustness of their mapping, since correctness is achieved in spite of unpredictable communication delays and highly different clock rates in the different processes. However, due to the minimal assumptions made about the asynchronous architecture, their approach does not provide the hard real-time guarantees provided by PALS. Also, their use of acknowledgments may lead to greater communication overhead than in PALS.

The PALS pattern also shares many similarities with the time-triggered architecture, such as TTA [13]. Time-triggered architectures address the issue of global state synchronization through the introduction of a single global clock implemented in the system bus. Rushby [14] compared different time-triggered bus architectures. Many of these bus architectures have built-in hardware support for the development of fault-tolerant applications. These solutions assume that the hosts and the networks are replicated. They support the basic service of guaranteeing consistent message transfer to make sure that replicated hosts maintain consistent state. However, they support synchronization and fault-tolerance at a very low level and require specialized components to implement the services. In contrast, PALS is a time-based synchronization pattern built on top of a fault-tolerant real-time network, where reliable delivery of real-time message is assured by the communication subsystem. This allows PALS to be implemented over a wide variety of distributed architectures, including the event-triggered architectures so long as the PALS assumptions are satisfied. Moreover, the PALS pattern does not require tight network synchronization of the nodes as compared to a time-triggered architecture.

The PALS pattern separates the real-time synchrony mechanism from the application logic, including the fault-tolerance mechanism as shown Section III. Many protocols have been developed to provide distributed processes with consistent views in general-purpose distributed systems. Birman and Joseph [15] introduced the process group abstraction to achieve fault-tolerant virtual synchrony for general-purpose computation. Abdelzaher, et al. [16] developed a fault-tolerant multicast and membership service for real-time process groups. In these process group approaches, synchrony management and fault-tolerance are bundled together. This is a good idea for many applications. However, in safety-critical applications such as avionics, a system has subsystems with different levels of reliability requirements. For example, avionics certification standard DO178B defines 5 design assurance levels [17]. On the other hand, the PALS pattern allows designers to combine the synchronization mechanism with preferred fault-tolerance mechanisms to meet different reliability requirements.

VIII. Conclusion

This paper introduces an architecture pattern for reducing the design, verification and validation complexity of real-time distributed systems. Design and verification is difficult in general in these systems due to the nondeterministic state interleaving resulting from asynchronous interactions. We demonstrated a semantics-preserving mapping of synchronous designs to asynchronous architectures that can drastically reduce this verification overhead.

Ongoing work is being done on automatically translating AADL models into Real-Time Maude [5] for model checking. Furthermore, our focus currently is on supporting the design transformations as a middleware without any coupling to a concrete implementation. We are also working on the composition of global computations executing at different PALS clock periods. Of course, with emerging practices of the model-driven design, there is also much fruitful work to be done on generating implementation from models themselves (e.g. the model output from the PALS transformation).

Acknowledgment

The authors would like to thank José Meseguer, Peter Ölveczky, Dionisio de Niz, Min Young Nam, Jean-Paul Bodeveix, Sibin Mohan and Heechul Yun for their help and suggestions. This work is sponsored in part by Rockwell Collins Inc., the Office of Naval Research, the National Science Foundation, and Lockheed Martin Corporation.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] L. Sha, A. Al-Nayeem, M. Sun, J. Meseguer, and P. C. Ölveczky, "PALS: Physically Asynchronous Logically Synchronous Systems," University of Illinois at Urbana-Champaign, http://hdl.handle.net/2142/11897, Tech. Rep., 2009.

[3] S. P. Miller, M. W. Whalen, D. OBrien, M. P. Heimdahl, and A. Joshi, "A Methodology for the Design and Verification of Globally Asynchronous/Locally Synchronous Architectures." NASA Contractor Report CR-2005-213912, 2005.

[4] Society of Automotive Engineers, "SAE Standards: Architecture Analysis & Design Language (AADL)," *AS5506*, 2004.

[5] P. C. Ölveczky and J. Meseguer, "Semantics and Pragmatics of Real-Time Maude," *Higher-Order and Symbolic Computation*, vol. 20, no. 1-2, pp. 161–196, 2007.

[6] N. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers Inc: San Fransisco, CA, 1996.

[7] OSATE, "www.aadl.info."

[8] Eclipse Modeling Framework, "www.eclipse.org/modeling/emf/."

[9] B. P. Douglass, *Real-time Design Patterns Robust Scalable Architecture for Real-time Systems*. Addison-Wesley, 2006.

[10] N. Halbwachs and L. Mandel, "Simulation and Verification of Asynchronous Systems by means of a Synchronous Model," in *Proceedings of the 6th International Conference on Application of Concurrency to System Design*, 2006.

[11] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens, "Virtual Execution of AADL Models via a Translation into Synchronous Programs," in *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, 2007.

[12] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. D. Natale, "Implementing Synchronous Models on Loosely Time Triggered Architectures," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1300–1314, 2008.

[13] H. Kopetz, "The Time-Triggered Architecture," in *Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 1998.

[14] J. Rushby, "A Comparison of Bus Architectures for Safety-Critical Embedded Systems." NASA Contractor Report CR-2003-212161, 2001.

[15] K. Birman and T. Joseph, "Exploiting Virtual Synchrony in Distributed Systems," *ACM SIGOPS Operating Systems Review*, vol. 21, no. 5, pp. 123–138, 1987.

[16] T. Abdelzaher, A. Shaikh, S. Johnson, F. Jahanian, and K. Shin, "RTCAST: Lightweight Multicast for Real-Time Process Groups," in *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, 1996.

[17] RTCA, "DO-178B - Software Considerations in Airborne Systems and Equipment Certification," *RTCA Inc.*, 1992.