Pattern-based Composition and Analysis of Virtually Synchronized Real-Time Distributed Systems

Abdullah Al-Nayeem, Lui Sha
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{*aalnaye2, lrs*}*@illinois.edu*

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

***Abstract*—Designing and verifying distributed protocols in a multi-rate asynchronous system is, in general, extremely difficult when the distributed computations require consistent input views, consistent actions and synchronized state transitions. In this paper, we address this problem and introduce a formal, complexity-reducing architectural pattern, called *Multi-rate PALS* system, to support virtual synchronization in multi-rate distributed computations. The pattern supports a component to be virtually synchronized with other components in different instantiations of this pattern. We present an application of a hierarchical control system to show that the composition of these instantiations can be used to achieve desired system-level properties, such as distributed consistency and distributed coordination. We verify the logical synchronization guarantee of this pattern which holds as long as the pattern assumptions are satisfied. We also discuss the correctness analysis necessary to validate these assumptions and provide a tool support to perform this analysis automatically on the AADL models.**

***Keywords*-Design patterns; virtual synchronization; complexity reduction**

I. Introduction

The size and complexity of the onboard software of cyber-physical systems, e.g. aerospace, automobiles, have grown exponentially in recent years [1][2]. If this growth continues, the cost and time of software development and verification will become the primary barrier to the development of future systems. In [3], we addressed this problem and proposed a model-based design methodology where *complexity-reducing, formal architectural patterns* are treated as first-class objects. In this approach, the system architecture is composed from libraries of architectural patterns that have been formally proven to reduce unnecessary complexity and coupling between components. The use of these patterns with formally specified components and easily verifiable compositional analysis ensures that new system designs are developed rapidly in a correct-by-construction manner.

In this work, we apply this design principle and develop a complexity-reducing architectural pattern to support architectural composition and analysis of virtually synchronized cyber-physical systems. We note that these systems are commonly implemented as networked real-time control systems consisting of a network of distributed devices controlled by a hierarchy of distributed controllers. These systems often require *distributed synchronization* to coordinate the hierarchical computations and guarantee consistency among the replicated computations. For example, in a fly-by-wire aircraft, the control surfaces such as ailerons, rudder, and elevator are each locally controlled by higher-level supervisory controllers operating at different rates. To initiate a turn, the actions of these control surfaces are coordinated by the synchronous changes of their setpoints generated from the supervisory controllers. Theoretically, the adjustment of these setpoints need not be synchronous since each of the elements under control is an analog device. However, asynchronous local actions increase coordination errors and thus, are undesirable. In addition, ensuring the consistency of input events to distributed controllers is important, especially when controllers replicated for fault-tolerance.

Developing synchronization protocols to guarantee consistency and coordination between the real-time computations in an asynchronous architecture is, in general, extremely difficult. Because of the clock skews, even a subtle timing difference in the execution and communication delay can lead to distributed race conditions and many serious, often non-reproducible, bugs. Consider the case of a triplicated redundant control. To guard against physical accidents, these 3 redundant controllers could be distributed at different processors. Because of the non-zero clock skews, one controller could be in period $k+1$, while the other two still be in period k. If a new setpoint is received by replicated controllers at different periods, one controller's command could be voted out resulting in an invalid failure detection. Same input for different periods or different inputs for the same period at 3 controllers violates consistency and thus, is unacceptable. The problem becomes more complex when these computations interact at different rates and concurrently participate in more than one distributed synchronization, such as leader election protocol and set-point synchronization.

In our earlier works on Physically-Asynchronous Logically-Synchronous (PALS) systems [4][5][6][7], we addressed this problem for real-time distributed computations

with *same period*. The PALS system is a formal architectural pattern for optimal real-time virtual (or logical) synchronization on top of asynchronously executing tasks. It allows engineers to design, verify and implement the distributed protocols as if the systems were driven by perfectly synchronized clocks. The pattern formally guarantees the same synchronous behavior that no further changes are required in the asynchronous system. As a result, it greatly benefits the software development and verification process since the state space of the synchronous design is often orders of magnitude smaller than that of an asynchronous system. For example, in a case-study of a dual-redundant avionics system, we found that model checking of the asynchronous design took over 35 hours even to discover a counter-example, whereas it finished in less than 30 seconds for the PALS design [4].

While the original PALS system is useful for some common fault-tolerant applications (executing at the same rate), it needs to be extended to support synchronization between multi-rate computations. In this paper, we extend the original PALS system and propose a complexity-reducing architectural pattern, called *Multi-rate PALS* system, to support virtual synchronization pattern with multi-rate (both harmonic and non-harmonic) computations. The Multi-rate PALS system guarantees that *a group of multi-rate distributed computations in a real-time asynchronous system have same behavior as the same set of computations in a synchronous system driven by perfectly synchronized clocks*. We verify this guarantee by showing that the pattern always establishes it if the system architecture satisfies the pattern assumptions. We discuss these assumptions in Section IV-B.

We also provide an architectural analysis to support composition of multiple instances of this pattern. For example, this pattern can be applied at different levels of the hierarchical control systems for consistent coordination of different computations. Our analysis framework assists the system-level verification of this system by validating different timing and environmental assumptions.

In this paper, we define the architectural annotations and specification of this pattern in AADL, an industry-standard architectural description language. These annotations and rules are used in the analysis tool to check the correctness of the pattern instantiation against any incorrect modifications during the system design.

II. Related Work

Virtual synchronization is a well-studied topic in distributed systems and theory [8][9]. The classical virtual synchronization is based on a communication service to guarantee consistent delivery of events to distributed processes using Lamport's vector clock. These synchronization techniques however do not satisfy hard real-time guarantees. Real-time versions of these communication services have been proposed in [10][11]. Although very useful in many contexts, these works are more suitable to synchronize replicated state machines (which commonly execute in the same rate). The Multi-rate PALS pattern based virtual synchronization supports different layers of application synchronization executing at different rates. A distributed middleware supporting the PALS systems is discussed in [12].

Our work is also related to another body of works that implement semantics preserving implementation of synchronous model onto different asynchronous architectures, e.g. Loosely Time-Triggered Architecture [13][14], Asynchronous Bounded Delay network [15]. Although correctness is achieved in spite of unpredictable communication delays and clock skews, these approaches do not provide the hard real-time guarantee required for synchronization and consistent views in networked real-time systems. In the absence of such guarantees, coordinating devices operating at different rates in a networked control system may be very complex even in a harmonic rate group. Coordination errors may be very large depending on the difference in controller periods and random variations of message arrival times.

The architectural assumptions of bounded end-to-end delay and clock skew is similar to Time-Triggered Architecture (TTA) [16]. Caspi et al. [17] gives a model-based approach for translating synchronous model onto TTA. J. Rushby [18] shows the formal verification of the algorithms for mapping of an asynchronous system onto a synchronous system for time-triggered architecture. In general, these solutions depend on a global schedule for distributed synchronization of applications through a tight node synchronization enforced by specialized hardware. On the other hand, our approach does not require any global schedule for synchronization, which itself can be very hard to maintain and compute for multi-rate distributed computations. We use the system parameters, e.g. end-to-end delay and clock skews, as the abstraction of the underlying architecture to define the constraints on applications' execution and synchronization periods. This abstraction works for the virtual synchronization in any real-time network architecture, whether it is time-triggered or event-triggered.

Our use of assumptions to define valid pattern instances and preserve the pattern guarantees is similar to the basic principle of *assume-guarantee* compositional reasoning [19]. For a realistic analysis, this approache decomposes a large, complex system into different subsystems and perform modular verification on each subsystem in the context of its environment interface. There is a fair amount research done on the topic of specification of assumptions and guarantees, automated assumption learning and abstraction refinement [20], [21]. Despite the advances in compositional reasoning, the validation and verification of real-time distributed systems is still subject to the state-space explosion problem. Complexity-reducing design patterns as ours can provide the necessary abstractions, such as the synchronous abstraction by the PALS patterns, which can simplify much of the

verification efforts. We believe that more research still needs to be done to integrate these patterns and their analysis in existing formal verification frameworks.

III. An Illustrative Example

In this section, we present an example of a hierarchical control system to illustrate the basic concepts of the Multi-rate PALS pattern.

A. Problem description

In networked control systems, multiple devices must be coordinated in a timely and synchronized manner to achieve desired operation of the system. For example, in practice, ailerons and rudders are used together to turn an aircraft. Ailerons, attached to the left and right wings of an aircraft, coordinate with each other to roll an aircraft about the longitudinal axis by changing the *lift* on two wings. Since these ailerons move in different directions (upward or downward) to create a differential lift on the wings, they also cause a difference in the *drag* on the wings. This unwanted side effect, commonly known as *adverse-yaw*, produces a yawing motion in a direction opposite to the desired roll. One of the commonly applied techniques to counteract this undesired yawing motion is to use the rudder attached to the vertical stabilizer of the aircraft. Proper, synchronized coordination of both ailerons and the rudder at the right speed is important for the safety of the aircraft. Otherwise, improper turn of the rudder or the ailerons may result in undesired and dangerous sideways movement, known as sideslip.

The coordination of these control surfaces is accomplished by a fault-tolerant, hierarchical control system, in which replicated supervisory controllers are responsible for coordinating the set-points of the position, velocity of ailerons and rudder at a desired speed based on the flight mode. The local servo controllers of each control surface, which are also replicated, use the set-point commands, compute local tracking errors with respect to the set-points, and generate actuator commands at the acceptable rate for the devices.

In order to prevent any incorrect device coordination and any single-point failure as a result of inconsistent actions at the replicated local controllers, the design must satisfy the system-level properties that (1) the replicated servo controllers receive the supervisory controller updates approximately at the same time as other device controllers and (2) a consistent view of inputs is guaranteed at the replicated controllers irrespective of the input data rates.

B. Multi-rate PALS protocol

The proposed *Multi-rate PALS* pattern can be applied to guarantee a logically synchronous coordination of these devices and prevent any inconsistency. The system would operate in the same way as it would do in a synchronous distributed system (with zero clock skew).

For illustration purposes, we assume that the ailerons are controlled at 66.67Hz (15ms), the rudder is controlled 50Hz (20ms)[1]. For simplicity, we only show the active-standby replication for the rudder control, where two servo controllers execute at the same rate. While both controllers receive the sensor data and supervisory commands, only the active controller sends the actuator command to the rudder.

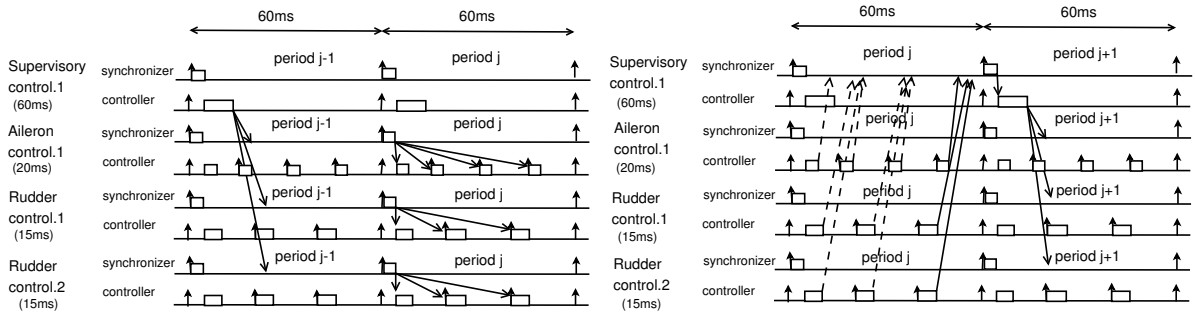
Logical synchronization period: In a perfectly synchronous distributed system, the synchronous changes in the set-points of the local control applications can happen only at the hyper-period boundary, i.e. at an interval equal to the LCM (least common multiple) of the local control periods. This is unavoidable in a synchronous design since there is no simple scheduling solution to change the setpoints at the same time with a smaller synchronization period. A smaller synchronization period may also potentially result in asynchronous actions, such as ailerons changing their set-points or other discrete commands first before the rudder, and vice versa. Such asynchronous changes are not desired as they could potentially lead to adverse-yaw during the aileron-rudder synchronization.

To preserve the same synchronous semantics in a Multi-rate PALS system, the supervisory synchronization period of these devices is also set to the LCM of the periods. In this example, the rudder and aileron servo controllers receive the setpoint updates at a period of 60ms. The supervisory controller itself may execute at a faster or slower rate. However, if it needs to receive synchronous updates of the status of the device controllers, it can do so at 60ms.

We also note that harmonic rates have been traditionally favored for hierarchical control in industrial systems, as they simplify the scheduling. However, the rates offered in such design may not be the best from a control perspective (considering the difference in the physical dynamics of the devices). On the other hand, picking locally optimal control periods may result in a very long LCM and slow the supervisory control. Therefore the trade-off between local, optimized control computations and longer supervisory control period needs to be considered when designing this hierarchical control. A key benefit of using our proposed pattern is that it provides the simplicity of the synchronous design and does not require the devices to operate in a strictly harmonic rate since there are no direct communication between them. Thus, engineers can address this trade-off and explore an extended design space with our approach.

Synchronization interface: In order to ensure the supervisory control synchronization in a period of 60ms, the pattern defines a *synchronization interface*, called *multi-rate synchronizer* for each servo controller. These synchronizers execute periodically at 60ms. They execute at a higher priority than the control application so that commands of the

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



(a) Supervisory set-point commands going downward the hierarchy

(b) Device controller status going upward the hierarchy

Figure 1. Logical synchronization in the hierarchical control system

supervisory controller of the synchronization period $j-1$ can be consistently processed by the device controllers during the synchronization period j.

Figure 1(a) shows the equivalent logically synchronous execution of these controllers. In this figure, the aileron and the replicated rudder controllers obtain synchronous updates of the set-point commands in every $60ms$ period. Since there are many executions of a servo controller in each synchronization period, the same set-point is used during these executions. In this example, there are 4 executions of the rudder servo controllers and 1 execution of the supervisory controller in each synchronization period j. Suppose that the setpoint generated by the supervisory controller in its execution period p for the rudder is given by $Supervisor.Setpoint_R^{out}(p)$ and the setpoint used by the rudder controller in its execution period q is given by $Rudder_i.Setpoint_R^{in}(q)$, $i = 1, 2$. The logically synchronous supervisory control of the rudder controllers can then be shown by $Rudder_1.Setpoint_R^{in}(4.j+k) = Rudder_2.Setpoint_R^{in}(4.j+k) = Supervisor.Setpoint_R^{out}(j-1), k = 0..3$ for each synchronization period j. (It can similarly be shown for the aileron controllers). In the asynchronous system, the system clock of each node has bounded clock skew of ϵ. The pattern abstracts the impact of the clock synchronization errors and produces an equivalent execution, which happens within an interval 2ϵ (as bounded by the clock skew).

Similarly, the aileron and the rudder controllers send their status to the supervisory controller[2]. These responses flow upward in the hierarchy as shown in Figure 1(b). It shows that the status at each synchronization period j is propagated to the supervisory controller in the same synchronization period. Based on these inputs, the supervisory controller may take any correction necessary to coordinate these devices. While there are many executions of each control application during a synchronization period, only the update from the last execution matters for the coordination. We show this communication with a solid line in the figure. However, the status from previous executions (as shown by the dashed lines) may be relevant depending on the application requirements, e.g. debugging. If they are transmitted, then the synchronization interface of the supervisory controller may be responsible to deliver the correct input. In this paper, the Multi-rate PALS pattern assumes that outputs from other executions are also delivered, but the synchronizer filters the outputs to deliver only the last one.

[2]The multi-rate synchronizer may be omitted for the supervisory controller unless it operates in a different rate and requires synchronous updates. We use it for uniformity of the design.

C. Pattern composition

A particular benefit of using the Multi-rate PALS pattern is that it allows designers to form logical synchronization groups where a component may participate in different groups. Thus the components have the flexibility to receive their messages logical synchronously at different rates. For example, in addition to the Multi-rate PALS synchronization for supervisory control, the replicated rudder servo controllers participate in another synchronization group with the sensors and the actuator. In this instantiation, these servo controllers receive the sensor data and perform discrete mode changes, e.g. changing the mode to standby upon the failure of the active controller, logically synchronously.

We show a Multi-rate PALS pattern instantiation for the rudder servo control later in Figure 2. (We will explain the structural specifications later in the paper). This instantiation for rudder servo control is a special case of this pattern with each task operating at 20ms. In this case, the logically synchronous interaction is simple. For the rudder sensor data (RSD), the pattern guarantees that $Rudder_1.RSD^{in}(j) = Rudder_2.RSD^{in}(j) = Sensor.RSD^{out}(j-1)$ in each 20ms synchronization period j.

From compositional verification perspective, the pattern greatly simplifies the system verification process. In this example, designers can reuse the formal pattern guarantee of virtual synchronization with respect to the input data, such as rudder sensor data (RSD) and supervisory setpoint command ($Setpoint_R$), to show that the servo controllers operate consistently by receiving identical inputs despite the differences in the rates. The only overhead for validating this system-level property of consistency is that these

instantiations indeed follow the pattern requirements. In Section VI, we discuss how the pattern requirements can be mechanically validated.

IV. The Multi-rate PALS Pattern

The *Multi-rate PALS* pattern is a formal architectural transformation that transforms an input system model to a new system model with guaranteed properties. In this section, we give a description of the *Multi-rate PALS* pattern's behavior as well as how a developer would use the pattern in a AADL system design specification[3]. We also describe the assumptions that must be satisfied before the pattern can be applied. These assumptions include constraints over the system architecture model, e.g. timing constraints, structural requirements for the relevant components and their connections. Later in Section V, we prove its logical synchronization guarantee for multi-rate real-time distributed systems. The key property of this pattern, as well as other formalized architectural patterns [3], is that this verification effort is amortized over all valid pattern instances. Analysis of system-level behavior can subsequently make use of the proven pattern guarantees without having to reprove them.

A. Pattern parameters

The pattern is applied to a group of periodic, distributed computation components, $M_1, ..., M_N$ modeled as AADL threads or thread groups[4]. They execute at a period of $T_1, ..., T_N$ respectively. The hyper-period, denoted by T_{hp}, is equal to the LCM of these periods. These components form a synchronization group defined by a property, *PALS_ID* after the pattern application. A set of (output port, input port) pairs, i.e. AADL port connections, used in the multi-rate synchronization is also provided as the parameters of this pattern. These connections are annotated by the property called *PALS_Connection_ID*. The value of *PALS_Connection_ID* is equal to *PALS_ID*.

B. Assumptions

The assumptions of the pattern are classified into three categories: system context (any requirements on the initial system model), timing, and external interface constraints. This detailed list of assumptions does not mean that the pattern is very restrictive. They rather provide the basis for the structural analysis relevant to this pattern. The other constraints, such as those based on physical properties of the controlled devices are out of scope of this pattern.

System context: This pattern is applicable in hard real-time, networked systems with following characteristics:

- *Bounded local clock skew*. Each node i has access to an approximation of the true global time t via a local clock $C_i(t)$, where the maximum skew of each local clock is ϵ, i.e., $|C_i(t) - t| < \epsilon$.
- *Monotonic local clocks*. Each node may adjust its clock rate, but it may never decrease the local clock value.
- *Bounded computation time*. The computation of a component completes within a specified time. Typically this is the scheduling deadline of a thread, α_i^{max}.
- *Bounded message delivery*. Messages from M_i, $i = 1 \dots N$, are reliably delivered to their destinations with a latency μ_i[5], where $\mu_i^{min} \leq \mu_i \leq \mu_i^{max}$.
- *Node fault assumptions*. Nodes are crash-stop and may recover later. The output of a crashed component is assumed to be 'null'. A failed node must not be able to send extra erroneous messages during a period. This could result in nodes receiving different messages, even though the network delivered each message correctly.

Timing constraints: The following constraints relating system parameters of each computation must be also satisfied. We show in next section that these constraints are required to satisfy the requirement that messages generated during the logical synchronization period $j-1$ are consumed by their destination nodes in the synchronization period j.

- *Computation period constraint*. The period of this computation gives the upper bound on the worst-case end-to-end delay from a component. A message must not be sent after its deadline so that the receiving nodes receive them before the next dispatch of this component.

$$Max(eout_i) < \alpha_i^{max} \leq T_i - \mu_i^{max} - 2\epsilon \tag{1}$$

  $eout_i$ denotes the execution time range when a component i delivers a message. Max(.) gives the upper bound of the time range.
- *Causality constraint*. In order to account for the clock skews, messages should not be delivered too early which might violate the causality.[6]

$$H_i \geq max(2\epsilon - \mu_i^{min}, 0) \tag{2}$$

  H_i is earliest time during the execution of a component i. $max(a, b)$ returns the larger value of a and b.

The initial system model must define necessary properties for period, deadline, output time, latency, and clock skew for these periodic components. In the AADL model, these can be specified by standard AADL properties e.g. *Period*, *Deadline*, *Output_Time*, *Latency*, and *Clock_Jitter*.

External interface constraints The last set of assumptions is associated with external inputs that are received from any component outside the PALS synchronization group, but are used in a multi-rate synchronization. The pattern assumes

[3]The pattern can similarly be specified and implemented in other programming environments, e.g. PTIDES [22], with a similar set of analysis.

[4]In AADL, a *thread group* gives the component abstraction for logically organizing threads and other thread group within a process

[5]The values must be normalized to take into account the minimum and maximum clock drift rate as suggested in [23].

[6]This constraint can be relaxed by using timestamps to prevent causability violation. It however increases the message size and requires message buffers at the receivers.

that the components consume these external inputs, such as a user input that changes the global system mode, in the same synchronization period.

These external connections are identified by the annotation property *PALS_Connection_ID*, which is provided as part of the pattern parameters. However the source component of these connections do not either define the *PALS_ID* property or has a different value than the identifier of this synchronization group.

C. Guarantees

As illustrated in Section III, the pattern guarantees logical synchronization between multi-rate asynchronous computations at a period of T_{hp}. Suppose that A is a sending component (period=T_a) and it sends messages to other components, B and C of period T_b and T_c respectively. There are $n_a = T_{hp}/T_a$, $n_b = T_{hp}/T_b$ and $n_c = T_{hp}/T_c$ executions of A, B and C during a synchronization period T_{hp}. The pattern guarantees that the receiving components receive all $n_a = T_{hp}/T_i$ messages from A during the synchronization period $j-1$, based on the timing assumptions (see Lemma 1 in Section V). The pattern filters these received messages (identically at both B and C) and delivers the selected message to the computation components of B and C during the period j. If the last received message is selected, then the pattern ensures that

$$B.in(j.n_b + k_b) = C.in(j.n_c + k_c) = A.out(j.n_a - 1);$$

where $k_b = 0 \ \ldots \ n_b - 1, k_c = 0 \ \ldots \ n_c - 1$.. Here $A.out(i')$, $B.in(j')$ and $C.in(k')$ corresponds to the input and output port data of the corresponding components in their execution period i', j', k'.

This logically synchronous interaction in the asynchronous system is equivalent to a group of perfectly synchronized nodes executing at the same rates, as long as the pattern assumptions are satisfied.

D. Pattern instantiation

In order to guarantee logical synchronization, the pattern attaches a *multi-rate synchronizer*, $M_{i,syn}$ at each component M_i that serves as a synchronization interface and manages only the input data used during the multi-rate synchronization (annotated by the connection property, *PALS_Connection_ID*). It does not affect other inputs that are not used in this instantiation.

The pattern models the synchronizer as an AADL thread component and binds it to the same processor as M_i. We use a set of pre-defined properties to model the expected scheduling and communication characteristics, which can later be used to validate the pattern instantiations:

- *PALS_Synchronizer_Type*: This is used to distinguish a thread as a multi-rate synchronizer. The values are set to *Multi_Rate_Synchronizer*.
- *Dispatch_Protocol*, *Period*: The synchronizer is a $Periodic$ thread, with its *Period* being set to T_{hp}.
- *PALS_Period*: This is the synchronization period of this group. Its value is set to T_{hp}. The period of the synchronizer must be equal to this value, too.
- *Priority*: As discussed in the example section, the thread priority of the multi-rate synchronizer is set to a higher value than that of M_i.

In addition to these properties, other standard AADL properties, such as *Output_Time* (the interval during an execution when output is transmitted) and *Deadline*, must be defined for this thread. The pattern also defines the message selection criteria of the input data ports of the synchronizer with a property, called *PALS_Synchronizer_Operation*. Currently, its value is set to *Last_Message_Only* to indicate that the synchronizer only propagates the last message it received in this port during a synchronization period. It can be changed to model other alternatives, such as delivering a vector or as a function of the received messages.

Composition of M_i and $M_{i,syn}$: In order to facilitate the use of this component in subsequent pattern instantiations, the pattern forms a new AADL thread group, M_i' with the computation component, M_i and the multi-rate synchronizer $M_{i,syn}$ as its subcomponents. This newly formed thread group has exactly the same input-output interface as the original component, M_i. The pattern defines the internal connections of its subcomponents with its input/output ports. The input ports of M_i' are connected to the input ports of $M_{i,syn}$ if they are relevant to the current pattern instantiation (as identified by the *PALS_Connection_ID*), otherwise these input ports are connected with corresponding input ports of M_i. The outputs of M_i are directly propagated through the corresponding outputs of M_i'.

The pattern also annotates M_i' with AADL properties: *Period*, *Deadline*, *Computation_Time*, *Output_Time*, *Priority*, and *PALS_ID*. These property values capture the timing characteristics of the combined execution of these two components and are derived from the properties of M_i and $M_{i,syn}$. In this case, the *Period*, *Deadline*, *Priority* of M_i' are set to same values as M_i since the computations of this new thread group do not change with this composition. However, adding a multi-rate synchronizer to generate the new system model adds some small computation overhead for the first execution of M_i in a synchronization period. We therefore update the *Computation_Time* and *Output_Time* with this added overhead. $M_i'.Output\_Time$ is set to the time range $Min(Output\_Time_i + Output\_Time_{i,syn}) \ldots Max(Output\_Time_i + Output\_Time_{i,syn})$ to denote the time range when this combined computation sends the message. Finally we set the connection property, *PALS_Connection_ID* of the incoming connections to $M_{i,syn}$ with the synchronization group identifier so that they can be differentiated in subsequent pattern applications.

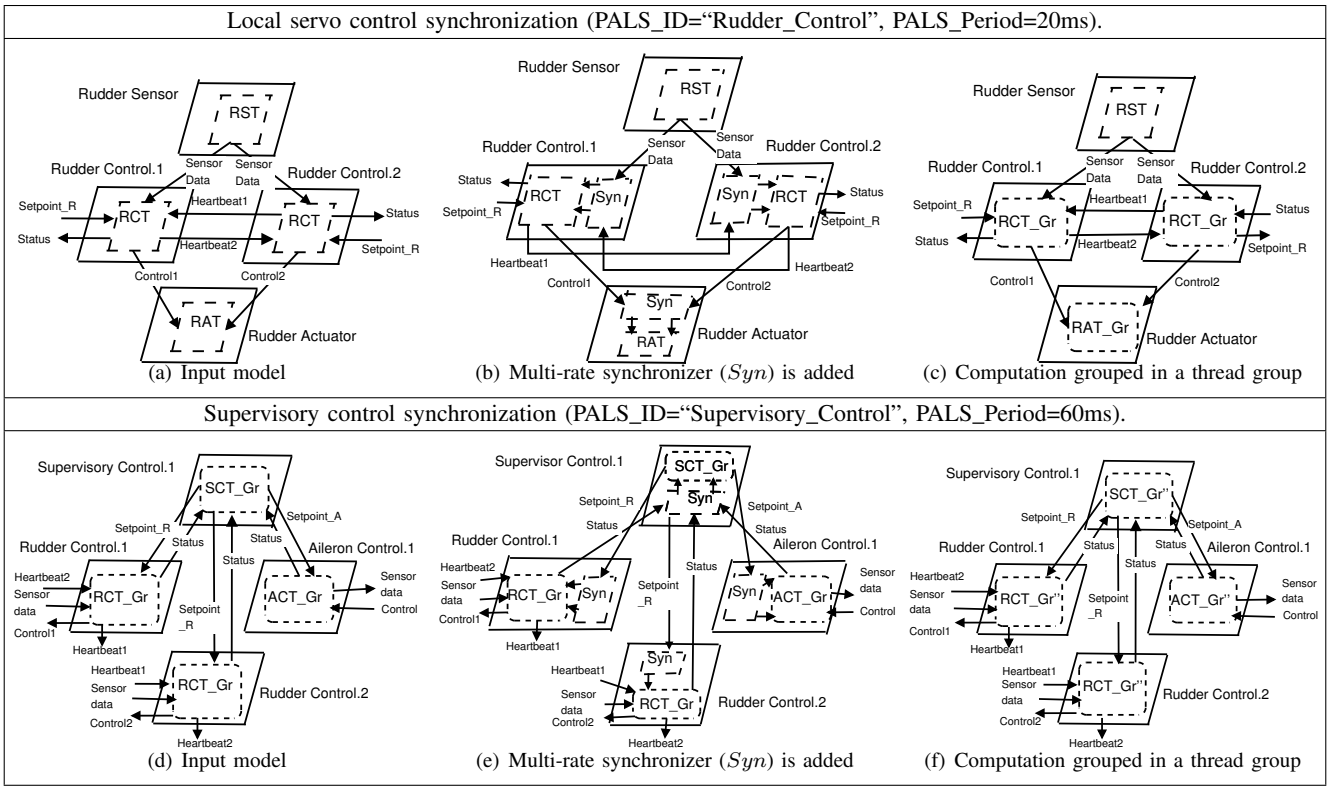


Figure 2. Composition of Multi-rate PALS pattern instances

Alternative implementation: Some of the suggested implementations may have alternative modeling solutions with equivalent results. In particular, with respect to the multi-rate synchronizer, instead of defining the multi-rate synchronizer as a separate thread element, this synchronizer may be modeled as a middleware subprogram that reads new data from the port at the specified period in the same computation thread. Even in this case, the effect of implementing the subprograms must be included in the pattern modeling and analysis, particularly the same computation overhead and the support for reusable message selection logic. Similarly, another standard property, called Timing on the port connection can be used instead of the thread priorities to define the scheduling order. In any case, the compositional analysis should be extended for these alternative solutions for preserving the synchronous semantics.

External input synchronization: The pattern assumes that inputs arrive logically synchronously. Here we discuss a related concept for synchronization of external inputs. In this case, the sources of these external inputs are not part of the synchronization group. In order to guarantee that external inputs arrive logically synchronously, we extend the concept of *environment input synchronizer* of the original PALS system [5].

An environment input synchronizer, M_{env} is a component that follows the timing constraints of Equation 1 and 2. It requires an additional constraint on its period T_{env} such that the synchronization period T_{hp} is perfectly divisible by T_{env}. With this solution, the original source component can execute at any period asynchronously with respect to the receiving components of the given synchronization group. The source component transmits its outputs asynchronously to an environment input synchronizer (which may also be replicated, if needed). The input synchronizer then delivers the messages to the receiving nodes logically synchronously. To use this concept in the analysis, the pattern requires the environment input synchronizer be annotated with the property, *PALS_Synchronizer_Type* with the value being set to *Environment_Input_Synchronizer*.

E. Exemplar model

Figure 2 gives simplified AADL diagrams of the pattern instantiations for local rudder servo control synchronization and supervisory control synchronization. The code snippet of this example is given in the appendix. In this figure, we only show the distributed process elements of different subsystems and the threads inside them. It does not show the distribution of these processes in dedicated system models with the hardware specification.

In the first synchronization in Figure 2(a)2(b)2(c), two replicated rudder controller threads (RCT) receive sensor data from the rudder sensor thread (RST) and propagates their actuator commands to the actuator thread (RAT). They also exchange the heartbeat messages to each other as part of

the active-standby replication protocol. After the pattern is applied, multi-rate synchronizers (denoted by a short name, *Syn*) are added to the processes. These synchronizers affect only the input data that are relevant to this synchronization instantiation. In this instantiation, the setpoint commands from the supervisory controllers are not directly involved so they are not passed through the multi-rate synchronizer. After this, we create an AADL thread group component, e.g. *RCT_Gr* at the servo controller composing both the controller thread (RCT) and the synchronizer (Syn).

In the second synchronization in Figure 2(d)2(e)2(f), the system models obtained after the previous instantiation are used for the supervisory control synchronization. In this case, two rudder controller and an actuator controller receive the setpoint commands from the supervisor. The pattern instantiation follows the same rule as above without affecting the non-participating inputs of a component.

V. Verification of the Multi-rate PALS Pattern

This section describes the timing model of this pattern and uses this model to prove the virtual synchronization guarantee based on the assumptions given in Section IV-B.

A. Multi-rate PALS timing model

Each node in the pattern has two components involved in the pattern instantiation: M_i (computation component) and $M_{i,syn}$ (multi-rate synchronizer). M_i and $M_{i,syn}$ are driven periodically by two local logical clocks or timers, C_i and $C_{i,syn}$ executed at a rate of T_i and $T_{hp} = n_i.T_i$ respectively.

We assume that the j^{th} synchronization period at each node begins at the local system clock time $j.T_{hp}$. Since each node has access to the global clock with a maximum clock skew of ϵ, in true time, the period j may begin in any time between $(j.T_{hp} - \epsilon)$ and $(j.T_{hp} + \epsilon)$, given by $t^i_{j,0}$.

Both M_i and $M_{i,syn}$ execute on the same processor. Thus, their logical clocks C_i and $C_{i,syn}$ may be derived from the same system clock so that they start at the beginning of a synchronization period. This also ensures that the j^{th} execution of $M_{i,syn}$ coincides with the $j.n_i^{th}$ execution of M_i, since there are n_i executions of M_i in a synchronization period. The other $n_i - 1$ executions of M_i happen at $j.T_{hp}+k.T_i$ or in true time at $t^i_{j,k} \in [j.T_{hp}+k.T_i-\epsilon, j.T_{hp}+k.T_i+\epsilon), k = 1..n_i - 1$.

Figure 3 shows a timeline of the computation and communication in a Multi-rate PALS pattern instance. The deadline of M_i is given by α_i^{max}; thus it must be scheduled to complete its execution by $t^i_{j,k} + \alpha_i^{max}$; for $k = 0 \ldots\ n_i - 1$. Let the output message be delivered after at least an interval of $H_i \geq eout_{i,syn} + eout_i$. ($eout_i$ and $eout_{i,syn}$ denote the execution times of M_i an $M_{i,syn}$ at which they send an output message.) This output is propagated to the multi-rate synchronizer of a receiving node after a latency of μ_i and is expected to arrive at $t^i_{j,k} + H_i + \mu_i$.

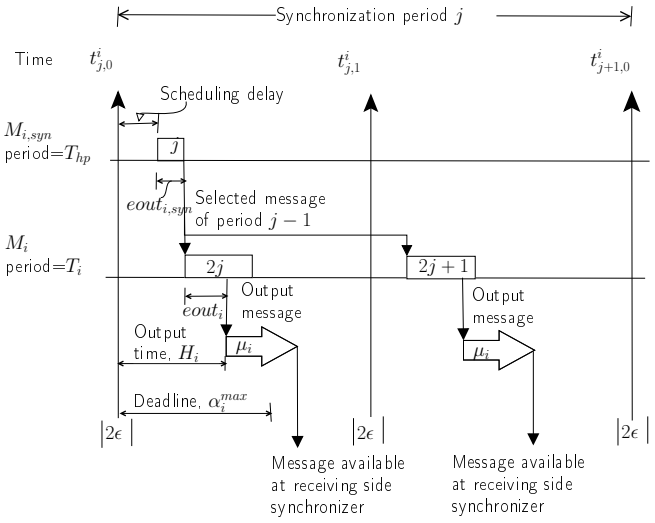


Figure 3. Multi-rate PALS timeline for $n_i = 2$

Lemma 1: *In a multi-rate asynchronous system, when a non-failed component M_i sends its messages to the multi-rate synchronizers $M_{r,syn}$ of receiving components M_r, these synchronizers receive exactly $n_i = T_{hp}/T_i$ messages in each synchronization period j if the timing constraints $\alpha_i^{max} \leq T_i - 2\epsilon - \mu_i^{max}$ (Equation 1) and $H_i \geq max(2\epsilon - \mu_i^{min}, 0)$ (Equation 2) are satisfied. In other words, messages generated during the synchronization period j are received by the receiving sides in the same synchronization period j if these timing constraints are satisfied.*

Proof: There are exactly n_i executions of M_i in each synchronization period j. We prove this lemma by showing that the first and n_i^{th} messages generated during the synchronization period j are indeed received in the same synchronization period at the receiving node.

The first execution of M_i during the synchronization period happens at $t^i_{j,0}$ and the message is expected to arrive at a receiving synchronizer at least after $t^i_{j,0} + H_i + \mu_i$ where $H_i \geq eout_i + eout_{i,syn}$ and μ_i is the message transfer latency. This can happen as early as $j.T_{hp}+H_i+\mu_i-\epsilon$ in true time. Given the minimum latency, μ_i^{min}, the earliest message arrival time is $j.T_{hp} + H_i + \mu_i^{min} - \epsilon$. At the receiving node, due to the clock skew, the execution of the multi-rate synchronizer can be delayed as late as $j.T_{hp} + \epsilon$ during the synchronization period j. Since $H_i \geq 2\epsilon - \mu_i^{min}$,

$$j.T_{hp} + H_i + \mu_i^{min} - \epsilon \geq j.T_{hp} + \epsilon.$$

It implies that this message is indeed received in synchronization period j, which happens after the dispatch of $M_{r,syn}$ in this period.

The n_i^{th} execution, i.e. the last execution, of M_i in the synchronization period j occurs at t^i_{j,n_i-1} or in true time between $j.T_{hp} + (n_i - 1).T_i \pm \epsilon$. Since the output of this execution must be sent before its deadline α_i^{max} and the worst-case message transfer delay is μ_i^{max}, the latest output

arrival time is given by t_{arr}, i.e. $t_{arr} < j.T_{hp} + (n_i - 1).T_i + \epsilon + \alpha_i^{max} + \mu_i^{max}$. The synchronization period $j+1$ in the receiving component may begin as early as at $(j+1).T_{hp} - \epsilon = j.T_{hp} + n_i.T_i - \epsilon$. Given that $T_i \geq 2\epsilon + \alpha_i^{max} + \mu_i^{max}$, it is easy to show that

$$t_{arr} < j.T_{hp} + n_i.T_i - \epsilon,$$

This implies that the n_i^{th} message is received in period j.

The proof immediately follows since outputs of the remaining $(n_i - 2)$ executions are received in FIFO order between the 1^{st} and n_i^{th} executions.

Theorem 1: The proposed pattern specification satisfies the same message communication guarantee as the perfectly synchronous system.

Proof: The proof is very simple. Based on Lemma 1, the receiving multi-rate synchronizers receive the same set of messages at a synchronization period j. The logic of the synchronizers are same at the receiving nodes. Since the multi-rate synchronizer $M_{r,syn}$ executes always before M_r at the beginning of synchronization period and they select the same message, such as the last received message, the receiving nodes M_r apply the same input during its executions in a synchronization period, in the same ways as it would do in a perfectly synchronous system.

VI. Compositional Analysis

In order to preserve the original Multi-rate PALS pattern guarantees in an evolving system architecture, we provide an analysis framework to validate its assumptions and structural specifications. We have implemented a prototype tool in OSATE (the Eclipse-based AADL development environment) to perform this analysis automatically. The tool reads the system model instances and performs static analysis of the AADL components and connections. Once the model is validated, designers can safely use the pattern guarantees.

A. Analysis procedure

Table I lists the main analysis rules used to validate the pattern specifications rules, the timing and external input assumptions for different Multi-rate PALS instantiations. (We do not however list other trivial rules, e.g. sanity checks of identifier, message selection properties, or the properties of the pattern-added thread groups etc.)

Explanation of the notation symbols: We consider that system AADL configuration, $G = (Comp, Conn)$ consists of the set of all thread and thread group components $Comp = \{M_i\}$ and the set of all port connections, $Conn = \cup_{i,j} Conn(M_i, M_j)$, where $Conn(M_i, M_j)$ is the set of all connections from component M_i to M_j. The set of all connections to a component M_j is also given by $Conn(*, M_j)$. For each connection Cn, we assume that $Cn.Dst$ provides information on the destination port. We use some data enumerating functions, such as $PALS\_Id(G)$, which gives all synchronization group identifiers in G, with the synchronization period of *PALS_Period(id)* and $Comp^P(G, id)$, which gives the list of components M_i with *PALS_ID* property being set to id. If $id = *$, then all components with a defined *PALS_ID* are returned by $Comp^P(G, *)$. For each component, we use the notation $M_i.Property$ to give the values of an AADL property *'Property'* of a component M_i. As shown earlier, a thread group M_i is formed from a computation component and its multi-rate synchronizer. Here, they are denoted by $M_i.Sync$ and $M_i.Comp$ respectively. In this system model, all three components are assumed to a member of $Comp$.

Sanity check of the Multi-rate PALS specifications
R1. $\forall_{id \in PALS\_Ids(G)} \forall_{M_i \in Comp^P(G,id)}$ $PALS\_Period(id) = (M_i.Sync).Period$
R2. $\forall_{M_i \in Comp^P(G,*)}$ $(M_i.Sync).Priority > (M_i.Comp).Priority$
R3. $\forall_{M_i \in Comp^P(G,*)} (M_i.Sync).Processor =$ $(M_i.Comp).Processor$
R4: $\forall_{M_i \in Comp^P(G,*)} \forall_{Cn \in Conn(M_i.Sync, M_i.Comp)}$ $Cn.PALS\_Connection\_ID = M_i.PALS\_ID$
R5. $\forall_{M_i \in Comp^P(G,*)}$ $\forall_{Cn \in Conn(*, M_i)}$ $Cn.PALS\_Connection\_Id = M_i.PALS\_ID \rightarrow$ $(\exists_{Cn' \in Conn(M_i.Sync, M_i.Comp)} Cn.Dst = Cn'.Dst$ $\wedge \exists_{Cn' \in Conn(M_i, M_i.Sync)} Cn.Dst = Cn'.Dst)$
Timing assumptions
R6. $\forall_{M_i \in Comp^P(G,*)} M_i.Period \geq 2 \times G.Clock\_Skew$ $+ Max((M_i).Latency) + M_i.Deadline$
R7. $\forall_{M_i \in Comp^P(G,*)} Min(M_i.Output\_Time) \geq$ $max(2 \times G.Clock\_Skew - Min(M_i.Latency), 0)$
External input assumptions
R8. $\forall_{M_i \in Comp^P(G,*)}$ $\forall_{Cn=(M_k, M_i) \in Conn(*, M_i)}$ $Cn.PALS\_Connection\_Id = M_i.PALS\_ID \rightarrow$ $(M_k.PALS\_Synch.\_Type = Env.\_Input\_Synch$ $\wedge\ M_i.PALS\_Period\ \%\ M_k.Period = 0$ $\wedge$ (M_k satisfies the predicates of R6, R7))

Table I
Multi-rate PALS PATTERN ANALYSIS

Pattern specification rules: The rules R1-R5 are related to the scheduling and communication characteristics associated with the pattern instantiated multi-rate synchronizer and the formed thread group. For example, R4-R5 gives the condition that all messages related to a Multi-rate PALS synchronization instance are indeed flown through the multi-rate synchronizer. We analyze the data flow between components and detect if the multi-rate synchronized is bypassed. This is important since any violation of these constraints may potentially break the logical synchronization guarantee if the data is relevant to the instantiation.

Timing assumptions: The rules R6 and R7 describe the timing assumptions we have defined in Equation 1 and 2. We have shown their usage in Lemma 1.

External input assumptions: The rule R8 validates the external input assumptions in a pattern instantiation. It validates that synchronization of external inputs are originated from an environment input synchronizer. This rule can be useful to validate the scenario when the components of two separate synchronization groups interact but do not themselves from a bigger synchronization group.

VII. Conclusion and Future Work

This paper presents an architectural pattern to support virtual synchronization among a group of distributed computations running at different rates. In this approach, the amount of effort spent on distributed system verification can be reduced by the re-usage of pattern guarantees, which are pre-proven to be correct.

We are extending this pattern to support distributed synchronization at a period smaller than the hyper-period. This, however, requires additional constraints on the execution period and scheduling. Currently, the pattern only supports synchronization based on point-to-point communications. Future work will support synchronization of distributed flows spread across multiple nodes and relevant constraints on the end-to-end scheduling. In [24], we collaborated with our colleagues at UIUC and University of Oslo to develop a model checking and validation framework for verifying synchronous AADL models of a single-rate PALS system in Real-time Maude. We also plan to integrate the Multi-rate PALS pattern in this framework.

Acknowledgment

This work was sponsored in part by AFRL under contract FA8650-10-C-7081 as part of the DARPA META program. The authors would like to thank anonymous reviewers, Scott Nagel of Rockwell Collins Inc, Min Young Nam, Mustafa Dursun, Po-Liang Wu, Kyungmin Bae and Cheolgi Kim of UIUC for their help and suggestions.

References

[1] P. H. Feiler, J. Hansson, D. de Niz, and L. Wrage, "System architecture virtual integration: An industrial case study," *Technical Note, CMU/SEI-2009-TR-017, ESC-TR-2009-017, http://www.sei.cmu.edu/reports/09tr017.pdf*, 2007.

[2] R. N. Charette, "This car runs on code," *IEEE Spectrum*, 2009.

[3] D. D. Cofer, "Complexity-reducing design patterns for cyber-physical systems," *AFRL Technical Report AFRL-RZ-WP-TR-2011-2098*, 2011.

[4] S. P. Miller, D. D. Cofer, L. Sha, J. Meseguer, and A. Al-Nayeem, "Implementing logical synchrony in integrated modular avionics," in *Proceedings of DASC*, 2009.

[5] 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 *Proceedings of RTSS*, 2009.

[6] J. Meseguer and P. C. Ölveczky, "Formalization and correctness of the pals architectural pattern for distributed real-time systems," in *Proceedings of ICFEM*, 2010.

[7] L. Sha, A. Al-Nayeem, M. Sun, J. Meseguer, and P. C. Ölveczky, "PALS: Physically Asynchronous Logically Synchronous Systems," *UIUC Technical Report http://hdl.handle.net/2142/11897*.

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

[9] R. van Renesse, K. P. Birman, and S. Maffeis, "Horus: A flexible group communication system," *Communications of the ACM*, vol. 39, April 1996.

[10] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time corba event service," in *Proceedings of OOPSLA*, 1997.

[11] T. Abdelzaher, S. Dawson, W.-C. Feng, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. Shin, Z. Wang, H. Zou, M. Bjorkland, and P. Marron, "ARMADA Middleware and Communication Services," *Real-Time Systems*, vol. 16, no. 2/3, pp. 127–153, 1999.

[12] C. Kim, A. Al-Nayeem, H. Yun, P.-L. Wu, and L. Sha, "PALS/PRISM software design description (sdd): Ver. 0.51," *UIUC Technical Report http://hdl.handle.net/2142/25987*.

[13] S. Tripakis, C. Pinello, A. Benveniste, A. S. 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.

[14] A. Benveniste, A. Bouillard, and P. Caspi, "A unifying view of loosely time-triggered architectures," in *Proceedings of EMSOFT*, 2010.

[15] G. Tel, *Introduction to Distributed Algorithms*, 2nd ed. Cambridge University Press, 2000.

[16] H. Kopetz, "The time-triggered architecture," in *Proceedings of ISORC*, 1998.

[17] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From simulink to SCADE/lustre to TTA: a layered approach for distributed embedded applications," in *Proceedings of LCTES*, 2003.

[18] J. Rushby, "Systematic formal verification for fault-tolerant time-triggered algorithms," *IEEE Transactions on Software Engineering*, vol. 25, pp. 651–660, September 1999.

[19] A. Pnueli, *In transition from global to modular temporal reasoning about programs*. Springer-Verlag, 1985.

[20] C. S. Pasareanu, M. B. Dwyer, and M. Huth, "Assume-guarantee model checking of software: A comparative case study," in *In Theoretical and Practical Aspects of SPIN Model Checking, Lecture Notes in Computer Science*, 1999.

[21] M. Gheorghiu Bobaru, C. S. Păsăreanu, and D. Giannakopoulou, "Automated assume-guarantee reasoning by abstraction refinement," in *Proceedings of CAV*, 2008.

[22] Y. Zhao, J. Liu, and E. Lee, "A programming model for time-synchronized distributed real-time systems," in *Proceedings of RTAS*, 2007.

[23] W. Steiner and J. Rushby, "TTA and PALS: Formally verified design patterns for distributed cyber-physical systems," in *Proceedings of DASC*, 2011.

[24] K. Bae, P. Ölveczky, A. Al-Nayeem, and J. Meseguer, "Synchronous AADL and its formal analysis in real-time maude," in *Proceedings of ICFEM*, 2011.

[25] Paparazzi, "http://paparazzi.enac.fr."

[26] F. Nemer, H. Cassé, P. Sainrat, J. P. Bahsoun, and M. D. Michiel, "Papabench: a free real-time benchmark," in *International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

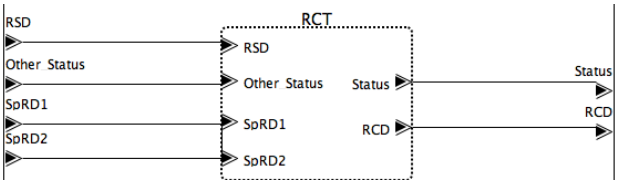
APPENDIX

AADL Models of Some Example Systems

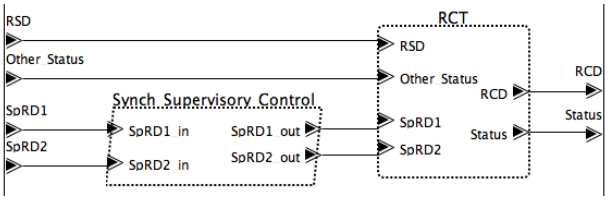
A. Hierarchical Control System

In this section, we discuss the example, mentioned inside the paper in more detail. The top-level AADL diagram is presented in Figure 4(a). This system consists of two supervisory controller defined under the hood of a system component (SCS), two rudder servo controllers (RCS), and three aileron servo controllers (ACS). In Figure 4(b) we show the internal specification of the system component RCS, where we show the system components of the two servo controllers RCS1 and RCS2, the actuator RA and the sensor RS. Similar to the description inside the paper, both rudder servo controllers receive the setpoints command from the supervisory controllers (denoted by SpRD1, SpRD2), sampled sensor data (RSD). Since the heardbeat information used in the active-standby replication management are given by the output port Status. Output of the rudder servo controllers (RCD1, RCD2) are passed to the supervisory controller and the actuator. Only the active controller's output is used to control the actuator.

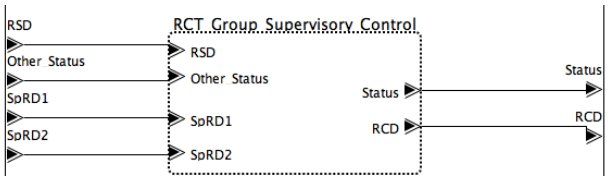
In this section, we show the AADL code snippet of the application of the Multi-rate PALS pattern in the supervisory control synchronization discussed in the paper. The structural modeling of this pattern application is shown in Figure **??**. A multi-rate synchronizer, named *Synch_Supervisory_Control* is implemented. The synchronizer only affects the data ow of the input data SpRD1, SpRD2 which are used in the supervisory control synchronization so that supervisory commands are processed consistently and logically synchronously at the local servo controllers. With this synchronizer, we form a new thread group (as discussed in the pattern description) and replace the original component in the process element with the new thread group.



(a) A computation component, RCT inside the process element of RCS1 (before pattern application)



(b) A thread group, named RCT_Group_Supervisory_Control is created with a multi-rate synchronizer and RCT.



(c) RCT is replaced with RCT_Group_Supervisory_Control (after pattern application)

Figure 5. Application of Multi-rate PALS pattern on the hierarchical control system

Input model The AADL code snippet of the rudder control process prior to the application of the Multi-rate PALS pattern for supervisory contro is given below. Here we only provide the AADL code without any hardware bindings.

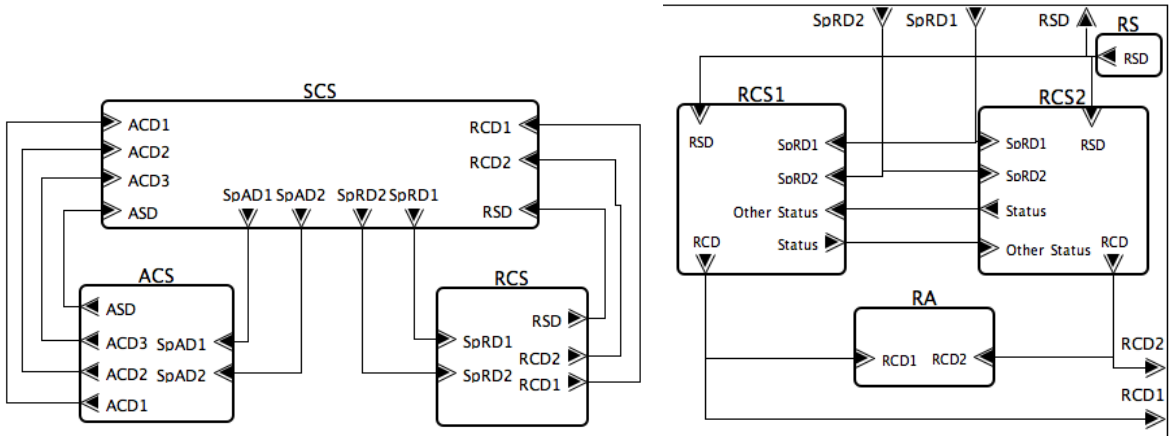
```
process Rudder_Control_Process
  features
    RSD: in event data port;
    SpRD1: in event data port;
    SpRD2: in event data port;
    RCD: out event data port;
    Status: out event data port;
    Other_Status: in event data port;
end Rudder_Control_Process;

-- input process implementation
process implementation Rudder_Control_Process.old
  subcomponents
    RCT: thread group Rudder_Control_Threads.servo;
  connections
    C1: event data port SpRD1 -> RCT.SpRD2;
    C2: event data port SpRD2 -> RCT.SpRD2;
    C3: event data port RCT.Status -> Status;
    C4: event data port RSD -> RCT.RSD ;
    C5: event data port Other_Status
        -> RCT.Other_Status;
    event data port RCT.RCD -> RCD;

  properties
    PALS_Properties::PALS_Connection_Id
        => "Rudder_Control" applies to C4;
    PALS_Properties::PALS_Connection_Id
        => "Rudder_Control" applies to C5;
    -- connections used in this logical
    -- synchronization.
    PALS_Properties::PALS_Connection_Id
        => "Supervisory_Control" applies to C1;
    PALS_Properties::PALS_Connection_Id
        => "Supervisory_Control" applies to C2;
end Rudder_Control_Process.old;

-- generic deition of the rudder control
-- threads and the thread group used
-- to define the pattern instantiation.
thread group Rudder_Control_Threads
  features
    RSD: in event data port;
    SpRD1: in event data port;
    SpRD2: in event data port;
    RCD: out event data port;
    Status: out event data port;
    Other_Status: in event data port;
end Rudder_Control_Threads;

-- input computation component.
-- it already has been formed after
-- pattern instantiation of "Rudder_Control".
thread group implementation
        Rudder_Control_Threads.servo
  subcomponents
    RCT: thread Rudder_ServoControl_Thread.impl;
    Synch_Servo:
            thread Rudder_Servo_SynchThread.impl;
  connections
    C1: event data port RSD
        -> Synch_Servo.RSD_in;
    C2: event data port Synch_Servo.RSD_out
            -> RCT.RSD;
    C3: event data port Other_Status
       -> Synch_Servo.Other_Status_in;
    C4: event data port Synch_Servo.Other_Status_out
            -> RCT.Other_Status;
```



(a) Top-level system component

(b) Replicated rudder control subsystem

Figure 4. Application of Multi-rate PALS pattern on the hierarchical control system

```
    C5: event data port RCT.RCD -> RCD;
    C6: event data port SpRD1 -> RCT.SpRD1;
    C7: event data port SpRD2 -> RCT.SpRD2;
    properties
      PALS_Properties::PALS_Period => 20 Ms;
      PALS_Properties::PALS_Id
              => "Rudder_Control";
      PALS_Properties::PALS_Offset => 0 Ms;
      Period => 20 Ms;
      -- SEI::Priority definition extended to
      -- define priority on thread group.
      SEI::Priority => 20;
      Deadline => 11 Ms;
      -- In this case, Output_Delay is a short
      -- form of the Min(Output_Time) property.
      META_Properties::Output_Delay => 4 Ms;

      PALS_Properties::PALS_Connection_Id
              => "Rudder_Control" applies to C1;
      PALS_Properties::PALS_Synchronizer_Operation
        => Last_Message_Only applies to C1;
      PALS_Properties::PALS_Connection_Id
              => "Rudder_Control" applies to C2;
      PALS_Properties::PALS_Connection_Id
              => "Rudder_Control" applies to C3;
      PALS_Properties::PALS_Synchronizer_Operation
        => Last_Message_Only applies to C3;
      PALS_Properties::PALS_Connection_Id
        => "Rudder_Control" applies to C4;
end Rudder_Control_Threads.servo;

thread Rudder_ServoControl_Thread
  features
    RSD: in event data port;
    SpRD1: in event data port;
    SpRD2: in event data port;
    RCD: out event data port;
    Status: out event data port;
    Other_Status: in event data port;
end Rudder_ServoControl_Thread;

thread implementation
        Rudder_ServoControl_Thread.impl
  properties
    Dispatch_Protocol => Periodic;
    Period => 20 Ms;
    Deadline => 11 Ms;
    SEI::Priority => 20;
    META_Properties::Output_Delay => 3 Ms;
    Timing_Properties::Dispatch_Offset
        => 0 Ms;
end Rudder_ServoControl_Thread.impl;

thread Rudder_Servo_SynchThread
  features
    RSD_in: in event data port;
    RSD_out: out event data port;
    Other_Status_in: in event data port;
    Other_Status_out: out event data port;
end Rudder_Servo_SynchThread;

thread implementation
    Rudder_Servo_SynchThread.impl
  properties
    Dispatch_Protocol => Periodic;
    Period => 20 Ms;
    Deadline => 3 Ms;
    SEI::Priority => 21;
    Timing_Properties::Dispatch_Offset
        => 0 Ms;
    META_Properties::Output_Delay => 1 Ms;
    PALS_Properties::PALS_Period => 20 Ms;
    PALS_Properties::PALS_Synchronizer_Type
        => Multi_Rate_Synchronizer;
end Rudder_Servo_SynchThread.impl;
```

Pattern instantiated components

The AADL code of the multi-rate synchronizer and the instantiated thread group are shown below:

```
process implementation Rudder_Control_Process.new
  subcomponents
    RCT_Group_Supervisory_Control:
        thread group Rudder_Control_Threads.supv;
  connections
    C1: event data port SpRD1
        -> RCT_Group_Supervisory_Control.SpRD2;
    C2: event data port SpRD2
        -> RCT_Group_Supervisory_Control.SpRD2;
    C3: event data port
        RCT_Group_Supervisory_Control.Status
        -> Status;
    C4: event data port RSD
        -> RCT_Group_Supervisory_Control.RSD ;
    C5: event data port Other_Status
        -> RCT_Group_Supervisory_Control.Other_Status;
```

```
    event data port
        RCT_Group_Supervisory_Control.RCD -> RCD;

  properties
    PALS_Properties::PALS_Connection_Id
        => "Rudder_Control" applies to C4;
    PALS_Properties::PALS_Connection_Id
        => "Rudder_Control" applies to C5;
    -- connections used in this logical
    -- synchronization.
    PALS_Properties::PALS_Connection_Id
        => "Supervisory_Control" applies to C1;
    PALS_Properties::PALS_Connection_Id
        => "Supervisory_Control" applies to C2;
end Rudder_Control_Process.new;

thread group implementation
        Rudder_Control_Threads.supv
  subcomponents
    RCT: thread group Rudder_Control_Threads.servo;
    Synch_Supervisory_Control:
        thread Rudder_Supervisor_SynchThread.impl;
  connections
    C1: event data port RSD -> RCT.RSD;
    C2: event data port RCT.RCD -> RCD;
    C3: event data port RCT.Status -> Status;
    C4: event data port Other_Status
        -> RCT.Other_Status;
    C5: event data port SpRD1
        -> Synch_Supervisory_Control.SpRD1_in;
    C6: event data port
        Synch_Supervisory_Control.SpRD1_out
        -> RCT.SpRD1;
    C7: event data port SpRD2
        -> Synch_Supervisory_Control.SpRD2_in;
    C8: event data port
        Synch_Supervisory_Control.SpRD2_out
        -> RCT.SpRD2;
  properties
    PALS_Properties::PALS_Period => 60 Ms;
    PALS_Properties::PALS_Id
        => "Supervisory_Control";
    PALS_Properties::PALS_Offset => 0 Ms;
    Period => 20 Ms;
    Deadline => 11 Ms;
    -- SEI::Priority definition extended to
    -- define priority on thread group.
    SEI::Priority => 20;
   META_Properties::Output_Delay => 5 Ms;

    PALS_Properties::PALS_Connection_Id
        => "Supervisory_Control" applies to C5;
    PALS_Properties::PALS_Synchronizer_Operation
        => Last_Message_Only applies to C5;
    PALS_Properties::PALS_Connection_Id
        => "Supervisory_Control" applies to C6;
    PALS_Properties::PALS_Connection_Id
        => "Supervisory_Control" applies to C7;
    PALS_Properties::PALS_Synchronizer_Operation
        => Last_Message_Only applies to C7;
    PALS_Properties::PALS_Connection_Id
        => "Supervisory_Control" applies to C8;
end Rudder_Control_Threads.supv;

thread Rudder_Supervisor_SynchThread
  features
    SpRD1_in: in event data port;
    SpRD1_out: out event data port;
    SpRD2_in: in event data port;
    SpRD2_out: out event data port;
end Rudder_Supervisor_SynchThread;

thread implementation
    Rudder_Supervisor_SynchThread.impl
  properties
    Dispatch_Protocol => Periodic;
    Period => 20 Ms;
    Deadline => 5 Ms;
    SEI::Priority => 22;
    Timing_Properties::Dispatch_Offset => 0 Ms;
    META_Properties::Output_Delay => 1 Ms;
    PALS_Properties::PALS_Period => 60 Ms;
    PALS_Properties::PALS_Synchronizer_Type
        => Multi_Rate_Synchronizer;
end Rudder_Supervisor_SynchThread.impl;
```

B. Second illustrative example

In this section, we describe an example of a fault-tolerant UAV system to model our proposed Multi-rate PALS pattern. This example is derived from an open-source UAV architecture, called Paparazzi [25]. Paparazzi is a low-cost UAV system with the support for both manual and autonomous flight navigation based on variety of onboard sensor modules, e.g. GPS, infrared, gyroscope. Its on-board autopilot software performs two major operations: flight navigation and flight stabilization. During the autonomous flight operation, the navigation controllers provides flight guidance commands for roll, pitch and throttle setpoints to the stabilization controllers which then compute the control commands and sends to the aircraft actuators, e.g. aileron, motor and elevator. In this system, the UAV can also be manually controlled from the ground station. The ground station sends radio commands to manage the flying modes and any necessary flight guidance commands or flight plans to fly the aircraft.

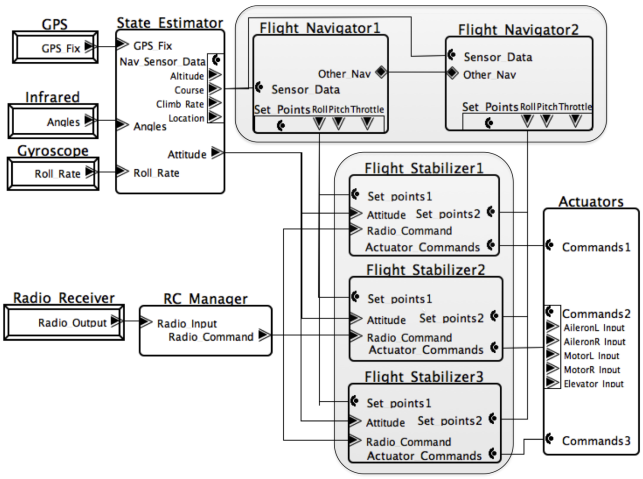


Figure 6. AADL diagram of the extended Paparazzi architecture.

Although the Paparazzi architecture has some built-in support for failsafe operation, e.g. flying the aircraft according to a safe (pre-defined) plan in the case of radio communication failure or just gliding the aircraft in case of hardware failures, it does not implement any standard fault-tolerance practice of hardware or software redundancy for any of its safety and mission critical components.

To describe the synchronization problem in a multi-rate distributed operations, we therefore have extended design of the Paparazzi architecture with different fault-tolerant mechanism for the major functional components. Since extending Paparazzi to

become a full-fledged avionics system is not the primary goal of this paper, we focus only on the autopilot software and replicated its navigation and stabilization controllers.

Extended Paparazzi architecture: Figure 6 gives the high-level AADL block diagrams of the extended design[7].

The sensors include GPS, Infrared and a Gyroscope which supply the position, altitude, attitude and roll rate measurement to be used by the flight navigation controllers, Flight_Navigator$_i$ and flight stabilization controllers, Flight_Stabilizer$_j$. Ground station commands are received by the Radio_Receiver and propagated to the Flight_Stabilizer by the radio command manager, RC_Manager. In the original design, the sensor state estimation, flight navigation, stabilization are done by a single set of controllers executing on the single processor. In this design, we replicate the flight navigation controllers as an active-standby system to tolerate any single hardware failure and the flight stabilization components using a triple-redundant system to tolerate any single hardware/software failure with the voting logic implemented at the actuators. The State_Estimator supplying sensor state estimation must also replicated to prevent any single-point failure (not shown in the figure to simplify the diagram).

We consider the original task rates of these component. In this system, Flight_Stabilizer operates at 20Hz (50ms), Flight_Navigator operates at 4Hz (250ms) and Radio_Manager operates at ,

Mode consistency problem: The flight can operate in two modes: *auto* and *manual*. In the $auto$ mode, each flight stabilizer must use the flight guidance command computed by the flight navigators, while in the $manual$ mode flight stabilizers use the flight guidance commands of the ground station. Flight navigators can also operate in two modes: $normal$ and $home$. In the $normal$ mode, it navigates the flight according to fight plan and the waypoint trajectory; however if certain categories of failure occurs, such as low battery, GPS failure or the aircraft going out of the expected fligh region, the navigator may switch to the $home$ mode and navigates toward the safe location, e.g. ground station.

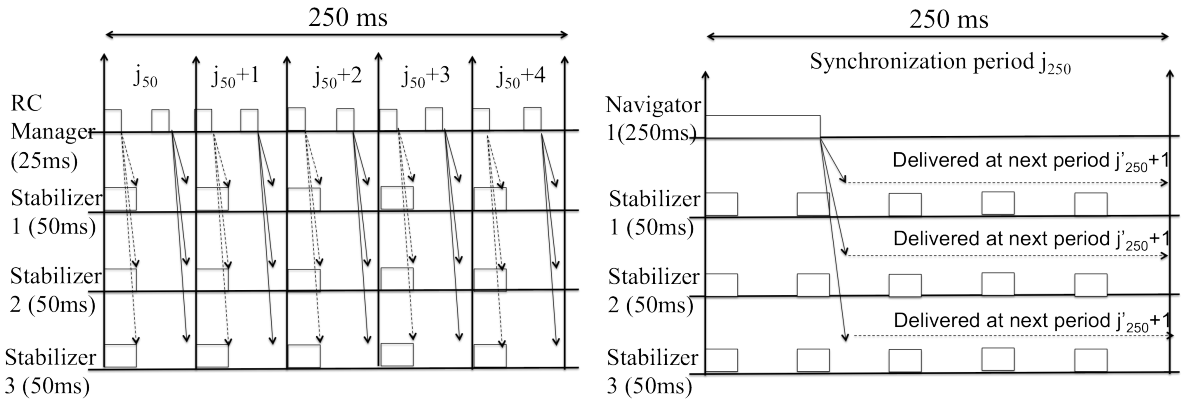
To illustrate the challenge to guarantee consistency in this design, consider that the current mode is *auto* and the aircraft is flying near the edge of the planned flying zone. At this momen, the ground station sends a command to switch to *manual* mode. Due to the asynchronous interaction, Flight_Stabilizer$_1$ receives the command in period j, but Flight_Stabilizer$_2$ and Flight_Stabilizer$_3$ receive in period $j + 1$. Nearly at the same time, the Flight navigators received the GPS location and decided to route the aircraft toward the ground station. Again due to the asynchrony, while operating in the *auto* mode, Flight_Stabilizer$_2$ reads the new flight navigation command, but not the Flight_Stabilizer$_3$. As a result, the Flight_Stabilizer$_3$ still operates on the old navigation command. It becomes clear that the voting logic will not be able to decide upon a valid actuator command and will force the UAV to trigger other failsafe mode, although none of the stabilizers are indeed failed!

Multi-rate PALS system based mode consistency: In Figure 8, we show the provided logical synchronization between the radio control manager (RC_Manager), flight stabilizers and flight navigators.

In Figure 7(a), we show the logically synchronous interaction between the three flight stabilizers with a synchronization period of 50ms. In each synchronization period j_{50}, each flight stabilizer receive two commands from the radio control manager. The Multi-rate PALS guarantees that these redundant stabilizers receive identical updates from the radio control manager. Similarly in Figure 7(b), the flight stabilizers receive identical updates of the setpoints from a flight navigator. Similar execution behavior is guaranteed with respect to the updates from other flight navigator.

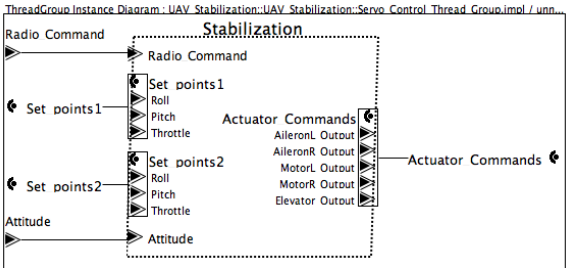
Pattern instantiation: Similar to the previous example, Figure 8 illustrates the instantiation of a multi-rate synchronizer in this example.

[7]The AADL thread description of the original paparazzi UAV is described at [26]

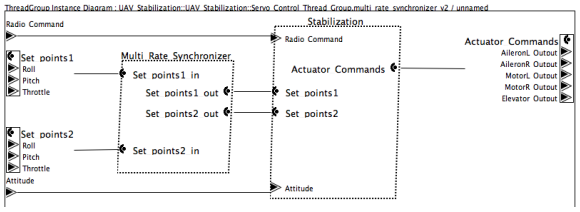


(a) Interaction between RC_Manager and Flight_Stabilizer$_j$, j=1...3 (b) Interaction between Flight_Navigator$_1$ and Flight_Stabilizer$_j$, j=1...3

Figure 7. Logical synchronization in the extended UAV architecture



(a) AADL diagram of a Flight_Stabilizer process



(b) AADL diagram of a Flight_Stabilizer process with the multi-rate synchronizer

Figure 8. Instantiation of the multi-rate synchronizer in the UAV example