COMPLEXITY-REDUCING DESIGN PATTERNS FOR CYBER-PHYSICAL SYSTEMS

Dr. Darren Cofer

Rockwell Collins, Inc.
400 Collins Rd. NE
Cedar Rapids, IA 52498

September 2011

FINAL REPORT

Approved for Public Release, Distribution Unlimited.

TABLE OF CONTENTS

Section **Page**

1.0 Summary 1
2.0 Introduction 2
3.0 Methods, Assumptions, and Procedures 4
3.1 Design Problems and Characteristics 4
3.2 Design Flow and Methodology 4
3.3 Design Pattern Models 4
3.4 System Architecture Models 4
3.5 Pattern Verification 5
3.6 System Verification 5
4.0 Results and Discussion 6
4.1 Design Problems and Characteristics 6
4.2 Design Flow and Methodology 35
4.3 Design Pattern Models 55
4.4 System Architecture Models 72
4.5 Pattern Verification 86
4.6 System Verification 118
5.0 Conclusions 131
6.0 References 132
APPENDIX A AADL MODELS OF AVIONICS SYSTEM 136
APPENDIX B Mode Logic Overview 152
List of Acronyms 154

LIST OF FIGURES

Figures **Page**

Figure 1 – System Development Process Design Flow 2
Figure 2 – Design Flow 35
Figure 3 – Specification Phase Design Flow and Tools 36
Figure 4 – System Development Phase Design Flow and Tools 43
Figure 5 – Pattern Transform Editor: Transform Tab 47
Figure 6 – Pattern Transform Editor: Instantiation Tab 48
Figure 7 – Pattern Verifier for Instantiation 49
Figure 8 – Model Verification View 50
Figure 9 – System Analysis Concerns View 51
Figure 10 – Lute Theorem for Verifying Process Deadlines 53
Figure 11 – AADL graphical model for PALS pattern 57
Figure 12 – AADL graphical model for Replication pattern (input context) 59
Figure 13 – After application of Replication pattern (replicate inputs and outputs) 59
Figure 14 – After application of Replication pattern (shared inputs and merged outputs) 60
Figure 15 – Leader Thread T Inserted Into Each Process 62
Figure 16 – Voter thread inserted into process by Fusion pattern 65
Figure 17 – AADL graphical model of the system blocks for multi-rate PALS 70
Figure 18 – Process component before inserting the multi-rate synchronizer 70
Figure 19 – Process component after inserting the multi-rate synchronizer 70
Figure 20 – Top Level Logical and Physical Overview 72
Figure 21 – Avionics System Architecture 73
Figure 22 – Typical Primary Flight Display 74
Figure 23 – Flight Control System Overview 75
Figure 24 – Typical Flight Control Panel 76
Figure 25 – Flight Guidance System Overview 77
Figure 26 – Flight Guidance Process Overview 78
Figure 27 – Autopilot System Overview 79
Figure 28 – Autopilot Process Overview 80
Figure 29 – Air Data System Overview 80
Figure 30 – Air Data Process Overview 81
Figure 31 – IMA Platform Overview 82

Figure 32 – Fast CCM Architecture 83
Figure 33 – System Design through Pattern Application 84
Figure 34 – Verification in Design Flow 87
Figure 35 – PALS design pattern 89
Figure 36 – PALS timeline 91
Figure 37 – PALS causality constraint 92
Figure 38 – PALS period constraint 92
Figure 39 – A System Using Perfectly Synchronized Clocks 94
Figure 40 – Logical Equivalence to Causality Violation 95
Figure 41 – Clock C_1 leads Clock C_2 95
Figure 42 – Clock C_2 leads Clock C_1 96
Figure 43 – Synchronous Leader Selection algorithm in C++ 102
Figure 44 – NuSMV implementation of Node (called device) 106
Figure 45 – Main NuSMV Module 107
Figure 46 – Multi-rate PALS design pattern 110
Figure 47 – Multi-rate PALS timeline 113
Figure 48 – Final FCS System Architecture 119
Figure 49 – Contract for Flight Control System 121
Figure 50 – AGAT PSL Fragment 122
Figure 51 – Facts for Leader Select implemented in FCS 123
Figure 52 – Architecture Data Dependencies 124
Figure 53 – FGS contract 126
Figure 54 – AP Contract 126
Figure 55 – AGAT Plug-In 128
Figure 56 – Verification Results 128
Figure 57 – Counterexample 129
Figure 58 – Final Proof Result 130
Figure 59 – Impact of correct-by-construction development process 131
Figure 60 – A Simple Mode 152
Figure 61 – An Arming Mode 153
Figure 62 – A Capture/Track Mode 153

LIST OF TABLES

Table **Page**

Table 1 – Summary of System Design Problems 7
Table 2 – Information Needed to Define a Component..... 38
Table 3 – Fundamental Model Transformations..... 39
Table 4 – Information Needed to Define a Design Pattern..... 41
Table 5 – AADL Constructs Supported by the SysML Translator..... 52
Table 6 – Analysis times to check requirements R1–R4 on group of *N* nodes 108

1.0 SUMMARY

Advanced capabilities planned for the next generation of commercial and military aircraft will be based on complex new software. These aircraft will incorporate adaptive control algorithms and sophisticated mission software providing enhanced functionality and robustness in the presence of failures and adverse flight conditions. Unmanned aircraft have already displaced manned aircraft in most surveillance missions and will soon do so in combat missions with increasing levels of autonomy. Manned and unmanned aircraft will be required to coordinate their activities safely and efficiently in both military and commercial airspace.

The *cyber-physical systems* that provide these capabilities are so complex that software development and verification is one of the most costly development tasks and therefore poses the greatest risk to program schedule and budget. Without significant changes in current development processes, the cost and time of software development will become the primary barriers to the deployment of the advanced capabilities needed for the next generation of military aircraft.

DARPA's META program seeks to significantly improve the design, manufacture, and verification of complex cyber-physical systems. The work described in this report directly addresses this goal by allowing the system architecture to be composed from libraries of complexity reducing design patterns with formally guaranteed properties. This allows new system designs to be developed rapidly using patterns that have been shown to reduce unnecessary complexity and coupling between components. This work also deeply embeds formal verification into the design process to enable correct-by-construction development of systems that work the first time. The use of components with formally specified contracts, design patterns that provide formally guaranteed properties, and an architectural modeling language with a formal semantics ensures that the system design is known to meet its requirements even before it is implemented.

Rockwell Collins and its teammates, the University of Illinois at Urbana-Champaign (UIUC), the University of Minnesota (UMN), and the WW Technology Group (WWTG), addressed these challenges by developing capabilities in three key areas:

1. Complexity-reducing system design patterns with formally guaranteed properties, supporting correct-by-construction composition of system designs
2. Architectural modeling and analysis to support virtual integration, composition, and verification of system-level properties
3. Automated formal verification deeply embedded in the system design process in order to prevent errors, resulting in dramatic schedule efficiencies

2.0 INTRODUCTION

The design flow and supporting tool chain for the project are shown in Figure 1. The overall development process can be divided three phases, Specification, System Development, and Foundry.

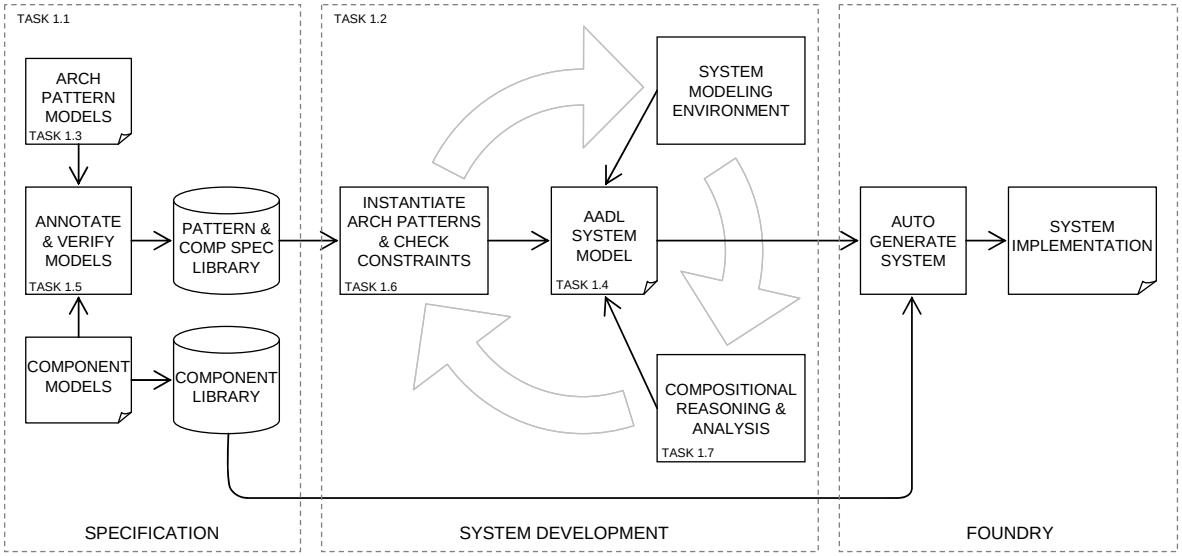


Figure 1 – System Development Process Design Flow

In the Specification phase, models of architecture patterns and components are developed and verified. Each model is annotated with its verified properties and the constraints that must be satisfied for those properties to hold. Creation and verification of these patterns is an infrastructure activity outside of the main development path. As new patterns are identified, they will be formally modeled and key properties specified. The properties will be verified through a combination of test and formal methods, and constraints for their correct usage specified. These constraints may include the external environment of the system, the physical system being controlled (the *plant*, in control theory terms), or interfaces with other systems. Existing component models that will be used in system design must also have their properties (requirements) specified and verified, with necessary usage constraints identified. The resulting annotated models of system architecture patterns and components are published in a library for consumption by the system development phase.

In the System Development phase, architecture patterns are instantiated to create a system model. Starting at a relatively high level of abstraction, a system architecture is created or instantiated from the library to create an initial system model. This model is then refined by repeated applications of design patterns and manual modeling to create a detailed system model from which an implementation can be generated. During system development, a transformed model may be loaded into the architectural modeling environment and checked using the performance, security, and safety tools available in that environment. At the appropriate levels of refinement, requirements for the system model itself are formalized and mechanically verified.

Finally, the Foundry phase is entered and the system implementation for a given target is generated from the system model and referenced component models. The Foundry phase draws upon component models and the fully elaborated system model to auto-generate an implementation. This may be a prototype system to support evaluation in a lab or other test

environment, or the final system for delivery and deployment. The system model provides sufficient level of detail to generate component interface code ("glue code"), along with network or system calls to services provided by the underlying platform. Component specifications must provide sufficient level of detail for their implementation to be linked in with the interface code.

This project developed technology for the Specification and System Development phases.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

This section provides a summary of the main project results. These are described in greater detail in the following sections.

3.1 Design Problems and Characteristics

This study identified the system design problems to be addressed through the use of complexity-reducing patterns, specified sources of complexity, relevant parameters, and desired resolution, and selected candidate problems to be addressed in the project. The study was conducted by surveying design data and development process feedback for a set of existing aircraft systems. Based on the data collected, the most critical problems in developing and verifying complex cyber-physical systems were determined to be 1) asynchronous computing, 2) unreliable computing platforms, and 3) untrusted or unreliable components.

3.2 Design Flow and Methodology

This section defines a design flow methodology for the rapid development of cyber-physical systems based on the application of verified design patterns on verified system component models. This design flow models the system at increasing levels of abstraction and embeds verification at all stages through automated support for compositional reasoning about system correctness, formally verified design patterns, and components with guaranteed properties.

This project developed technology for the specification and system development phases. Tools developed to support this design flow include 1) an Eclipse plug-in to translate SysML models to and from AADL, 2) and Eclipse plug-in to check structural properties of an AADL model, 3) a pattern application tool implemented as part of the EDICT tool suite, and 5) an Eclipse plug-in to generate system architectural models for which behavioral properties can be formally verified using the NuSMV [30] or Kind [35] model checkers.

3.3 Design Pattern Models

This section describes the design patterns developed in the project. Patterns were developed for PALS, Replication, Leader Selection, and Fusion (voting or source selection). For each pattern, a description of the pattern's behavior is provided, as well as how a developer would use the pattern in a system design. Also provided for each pattern are the arguments for the pattern, the environmental assumptions that must be satisfied before the pattern can be applied, the guarantees that are provided by the pattern after it is applied, a textual description of the algorithm for applying the pattern to a system model, and exemplar before/after AADL models that illustrate the effect of the pattern application.

3.4 System Architecture Models

This section describes the architectural models developed in the project. These architectural models provide examples that can be used to evaluate the design and verification tools created. The architectural models describe a Flight Control System (FCS) for a typical regional jet aircraft. An overview of this architecture is provided along with detailed SysML and AADL specifications. An example shows how this architectural model can be generated from simpler "sunny day" architecture through application of a sequence of patterns.

3.5 Pattern Verification

This section describes the verification of the design patterns developed in the project. A key element of these design patterns is that they provide guarantees of correct behavior when used in accordance with their specifications. Their behavior is proven through the use of formal methods as part of the pattern development process. The verification effort for the generic pattern is amortized over all subsequent instantiations of the pattern in specific system models. This amounts to *reuse* of the initial pattern verification. Analysis of system-level behavior can subsequently make use of the proven pattern guarantees without having to reprove them.

3.6 System Verification

This section describes the compositional verification of system level properties. One of the key goals of this process is to reuse the verification already performed on the components and design patterns. To do this, it is first necessary to prove that that the system architecture satisfies the assumptions made by each component or design pattern. Once this is done, the guarantees provided by the component or design pattern can be used in proving that the system constructed from design patterns and components provides the desired system behavior. To do this, a formal semantics must be assigned to the system architectural modeling language that correctly incorporates the assumptions and guarantees of the system components and design patterns.

4.0 RESULTS AND DISCUSSION

This chapter discusses the main project results in detail.

4.1 Design Problems and Characteristics

This study identified the system design problems to be addressed through the use of complexity-reducing patterns, specified sources of complexity, relevant parameters, and desired resolution, and selected candidate problems to be addressed in the project. The study was conducted by surveying design data and development process feedback for a set of existing aircraft systems. Based on the data collected, the most critical problems in developing and verifying complex cyber-physical systems were determined to be:

- Asynchronous computing
- Unreliable computing platforms
- Untrusted or unreliable components

4.1.1 Sources of Data

The sources consulted in compiling the design problems described below are listed in the References (Chapter 6.0) in indicated by reference in the tables below. These documents, presentations, and interviews describe different types of errors encountered in the development of complex systems, their characteristics, and (sometimes) their causes.

4.1.2 Catalog of System Design Problems

Table 1 is a summary of the system design problems identified during our survey, along with their possible solutions. The problems listed here are candidates to be addressed through the use of complexity-reducing design patterns. A detailed description of each design problem is provided in the following subsections.

Table 1 – Summary of System Design Problems

	Design Problem	Description	Approach
1	Asynchronous Computation	Race conditions lead to inconsistent state information across asynchronous nodes in distributed systems.	Implement real-time logical synchrony to simplify agreement on global state.
2	Unreliable Computing Platform	Incorrect designs for fault tolerance result in system failures.	Use of verified architectural design patterns for fault tolerant design.
3	Unreliable Data Sources	Incorrect designs for voting or selection of redundant sensors lead to use of invalid inputs.	Use of verified architectural design patterns for source selection and voting.
4	Unreliable Actuators	Failure of incorrect arrangement of actuators leads to improper actuation of controlled devices.	Use of redundant actuators with feedback monitoring and logic to override actuator failures.
5	Unintended Interaction/ Interference	Failure or unexpected behavior of a component causes the failure of a logically unrelated component.	Use of verified architectures that prevent interference. Use of assume/guarantee contracts.
6	Resource Contention	Incorrect use of shared resources leads to inconsistent global state, corrupted resource, or denial of service.	Use of verified architectural design patterns for concurrent access to shared resources.
7	Platform Hardware Dependencies	Dependencies on low-level hardware or platform behavior make it prohibitively expensive to port systems to new platforms.	Use of layered virtual interfaces based on formally specified assume/ guarantee contracts. Automated generation of correct by construction implementations for each level of design refinement.
8	Allocation to the Target Platform	Incorrect prediction of required physical resources leads to late changes to system architecture and extensive rework..	Use of system design patterns and auto-generation of correct-by-construction implementations.
9	Untrusted/ Unreliable Components	Use of complex components that provide highly desirable performance are prohibitively expensive to verify.	Use of formally verified simplex design pattern that monitors boundary conditions and switches to a simpler reliable controller.

	Design Problem	Description	Approach
10	Requirements Errors	Missing or incorrect requirements are not discovered until system integration resulting in extensive rework.	Use executable system models and automated compositional verification to enable early identification of requirements errors.
11	Unspecified Dependencies	Missing or incorrect dependencies of components on other components are not discovered until system integration.	Use of formal assume/ guarantee contracts and automated compositional verification to enable early identification of unspecified dependencies.
12	Poorly Defined Interfaces	Lack of industry standards for certain classes of interfaces leads to significant system redesign even for small changes.	Use of standard design patterns for interface definitions to guide developers.

4.1.2.1 Asynchronous Computation

Problem	Asynchronous Computation
Description	Race conditions lead to inconsistent state information across asynchronous nodes in distributed systems.
	Many cyber-physical systems must be implemented as redundant, distributed systems in order to provide the necessary level of reliability. To correctly perform their function, the individual nodes of these systems must agree on some part of the global system state. Developing protocols to achieve such agreement in an asynchronous environment can be extremely difficult. Great care must be taken to establish the necessary coordination between the distributed components to avoid race and deadlock conditions and to implement the correct behavior. Failure to do this can result in subtle timing dependences between components that manifest themselves as system failures that are extremely difficult to reproduce and debug, often referred to as "no fault found" (NFF) errors.
Approach	Implement real-time logical synchrony to simplify agreement on global state.
	Use a design pattern that implements logical synchrony in real time over the physically asynchronous platform to simplify achieving agreement on the relevant portion of the global system state. This allows developers to design and verify a distributed, redundant system as though all nodes execute synchronously, making design and verification much simpler. This synchronous design can then be distributed over a physically asynchronous architecture following constraints that ensure that the logical correctness of the synchronous design is preserved. As the design pattern is mapped onto a specific architecture, the characteristics of that architecture (e.g., maximum clock error, maximum communication delay) can be checked against the timing constraints specified in the pattern.
Parameters	• Global state to be synchronized • Nodes to be synchronized • Timing constraints for synchronization

Documentation – Asynchronous Computation

Source	Supporting Data
[2]	*The need to consider failure of software in the system MTBF calculations has arisen, even though software doesn't "fail" in the traditional hardware sense.*
[7]	*Airlines voted NFF "the most important issue" in a poll of delegates at the annual Avionics Maintenance Conference (AMC) in 2004 and it continues to be amongst the top-most items AMC discusses each year.*
[7]	*At that time (1997), ATA estimated annual NFF costs for an airline operating 200 aircraft at $20 million, or $100,000 per aircraft per year. Remove the economies of scale and add a decade of inflation, and at least twice that figure could be closer to the truth today.*
[7]	*The chief culprits are avionics systems, which by function, account for 74 percent of NFF events. By comparison, pneumatics add 19 percent, hydraulics 4 percent and others 3 percent, according to Air Canada. Moreover, the typical, industry-wide average NFF-rate for avionics systems is around 30 percent, according to Axel Müller, manager processes and controlling, Lufthansa Technik Component Maintenance Services, who chairs the ARINC 672 working group. But this is down from an average NFF rate of 50 percent some 20 years ago.*
[16]	*Many avionics systems must be implemented as redundant, distributed systems in order to provide the necessary level of fault tolerance.*
[16]	*In many IMA architectures, each processing resource is driven by its own clock. While these clocks may have the same period, they execute asynchronously relative to each other with their own offset, drift, and jitter. This results in an architecture in which synchronous components execute asynchronously relative* *to each other.*
[16]	*Great care must be taken to establish the necessary coordination between the distributed components to avoid race and deadlock conditions and to implement the correct behavior*
[16]	*This paper presents a simple design pattern, Physically Asynchronous/Logically Synchronous (PALS) that allows developers to design and verify a distributed, redundant system as though all nodes execute synchronously. This synchronous design can then be distributed over a physically asynchronous architecture following a few simple constraints that ensure that the logical correctness of the synchronous design is preserved.*
[16]	*Thirty hours required to formally verify (model check) an asynchronous implementation of a protocol as compared to approximately 30 seconds to formally verify a synchronous implementation of the same protocol.*

[3]	*"… the process model community has found that software engineering, software development, system engineering, and other activities are integrated, have dependencies, and cannot be adequately performed and optimized independently of each other …"*

4.1.2.2 Unreliable Computing Platform

Problem	Unreliable Computing Platform
Description	Incorrect designs for fault tolerance result in system failures.
	Cyber-physical systems with high reliability requirements must be implemented as redundant, distributed systems due to the failure rates of individual hardware computing elements. Standard fault-tolerant designs have been developed over the years to provide high levels of reliability for various fault models. Often, these designs are implemented over bounded asynchronous networks. Correct development of concurrent, asynchronous, distributed algorithms is one of the most difficult design problems in software engineering. Unfortunately, engineers frequently develop fault-tolerant algorithms as a variation of a design they're familiar with. The debugging of these algorithms can be extremely expensive and often fails to find all errors. The resulting systems appear to work in the lab but do not actually provide the required level of reliability in the field.
Approach	Use of verified architectural design patterns for fault tolerant design.
	Develop a library of formally verified reusable design patterns that can be applied to a functional element to produce a correct fault tolerant design with known reliability for specific fault models. Formal verification of fault tolerant design patterns is greatly simplified if the nodes execute synchronously rather than asynchronously. For this reason, the fault tolerant design patterns may be based on the logical synchrony design pattern.
Parameters	• Functional logic to be made fault tolerant • Required reliability of fault-tolerant design • Reliability of each computing element • Failure model of each computing element

Documentation – Unreliable Computing Platform

Source	Supporting Data
[1] [12]	*[Aerospace systems] design is intrinsically complex due to high reliability requirements, …*
[2]	*The need to consider failure of software in the system MTBF calculations has arisen, even though software doesn't "fail" in the traditional hardware sense.*
[2]	*The implementation of a reliable, well tested single software design system is a cost effective alternative to costly multiple "independent" program developments.*
[2]	*Recent developments show unexpected correlation of errors in independently developed software designs, reducing expected improvements in reliability. This, coupled with the significant portion of errors imbedded in requirements, points toward well tested single software designs as a method of meeting software reliability requirements.*
[2]	*Models for predicting reliability of hardware exist but the software intensive system doesn't have the benefit of realistic reliability models.*
[16]	*In many IMA architectures, each processing resource is driven by its own clock. While these clocks may have the same period, they execute asynchronously relative to each other with their own offset, drift, and jitter. This results in an architecture in which synchronous components execute asynchronously relative* *to each other.*
[8] [9] [10]	*Insufficient validation of new concepts*
[8] [9] [10]	*Pushing too many technology envelopes during a production program*

4.1.2.3 Unreliable Data Sources

Problem	Unreliable Data Sources
Description	Incorrect designs for voting or selection of redundant sensors lead to use of invalid inputs.
	Sensors for cyber-physical systems often have higher failure rates than the system itself, requiring that redundant, distributed sensors be used to provide information about the environment and the physical system being controlled. Components for voting or selecting among these redundant sensors are often quite complex, using history and complex algorithms to estimate the true value, logic to declare sensors failed when their outputs deviate too much, and logic to readmit sensors when their outputs have healed. Often, the precise behavior of these critical components is not well understood. For example, the magnitude of the transient change in voter outputs may not be known for all combinations of voter state and sensor failures. Pathological conditions in which the voter declares a good sensor failed and instead uses a failed sensor are also surprisingly common. This problem may be further complicated by the fact that there are often multiple consumers of data sources that may use different voting, selection, or estimation techniques, resulting in divergent views of the sensed state.
Approach	Use of verified architectural design patterns for source selection and voting.
	Develop a library of reusable design patterns for sensor voting or selection that have been formally verified to be correct for a given fault model of the sensors. Each pattern should specify the possible error in the voter output for all possible combinations of sensor outputs. Pathological conditions in which failed sensors are used or good sensors are declared failed should be eliminated through formal verification.
Parameters	• Number of sensors • Range and type of sensor inputs • Range and type of sensor outputs • Fault/error model of each sensor • Required accuracy of voter output • Thresholds used in declaring sensors failed • Readmission policy for healed sensors

Documentation – Unreliable Data Sources

Source	Supporting Data
[1] [12]	*[Aerospace systems] design is intrinsically complex due to high reliability requirements, …, and extensive safety-critical sensor dependencies.*
[2]	*The need to consider failure of software in the system MTBF calculations has arisen, even though software doesn't "fail" in the traditional hardware sense.*
[8] [9] [10]	*Difficult to import legacy product data into architecture modeling tools*

4.1.2.4 Unreliable Actuators

Problem	Unreliable Actuators
Description	Incorrect designs for coordination and selection of redundant actuators leads to loss of control or degraded system performance
	Cyber-physical systems are required to interact with the world around them and often use actuators to dynamically control system behavior and effect changes in interfacing systems. When reliable actuation is required for safety or overall system reliability redundant actuators are often used to provide "operate through" capabilities in the presence of an actuator failure. The arrangement of the actuators on the physical device, the selection logic and actuator coordination strategy, and the information used to monitor the actuator performance and determine failure all have major impacts on the ability of the system to overcome actuator failures. Improper alignment of the system requirements for control stability, error recovery and fault tolerance with the actuator management approach can lead to situations where actuator failures are not detected or the switching logic does not provide robust and rapid fail over, leading to system failure.
Approach	Use of verified architectural design patterns for actuator feedback monitoring and coordination and selection of multiple actuators
	Develop a library of reusable design patterns for actuator monitoring and control strategies that have been formally verified to be correct. Each library element will address the type of coverage required for actuator failures, the allowable latency in detection of actuator failures and requirements for actuator recovery. Design patterns may be refined to address classes of actuators to address domain specific issues.
Parameters	• Error coverage for actuation system • Allowable detection latency • Fault/error model of each actuator • Required data to determine actuator failure • Readmission policy for healed actuators

Documentation – Unreliable Actuators

Source	Supporting Data
	C. Walter, B. LaValley, WW Technologies, industrial experience, Virginia Class ship control system
	Steve Miller, Industrial experience, autopilot actuator.

4.1.2.5 Unintended Interaction/ Interference

Problem	Unintended Interaction/ Interference
Description	Failure or unexpected behavior of a component causes the failure of a logically unrelated component.
	To reduce verification costs, architectural designs are often chosen that isolate components from other components. Typically, high-criticality, highly verified components are protected from low-criticality, lightly verified components. These mechanisms for non-interference allow one part of the system to be modified and verified without having to verify the other parts. Interference can occur in many ways, but they all involve interactions through shared resources, such as altering the memory space of another task, shifting the time at which a task executes, restricting access to the bus, thermal interference through heat conducting materials, or EMF interference through unshielded space. In many cases, interference occurs simply because all the dependencies of a component were never explicitly and comprehensively defined. In these cases, interference occurs because the designers of other components were unaware of the undocumented constraints on their design.
Approach	Develop design patterns for the specification of non-interference in the system architecture.
	Design patterns for specifying non-interference of each desired type should be defined and used in specifying a system architecture. Refinement of the system design should automatically generate a correct-by-construction implementation that enforces the specified non-interference or tools should check that non-interference is maintained provided the fault-model of each component is not exceeded. At the same time, components in a system must interact if the overall system is to do anything useful. Precisely defining these *intended* interactions should be done as part of the specification of the system architecture. This defines the contract between a module and the rest of the system that can be used in the formal verification of the component and in the compositional verification of the system.
Parameters	• Partition Boundary • Type of Interference to be Prevented • Criticality of Region • Fault Model of Each Component • System Fault Propagation Model

Documentation – Unintended Interaction/ Interference

Source	Supporting Data
[2]	*The need to consider failure of software in the system MTBF calculations has arisen, even though software doesn't "fail" in the traditional hardware sense.*
[2]	*Models for predicting reliability of hardware exist but the software intensive system doesn't have the benefit of realistic reliability models.*
[2]	*The lack of software failure data for highly reliable systems has been a hindrance.*
[2]	*A reliability growth curve for a high use system is needed.*
[16]	*The introduction of Integrated Modular Avionics (IMA) [1], [2] has allowed developers to implement increasingly sophisticated avionics systems by taking advantage of the growing processing power and memory available in modern integrated circuits.*
[16]	*In many IMA architectures, each processing resource is driven by its own clock. While these clocks may have the same period, they execute asynchronously relative to each other with their own offset, drift, and jitter. This results in an architecture in which synchronous components execute asynchronously relative* *to each other.*
[3]	*"… the process model community has found that software engineering, software development, system engineering, and other activities are integrated, have dependencies, and cannot be adequately performed and optimized independently of each other …"*
[8] [9] [10]	*Poor understanding of architecture caused by ambiguous / incomplete architecture representation*

4.1.2.6 Resource Contention

Problem	Resource Contention
Description	Incorrect use of shared resources leads to inconsistent global state, corrupted resource, missed deadlines, or denial of service.
	Components of cyber-physical systems routinely access shared resources such as buses, memory, processors and data bases. Due to the inherently distributed, redundant, and concurrent nature of these systems, access to these shared resources must be controlled to ensure the resource is not corrupted, that users of the resource are not provided incorrect information by the resource resulting in inconsistent global state, or that access to the resource is denied to other components such that real-time deadlines are missed.
Approach	Use of formally specified and verified design patterns for concurrent real-time access to shared resources.
	Develop a library of verified reusable design patterns that enforce correct access to the shared resource (i.e., resource managers). While the specific means of accessing a shared resource will vary from resource to resource, it is likely that generic patterns can be developed to protect entire classes of resources. In addition to formally specifying the environmental assumptions that must be satisfied for each resource and the guarantees to be provided by each resource, there will be invariants on the resource itself that must be maintained. Each design pattern should be formally verified to ensure that it provides its guarantees if these assumptions are met and that the resource invariants are maintained. Finally, tools must be developed that take account of the time required to access shared resources and ensure that the system's real-time constraints are still met.
Parameters	• Allowed Accesses • Assume/Guarantee Contract for each Access • Resource Invariants

Documentation – Resource Contention

Source	Supporting Data
[2]	*The need to consider failure of software in the system MTBF calculations has arisen, even though software doesn't "fail" in the traditional hardware sense.*
[2]	*The implementation of a reliable, well tested single software design system is a cost effective alternative to costly multiple "independent" program developments.*
[16]	*In many IMA architectures, each processing resource is driven by its own clock. While these clocks may have the same period, they execute asynchronously relative to each other with their own offset, drift, and jitter. This results in an architecture in which synchronous components execute asynchronously relative to each other.*
[3]	*"… the process model community has found that software engineering, software development, system engineering, and other activities are integrated, have dependencies, and cannot be adequately performed and optimized independently of each other …"*

4.1.2.7 Platform Hardware Dependencies

Problem	Platform Hardware Dependencies
Description	Dependencies on low-level hardware or platform behavior make it prohibitively expensive to port systems to new platforms.
	Cyber-physical applications that directly manipulate and access the hardware platform are dependent on the availability of those exact hardware structures. This makes it prohibitively expensive to port applications to different hardware platforms and to exploit continual improvements in hardware speed, reliability, and capability. Too often, abstractions of the hardware platform are bypassed to meet system performance requirements.
Approach	Use of layered virtual interfaces based on formally specified assume/ guarantee contracts. Automated generation of correct-by-construction implementations for each level of design refinement.
	By defining standard interfaces that hide the details of the hardware structure, new hardware can be introduced without changing the application software by adding a small amount of software or firmware that implements the interface over the new hardware. This technique can be used not just for hardware, but also for layers of software, where each layer implements its interface over the services provided by the next lower layer. This allows entire layers of standard services to be reused. Each interface should be defined in terms of the services it provides and should hide as much of the implementation as possible from users of the interface. These services should be formally specified using an assume/guarantee contract that specifies the assumptions made by the service and the guarantees provided by the service. Ideally, a correct-by-construction implementation of each layer would be automatically generated from the specification of the services it provides and the services of lower layers it uses. Finally, support for real-time virtual integration should be provided so that prediction of system performance and capacity can be performed continuously during system development.
Parameters	• Standard Virtual Interfaces for Cyber-Physical Systems • Formally Specified Contract for Each Virtual Interface • Specification of Performance Overhead

Documentation – Platform Hardware Dependencies

Source	Supporting Data
[16]	*In many IMA architectures, each processing resource is driven by its own clock. While these clocks may have the same period, they execute asynchronously relative to each other with their own offset, drift, and jitter. This results in an architecture in which synchronous components execute asynchronously relative to each other.*
[3]	*"… the process model community has found that software engineering, software development, system engineering, and other activities are integrated, have dependencies, and cannot be adequately performed and optimized independently of each other …"*
[6]	*A specialized hardware platform was found to work much better when its application was built with a monolithic build. This had implications for other applications on the same platform that had to be reworked in order to be built with that application, and this caused extra development and certification effort.*

4.1.2.8 Allocation to the Target Platform

Problem	Allocation to the Target Platform
Description	Incorrect prediction of required physical resources leads to late changes to system architecture and extensive rework.
	In current processes, development of the target platform and the logical system typically proceed in parallel. As the logical system is mapped down onto the physical platform, tasks are manually allocated and scheduled onto specific virtual machines or processing elements by the system designer. Communication paths between functions are also defined manually. Analytic tools may be used to verify virtual machine and network schedules, but most of the system verification is done through system testing of the integrated system. This makes it extremely difficult to add new tasks unless sufficient processing and communication capacity was reserved for future expansion. If existing schedules or communication paths need to be changed, verification of the entire system may have to be repeated. As a result, many systems are deployed with more than half of their hardware capacity unused. Lack of strong analysis tools also makes it very costly to conduct reliable trade studies to find an optimal system configuration. Typically, a configuration is selected based on engineering judgment or a few paper-and-pencil trace studies.
Approach	Use of system design patterns that enable early prediction of system performance.
	At each level of design, standard patterns are used to systematically collect the relevant processing or communication requirements for that level. Analysis tools are used to predict performance and utilization of system resources for that level. The design is progressively refined into a full implementation, with each refinement taking into account more details about the target platform. Unverified analysis tools may be used to find optimal system configurations that are verified with highly trusted checking tools. Specific design patterns and their implementations will depend on the specific target platform. For example, very different design patterns may be needed for refinement onto a large multi-core platform than would be needed for refinement onto a traditional RTOS-based platform.
Parameters	For each level of refinement: • Specification of Services Provided by Level N of System Architecture • Specification of Services Provided by Level N+1

Documentation – Allocation to the Target Platform

Source	Supporting Data
[1] [12]	*Twp scenarios based on previous generation aircraft systems pointed to a future system [aircraft] system containing approximately 60 million SLOC. That level of software growth is self-limiting, given that the cost of such a complex system is in excess of $10B, likely exceeding a limit of affordability.*
[1]	*…and thus an aircraft consists of systems of systems. There may be on the order of 13 levels in the overall aircraft system, resulting in thousands of system elements at the lowest levels.*
[2]	*Over sixty percent of the failures were traced to incorrect requirements.*
[2]	*Models for predicting reliability of hardware exist but the software intensive system doesn't have the benefit of realistic reliability models.*
[16]	*In many IMA architectures, each processing resource is driven by its own clock. While these clocks may have the same period, they execute asynchronously relative to each other with their own offset, drift, and jitter. This results in an architecture in which synchronous components execute asynchronously relative to each other.*
[3]	*"… the process model community has found that software engineering, software development, system engineering, and other activities are integrated, have dependencies, and cannot be adequately performed and optimized independently of each other …"*

4.1.2.9 Untrusted/ Unreliable Components

Problem	Untrusted/Unreliable Components
Description	Use of complex components that provide highly desirable performance are prohibitively expensive to verify.
	Highly desirable improvements in system performance or capability can often be achieved with sophisticated subsystems that are prohibitively expensive to verify due to their size and complexity.
Approach	Use of formally verified simplex design pattern that switches to a simpler trusted controller before the system's acceptable envelope of behavior is violated.
	The simplex architectural design pattern allows a system to use a high performance but untrusted controller by using run-time monitoring to switch to a low performance, trusted controller before the system's acceptable envelope of behavior is violated. Use of the simplex pattern requires: • An untrusted controller that provides superior performance most of the time. • A trusted controller that provides acceptable performance all of the time. • A monitor function that can determine when the acceptable envelope of system behavior is about to be violated. The simplex pattern provides a formally specified, formally verified pattern for transferring control from the untrusted controller to the trusted controller before the system's acceptable envelope of behavior is violated.
Parameters	• Untrusted Control Function • Trusted Control Function • Monitor Function

Documentation – Untrusted/Unreliable Components

Source	Supporting Data
[2]	*The need to consider failure of software in the system MTBF calculations has arisen, even though software doesn't "fail" in the traditional hardware sense.*
[2]	*The lack of software failure data for highly reliable systems has been a hindrance.*
[2]	*A reliability growth curve for a high use system is needed*
[16]	*The introduction of Integrated Modular Avionics (IMA) [1], [2] has allowed developers to implement increasingly sophisticated avionics systems by taking advantage of the growing processing power and memory available in modern integrated circuits.*
[3]	*"… the process model community has found that software engineering, software development, system engineering, and other activities are integrated, have dependencies, and cannot be adequately performed and optimized independently of each other …"*
[8] [9] [10]	*Insufficient validation of new concepts*
[8] [9] [10]	*Pushing too many technology envelopes during a production program*
[8] [9] [10]	*Poor understanding of architecture caused by ambiguous / incomplete architecture representation*

4.1.2.10 Requirements Errors

Problem	Requirements Errors
Description	Missing or incorrect requirements are not discovered until system integration resulting in extensive rework.
	Missing or incomplete requirements are well known to be one of the most significant cost drivers in the development of cyber-physical systems. Unfortunately, the majority of requirements errors are not detected until system integration when the cost of fixing the error has increased by two orders of magnitude.
Approach	Use executable system models and automated compositional verification to enable early identification of requirements errors.
	By constructing system design models with well defined semantics and formally defined requirements for the components, developers can perform simulations of the overall system. These help developers to identify many cases of over-utilization of processing, communication, or memory resources and to identify instances in which system safety and security requirements are violated. More subtle errors can be found using compositional verification. Compositional verification takes an architectural model with well defined semantics and components with formally defined assume/guarantee requirements and proves that the overall system requirements are satisfied by the model. Almost always, formal verification of this sort uncovers missing assumptions, missing requirements, and incorrect system designs.
Parameters	• System Architectural Model • Formal Assume/Guarantee Specification of System Components • Formal Assume/Guarantee Specification of System Requirements

Documentation – Requirements Errors

Source	Supporting Data
[1]	*The majority of software defects are introduced prior to actually writing code during such a development, specifically in requirements and design, but only a fraction of these defects are detected and addressed early.*
[1]	*Currently, about half of all software defects are not found until hardware/software (physical) integration is completed.*
[1]	*Rework cost is dominated by the cost of removing defects generated during the requirements phase and the design phase but not detected until later in the [system life cycle].*
[1] [12]	*The nominal cost for removing a defect introduced in pre-coding phases but detected in post-coding phases is, for safety-critical systems, often two orders of magnitude higher relative to the cost of removing it prior to code development.*
[1] [12]	*Seventy-nine percent and sixteen percent of the rework cost … is due to defects in the requirements and design phases, respectively.*
[1] [12]	*Rework cost normally is 30%-60% of total development cost, and large systems and quality attribute-intensive systems trend toward the higher values.*
[2]	*Over sixty percent of the failures were traced to incorrect requirements.*
[8]	*The common wisdom is to find and fix requirements errors early in the lifecycle of a project, but that is easier said than done.*
[8]	*The cost to fix a software defect varies according to how far along you are in the cycle, according to authors Roger S. Pressman and Robert B. Grady.*
[8]	*Recall the statistic that said that the US wastes $30 billion annually in rework due to requirements errors. That works out to 300,000 person-years of effort every year ($30 billion /$100,000).*
[15]	*Fred Brooks states the problem succinctly: "The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements...No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later."*
[15]	*While numerous methodologies for REM have been developed over the years, the results of an industry survey, described in reference 1, indicate the best of these practices are rarely being used, if at all, and digital system developers have many questions on how to effectively collect, document, and organize requirements.*

Source	Supporting Data
[15]	*As illustrated in this Handbook, a good set of requirements consists of much more than just a list of shall statements and is not easy to produce. However, investing the time and effort at the start of a project to produce good requirements was shown to ultimately reduce costs while improving the quality of the final product [22-24 and 47].*
[4]	*A principal software engineer with extensive experience on development programs reviewed a draft of the present document and stated that requirements errors are "a biggie." In particular, he noted the difficulty (and importance) of accurately estimating resource requirements such as processor cycles or network bandwidth. "A recurring theme on most every project," he said, "is, 'Hey, I'm out of gas.'"*
[8] [9] [10]	*Not enough focus on Key Program Parameter (KPPs)*
[8] [9] [10]	*Incomplete analysis of impacts to requirements & design changes*
[8] [9] [10]	*Inability to prioritize requirements*

4.1.2.11 Unspecified Dependencies

Problem	Unspecified Dependencies
Description	Missing or incorrect dependencies of components on other components are not discovered until system integration.
	The most common form of requirements error is to fail to specify the dependencies, or assumptions, a component makes about its environment, including other components. Failure to identify and document a components assumption has been the cause of several dramatic system failures. In many cases, the failure occurred because a subsystem developed by one team did not meet the assumptions made by another team. Documenting all assumptions is essential for reuse of a component. Without this information, the system designer has no way to know if a component can be safely reused in a new context.
Approach	Use of formal assume/guarantee contracts and automated compositional verification to enable early identification of unspecified dependencies.
	Requirements for all components and subsystems should be specified using formal assume/guarantee contracts that formally state the assumptions that must be satisfied by the environment and the guarantees provided by the component or subsystem. Formal verification (proof) of component behavior will identify many unstated dependencies since proof of the behavior guaranteed by the component will not be possible without the assumptions. Formal verification of system behavior will also identify missing or incorrect dependencies.
Parameters	• System Architectural Model • Formal Assume/Guarantee Specification of System Components • Formal Assume/Guarantee Specification of System Requirements

Documentation – Unspecified Dependences

Source	Supporting Data
[2]	*Over sixty percent of the failures were traced to incorrect requirements.*
[2]	*The lack of software failure data for highly reliable systems has been a hindrance.*
[2]	*A reliability growth curve for a high use system is needed.*
[13]	*On a typical project, 49% of requirements errors are incorrect facts and 31% are omissions – Table 1-1, page 6.*
[15]	*Identifying the environmental assumptions is also essential for reuse of a component. In several cases, dramatic failures have occurred because an existing system was reused in a different environment and the developers were unaware of the assumptions made by the original developers. Documenting the environmental assumptions is a necessary perquisite for component reuse.*
[16]	*In many IMA architectures, each processing resource is driven by its own clock. While these clocks may have the same period, they execute asynchronously relative to each other with their own offset, drift, and jitter. This results in an architecture in which synchronous components execute asynchronously relative* *to each other.*
[18]	*Airborne software is an increasingly large and important system element. However, it is only one element of the aircraft systems. and only one element of the aviation software environment. Consequently, it must be considered similarly to the other system and software elements, and cannot be considered in isolation.*
[3]	*"… the process model community has found that software engineering, software development, system engineering, and other activities are integrated, have dependencies, and cannot be adequately performed and optimized independently of each other …"*
[8] [9] [10]	*Poor understanding of architecture caused by ambiguous / incomplete architecture representation*

4.1.2.12 Poorly Defined Interfaces

Problem	Poorly Defined Interfaces
Description	Lack of industry standards for interfaces leads to significant system redesign even for small changes.
	When an accepted industry standard does not exist for interfaces, interfaces must be defined and agreed upon by all affected parties. However, the cost and difficulty of defining robust, well-understood interfaces is often under-estimated by project teams. It is usually not sufficient to specify just the values and types to be exchanged. Ranges, units, precision, accuracy, error status, physical format, and constraints on combinations of values should also be defined. Often, protocols specifying acceptable sequences of values across interface (e.g., an acknowledgement will be sent on message receipt). For real-time systems, accepted latencies must be specified. While these issues have often been addressed in the definition of industry standards, they are often overlooked in the development of proprietary interfaces. The problem is exacerbated when components are being developed by separate companies.
Approach	Use of standard design patterns for interface definitions to guide developers.
	Create design patterns for common classes of interfaces with formally verified guarantees of the relationships that will be maintained by the interface.

Parameters	• For each value to be exchanged:
	  o Type, range, precision, accuracy, and unit
	  o Error status of each value
	  o Acceptable latency of each value
	  o Constraints to be maintained across multiple values
	  o Groups of values that must change as an atomic unit
	• Invariants to be maintained across interface
	• Acceptable sequences of interactions across interface

Documentation – Poorly Defined Interfaces

Source	Supporting Data
[15]	*Recommended Practice 2.2.1: Define the system boundary early in the requirements engineering process by identifying a preliminary set of monitored and controlled variables.*
[15]	*Recommended Practice 2.2.7: Completely define all physical interfaces to the system, including definitions for all discrete inputs, all messages, all fields in a message, and all protocols followed.*
[15]	*Recommended Practice 2.4.1: Define the type, range, precision, and units required for all monitored and controlled variables as part of the system's environmental assumptions.*
[15]	*Recommended Practice 2.8.7: Define the acceptable latency for each controlled variable along with the rationale for its value as part of the detailed system requirements.*
[15]	*Recommended Practice 2.8.8: Define the acceptable tolerance for each numerical controlled variable.*
[5]	*For a certain type of device, the control interface is "a giant state machine" written to the ICD (Interface Control Document) of the device. The ICD's that are created require us to design an entire state machine to keep the device from crashing.*
[5]	*A large development program has struggled because the development team did not understand the interfaces of components, developed by other companies, to which their components much connect.*
[3]	*"… the process model community has found that software engineering, software development, system engineering, and other activities are integrated, have dependencies, and cannot be adequately performed and optimized independently of each other …"*
[8] [9] [10]	*Poor understanding of architecture caused by ambiguous / incomplete architecture representation*

4.2 Design Flow and Methodology

This section defines a design flow methodology for the rapid development of cyber-physical systems based on the application of verified design patterns on verified system component models. This design flow models the system at increasing levels of abstraction and embeds verification at all stages through automated support for compositional reasoning about system correctness, formally verified design patterns, and components with guaranteed properties.

An overview of the design flow is shown in Figure 2. The overall development process can be divided into three phases.

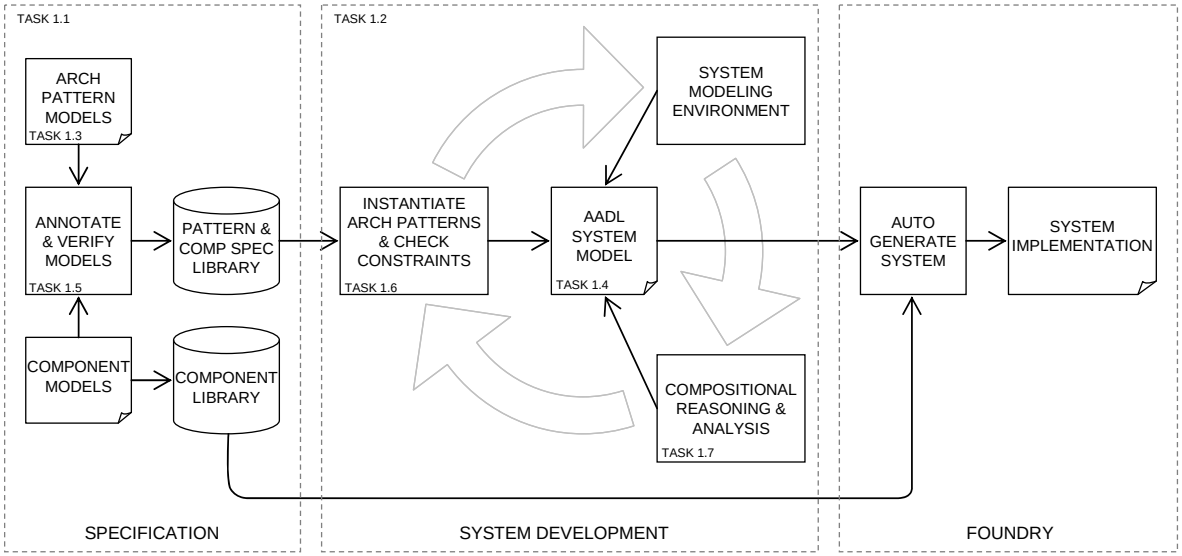


Figure 2 – Design Flow

In the Specification phase, models of architecture patterns and components are developed and verified. Each model is annotated with its verified properties and the constraints that must be satisfied for those properties to hold. In the System Development phase, architecture patterns are instantiated at a given level of abstraction to create a system model. An iterative process further elaborates the design, analyzes its properties, and refines it until sufficient detail for implementation is achieved. At this point, the Foundry phase is entered and the system implementation for a given target is generated from the system model and referenced component models.

This project developed technology for the specification and system development phases. Tools developed to support this design flow include 1) an Eclipse plug-in to translate SysML models to and from AADL, 2) and Eclipse plug-in to check structural properties of an AADL model, 3) a pattern application tool implemented as part of the EDICT tool suite, and 5) an Eclipse plug-in to generate system architectural models for which behavioral properties can be formally verified using the NuSMV [30] or Kind [35] model checkers.

4.2.1 Specification Phase

In the Specification Phase, architectural patterns and components are developed and verified. The Specification Phase for design patterns is shown in more detail in Figure 3.

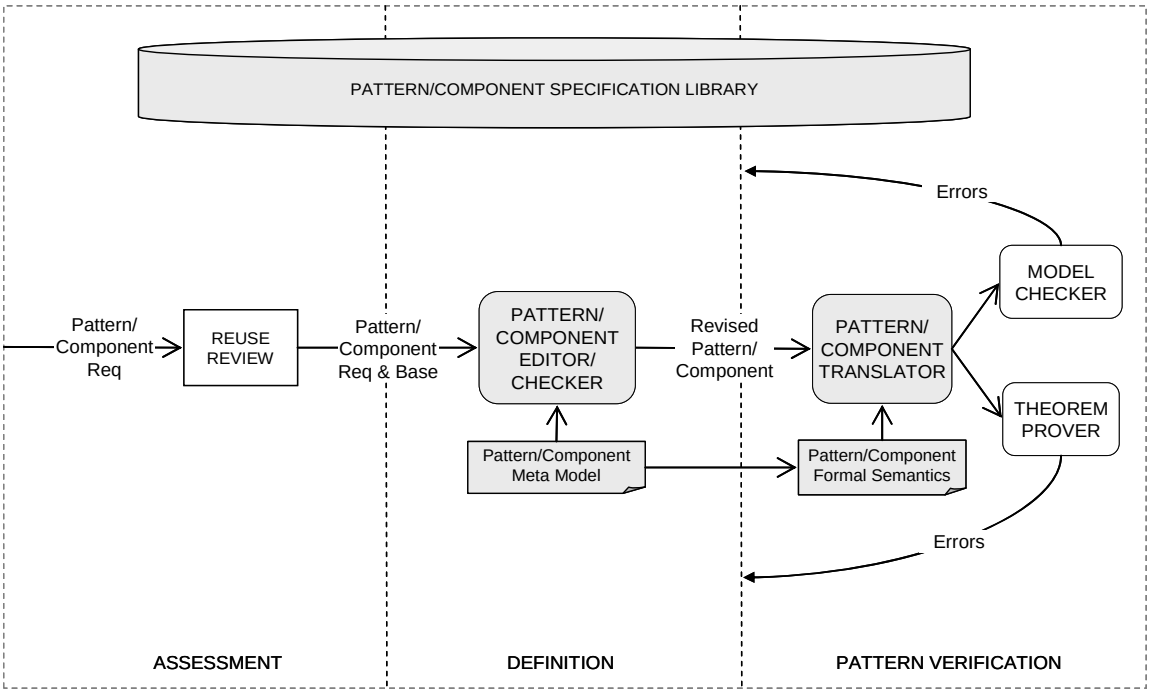


Figure 3 – Specification Phase Design Flow and Tools

A single framework can be used for the specification of design patterns and components since at the architectural level a component is just a design pattern with a single element. The Specification Phase is broken down into three sub-phases: Assessment, Definition, and Pattern Verification.

4.2.1.1 Assessment

The purpose of the Assessment Phase is to determine whether and how a request for a new component or design pattern is to be met. During Assessment, a request for a new component or design pattern is evaluated against the library of existing design patterns to determine if any of the following solutions are feasible:

Direct Reuse – An existing component or design pattern meets the request and can be used without modification.

Reuse through Generalization – An existing component or design pattern would meet the request if it were generalized.

Reuse through Variation – A new variant of an existing component or design pattern can be created to meet the request.

Creation of a New Pattern – A new component or design pattern must be created to meet the pattern request.

Normally, the first feasible option in the list above would be chosen. However, there may be some cases where choosing between generalization and variation may not always be obvious since generalization may increase the coupling between design instances created from a more general pattern.

In a full system development effort, assessment is performed to ensure that new components and patterns are not created when an existing pattern could be used. Assessment also helps to ensure the design pattern library grows in an orderly way. This implies the existence of a change control organization that reviews requests to create or add new components and design patterns to the library. This is a normal consequence of a product family approach where the conflicting goals of the product family and product instances lead to separate organizational structures for the product family and each new product. Due to the small size of the component and design pattern library in this proof-of-concept, issues of change control are dealt with informally.

As the size of the component and design pattern library grows, tools to catalog components and design patterns and to search for relevant matches may become desirable. Much research has been done in this area that may be relevant in full developments. Again, due to the small size of the component and design pattern library in this proof of concept, no tools were developed to assist in the Assessment Phase.

The outputs of the assessment phase are the original component/design pattern request, a recommendation of how to address the request, and possibly a base component or pattern to be generalized or used in creating a new variant.

4.2.1.2 Definition

In the Definition Phase, a specification for the new component or design pattern is created. A component is a single architectural entity that has no internal structure modeled at the architecture level. It may have externally visible features such as input and output ports, as well as a separately specified implementation. Associated with each component are a set of assumptions and guarantees that characterize the behavior of the component. If the component's assumptions are satisfied by the component's environment, then it will provide the behavior specified in its guarantees. Any implementation of a component that provides the specified inputs and outputs and satisfies the component's guarantees when the environment satisfies the components assumptions should be equally acceptable.

A pattern is a transformation that can be applied to an architectural model to obtain a new model. Application of a pattern may transform a model in a variety of ways, such as inserting components into the model, replicating portions of the model, adding information (i.e., properties) to the model, or creating new connections between components in the model. Application of a pattern may also insert assumptions and guarantees about the system's behavior into the model, where the guarantees can be used in proofs about the system's behavior so long as the assumptions can be proven to hold. Often, the assumptions and guarantees associated with a pattern are its most important aspects.

4.2.1.2.1 Component Definition

The information needed to define a component consists of its type (the name used to reference the component definition), parameters, inputs, outputs, and any internal state variables (such as component modes) used in specifying the component's behavior, assumptions, and guarantees.

A component's assumptions define constraints on its inputs and internal state that must be satisfied for the component to behave correctly. A component's guarantees are constraints on the internal state and outputs that the component ensures when all its assumptions are satisfied. The assumptions and guarantees of a component form the contract that must be satisfied by an implementation of the component. They may be viewed as comprising the requirements for that

component. Conversely, a component's assumptions and guarantees also specify the behavior of the component that can be used when reasoning about overall system behavior. These may specify timing and reliability requirements as well as functional behavior.

The specification of a component provides a contract that allows analysis of the system design to be decoupled from the details of component implementation. Any implementation of a component that provides the specified inputs and outputs and satisfies the components guarantees when the environment satisfies the components assumptions should be equally acceptable.

The information that is to be collected and checked to specify a component during the Definition Phase is summarized in Table 2.

Table 2 – Information Needed to Define a Component

Data	Description
Type	A name identifying the component definition
Parameters	Name, value, and type of constants used to configure/parameterize the component
Inputs	Names and types of component input signals
Internal State	Names and types of internal state values used in formal specification (e.g., component modes)
Outputs	Names and types of component output signals
Assumptions	Constraints on inputs and internal state that the component depends on
Guarantees	Constraints on outputs and internal state that the component provides when all assumptions are satisfied

4.2.1.2.2 Pattern Definition

A pattern is a transformation that can be applied to an architectural model to obtain a new model. Application of the pattern can transform the system model in a variety of ways, including:

- Annotating an architectural component or subsystem with information or properties that are used during system analysis or generation.
- Inserting a component or subsystem a single unit into the system architecture. The component or subsystem itself may first need to be instantiated with more fully defined components.
- Weaving a subsystem of components and connections into the system architecture.
- Replicating a portion of the system and its connections. Patterns of this sort are very common in creating fault tolerant systems.

Patterns are themselves composed from the small fundamental model transformations described in Table 3. These can be thought of as the primitive operations that a pattern can make use of. Each fundamental transformation has its own structural pre-conditions that must be satisfied by a model for it to be applied. Each fundamental transformation may also have options that vary how the model is transformed.

Table 3 – Fundamental Model Transformations

Name	Description
Insert Component	Inserts a component from the library into a model.
Remove Component	Removes a component and all connections to/from that component from the model.
Rename Component	Renames a component in a model.
Replicate Component	Replicates a component and its connections in a model.
Insert Data Specification	Inserts a data specification into a model
Create Feature	Creates a new port, bus access, or data access feature on a component in a model.
Replicate Feature	Replicates a feature across several components in a model.
Remove Feature	Removes a feature from a model component.
Create Connection	Creates a connection between two features or between a feature and an accessible component such as memory, data, or bus.
Remove Connection	Removes a connection from a model.
Insert Property Set	Inserts a new property set from the library in a model.
Assign Property	Adds a property to a model component.

The information needed to define a pattern consists of its type (name used to reference the design pattern), its purpose, rationale, parameters, specification of components, assumptions, guarantees, and a specification of how the pattern changes the model. This information is summarized in Table 4.

Note that component specifications can be important inputs to a pattern. The same information is collected for each component participating in a pattern as shown in Table 2. However, the assumptions and guarantees are used for different purposes than when a component is specified in a library. This is because the specification of a component in a library defines the maximal behavior guaranteed by the component, while the specification of a component in a pattern defines the minimal behavior required of the component. Thus, for a component specification in a pattern:

1. The assumptions promised to the component should be the strongest assumptions that the pattern can guarantee based on its assumptions, and
2. The guarantees required from the component should be the weakest guarantees that the pattern needs to ensure the pattern's guarantees hold.

This allows the design pattern to be instantiated with the largest number of component instances.

In a similar fashion, the purpose of the assumptions and guarantees for the pattern itself is to provide a contract that allows analysis of the pattern to be decoupled from the analysis of the system architecture. To this end, the assumptions and guarantees for the pattern are separated into structural constraints on the system architecture and behavioral constraints over the system execution.

The structural assumptions describe constraints over the system architecture model that must be satisfied to apply the pattern. These may specify variables, components, connections, interfaces, or attributes that need to be present in the architecture, or relationships between these entities. The behavioral assumptions describe constraints over the execution of the system that must be satisfied for the application of the pattern to behave correctly in the system architecture. These may specify constraints such as the ranges architecture variables, invariant relationships between system variables, and temporal constraints specifying the dynamics of the system variables.

The structural guarantees describe constraints over the system architecture after the design pattern is applied. These may specify variables, components, connections, interfaces, or attributes that will be present in the architecture. Of particular interest are verification conditions that be easily checked to ensure that subsequent modifications to the architecture have not invalidated the design pattern. The behavioral guarantees describe constraints over the execution of the system that will be enforced by application of the design pattern and that can be used in proving overall properties of the system. These may specify such things as invariant relationships between system variables and temporal constraints over system variables.

Patterns can be composed to form more complex patterns. When applying such a sequence of patterns to a model, the structural pre-conditions for each pattern must be satisfied by the intermediate model produced by the earlier transformations in the sequence.

Table 4 – Information Needed to Define a Design Pattern

Data	Description
Type	Name of the design pattern
Purpose	An informal description of what application of the pattern does.
Rationale	An informal explanation of how the pattern is to be used and why it is useful.
Parameters	Name, value, and type of constants used to configure/parameterize the design pattern
Components	Name and specification of each component to be participating in the design pattern, including all information identified in Table 2
Assumptions	Constraints that must be satisfied by the system architecture for correct use of the pattern. These may include: Structural constraints necessary to instantiate the pattern such as: • architectural variables or interfaces that the pattern references • architectural attributes such as security or criticality level Behavioral constraints necessary for correct pattern behavior such as: • range constraints on architecture variables or interfaces • invariant relationships between system variables • temporal constraints over system variables
Guarantees	Constraints that will be ensured by the system architecture after the pattern has been applied providing the behavioral assumptions are satisfied. These may include: Structural constraints over the system architecture such as: • components, connections, variables, and interfaces that will exist in the system architecture • verification conditions that should not be invalidated by modifications to the system architecture Behavioral constraints guaranteed by the pattern (providing all its assumptions are satisfied) such as: • invariant relationships between system variables • temporal constraints over system variables
Specification	Description of how the pattern modifies the architectural model. This may consist of a collection of the fundamental model transformations described earlier.

4.2.1.3 Pattern Verification

Formal methods and testing are used to reason about system architectures throughout the design flow. Formal methods allow one to *prove* certain properties of systems and software using mathematical techniques such as theorem proving and model checking. These techniques are critical for establishing that patterns work correctly in the context of any system architecture where they are intended to be used, and that system architectures meet their overall system requirements.

Depending on the stage of development, there are different verification activities related to patterns that must be performed. These fall broadly into three categories:

- Assuring that a design pattern is correct.
- Assuring that the software architecture establishes the assumptions required to use a design pattern.
- Assuring that a system constructed from design patterns establishes global behavioral guarantees.

The first is a *pattern verification* activity, while the last two are *system verification* activities that will be described in Section 4.2.2.3.

Verification of a pattern consists of showing that the pattern always establishes its *guarantees* if its *assumptions* about the system architecture are satisfied. The guarantees codify properties established by the pattern such as structure of the resulting architecture, logical synchrony, data synchronization, or fault tolerance. These assumptions define restrictions placed on the system architecture for the pattern to work correctly. For example, in PALS, we require certain timing assumptions on thread and communication rates. The pattern guarantees are defined and proven at the time that the pattern is specified and are independent of a specific architecture. That is, a pattern should be "pre-supplied" with a proof that it establishes the guarantee of interest, given certain constraints on the architecture for *any valid* instantiation of the pattern.

As indicated in Figure 3, errors found in the specification of the component or design pattern are fed back into the Definition Phase for correction.

4.2.1.4 Tool Support for Pattern Definition and Verification

Patterns are implemented as an extension of the EDICT tool (Section 4.2.2.4) as needed to support the META project. The EDICT tool checks before pattern application that the structural assumptions (pre-conditions) of the pattern are satisfied by the system architecture and that the structural guarantees of the pattern are satisfied by the system architecture after pattern application. The EDICT tool also inserts the assumptions and guarantees of the pattern in the system architectural model when the pattern is applied.

Each pattern is formally verified during its development to prove that its behavioral guarantees hold if its behavioral assumptions are satisfied and the structural assumptions and guarantees hold. This verification is valid for any model in which the structural assumptions and guarantees hold, providing the link between the verification of the pattern specification and its instantiation in a specific system model.

Architectural design patterns can vary greatly as to the information required to specify the model transformation performed and the nature of the behaviors they enforce. Therefore, pattern

verification must be done on an individual basis using different tools and techniques. For example, we have used BDD-based model checking tools (NuSMV [30]), SMT-based model checkers (primarily Kind [23] and SAL [34]) and theorem provers (primarily PVS [33] and ACL2 [25]). SMT-based tools have the advantage of more automation for systems and architectures that involve decidable theories. Theorem provers, while requiring more manual effort, can reason about certain classes of systems, such as non-linear systems, that are not analyzable using SMT-based tools. In either case, an important aspect of using patterns in system development is the ability to amortize the verification effort over the many systems in which the pattern will eventually be deployed.

4.2.2 System Development Phase

The System Development Phase is the principal phase in which the system design is accomplished. Starting at a relatively high level of abstraction, a system architecture is created or instantiated from the library to create an initial system model. This model is then refined by repeated applications of design patterns and additional manual modeling to create a detailed system model from which an implementation can be generated. This is illustrated for one iteration in Figure 4.

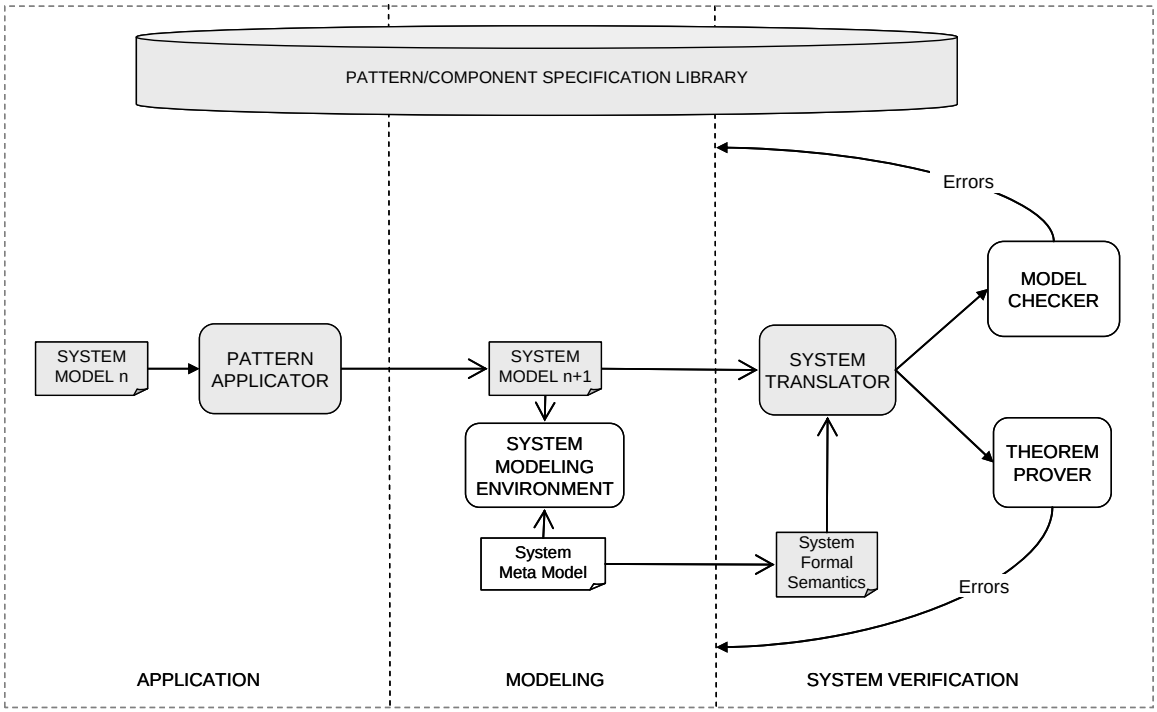


Figure 4 – System Development Phase Design Flow and Tools

The system model after "n" pattern applications is the initial input in Figure 4. During the Application phase, a new pattern is chosen from the pattern/component library and applied to create system model n+1. In the Modeling phase, this model is loaded into the architectural modeling environment and checked using the performance, security, and safety tools available in that environment. At the appropriate levels of refinement, requirements for the system model are formalized and mechanically verified in the System Verification phase.

4.2.2.1 Application

In the Application phase, the system developer selects a pattern to be applied to the existing system model. This pattern is then applied to system model n using a Pattern Application tool to generate system model n+1. Additional information, such as parameters or component specifications to be used by the pattern, may also need to be collected. Prior to application of the pattern, all pattern structural assumptions are checked to make sure the pattern can be applied to the system model. After application, the guarantees provided by the pattern are added to the list of known system properties for use during the Modeling and Verification phases.

As the size of the component and design pattern library grows, it may be helpful to develop tools to search for patterns based on different search criteria. This is similar to the issues discussed in the Assessment sub-phase of the Specification Phase. More importantly, guidelines may be needed to help developers understand how to apply patterns to obtain optimal architectures. For example, the order in which patterns are applied will certainly matter, i.e., pattern application will not normally be commutative. Due to the small size of the component and design pattern library in this proof of concept, no tool support for searching and selecting a pattern is planned.

The output of the Application phase is a revised model that is closer to being implementable.

4.2.2.2 Modeling

In the Modeling phase, the new model generated by application of a pattern is loaded into the architectural modeling environment. This is driven by the System Meta-Model (see Figure 4) that defines syntactically valid system models. If necessary, the model can be modified using the modeling environment to add capabilities not available in the pattern/component library, to change the layout to improve readability, or to add explanatory comments. At this time, all of the standard performance, security, and safety checks provided by the modeling environment can be used to check the model. Simulation capabilities of the modeling environment may also be used to check model behavior.

4.2.2.3 System Verification

In the System Verification Phase, formal verification tools are used to confirm important properties of the emerging system. This may include verification of functional behavior, performance, safety, or security properties of the system. As mentioned in 4.2.1.3 on pattern verification, there are two parts to system verification:

- Assuring that the software architecture satisfies the structural assumptions required to use a design pattern.
- Assuring that a system constructed from design patterns provides the system behavioral guarantees.

While the structural assumptions for applying a pattern can certainly be checked before the pattern is applied to a system architecture, the pattern guarantees may not hold if the if the system architecture is further modified by developers after pattern application. For this reason, the structural assumptions and guarantees for the pattern are attached to the system model so that they can be verified periodically. The behavioral guarantees and assumptions are also embedded in the architecture for similar reasons (Section 4.2.2.7).

The determination of architectural suitability is complicated by the fact that patterns can be built on top of (or in terms of) other patterns. For example, a leader-election protocol may have a

requirement that the nodes involved in the pattern communicate and execute synchronously. To establish logical synchrony, a PALS pattern is used, and this pattern requires certain timing assumptions on the threads and communications channels involved in the pattern.

Once the architecture has been shown to satisfy the assumptions for use of each pattern, global behavioral properties of the overall system can be verified. Some system properties will be immediately established from guarantees provided by the patterns, such as fault tolerance or synchronization, but others will require additional work. For example, to verify that redundancy management logic correctly diagnoses failures would require functional properties to be specified and separately proved for the processes involved in the redundancy management as well as guarantees established over the system architecture to ensure correct communication between the processes. The communication guarantees can be established by the system architecture, but the failure diagnosis depends on the functional behavior of the involved components.

The precise step of refinement in which a property is verified depends on the property and the sequence of refinements. Ideally, properties would be verified as early as possible since verification will probably require more time on a detailed model produced after several refinements than on its more abstract ancestor. On the other hand, it makes no sense to verify a property that has not emerged at the current level of refinement (e.g., to prove fault tolerance before redundancy has been introduced). Other properties may be easier to prove once certain refinements have been made (for example, after logical synchrony has been introduced into the model). A goal of this project is to start developing guidelines for what properties of architectural models can be formally verified and when they should be verified.

Of course, one of the primary goals of this approach is to reuse analyses previously performed on the design patterns. This supports a layered approach to reasoning that allows us to take advantage of proof results established for architectural patterns to allow system-level reasoning about functional behavior. We first check that the assumptions of the design pattern hold. Then we can use the guarantees established by the patterns to simplify reasoning about the system behavior.

4.2.2.3.1 Describing Time

One of the major issues for architectural verification has to do with the modeling of time. Many approaches for modeling and reasoning about dense real time have been proposed, including Timed Automata [20][27], Timed CSP [32], hybrid automata [24] and several other notations (e.g., [36][29]). In these notations, we can model time as a true real-valued variable and ascribe time ranges to computations and other activities within the system. The benefit of modeling real time directly is that it is the most natural and accurate model of system behavior. Unfortunately, reasoning about real time is computationally expensive, and proofs that make essential use of real-time are difficult to automate.

Alternately, we can leave time as an abstract quantity and talk instead about approximations of real time. One approximation is to approximate the real-time values with integers. Another is to describe the relative rates (in terms of a predicate) between threads. In order to make the approximations sound with respect to universal temporal properties, the approximate behaviors must simulate the original behaviors, i.e., the abstracted system must have at least as many behaviors (in terms of the relative invocations of processes) as the original system. To do this, we introduce non-determinism. For example, one safe approximation is to assume that each

thread can run or not run arbitrarily. This will model any possible execution sequence of threads from the actual architecture. Approximate approaches are simpler to automatically analyze, but introduce a form of *incompleteness* in that some properties that are true of the actual architecture cannot be verified on the simplified approximate model. However, as long as the approximate model satisfies all required system behaviors, the actual architecture will also satisfy all required behaviors.

Any representation of time in architectural models that makes the analysis tractable introduces complexity and possibly incompleteness into the analysis. When possible, we reason using an approach where time has been factored out of interesting portions of the system architecture. One way to do this is to assume that threads within the system communicate *synchronously.* That is, for the purposes of analysis, it is assumed that communications between threads is instantaneous and that all threads share a global clock. One of the benefits of the proposed design flow is that some of the patterns, such as PALS, allow one to view higher-level components of the system as interacting in a *logically synchronous* way. Synchronous systems are much simpler to analyze in an automated way than either real-time based or abstract-time-based approaches, because time is simply factored out: the components execute in logical lock-step, drastically reducing the number of possible system configurations.

The tools that we are using allow time to be encoded as an integer or real-valued variable to allow precise time reasoning, or to be abstracted out of the model if we can assume logical synchrony (such as is provided by a time-triggered architecture [26] or the PALS pattern [31], [28]). For properties involving real-time, only *safety properties* [21] are considered, i.e., "the system is always in a good state", or equivalently, "some bad thing never happens". Restricting the properties to safety properties allows real-time properties to be specified in terms of state invariants and avoids the need for real-time temporal logics.

4.2.2.4 Tool Support for Pattern Application

Tool support for the application of patterns is provided by the Pattern Transform Editor within the EDICT tool suite. The Pattern Transform Editor has three tabs as shown at the bottom of Figure 5: 1) the *Transform* tab provides overall support for the application of patterns to a system model, 2) the *Instantiation* tab allows parameters of a specific pattern to be supplied and 3) the *Save Copy* tab allows the user to save the transformed architecture model.

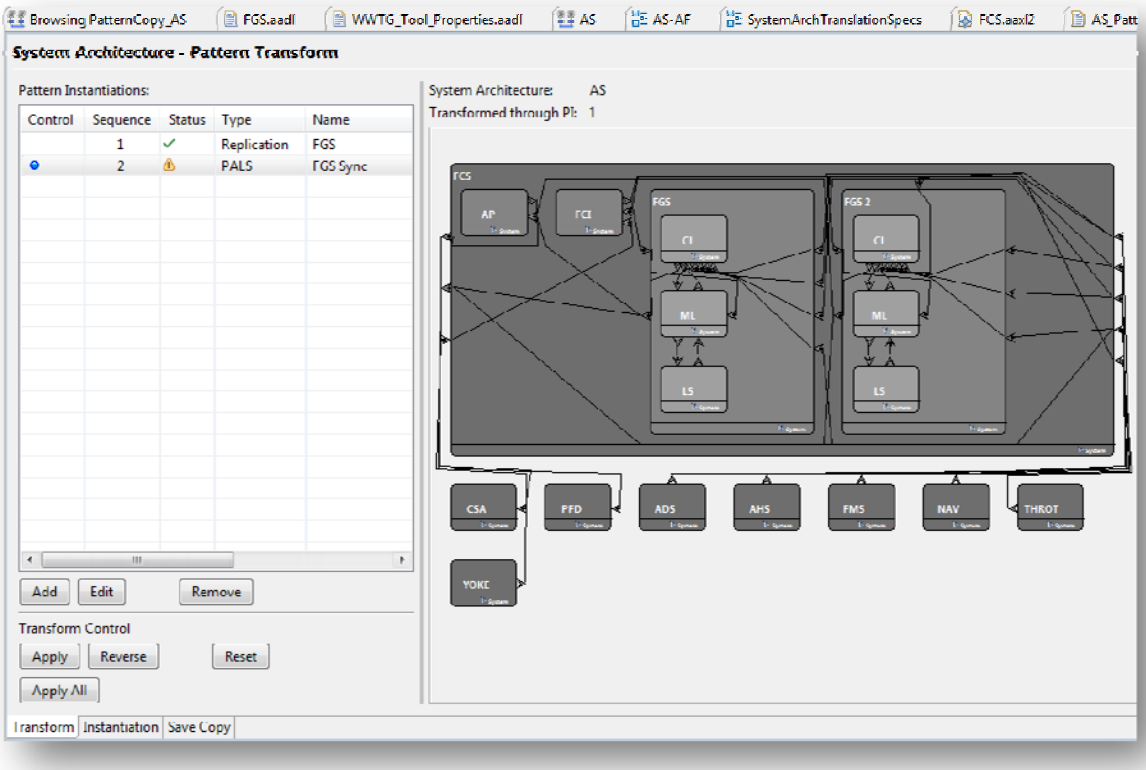


Figure 5 – Pattern Transform Editor: Transform Tab

Figure 5 depicts the editor when the *Transform* tab is selected. On the left hand side is an ordered list of patterns that are to be applied to the architectural model. There are buttons to *Add*, *Edit* and *Remove* patterns from the list. There are also *Transform Control* buttons: *Apply* to apply the next pattern in the list, *Reverse* to undo the last pattern applied, *Reset* to go back to the beginning with no patterns applied, and *Apply All* to apply all the unapplied patterns in the list. The right hand side of the tab displays a visualization of the architecture model that is dynamically updated after each pattern is applied. The user can easily view structural changes to the model at any point.

4.2.2.4.1 Configuring a Pattern for Application

Each of the pattern instances listed on the *Transform* tab requires pattern specific data before the model transformations defined in the pattern can be applied. When a pattern is selected from the list and the *Edit* button is selected, the *Instantiation* tab is displayed. The *Instantiation* tab contains controls specific to the selected pattern with which the user supplies the required parameters. Figure 6 shows the *Instantiation* tab for the Replication pattern.

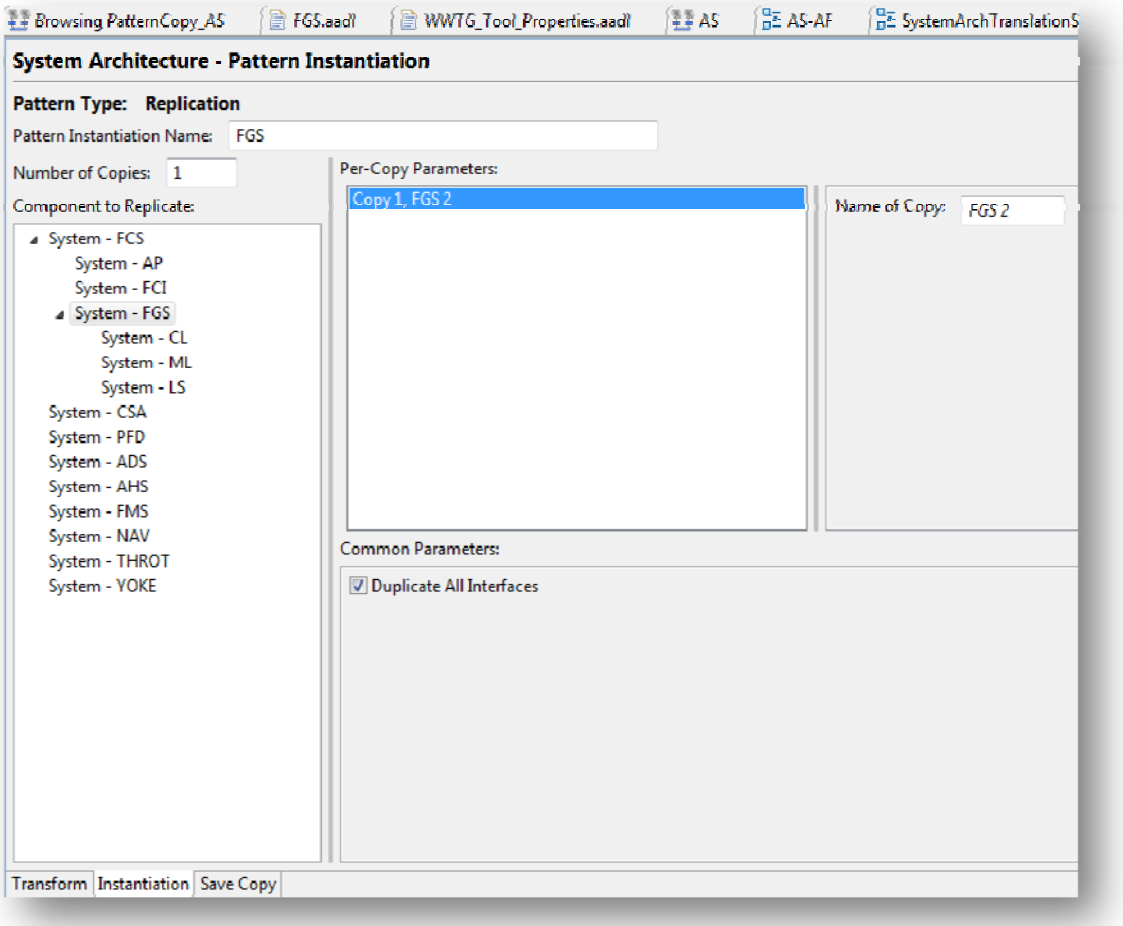


Figure 6 – Pattern Transform Editor: Instantiation Tab

For most patterns there are several types of parameters that must be supplied. For the Replication pattern, there are general parameters including the name of the pattern instance (to distinguish it from other instances of the Replication pattern), the architectural components to be replicated, and the number of copies to be created. There are also parameters applied on a per component basis (i.e., to a specific component in the transformed model) such as the name of the new components, and common parameters that are applied to all components such as whether the interfaces for each copy should be duplicated. Once all parameters are set the user can save the configuration and return to the *Transform* tab to apply the pattern.

4.2.2.4.2 Verification of Pattern Pre-Conditions

When a pattern is instantiated on a system model, structural pre-conditions for the application of the pattern are checked by a model verifier. The verifier ensures that the structural requirements for instantiation of the pattern are met before the pattern is applied to the system model. These include checks for the existence of required architectural components, constraints on component types, constraints on the connectivity of components, or constraints on component properties. These pre-conditions specify the architectural constraints that must be in place for the pattern to construct a correct new model.

The verifiers that are provided for each pattern are integrated with the Graphical User Interface (GUI) of the Pattern Transform Editor. The verifier is used to control the information displayed in the GUI and to provide live feedback to the user on the valid choices for instantiation. The GUI does this by highlighting missing or incorrect values and by limiting the choices of the user to those consistent with the pre-conditions. Figure 7 shows an example of the verifier working with the GUI during application of the Physically Asynchronous Logically Synchronous (PALS) pattern

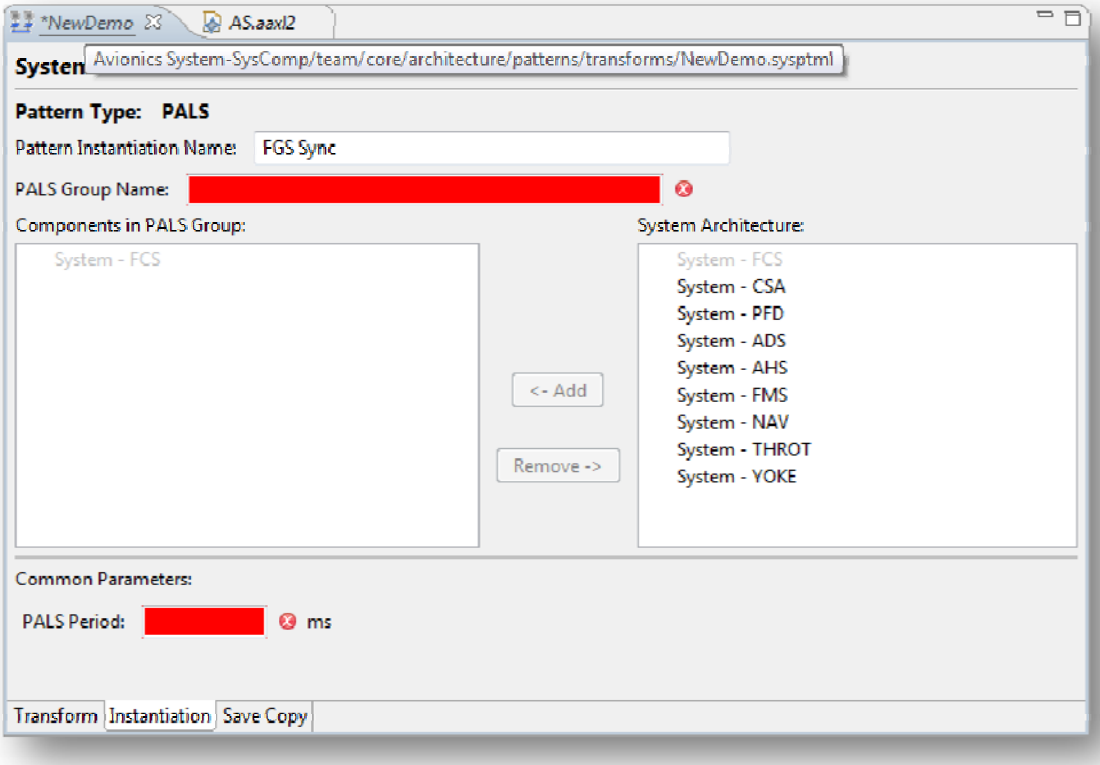


Figure 7 – Pattern Verifier for Instantiation

In this example the user has failed to provide and PALS Group Name that identifies the members of the PALS group and the period of the PALS synchronization. These fields are highlighted in red and will prevent instantiation of the pattern until valid entries are supplied. The PALS pattern has no architectural pre-conditions, so all of the model components can be selected for application of the PALS pattern. In Figure 7 the FCS subsystem has been selected already and is therefore grayed out. The verifiers are re-run each time the pattern instance is opened or saved so that any errors will be caught before the pattern is applied to the architecture.

4.2.2.4.3 Verification of Pattern Post-Conditions

The Pattern Transform Editor also verifies structural architecture properties after patterns have been applied to the system model. The application of patterns to an architecture results in both structural changes (additional components or interfaces) and behavioral changes (new or modified behavioral assumptions and guarantees). The properties associated with a pattern may be invalidated if subsequent changes to the system model are made, either due to additional pattern applications or changes made by system developers manually.

In a design flow in which changes to the system model can be made iteratively, either through additional pattern applications or through manual editing, it is necessary to verify the pattern assumptions and guarantees continuously as a part of the overall system verification process. Verifiers for structural post conditions can be built in a similar fashion to those described earlier for pre-condition verification. These post condition verifiers will use the information from the pattern instantiation to identify the architecture specific post conditions that should be present for each pattern that has been applied to the architecture. These verifiers can be run on the architecture to verify that the expected post conditions for each pattern have not been violated or corrupted by the application of subsequent patterns or by manual editing.

The results of the post-condition verifiers will be posted in a common view for easy access by the user. This view provides customized icons, highlighting and categorization to aid the user in identification and prioritization of problems. Figure 8 is an example of a model verification view from other architecture analysis features that currently reside in the EDICT tools.

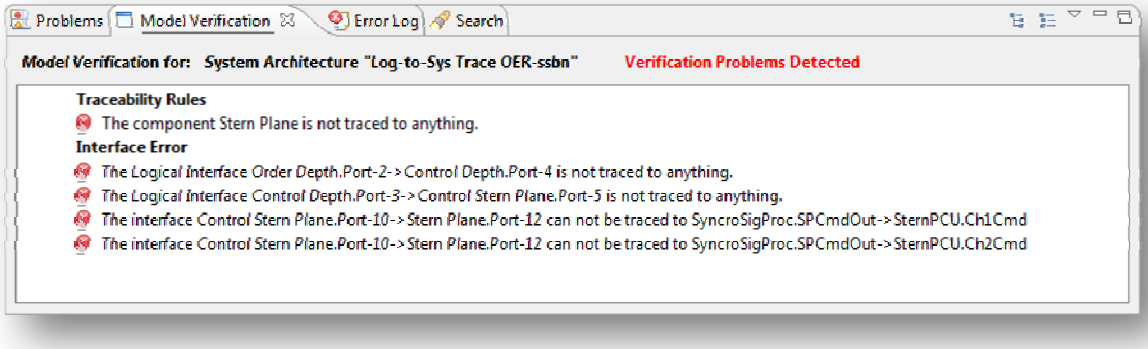


Figure 8 – Model Verification View

The type of checks that are provided by post-condition verifiers need not be limited to the checking of architectural structure. The EDICT tool suite performs many kinds of analyses on system models such as dependability, safety, performance and security. These more advanced forms of verification can also be performed on the architecture model after patterns have been applied to ensure that aspect specific properties, such as schedulability, are met after the application of patterns. The EDICT tools provide an analyzer management framework that can bring together the results of analyzers of many different aspects into a common view. The System Analysis Concerns View provides a single place where the user can go to check on the status of all analyses that are performed. Analysis results are ordered by system aspect and a high level status is provided for each analyzer.

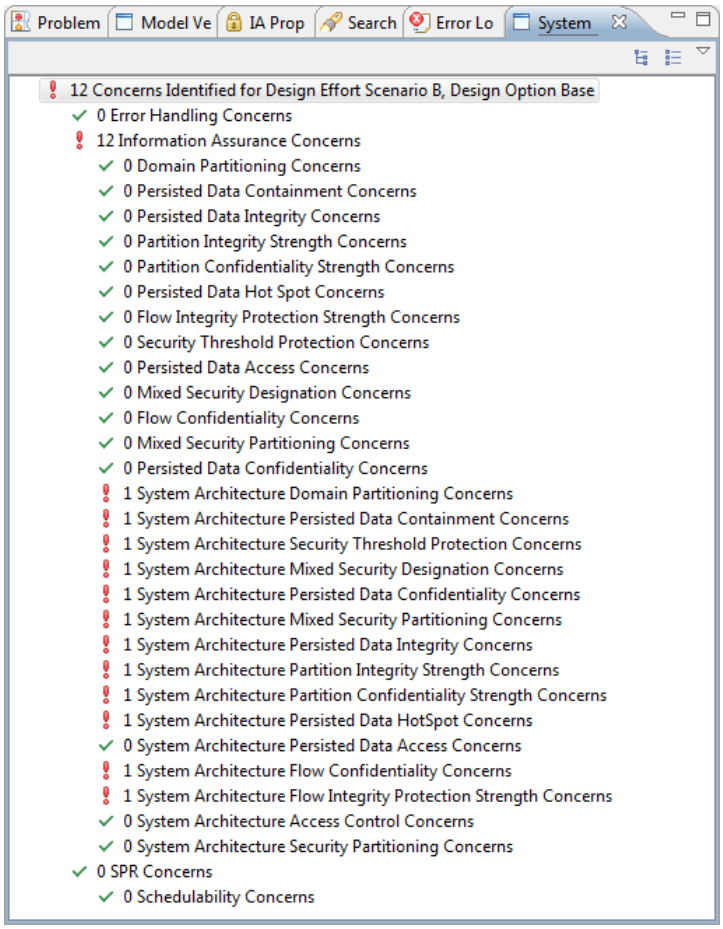


Figure 9 – System Analysis Concerns View

Figure 9 is an example of the System Analysis Concerns View showing information from analyzers for error handling, information assurance and schedulability. Each of the analyzers can be selected to review more details. The user can double click on a concern and the analyzer will be run so that analysis details can be viewed directly.

4.2.2.5 Tool Support for Importing/Exporting SysML

To facilitate the use of SysML as well as AADL for creating system models to which patterns can be applied, a translator allows system architectural models specified in a subset of SysML with the Sparx Enterprise Architect tool to be translated to and from AADL. SysML is an open standard published by the Object Management Group (OMG) for the specification of system architectures.

The translator is written in the Java programming language and is packaged as a plug-in for the Eclipse development environment, making it available to both the EDICT and OSATE tools. Once installed, new menu items are provided in OSATE and EDICT to import a SysML model as an AADL model and to export an AADL model as a SysML model.

The AADL constructs supported by the translator are shown in Table 5. Constructs marked with an asterisk are not currently supported.

Table 5 – AADL Constructs Supported by the SysML Translator

Category	Features	Parts	Parent	Notes
System	Port Portgroup* Subprogram* Provides Data Acc Requires Data Acc Provides Bus Acc Requires Bus Acc	System Data Process Processor Memory Bus Device	System Null	
Data	Subprogram* Provides Data Acc	Data	System Process Thread Data	Direct Access Connections Allowed
Process	Port Subprogram* Provides Data Acc Requires Data Acc	Thread Data Threadgroup	System	
Thread	Port Subprogram* Provides Data Acc Requires Data Acc	Data	Process	
Threadgroup*	Port Subprogram* Provides Data Acc Requires Data Acc	Thread Data Threadgroup*	Process	
Subprogram*	Port Portgroup* Requires Data Acc Parameter			
Processor	Port Portgroup* Subprogram* Requires Bus Acc	Memory	System	
Memory	Requires Bus Acc	Memory	System Processor Memory	
Bus	Requires Bus Acc		System	Direct Access Connection Allowed
Device	Port Portgroup* Subprogram* Requires Bus Acc		System	

* Not currently supported

To ensure correct translation to AADL, SysML blocks and ports are stereotyped with the desired AADL construct. If a SysML block is not stereotyped, it is translated to an AADL system. If a SysML port is not stereotyped, it is translated to an AADL port.

Graphical layout of a model in a SysML diagram is stored in a layout file when it is imported into AADL. When the AADL model is exported back to SysML, this layout information is used to reconstruct the layout of the SysML diagrams.

4.2.2.6 Tool Support for Structural Verification

Complex structural assumptions and guarantees are verified using the Lute checker developed as part of this project. While Lute is similar to the to the REAL verification system [22], it provides several enhancements needed for the META project for specifying and checking complex structural properties.

A Lute specification is made up of Lute theorems, which are computational checks over the structure of the model. A typical Lute theorem iterates over a select group of components and aggregates information about each before checking a Boolean condition. For example, a Lute theorem may iterate over each process and verify that the maximum deadline for all threads in the process is less than or equal to the process deadline. The Lute code for this theorem is shown in Figure 10.

```
theorem Process_Deadline_Greater_Or_Equal_Thread_Deadline
  foreach p in Process_Set do
    Thread_Deadlines := {Property(t, "Deadline") for t in Thread_Set | Owner(t) = p};
    check Max(Thread_Deadlines) <= Property(p, "Deadline");
end;
```

Figure 10 – Lute Theorem for Verifying Process Deadlines

The Lute checker will take the Lute specification and execute it over an AADL system instance. In the case of success it will report back the number of conditions which were checked. In the case of failure it will report the AADL object for which the theorem failed. Additional information can be attached to the Lute theorem in order to provide more detailed error messages in the case of failure.

Since Lute theorems are purely computational, they can be executed without user interaction. Thus it is feasible to re-verify the Lute specification every time a structural change is made to the model. This enables instant feedback during model development.

4.2.2.7 Tool Support for Behavioral Verification

An important goal of this project is to use guarantees provided by system components and architectural patterns to prove behavioral properties of complex system architectures. We used the Property Specification Language (PSL) [19] to define these behavioral properties. PSL is a temporal logic-based language with additional support for regular expressions and clocks, to define system properties. As an example, the following property defines leader agreement across a group of devices (numbered 0…MAX_ELEM). It states that all non-failed devices agree on the leader of the group.

```
PSLSPEC
   forall i in {0:MAX_ELEM} : forall j in {0:MAX_ELEM} :
      G((status[i].device_ok & status[j].device_ok) ->
      status[i].leader = status[j].leader);
```

We developed an Eclipse plug-in that translates AADL models and a subset of the PSL notation into the input languages of the NuSMV and KIND model checkers to support system verification. The NuSMV model checker can be used for the analysis of synchronous models (or of asynchronous systems that implement logical synchrony using PALS) specified using discrete (integer) time. The Kind model checker provides support for k-induction and can be used for analysis of asynchronous system specified using continuous time providing they do not involve non-linear arithmetic or unbounded (tree-like) data. If non-linear arithmetic is necessary to express properties, more sophisticated technology such as a theorem prover would be required.

4.2.3 Foundry Phase

One of the goals of all of the preceding activities is to enable a "no surprises" build of the system during the Foundry phase. The Foundry phase draws upon component models and the fully elaborated system model to auto-generate an implementation. This may be a prototype system to support evaluation in a lab or other test environment, or the final system for delivery and deployment. The system model provides sufficient level of detail to generate component interface code ("glue code"), along with network or system calls to services provided by the underlying platform. Component models must provide sufficient level of detail to be linked in with the interface code. Component models corresponding to electromechanical systems must specify an application interface for the rest of the system.

While the work performed in the project directly supports the generation of "no surprise" implementations, development of tools to support the Foundry phase are outside the project's scope.

4.3 Design Pattern Models

This section describes the design patterns that have been developed in the Complexity-Reducing Design Patterns for Cyber-Physical Systems project. Patterns were developed for PALS, Replication, Leader Selection, and Fusion (voting or source selection). For each pattern, a description of the pattern's behavior is provided, as well as how a developer would use the pattern in a system design. Also provided for each pattern are:

1. Arguments. The arguments for the pattern are given in the form of parameters that must be specified by the user.
2. Assumptions. This includes the system context (any requirements on the initial system model) for the pattern, and any behavioral constraints on the environment that must hold for the pattern guarantees to be valid.
3. Guarantees. These are the properties that are enforced by the pattern and which have been formally verified to hold given when the pattern assumptions are true.
4. Instantiation. A textual description of the algorithm for applying a pattern to a system model.
5. Exemplar AADL models. These are simple before/after models (graphical and textual) that show the effects of pattern application and help to illustrate the pattern design.

4.3.1 PALS

The purpose of the PALS pattern is to make portions of a distributed asynchronous system operate in virtual synchrony. This allows portions of the system logic to be designed and verified as though they will be executed on a synchronous platform, and then deployed in the asynchronous system with the same guaranteed behavior.

To use the pattern, a group of nodes (systems) is selected that are to execute at approximately the same time at period T. The outputs (ports) of these nodes are to be received by other nodes in the group such that all nodes will receive the same values at each execution step. The pattern does not add any new data connections to the model, but assumes that the required connections already exist.

4.3.1.1 Arguments

1. Set of system blocks to synchronize
2. PALS synchronization period T
3. Name of PALS group

4.3.1.2 Assumptions

The PALS assumptions are conditions that the system design model must satisfy, either when the pattern is instantiated or possibly at a later stage of system design.

1. *Bounded Local Clock Error* - Each node i has access to an approximation of the true global time t via a local clock c_j, where the maximum error (called either jitter or skew) of each local clock is ε, i.e.,

   $| C_i - t | < \varepsilon$.

2. *Monotonic Local Clocks* - The value of each local clock C_i is monotonically increasing. Each node may adjust its local clock rate, but it may never decrease the value of its local clock.
3. *Bounded Computation Time* – The computation of a node's new local state and outputs completes within a specified time. Typically this is the periodic scheduling deadline for a thread α_{max}.
4. *Bounded Message Delivery* – Messages are reliably delivered to their destinations with latency μ, where $\mu_{min} \leq \mu \leq \mu_{max}$. Depending on the system fault assumptions, this may require thus use of a fault-tolerant network.
5. *Node fault assumptions* – A failed node must not be able to send extra messages (more than one) during a PALS period. This could result in nodes receiving different messages, even though each was delivered correctly by the network.

In addition, there are some constraints relating system parameters that must be satisfied. These should appear in the model as "assumed" properties that must be verified. Two important constraints expressed in terms of AADL properties are:

1. *Causality constraint* – Messages cannot be sent too early.

   $\text{Min(Output time)} \geq 2\varepsilon - \mu_{min}$

2. *PALS period constraint* – Messages cannot be sent too late.

   $\text{Max(Output time)} \leq T - \mu_{max} - 2\varepsilon$

These two constraints can be described and verified using the Lute structural property checker that we have developed.

4.3.1.3 Guarantees

1. The synchronization logic for each node executes with period T at approximately the same global time as every other node.
2. Messages generated during synchronization step i are consumed by their destination nodes in synchronization step i+1. For two nodes A and B where A sends data to B, this may be expressed (depending on analysis tool syntax) as

   B.in(i) = A.out(i-1), or next(B.in) = A.out.

   In our model, this is will be expressed by a new AADL property called Synchronous_Communication which, when assigned the value "one_step_delay" will be used by system verification tools to generate an appropriate verification model. It may also be used to satisfy an assumption for Synchronous_Communication associated with another pattern (e.g., Leader Selection).
3. All nodes that have common external inputs start each synchronization step with identical values for those inputs. This feature is not implemented in the current version of the pattern, but can be enforced by requiring that each external input be processed by a single PALS node.

4.3.1.4 Instantiation

Add PALS properties to the specified system blocks and output ports. These properties will be used to enforce the thread and message scheduling constraints in the system implementation. In

addition, the PALS Group property will cause PALS middleware subprograms to be inserted in the implementation.

4.3.1.5 Exemplar AADL models

Figure 11 shows a graphical AADL model illustrating the PALS pattern for two system nodes. Note that there is no requirement for the nodes to be completely connected. That is, it is not necessary for NodeB.output to be part of the PALS group in this example.

The textual AADL model follows the figure. Items to be changed or added during pattern instantiation are highlighted.

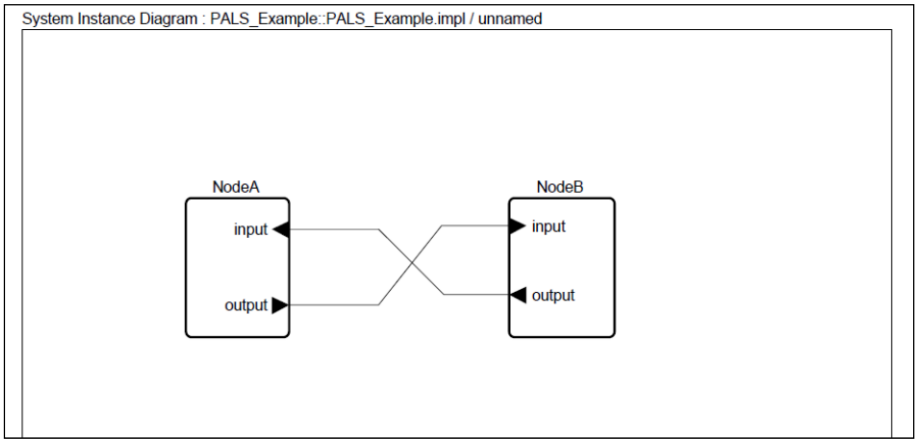


Figure 11 – AADL graphical model for PALS pattern

```
system PALS_Example
end PALS_Example;

system Node
  features
    input: in data port;
    output: out data port;
end Node;

system implementation PALS_Example.impl
  subcomponents
    NodeA: system Node;
    NodeB: system Node;
  connections
    DataConnection1: data port NodeA.output -> NodeB.input;
    DataConnection2: data port NodeB.output -> NodeA.input;
  properties
    --
    -- REQUIREMENTS
    --
    -- PALS Period must be the same for all members of a PALS group.
    -- The actual period of members of the group must be equal to this
    -- value.
    PALS_Properties::PALS_Group_Id => "Group1" applies to NodeA;
    PALS_Properties::PALS_Group_Id => "Group1" applies to NodeB;
    PALS_Properties::PALS_Period => 10 Ms applies to NodeA;
    PALS_Properties::PALS_Period => 10 Ms applies to NodeB;
    PALS_Properties::PALS_Lute_Property => "theorems.lute"
```

```
    PALS_Properties::PALS_PSL_Property =>
       "NodeA.input[i] = NodeB.output[i-1]  and
        NodeB.input[i] = NodeA.output[i-1]";
end PALS_Example.impl;
```

4.3.2 Replication

The purpose of the Replication pattern is to create identical copies of portions of the system. This is typically used to implement fault tolerance by assigning the copies to execute on separate hardware platforms with independent failure modes.

To use the pattern, one or more nodes (systems) are selected and the number of copies to create is specified. Optional arguments for each input and output port on the selected systems determine how these ports and their connections are handled in the replication process. Each new system and port created is given a unique name. When multiple outputs are created they may be merged by the addition of a new system block to select, average, or vote the outputs. This can be implemented using the Fusion pattern (section 4.3.4).

4.3.2.1 Arguments

1. Set of system components to replicate
2. N, the desired number of copies, including the original component(s)
3. For each new output port in the set of replicas, choose:
   a. replicate the destination of the data connection (default)
   b. add a new system block (see Fusion pattern) to merge the new outputs and connect to the destination of the original data connection
4. For each new input port in the set of replicas, choose:
   a. replicate source of the data connection (default)
   b. fan out the data connection from the source of the original input data connection

4.3.2.2 Assumptions

Replicated systems will be hosted on platform hardware with independent failure modes. This can be specified through use of the AADL property `Not_Collocated`.

4.3.2.3 Guarantees

If hardware failures are independent, and there are less than N failures, then at least one replica will be functional at all times. Guarantees associated with the use of various voting algorithms will be addressed separately, as part of the definition of voting design patterns.

4.3.2.4 Instantiation

Add copies of selected system blocks with unique names. Add data connections for new input and output ports specified in the pattern arguments. Add output merge system blocks, if specified in the pattern arguments. Behavioral properties that have been defined for the original system blocks will be copied in the replicas with renaming as needed.

4.3.2.5 Exemplar AADL models

Figure 12 shows a graphical AADL model illustrating a sample system before application of the Replicate pattern. Figure 13 shows the instantiation of the Replicate pattern with N = 2 and the

default options for both input and output ports. Figure 14 shows the instantiation of the Replicate pattern with N = 2 and the option to share the input connection source and add a merge block to reconcile the new output ports.

The textual AADL model with the default port options follows the figure. Items to be changed or added during pattern instantiation are highlighted.

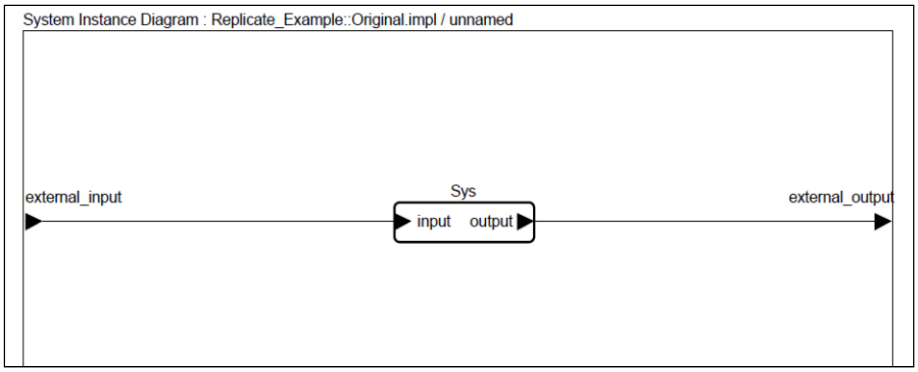


Figure 12 – AADL graphical model for Replication pattern (input context)

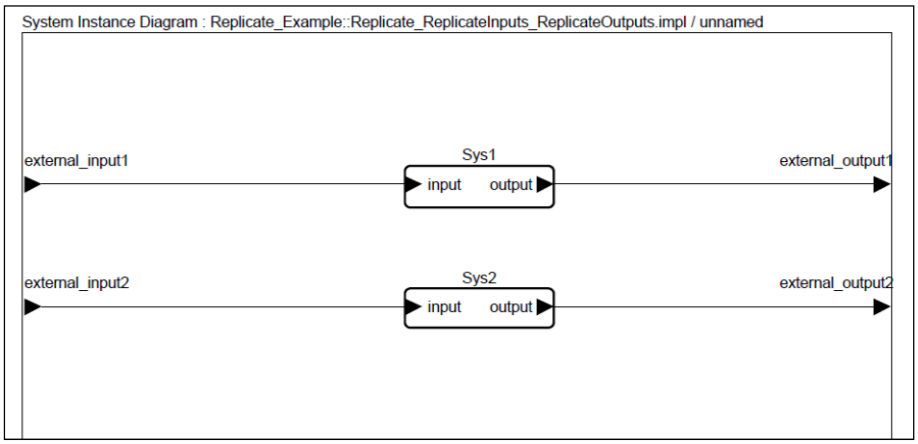


Figure 13 – After application of Replication pattern (replicate inputs and outputs)

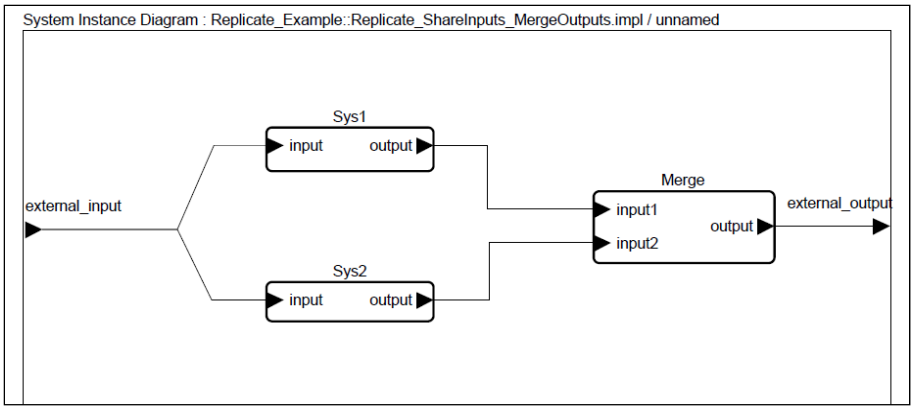


Figure 14 – After application of Replication pattern (shared inputs and merged outputs)

```
system Sys
  features
    input: in data port;
    output: out data port;
end Sys;

system implementation Sys.impl
end Sys.impl;

system Replicate_ReplicateInputs_ReplicateOutputs
  features
    external_input1: in data port;
    external_input2: in data port;
    external_output1: out data port;
    external_output2: out data port;
end Replicate_ReplicateInputs_ReplicateOutputs;

system implementation Replicate_ReplicateInputs_ReplicateOutputs.impl
  subcomponents
    Sys1: system Sys.impl {
       META_Properties::Not_Collocated => (reference (Sys2))
    };
    Sys2: system Sys.impl {
       META_Properties::Not_Collocated => (reference (Sys1))
    };
  connections
    DataConnection1: data port external_input1 -> Sys1.input;
    DataConnection2: data port external_input2 -> Sys2.input;
    DataConnection3: data port Sys1.output -> external_output1;
    DataConnection4: data port Sys2.output -> external_output2;
end Replicate_ReplicateInputs_ReplicateOutputs.impl;
```

4.3.3 Leader Selection

The purpose of the Leader Selection pattern is to coordinate a group of nodes so that a single node is agreed upon as the 'leader' at any given time. The nodes typically correspond to replicated computations hosted on distributed computing resources, and are used as part of a fault-tolerance mechanism. If a replicated node fails, this allows a non-failed node to be selected as the one which will interact with the rest of the system.

To use the pattern, a group of N nodes (systems or processes) is identified that are to select a leader from among themselves. The leader selection pattern will insert new leader selection threads into each of the systems/processes which are to participate in leader selection. Each thread will have a unique identifier (an integer) to determine its priority in selecting a leader. Connections will be added so that all leader selection threads are able to communicate with each other (N-1 input ports, 1 output port). In addition, each leader selection thread will have an input port from which it determines (from other local systems) if it is failed, and an output port which will say if it is the leader. These two ports are initially left unconnected.

4.3.3.1 Arguments

1. Set of N nodes to select a leader from. In AADL, these must correspond to processes since the leader selection thread to be inserted must be contained in a process.
2. Leader priority (unique integer) for each node.

4.3.3.2 Assumptions

1. The leader selection nodes must communicate synchronously with a one-step delay.
2. At least one node is functional (non-failed) at any given time.

4.3.3.3 Guarantees

1. All non-failed nodes shall agree on who is the leader.
2. If a node fails, leadership is transferred to a non-failed node in the next step.
3. If non-failed nodes exist, then in the next step one of them will be the leader.
4. A non-failed leader node shall remain leader as long as no user request to change leadership to a different non-failed node has been made.

4.3.3.4 Instantiation

Add leader selection threads to each node (process). Each leader selection thread has N-1 inputs ports and one output port to coordinate with the other leader threads. Add connections from each leader output to the other leader inputs. Add the required assumption and guarantee properties. Check uniqueness of leader priorities. Connections between each leader thread and local applications are added manually to provide any state information needed.

4.3.3.5 Exemplar AADL models

Figure 15 shows a graphical AADL model illustrating a sample system with the leader selection pattern applied to two nodes, systems S1 and S2. S1 and S2 each contain a process with one application thread A. The leader algorithm is inserted as new thread T in each process. T communicates with the other leader thread and is connected to the existing application thread to obtain any local state information needed by the leader algorithm.

The textual AADL model follows the figure. Items to be changed or added during pattern instantiation are highlighted.

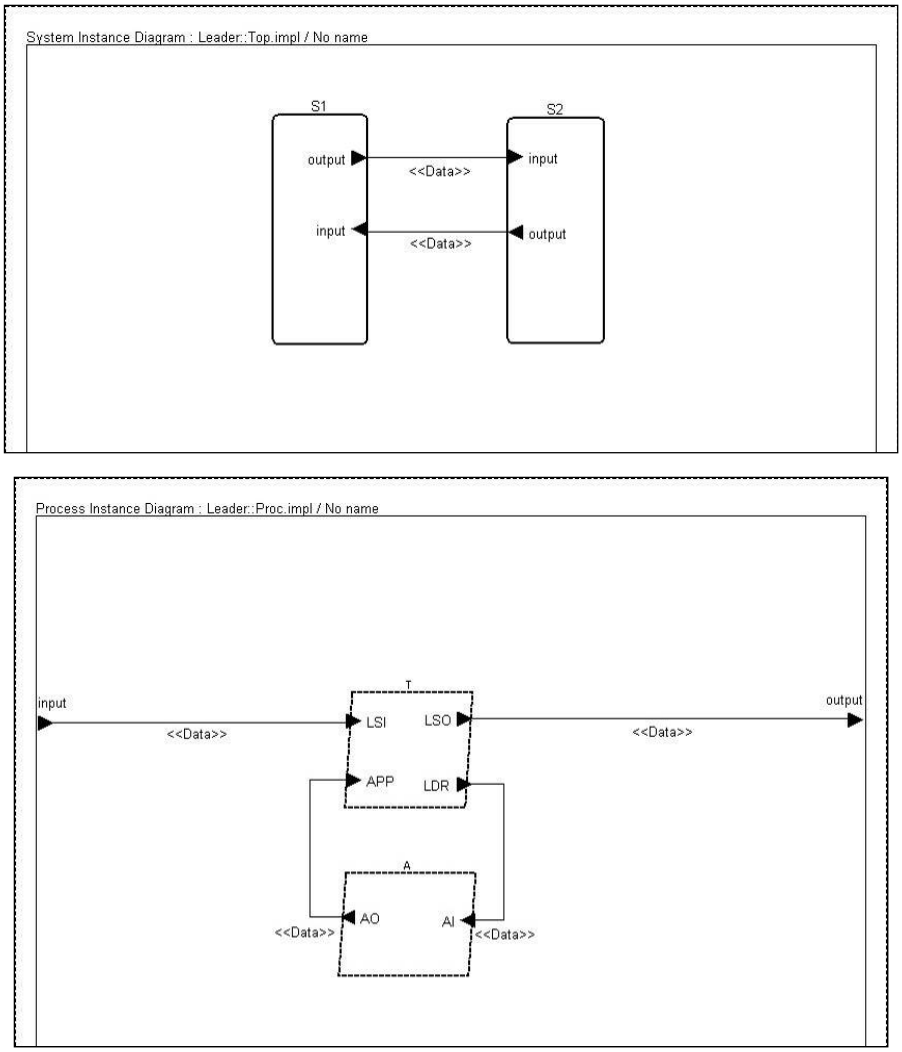


Figure 15 – Leader Thread T Inserted Into Each Process

```
system Top
end Top;

system Sys
  features
    input: in data port;
    output: out data port;
end Sys;

process Proc
  features
    input: in data port;
    output: out data port;
end Proc;

thread Ldr
  features
    LSI: in data port;
    LSO: out data port;
    APP: in data port;
```

```
    LDR: out data port;
end Ldr;

thread App
  features
    AO: out data port;
    AI: in data port;
end App;

thread implementation Ldr.impl
end Ldr.impl;

thread implementation App.impl
end App.impl;

process implementation Proc.impl
  subcomponents
    T: thread Ldr.impl;
    A: thread App.impl;
  connections
    con1: data port input -> T.LSI;
    con2: data port T.LSO -> output;
    con3: data port T.LDR -> A.AI;
    con4: data port A.AO -> T.APP;
end Proc.impl;

system implementation Sys.impl
  subcomponents
    P: process Proc.impl;
  connections
    con1: data port input -> P.input;
    con2: data port P.output -> output;
end Sys.impl;

system implementation Top.impl
  subcomponents
    S1: system Sys.impl;
    S2: system Sys.impl;
  connections
    con1: data port S2.output -> S1.input;
    con2: data port S1.output -> S2.input;
end Top.impl;
```

4.3.4 Fusion/Voting

The purpose of the Fusion pattern is to insert a component into the architecture that combines several component interfaces into a single interface. The component supplies properties that define the validation/selection algorithm that is used and its impact on the fault tolerance or performance properties of the interfaces. The fusion algorithm could provide voting through exact or approximate agreement or by mid-value selection. The output could correspond to one of the selected inputs or it could be a computing average.

To use the pattern, the user will select from a predefined set of fusion algorithms that are presented in a list. Each option will describe the properties and allow the user to browse these as part of the selection process. The user will select the type of component to be inserted in the model to perform the fusion algorithm. There are three initial choices: System (for abstract system designs), Thread (for software implemented voting), and Device for hardware implementations. Finally, the user will select the insertion point for the voter by first selecting an existing architecture component that is the current destination of the interfaces to be voted. After component selection the user will be presented with a list of input interfaces that match the constraints required for the voter that was selected. The user can then select the set of interfaces that the voter will be applied to.

4.3.4.1 Arguments

1. Set of interfaces which will be the sources of data for the voter and the destination of the output computation.
2. The type for the inserted component (System, Thread, Device)
3. A set of properties that describe the fusion component to be inserted
   a. Number of interfaces required
   b. Data type congruency required

4.3.4.2 Assumptions

1. The interfaces must terminate at the same destination component. The pattern only inserts the fusion component and does not impact data flow in the architecture.
2. The interfaces all carry the same data type.

4.3.4.3 Guarantees

Verified properties are specific to the type of voting or selection algorithm that is supplied with the new component. These are example properties for a three-input voter (properties are symmetric for the other inputs).

1. If all three sensors are valid then the voter output is valid and its value is the middle value of the three inputs.

```
IN1.Valid & IN2.Valid & IN3.Valid ->
   OUTPUT.Valid & OUTPUT.Val = Min(Max(IN1.Val, IN2.Val),
   Max(IN1.Val, IN3.Val), Max(IN2.Val, IN3.Val))
```

2. If only two sensors are valid then the voter output is valid and its value is the average of the good sensors. However, if there are only two valid sensors and they are far apart the voter should not produce a valid output.

```
(IN1.Valid & IN2.Valid & !IN3.Valid) &
  ABS(IN1.Val - IN2.Val) <= MaxDiff ->
  OUTPUT.Valid & OUTPUT.Val = (IN1Val + IN2.Val) / 2.0;
(IN1.Valid & IN2.Valid & !IN3.Valid) &
  ABS(IN1.Val - IN2.Val) > MaxDiff -> !OUTPUT.Valid;
```

3. If only one sensor is valid, then it is used for the voter output.

```
IN1.Valid & !IN2.Valid & !IN3.Valid ->
  OUTPUT.Valid & OUTPUT.Val = IN1.Val
```

4.3.4.4 Instantiation

Add the new fusion component. Remove old connections from source interfaces to destination. Replace them with connections that flow through the fusion component. Add behavioral assumption and guarantee properties describing the fusion/voting algorithm chose.

4.3.4.5 Exemplar AADL models

Figure 16 shows a graphical AADL model illustrating a sample system with the fusion pattern applied to three data sources. In this case, as voting algorithm has been inserted as a new thread. The output of the voter provides a single fused output to the input of the application thread A.

The textual AADL model follows the figure. Items to be changed or added during pattern instantiation are highlighted.

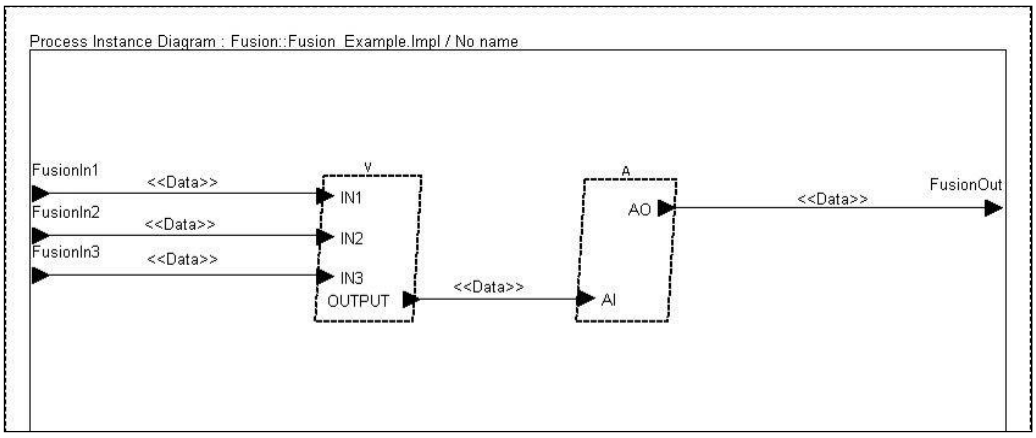


Figure 16 – Voter thread inserted into process by Fusion pattern

```
process Fusion_Example
  features
    FusionOut: out data port Fusion_Data;
    FusionIn1: in data port Fusion_Data;
    FusionIn2: in data port Fusion_Data;
    FusionIn3: in data port Fusion_Data;
end Fusion_Example;
```

```
thread Voter
  features
    IN1: in data port Fusion_Data;
    IN2: in data port Fusion_Data;
    IN3: in data port Fusion_Data;
    OUTPUT: out data port Fusion_Data;
end Voter;

thread App
  features
    AO: out data port Fusion_Data;
    AI: in data port Fusion_Data;
end App;

thread implementation App.impl
end App.impl;


data Fusion_Data
end Fusion_Data;

data Boolean
end Boolean;

data Real
end Real;

data implementation Fusion_Data.Impl
  subcomponents
    Val: data Real;
    Valid: data Boolean;
end Fusion_Data.Impl;

process implementation Fusion_Example.Impl
  subcomponents
    V: thread Voter.Impl;
    A: thread App.Impl;

  connections
    conn1: data port FusionIn1 -> V.IN1;
    conn2: data port FusionIn2 -> V.IN2;
    conn3: data port FusionIn3 -> V.IN3;
    conn4: data port V.OUTPUT -> A.AI;
    conn5: data port A.AO -> FusionOut;
 end Fusion_Example.Impl;

thread implementation Voter.Impl
end Voter.Impl;
```

4.3.5 Multi-Rate PALS

The purpose of the multi-rate PALS pattern is to support the virtual synchronization between multi-rate distributed computations. It supports the use of both harmonic and non-harmonic periods for these computations. The pattern guarantees that the design and verification complexities for distributed synchronization in a multi-rate system is still remains simple and is

equivalent to the corresponding single-rate perfectly synchronized system.

To use the pattern, a group of *n* nodes (systems) is selected that are to execute at different periods T_i, i = 1…N. In order to guarantee virtual synchronization, the pattern adds a *synchronization interface, called multi-rate synchronizer,* at each node. The synchronizer executes at a period equal to T_{hp} = least-common-multiple of (T_1,…, T_N). The pattern guarantees that other nodes in the group receive the outputs of a node at approximately the same global time.

The type of the synchronizer component (system, process, or thread), is determined by the user. The pattern assumes it will collocated with the receiver node; i.e., it executes on the same processor. Currently, the proposed instantiation of the pattern adds an AADL thread subcomponent for the synchronizer in the same process as the synchronization logic of each node. It propagates the message using AADL *immediate* connections. The pattern also defines the communication and scheduling characteristics between the synchronizer and the receiving computation logic.

4.3.5.1 Arguments

1. A group of n nodes. In AADL, this corresponds to a set of AADL thread elements distributed across defined process elements.
2. Period of each computation, T_i
3. Name of the multi-rate PALS group.
4. Set of (output port, input port) pairs used in the multi-rate synchronization.

4.3.5.2 Assumptions

Assumptions (1-5) of the multi-rate PALS pattern are similar to those of the original PALS pattern. The rest are specific to the multi-rate synchronizer component.

1. *Bounded Local Clock Error* - Each node *j* has access to an approximation of the true global time t via a local clock c_j, where the maximum error (called either jitter or skew) of each local clock is ε, i.e.,

   $| C_j - t | < \varepsilon$.

2. *Monotonic Local Clocks* - The value of each local clock C_j is monotonically increasing. Each node may adjust its local clock rate, but it may never decrease the value of its local clock.
3. *Bounded Computation Time* – The computation of a node's new local state and outputs completes within a specified time. Typically this is the periodic scheduling deadline for a thread α^i_{max}.
4. *Bounded Message Delivery* – Messages are reliably delivered to their destinations with latency μ, where $\mu_{min} \leq \mu \leq \mu_{max}$. Depending on the system fault assumptions, this may require thus use of a fault-tolerant network.
5. *Node Fault Assumptions* – A failed node must not be able to send extra messages (more than one) during a PALS period. This could result in nodes receiving different messages, even though the network delivered each correctly. By default, we assume that the output of a failed node is '*null.*'

6. *Output Assumptions of Multi-Rate Synchronizer* – Currently, the pattern assumes that in each step, multi-rate synchronizer only propagates the last message it received during its previous period. (This assumption can be relaxed so that the synchronizer provides a vector of received messages.). Based on the node fault assumption, if the sender fails, the synchronizer propagates 'null' message.
7. *First Execution of Multi-Rate Synchronizer* –The first dispatch time of both multi-rate synchronizer and the receiving computation logic are same.

Similar to the original PALS pattern, the computation logic at each node must also satisfy the following constraints relating the system parameters of the associated computation logic *i*:

1. *Causality constraint* – Messages cannot be sent too early.

   $\text{Min}(\text{Output time}_i) + \text{Min}(\text{Output time}_{sync}) \geq 2\varepsilon - \mu_{min}$

2. *Computation Period constraint* – Messages cannot be sent too late.

   $\text{Max}(\text{Output time}_i) \leq T_i - \mu_{max} - 2\varepsilon$

$\text{Output time}_{sync}$ is the output time of multi-rate synchronizer.

Additionally, the following constraints on the period of the multi-rate synchronizer must be verified.

3. Period of multi-rate synchronizer = LCM(T1, …, Tn) where LCM = Least common multiple.
4. Priority of multi-rate synchronizer > Priority of the receiving computation logic.

4.3.5.3 Guarantees

1. Synchronization between multi-rate computations happens only at the hyper-period boundary. This hyper-period, denoted by T_{hp}, is determined by the LCM of the computation logic.
   a. The source node sends its messages to the multi-rate synchronizers at the receiving nodes. The synchronizers execute at a period equal to $T_{sync} = T_{hp}$.
   b. In each period, a multi-rate synchronizer receives $n_i = T_{sync}/T_i$ number of inputs from a source node with period T_i.
   c. Of the n_i messages received during the multi-rate synchronizer period j, the synchronizer only propagates the last message and makes it available to the receiving computation logic during its period j+1.

   Suppose that A (Period T_A) sends messages to B (period T_B). Then, there are $n_A = T_{sync}/T_A$ and $n_B = T_{sync}/T_B$ executions of A and B in an interval T_{sync}. With the multi-rate synchronizer, these nodes interact only at every T_{sync} interval such that at the multi-rate synchronization period j,

   $A.in(j.n_A+k) = B.out((j.n_B -1); k = 0…n_A-1$

   The pattern guarantees that the interaction of these nodes is equivalent to a group of perfectly synchronized nodes executing at period T_{sync}. If the corresponding equivalent nodes of A and B are A' and B', then

   $A'.in(j) = B'.out(j-1)$

2. Similar to the original PALS pattern, external inputs are passed through environment input synchronizer. If the external inputs are sent to multiple computations with different periods, then the environment input synchronizer operates at the period equal to the LCM of their periods.

4.3.5.4 Instantiation

Add the multi-rate synchronizer component. Remove old connections from source interfaces to destination. Replace them with connections that flow through the synchronizer component. Add AADL properties to components, behavioral assumption and guarantee properties.

The following properties are added to the multi-rate synchronizer during the instantiation:

1. Synchronizer type
   a. Corresponding AADL property: `PALS_Synchronizer_Type`
   b. Value: `Multi_Rate_Synchronizer`
   c. This property distinguishes the component as a PALS synchronizer for multi-rate synchronization from other types of synchronizer, e.g. environment input/output synchronizer.
2. Synchronization Period
   a. Corresponding AADL property: `PALS_Period`
   b. Value: LCM of the period of the related groups in the hierarchical system.
3. Connection timing from synchronizer to receiving computation logic
   a. Corresponding AADL property: `Timing` (standard AADLv2 property)
   b. Value: Immediate
   c. This property defines the scheduling priority of the synchronizer with respect to the PALS synchronization logic at the receiving node based on standard AADL semantics.

These properties will be used to enforce the thread and message scheduling constraints in the system implementation. In addition, PALS middleware subprograms need to be inserted in the implementation to propagate the outputs to the receiving node.

4.3.5.5 Exemplar AADL models

Figure 17 shows the input graphical AADL model of an example hierarchical control system consisting of a supervisory controller system (SCS), rudder control system (RCS) and aileron control system (ACS). The supervisor synchronizes the control systems of rudder and aileron based on their feedback response. These three systems execute at different periods. Figure 18 shows the graphical model with the process component of the process element inside RCS before the multi-rate synchronizer thread is inserted. Similar structure exists for the other systems, SCS and ACS. Figure 19 shows this process component after the multi-rate synchronizer (RCT) is inserted.

A snippet of the textual AADL model follows the figure. Items to be changed or added due to the multi-rate synchronization during pattern instantiation are highlighted. The textual model

only includes the definition of one system block (RCS); similar changes are also expected for other system blocks.

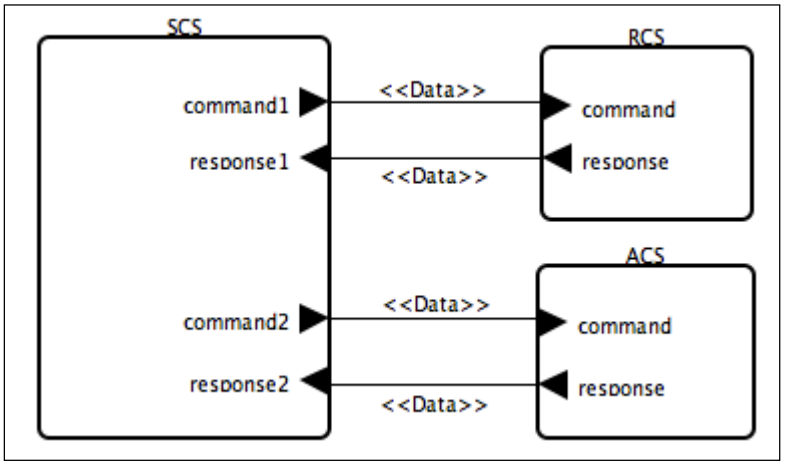


Figure 17 – AADL graphical model of the system blocks for multi-rate PALS

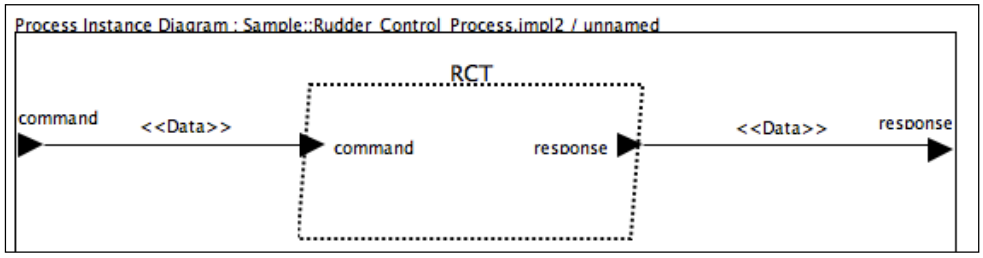


Figure 18 – Process component before inserting the multi-rate synchronizer

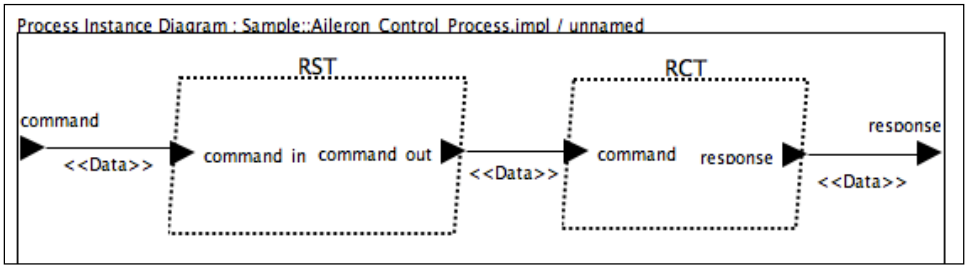


Figure 19 – Process component after inserting the multi-rate synchronizer

```
system MultiRateExample
end MultiRateExample;

system implementation MultiRateExample.impl
  subcomponents
    SCS: system Supervisory_Control_System.impl;
    RCS: system Rudder_Control_System.impl;
    ACS: system Aileron_Control_System.impl;
  connections
    SCStoRCS: data port SCS.command1 -> RCS.command;
    RCStoSCS: data port RCS.response -> SCS.response1;
    SCStoACS: data port SCS.command2 -> ACS.command;
    ACStoSCS: data port ACS.response -> SCS.response2;
```

```
end MultiRateExample.impl;

system Rudder_Control_System
  features
    response: out data port;
    command: in data port;
end Rudder_Control_System;

system implementation Rudder_Control_System.impl
  subcomponents
    RCP: process Rudder_Control_Process.impl;
  connections
    RCStoRCP: data port command -> RCP.command;
    RCPtoRCS: data port RCP.response -> response;
end Rudder_Control_System.impl;

process Rudder_Control_Process
  features
    command: in data port;
    response: out data port;
end Rudder_Control_Process;

process implementation Rudder_Control_Process.impl
  subcomponents
    RST: thread Rudder_Synchronizer_Thread;
    RCT: thread Rudder_Control_Thread;
  connections
    RCPToRST: data port command -> RST.command_in;
    RSTtoRCT: data port RST.command_out -> RCT.command;
    RCTtoRCP: data port RCT.response -> response;
  properties
    PALS_Properties::Multi_Rate_PALS_Id => "Supervisory_Control"
       applies to RCT;
    Period => 20ms applies to RCT;
    Deadline => 150 Ms applies to RCT;
    META_Properties::Output_Delay => 5 Ms applies to RCT;

    PALS_Properties::Multi_Rate_PALS_Id => "Supervisory_Control"
        applies to RST;
    PALS_Properties::PALS_Synchronizer_Type => Multi_Rate_Synchronizer
        applies to RST;
    PALS_Properties::PALS_Period => 100ms applies to RST;
    Timing_Properties::Timing => Immediate applies to RSTtoRCT;
end Rudder_Control_Process.impl;

thread Rudder_Control_Thread
  features
    response: out data port;
    command: in data port;
end Rudder_Control_Thread;

thread Rudder_Synchronizer_Thread
  features
    command_in: in data port;
    command_out: out data port;
end Rudder_Synchronizer_Thread;
```

4.4 System Architecture Models

This section describes the architectural models developed in the project. These architectural models provide examples that can be used to evaluate the design and verification tools created.

The architectural models describe a Flight Control System (FCS) for a typical regional jet aircraft. Section 4.4.1 provides an overview of this architecture. Section 4.4.2 provides detailed SysML and AADL textual specifications of the architecture. Section 4.4.3 describes how this architectural model can be generated from simpler "sunny day" architecture through application of a sequence of patterns. The SysML and AADL models for each step in this sequence are also provided with the changes caused by the pattern application highlighted. Since the architecture of a FCS is heavily influenced by the flight modes supported, an appendix providing more information about the mode logic of a typical FCS is provided in APPENDIX B.

4.4.1 Overview of the System Architectural Models

The example architectural model to be used for development and evaluation of the design and verification tools is a Flight Control System (FCS) for a typical regional jet aircraft. The FCS compares the measured state of the aircraft (position, speed, and attitude) to the desired state and generates guidance commands that are displayed as visual guidance cues on the left and right Flight Director (FD). The pilot or copilot can manually fly the aircraft to follow these guidance cues to achieve the desired reference speed and attitude. The FCS also provides an Autopilot (AP) function. When engaged, the AP generates commands to the actuators of the aircraft's control surfaces to automatically fly the aircraft to the reference attitude.

4.4.1.1 Top Level Logical and Physical Systems

The top level of the architecture divides the system into a logical architecture implemented primarily in software and a physical architecture implemented primarily in hardware. A SysML diagram of this structure is shown in Figure 20. The AADL textual specification is generated from the SysML model is shown in Appendix A.1.

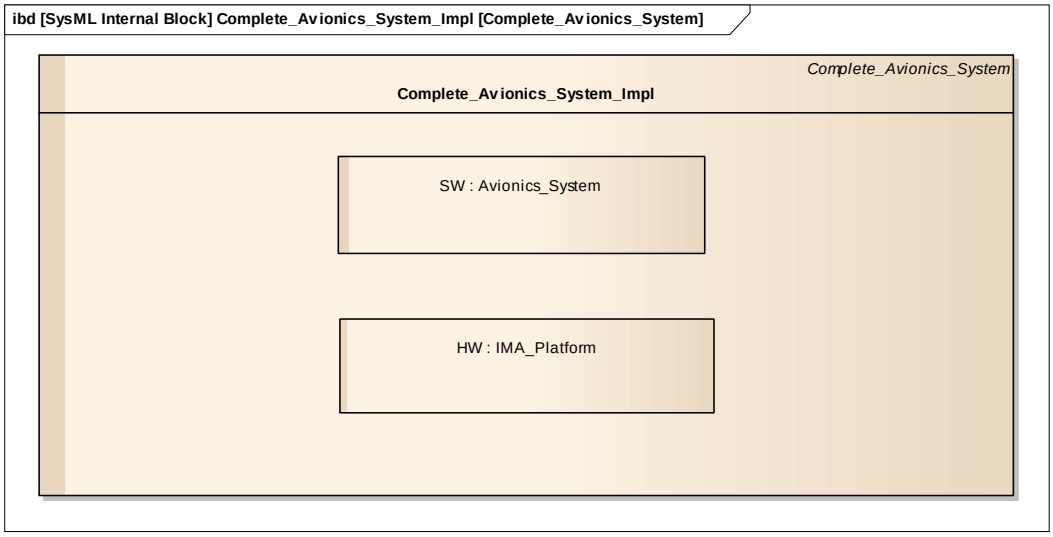


Figure 20 – Top Level Logical and Physical Overview

The system architecture is divided in this way to mirror the inherent difference in the logical architecture of the system functionality and the physical architecture of the platform on which

that functionality is implemented. This also allows bindings of logical components to physical components to be defined in a single place. These bindings are stored as *Tagged Values* in the SysML model and translated into *Connection Bindings* in the AADL model. For example, in the textual AADL specification it can be seen that the thread implementing the left Flight Guidance System (SW.FCS.FGS_L) is bound to processor A (HW.A.PRC) in the IMA platform.

4.4.1.2 Avionics Systems

A top level SysML diagram of the logical architecture of the Avionics System is shown in Figure 21. This view emphasizes the FCS even though some of the systems with which it interacts, such as the FMS, may be much larger. The AADL textual specification of the Avionics System generated from the SysML model is shown in Appendix A.2.

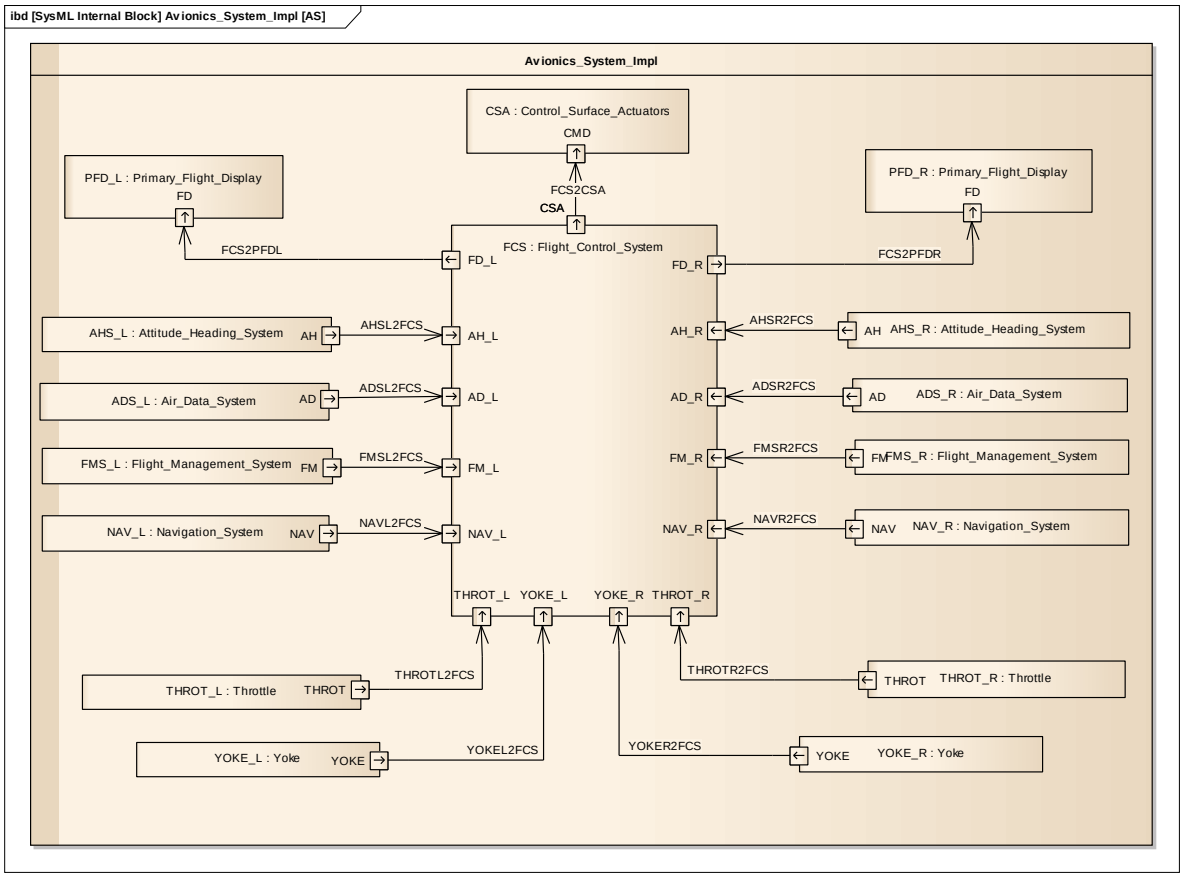


Figure 21 – Avionics System Architecture

The Control Surface Actuator (CSA) system positions the aircraft control surfaces based on the actuator commands generated by the FCS.

The left and right Primary Flight Displays (PFD) physically host the display of the FD guidance cues. A typical PFD is shown in Figure 22.

The FD guidance cues are the shaded wedge, or “V bars” in the center of the artificial horizon display. They are displayed directly above the white V bars showing the current attitude of the aircraft. In Figure 22, the current attitude indicates level flight (0 degrees of both roll and pitch) while the FD is indicating about 7.5 degrees of pitch and 0 degrees of roll. Also displayed on the PFD are the lateral and vertical mode annunciations of the Flight Guidance System (FGS), the

AP Engagement Indicator, and the Pilot Flying Indicator. These are described in more detail in later sections.

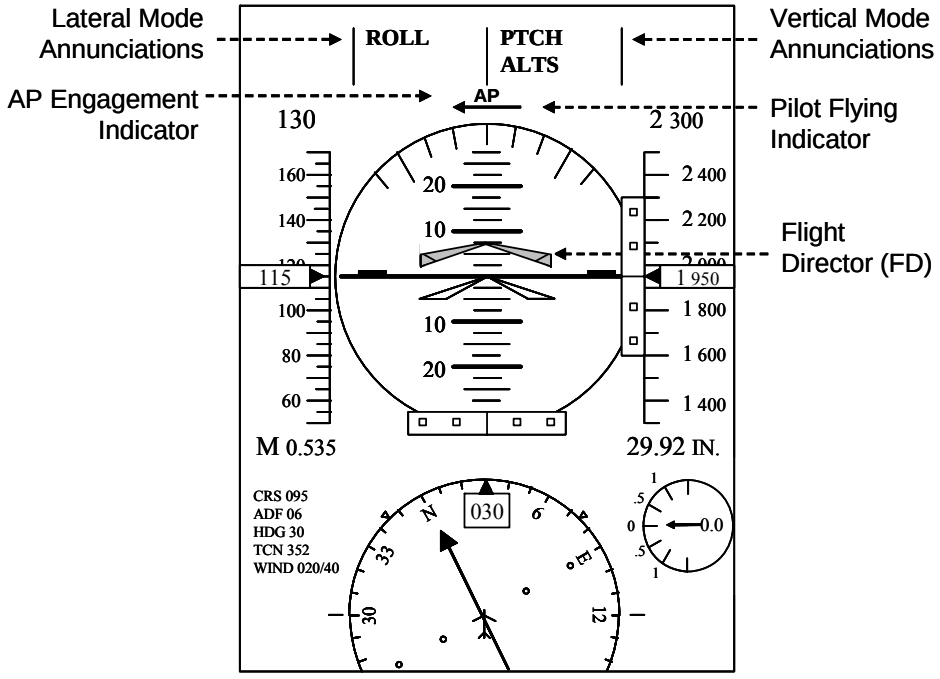


Figure 22 – Typical Primary Flight Display

As shown in Figure 21, the FCS accepts input about the aircraft's current attitude from the left and right Attitude Heading Systems (AHS). Information about the aircraft's airspeed relative to the surrounding air mass is provided by the Air Data Systems (ADS). Information about the aircraft's location relative to ground-based navigation sources such as Nondirectional Beacons (NB), VHF Omnidirectional Range (VOR) ground stations, and the Localizer (LOC) and Glideslope (GS) of Instrument Landing Systems (ILS) are provided by the Navigation (NAV) systems. The Flight Management Systems (FMS) provide new target reference settings (desired pitch and roll) for different flight modes as well as vertical guidance commands when Vertical Navigation (VNAV) mode is selected.

During manual operation, the pilot directly controls the aircraft through the Yokes and Throttles. When the AP function is engaged, the aircraft is controlled by the Flight Control System. The AP can be disengaged by the pilot by pressing the Disengage switch on the Throttle. Also while the AP Function is engaged, the pilot or copilot can initiate Control Wheel Steering (CWS) by pressing the SYNC switch on the yoke. This suspends the AP Function and allows the pilot to manually fly the aircraft to a new attitude. When the SYNC switch is released, the AP function is resumed using the new aircraft attitude as the reference attitude.

4.4.1.3 Flight Control System

A SysML overview of the internal organization of the FCS is shown in Figure 23. The AADL textual specification of the FCS is given in Appendix A.3.

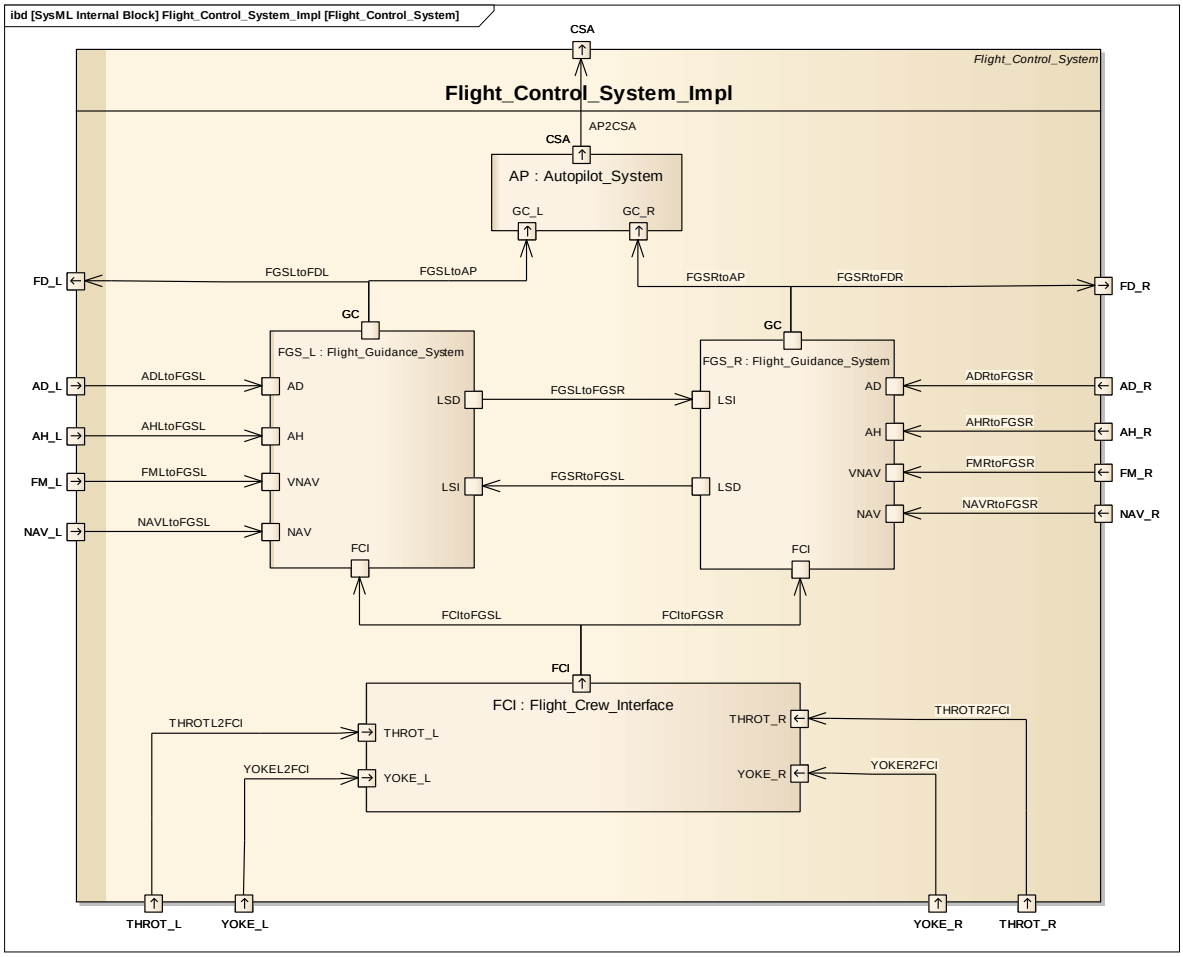


Figure 23 – Flight Control System Overview

To provide the necessary level of reliability and safety, the FCS is implemented as two redundant Flight Guidance Systems (FGS) that are required to fail independently (i.e., the probability of simultaneous failure must be acceptably improbable). This requirement is recorded by attaching a *Not_Collocated* AADL property to each FGS component in the FCS AADL textual specification (Appendix A.3).

Each FGS uses the inputs provided by the external AHS, ADS, FMS, and NAV systems to generate pitch and roll guidance commands.

Most of the time, the two FGS operate in *Dependent* mode in which only one FGS is active. The other FGS serves as a hot spare that can become active immediately on request by the pilot or in case of failure of the active FGS. However, in some critical modes of operation such as during approach or take off, both FGS are active and their outputs compared. This mode of operation is referred to as *Independent* mode.

The guidance commands computed by the FGS are used by the FD to position the FD guidance cues on the PFD. When engaged, the AP uses the same guidance commands to generate the actuator commands provided to the CSA system.

The flight crew interacts with the FCS primarily through the Flight Control Interface (FCI). This is often implemented as a single panel located over the glare shield in the cockpit. A typical Flight Control Panel (FCP) implementation of the FCI is shown Figure 24.

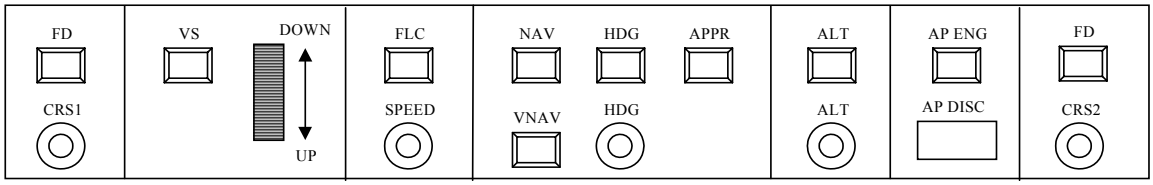


Figure 24 – Typical Flight Control Panel

The FCP includes switches for turning the Flight Director (FD) on and off, switches for selecting the different flight modes of the FGS such as vertical speed (VS), lateral navigation (NAV), heading select (HDG), altitude hold (ALT), and approach (APPR), and the Vertical Speed/Pitch Wheel. The FCP also often includes controls to engage the AP. A separate AP Disengage switch is provided on the FCP in case the Disengage switch on the Throttle should fail. The FCP may also provide feedback to the flight crew by lighting lamps on either side of a mode's button to indicate whether that mode is currently selected.

It should also be noted in Figure 23 that the AP Disengage switch on the throttle and the SYNC switch on the yoke are routed through the FCP.

4.4.1.4 Flight Guidance System

Each FGS will be implemented in software that will execute on a single processor, so the next level of decomposition makes explicit this transition from the system architecture to the software architecture by defining a single process FGP within the FGS as shown in Figure 25.

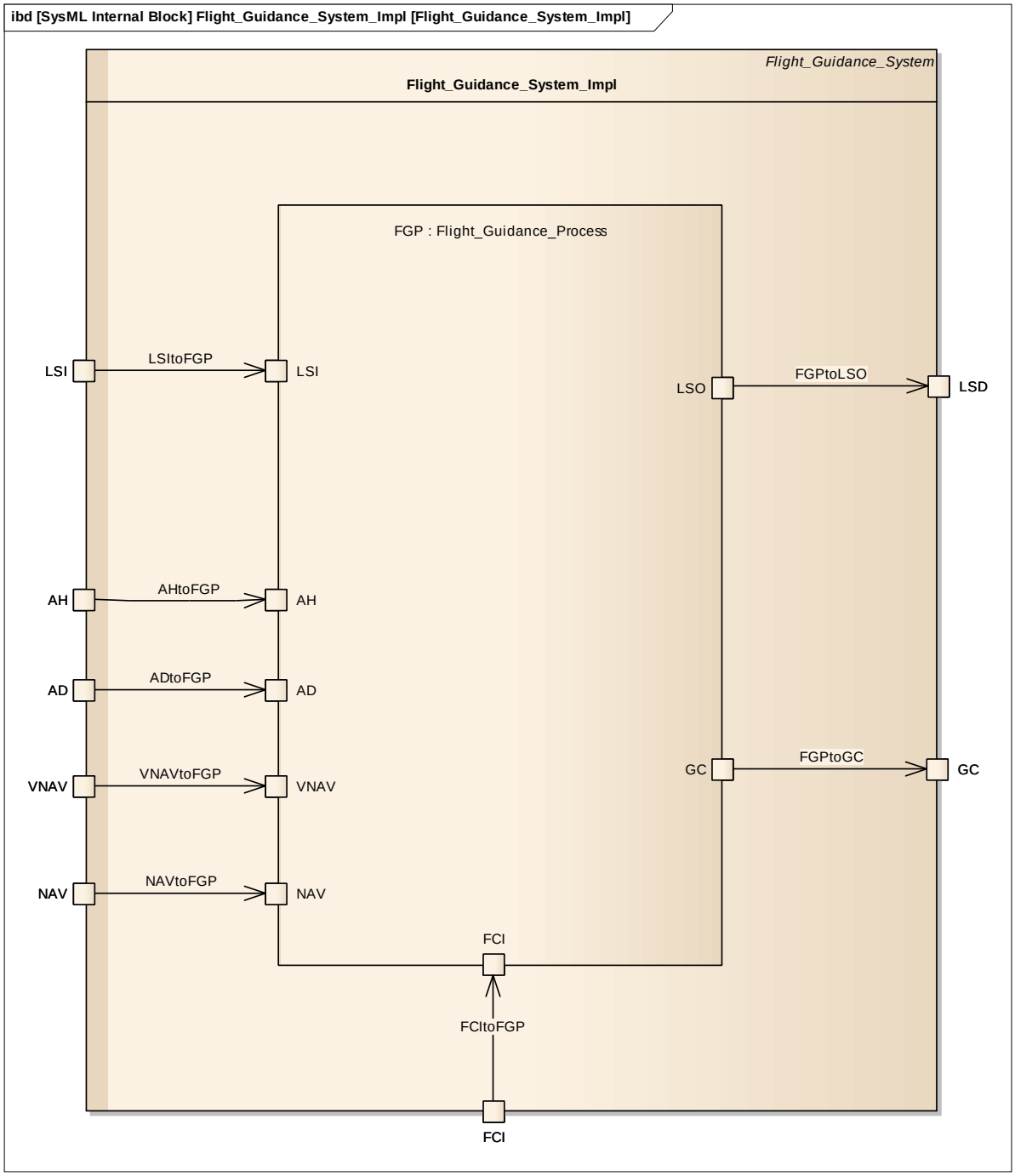


Figure 25 – Flight Guidance System Overview

A process is protected address space in which software components (such as threads) can execute. Typically, a process is bound to single virtual machine on a hardware processor. The FGP process component of Figure 25 is stereotyped as an AADL process.

The architecture of the Flight Guidance Process is further broken down into three threads that execute within its protected address space as shown in Figure 26.

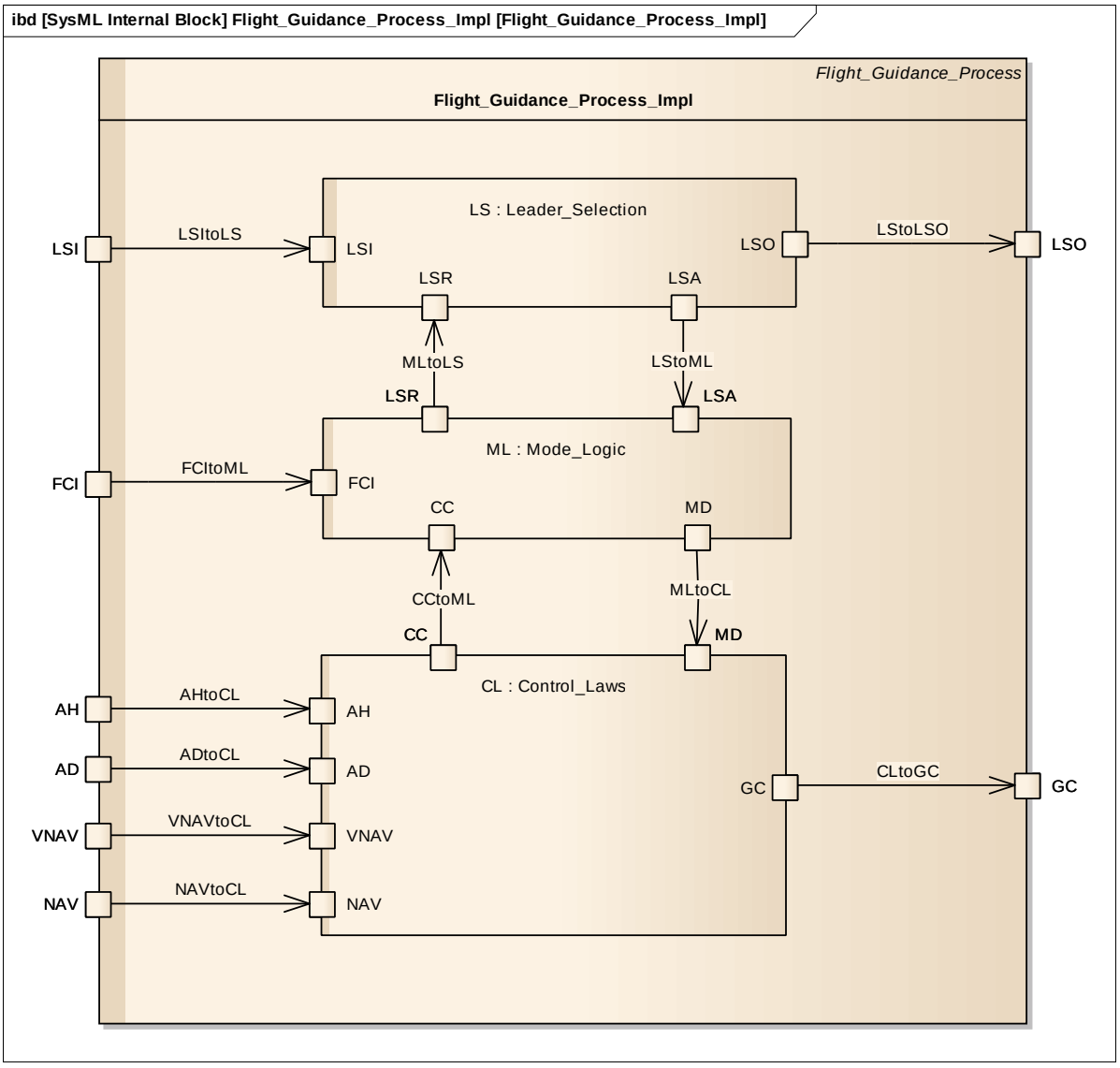


Figure 26 – Flight Guidance Process Overview

The Control Laws (CL) take information about the aircraft's current and desired state and compute the pitch and roll guidance commands that are ultimately used by the FD and the AP. The lateral control laws provide guidance about the roll axis, while the vertical control laws provide guidance about the pitch axis. Only one lateral control law generating a roll guidance command and one vertical control law generating a pitch guidance command can be *active* at any time. Additional control laws can be *armed* and accumulating state information in preparation for becoming active. The pitch and roll guidance commands are provided via the GC port. In addition, the CL provides *capture condition* status to the mode logic via its CC port and receives information about which control laws should be armed and active via its MD port.

The Mode Logic (ML) determines which lateral and vertical control laws are active and armed at any given time. It receives information about flight crew requests via its FCI port, the status of control law capture conditions via its CC port, and information about whether it is the "active"

side via its LSA port. It provides requests to become the active side via its LSR port and information about the current active and armed modes via its MD port.

The Leader Selection (LS) function determines which FGS is the current "active" side, i.e., the current leader. It is a two node version of the more general N node leader selection function. It receives requests from ML to elect a new leader via its LSR port, exchanges information with other LS nodes via its LSI and LSO ports, and indicates whether it believes it is the current leader through its LSA port. To simplify verification of the distributed algorithm for leader selection, each LS instance is implemented as Physically Asynchronous/Logically Synchronous (PALS) component and assigned a PALS period. This can be seen in the generated AADL specification of the entire FGS in Appendix A.4.

4.4.1.5 Autopilot System

The Autopilot System (APS) takes the Guidance Data generated by each FGS, selects the Guidance Data from the active side, and generates the Control Surface Actuator commands needed to move the aircraft control surfaces to achieve the desired pitch and roll. In an actual aircraft, the APS is implemented using fault-tolerant redundant hardware systems. However, in this example specification, it is simplified to single process that will execute on a single processor. The top level overview of the APS is shown in Figure 27.

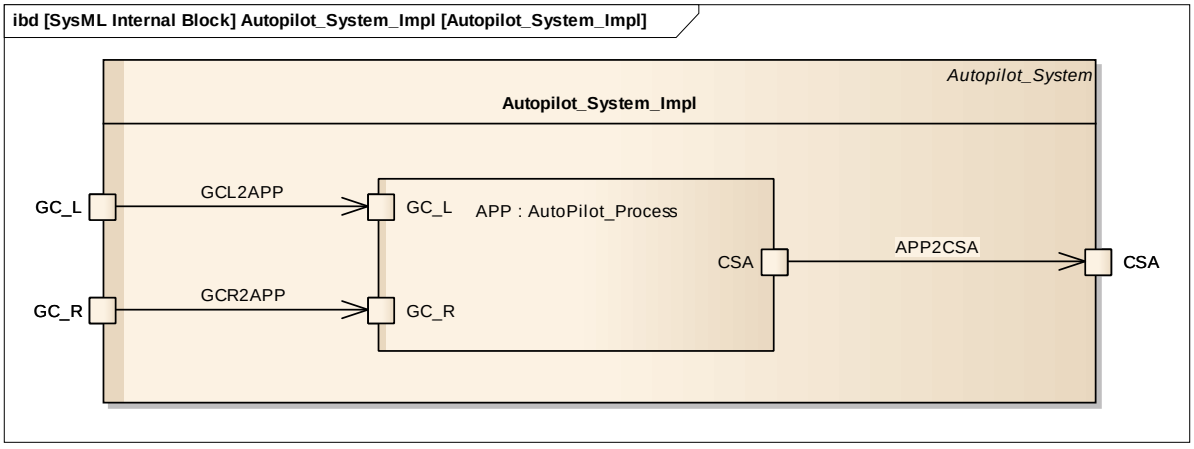


Figure 27 – Autopilot System Overview

Similar to the FGS, the APS has a single process APP that provides a protected address space. The internal structure of the APP process is shown in Figure 28.

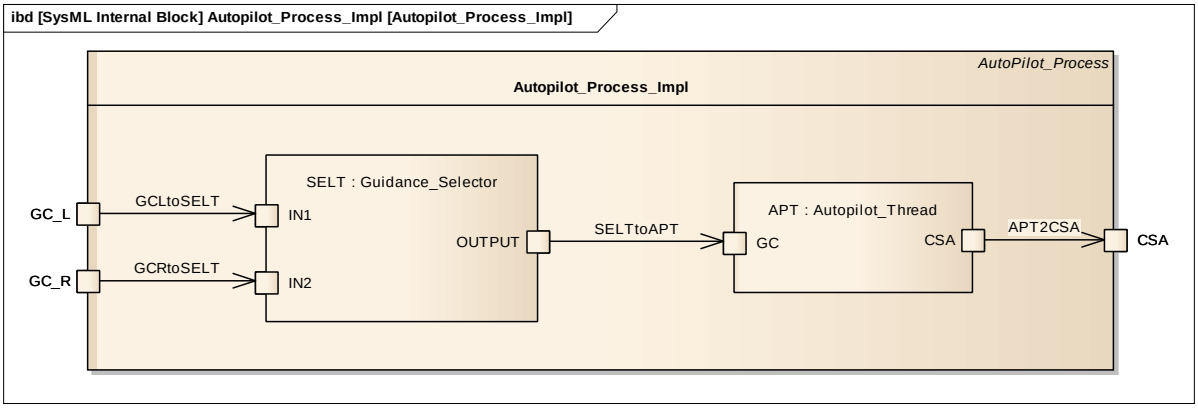


Figure 28 – Autopilot Process Overview

The SELT Guidance Selector thread selects the Guidance Data from the currently active FGS and passes it on to the APT Autopilot thread. The APT generates the Control Surfaces Actuator Commands that are passed to the servos on the aircraft's control surfaces. The AADL specification of the entire APS is shown in Appendix A.5.

4.4.1.6 Air Data System

To support experiments with probabilistic model checking, the Air Data System (ADS) has been elaborated to include three Airspeed Sensors as shown in Figure 29.

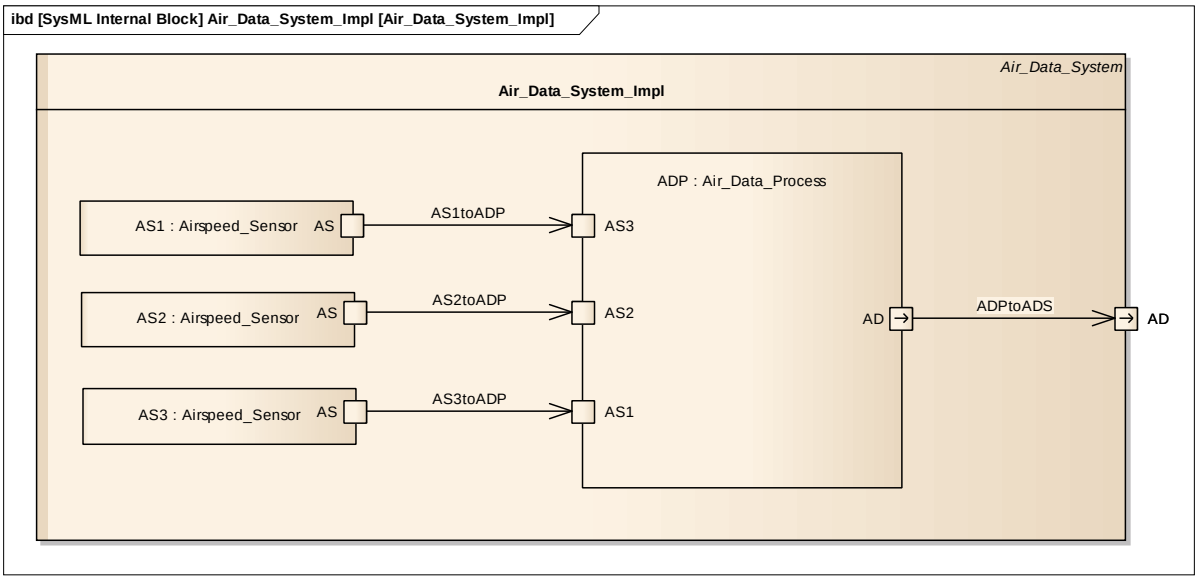


Figure 29 – Air Data System Overview

These sensors feed into a single Air Data Process (ADP). The internal structure of the ADP process is shown in Figure 30.

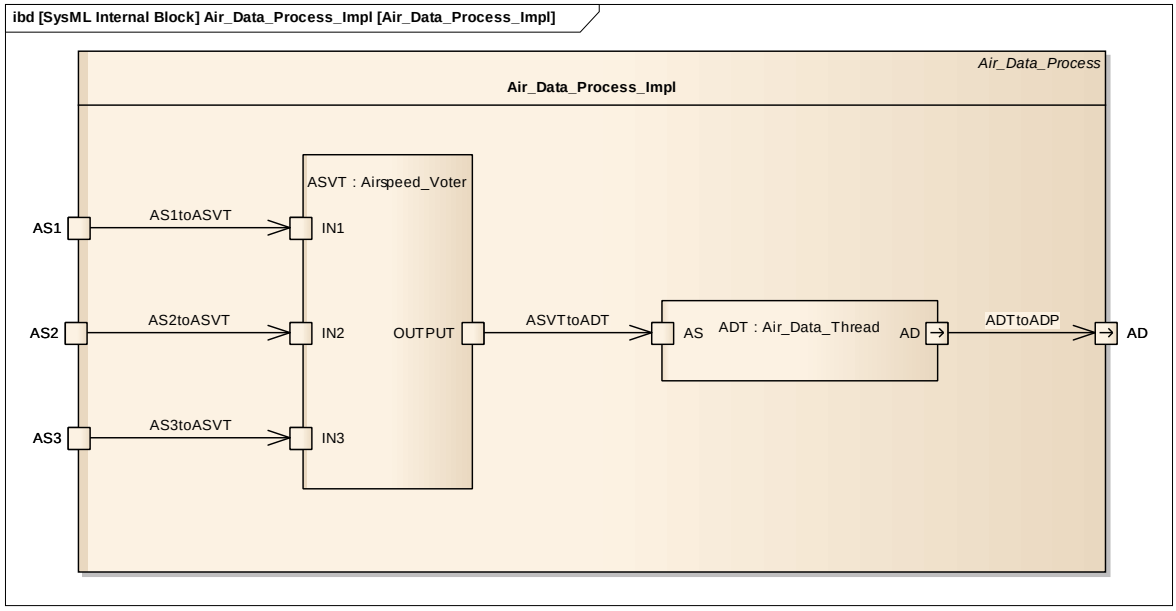


Figure 30 – Air Data Process Overview

The ADP process contains two threads. The ASVT Airspeed Voter thread votes the values of the three airspeed sensors and produces a single fault-tolerant airspeed that is fed to the Air Data Thread (ADT). The Air Data Thread combines the voted airspeed with data from other sensors and produces Air Data that is sent on to the FCS. The AADL specification for the entire ADS is shown in Appendix A.7.

4.4.1.7 Other Logical Systems

The other systems in the Avionics System Architecture of Figure 21 are unelaborated overviews. Systems such as the Attitude Heading System (AHS), Flight Management System (FMS), Navigation (NAV), Flight Crew Interface (FCI), and Primary Flight Display (PFD) consist of a singe process containing a single thread so that a complete AADL system instance can be generated. Other components such as the Yokes (YOKE), Throttles (THROT), and Control Surface Actuators (CSA) consist of a single AADL device. For this reason, they are not described in detail here, though full AADL textual specifications are provided in Appendix 4.4.2.

Note that the structure of the Avionics System Architecture allows them to be elaborated with more detail at any time without affecting the other subsystems.

4.4.1.8 Integrated Modular Avionics

As described in Section 4.4.1.1, the logical system architecture described in the preceding sections is mapped onto a model of the physical architecture, the Integrated Modular Avionics (IMA) platform. An overview of the IMA platform is shown in Figure 31.

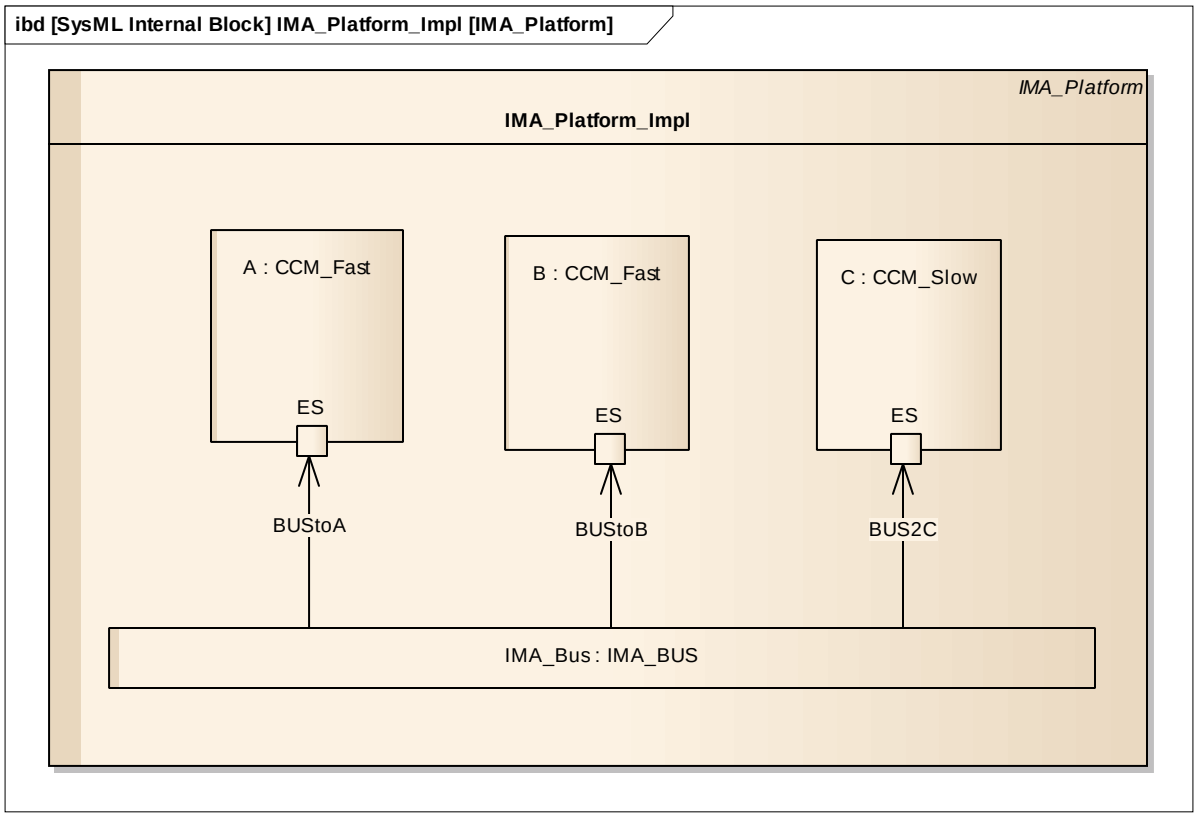


Figure 31 – IMA Platform Overview

The IMA platform consists of three Common Computing Modules (CCM) connected by a single IMA bus. The CCMs are stereotyped as AADL systems while the IMA bus is stereotyped as an AADL bus. The latency of the bus is set in a tagged value that is translated into an AADL property. CCM A and CCM B are fast computing modules while CCM C is a slow computing module.

The internal architecture of a fast CCM is shown in Figure 32. It consists of a single 350Mhz Power PC processor attached to an End System (ES) on the CCM that provides access to the IMA bus. The processor is stereotyped as an AADL processor. The slow CCM has an identical structure except that the processor is a 250Mhz Power PC.

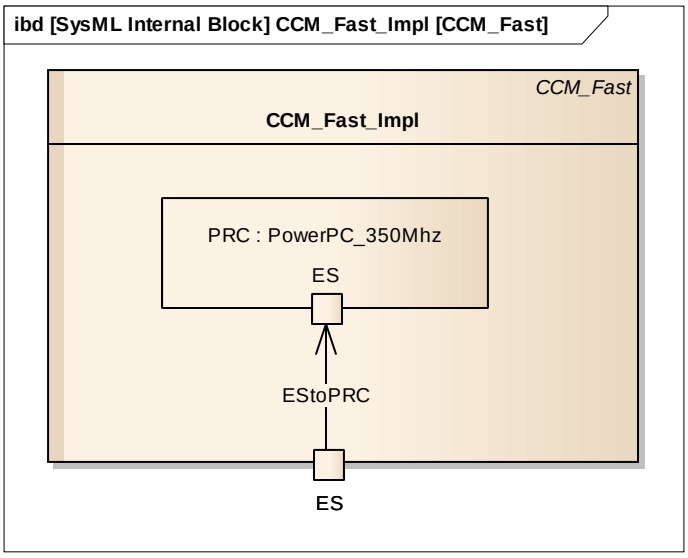


Figure 32 – Fast CCM Architecture

Since the bindings between the logical system architecture and the physical architecture are set in the topmost module, the IMA platform can be easily extended and modified without impacting the logical system architecture. For example, the single IMA bus could be replaced with a more detailed model that includes redundant channels for fault tolerance. More CCMs could be added and additional structure added to each CCM to enable analysis of CCM performance.

The AADL specification for the IMA platform is given in Appendix A.16.

4.4.2 AADL Specification of the System Architectural Model

The textual AADL specification of the system architectural model described in Section 4.4.1 is provided in the Appendix. This includes specifications for:

A.1 Complete Avionics System (TOP)
A.2 Avionics System (AS)
A.3 Flight Control System (FCS)
A.4 Flight Guidance System (FGS)
A.5 Autopilot System (APS)
A.6 Flight Control Interface (FCI)
A.7 Control Surface Actuators (CSA)
A.9 Attitude Heading System (AHS)
A.10 Flight Management System (FMS)
A.11 Navigation System (NAV)
A.12 Primary Flight Display (PFD)
A.13 Throttles (THROTTLES)
A.14 Yokes (YOKES)
A.15 Leader Selection (LDS)
A.16 IMA Platform (IMA)
A.17 META Property Set
A.18 PALS Property Set

4.4.3 System Design through Pattern Application

This section illustrates how increasingly complex systems can be constructed through repeated application of architectural patterns. Figure 33 shows an overview of how a sequence of pattern applications can be used to transform an initial non-fault-tolerant architecture into a fault-tolerant Flight Control System architecture.

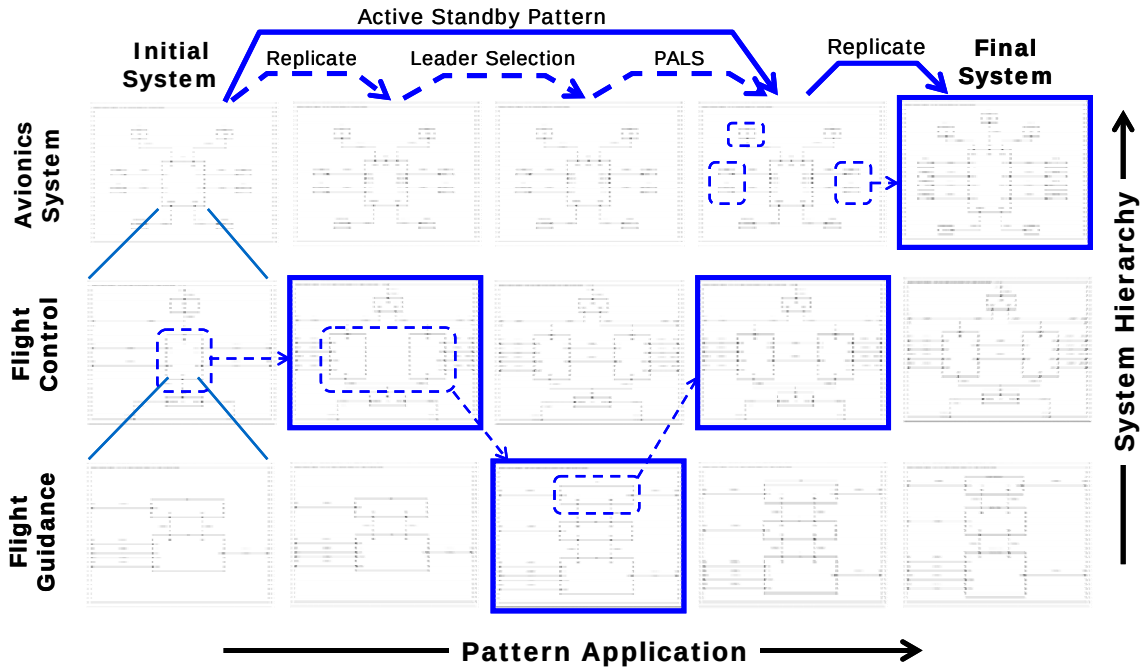


Figure 33 – System Design through Pattern Application

Three levels of the system architecture are shown: the Avionics System, the Flight Control System, and the Flight Guidance System. The transformation begins with the simple Initial System shown on the left, and concludes with the Final System described in Section 4.4.1.

The Initial System captures the functionality of the system under the assumption that nothing ever fails. It only has one set of inputs and outputs and has no redundancy in its implementation. It describes the system we would build in a perfect world from a perfect set of components. In this sense, it describes the essential functionality the customer desires.

Of course, components do fail and a critical system such as Flight Control must continue to function in spite of such failures. To move towards such a system, we first apply the replication pattern to the FGS component of the FCS to create a left and a right FGS. This pattern also replicates ports as necessary and applies the `not_collocated` property to each FGS. This property will be checked during implementation to make sure the functionality of each FGS is hosted in a separate fault-containment region. Note that the scope of the pattern's application terminates at the boundary of the FCS component. This creates inconsistencies in the system architecture at the Avionics System level and in the AP system (e.g., ports with no connections) that will be reconciled later.

Once the FGS is replicated, decisions must be made about how that redundancy is to be managed. For the current version of the FCS, we decide that it is sufficient for the *dependent mode* of operation discussed in Section 4.4.1.3 to have one FGS be the active FGS have the other

FGS serve as a hot spare. This requires that functionality be added select the current leader, i.e., the active FGS, and to ensure that one and only one FGS is the leader. This is done through application of the Leader Selection pattern for N nodes to each FGS with N = 2. This pattern inserts pre-verified leader selection functionality inside each FGS that determines the current leader. Application-specific changes then need to be included to add functionality for initiating a change in leadership, such as when the pilot or copilot requests a transfer of control to the other side.

In this example, we chose a component for the Leader Selection protocol that is correct only if all nodes execute synchronously. To satisfy this component assumption, we next apply the PALS synchronization pattern to the newly inserted Leader Selection (LS) nodes in each FGS. The pattern attaches the same *PALS Group Id* and *PALS Period* properties to each LS node. The *PALS Period* requirement will be verified during implementation to ensure it can actually be satisfied by the system implementation.

Finally, we use the replicate pattern to duplicate data sources in the Avionics System level and apply a yet to be specified pattern to the Autopilot System to deal with the extra set of guidance commands generated by the new FGS system.

The first three pattern applications can be grouped into a single *Active Standby* pattern that replicates the FGS, adds leader selection, and establishes logical synchrony. This more closely matches the goal of the system architect. It also better satisfies one of our main criteria for what constitutes a pattern, the reuse of verification. At the same time, patterns such as replication are fundamental to a variety of fault-tolerance patterns, such as triplex voting, dual-dual, or Byzantine fault resilience.

4.5 Pattern Verification

This section describes the verification of the architectural design patterns developed as part of the META project. A key element of these design patterns is that they provide guarantees of correct behavior when used in accordance with their specifications. Their behavior is proven through the use of formal methods as part of the pattern development process. The verification effort for the generic pattern is amortized over all subsequent instantiations of the pattern in specific system models. This amounts to *reuse* of the initial pattern verification. Analysis of system-level behavior can subsequently make use of the proven pattern guarantees without having to reprove them.

Section 4.5.1 provides an overview of architectural design patterns and the verification activities associated with their use in system development. Subsequent sections describe specific design patterns and document their formal models, assumptions and guarantees, and proofs of correctness. Verification of the following patterns is documented in this report:

1. Physically Asynchronous Logically Synchronous (PALS)
2. Leader Selection
3. Replication

4.5.1 Architectural Design Patterns

An architectural design pattern is a transformation applied to a system model that implements some desired functionality in a verifiably correct way. Each pattern can be thought of as a partial function on the space of system models. We will refer to the transformed system as the *instantiation* of a pattern.

Our principle objectives in creating architectural design patterns are to

1. Manage or reduce the overall system complexity, and
2. Allow the verification effort associated with critical design elements to be reused.

Both of these objectives contribute to the META goal of accelerating system development.

An excellent summary of our approach to verification of design patterns is found in [44]:

> A useful way to meet engineering challenges [of complex systems] is to amortize the use of formal methods not on an individual design, but on a generic family of system designs by means of a formal architectural pattern, that is, a generic formal specification of an engineering solution to a generic design problem that: (i) is shown to be correct by construction; (ii) comes with strong formal guarantees; and (iii) greatly reduces system complexity, making system verification and correct system implementation orders of magnitude simpler than if the pattern were not used.

Our pattern verification activities thus rely on formal methods as the means of providing the highest level of assurance, and capturing behavioral guarantees in a form that supports automation and verification reuse.

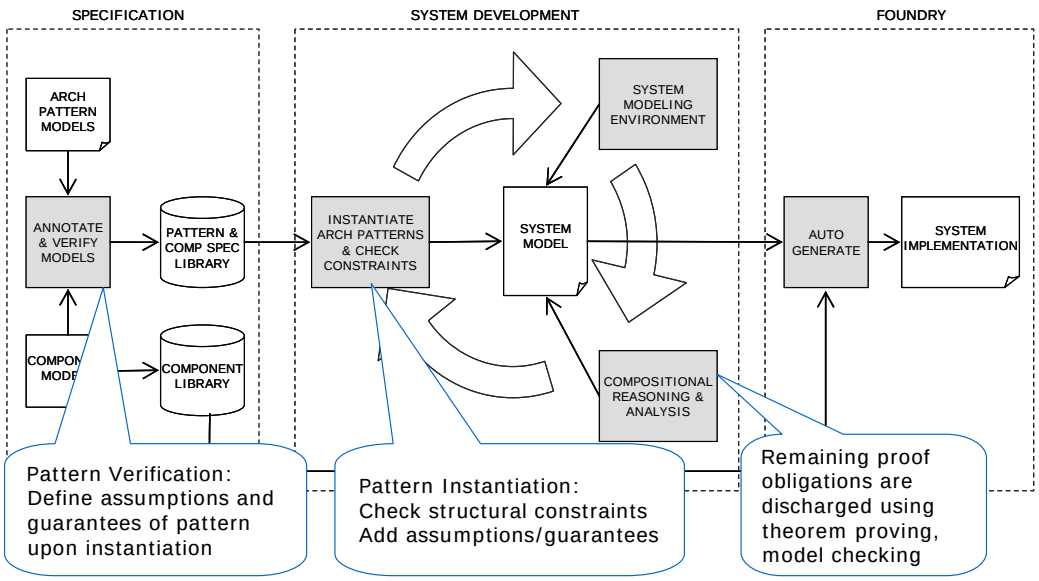


Figure 34 – Verification in Design Flow

Verification activities enter into our design flow at three distinct points (see Figure 34).

1. Pattern verification at design-time. A formal model of the pattern is created and analyzed to establish guaranteed behavioral properties about any valid instantiation of the pattern in a system model.
2. Pattern instantiation. A valid pattern instantiation must satisfy some specified constraints on the system model and the arguments of the pattern. These are checked at the time the pattern is instantiated.
3. System verification. System-level properties are proved by compositional reasoning about the guarantees provided by various design patterns that have been applied to the system, along with the structure of the system model and properties of components in the system model.

The focus of this document is pattern verification at design-time. To verify a pattern, we first must define correctness. By correctness, we mean that the pattern always establishes desired *guarantees* given certain *assumptions* about the system architecture. The guarantees codify properties established by the pattern such as logical synchrony (PALS), data synchronization, and fault tolerance; in other words they formally describe the reason that the pattern was created. The assumptions define restrictions placed on the system architecture that are necessary for the pattern to work correctly. For example, in the PALS pattern we require certain timing assumptions on thread and communication rates. The pattern guarantees are defined and proven at the time that the pattern is specified and are independent of a specific application of the pattern on a system model. That is, a pattern should be supplied with a proof that it establishes the guarantees of interest, given constraints on the system for *any valid* instantiation of the pattern.

The properties attached to patterns, either as guarantees or as assumptions, can take different forms. For example:

- They may be behavioral properties that describe the state of the system as it changes over time. Behavioral properties may be used to describe protocols governing component

interactions in the system, or the system response to combinations of triggering events. We will use the Property Specification Language (PSL) [40] to specify most behavioral properties.

- They may be structural properties of the system model to which the pattern is applied (pre-conditions), or of the transformed system model after pattern instantiation (post-conditions). Relationships among timing properties in the model or constraints on the numbers of various objects in the model are in this category. The REAL language for non-functional requirements [41] can be used to describe and check these properties.
- Some design patterns rely explicitly on resource allocation properties of the system, including real-time schedulability, memory allocation, and bandwidth allocation. There are many tools available to support verification of these properties, including the ASIIST tool developed by UIUC and Rockwell Collins [48].

In the following sections, we provide verification results for three patterns: PALS, Leader Selection, and Replication. In each case, we provide models and analysis results that demonstrate how the pattern guarantees have been proven under the specified system assumptions.

At the system level, pattern assumptions and guarantees are used to reason about the composition of the patterns to achieve desired system properties. For example, the Leader Selection pattern includes an assumption of synchronous data exchange which is satisfied by the PALS pattern guarantees, and an assumption of at least one working node, which is satisfied by the Replication pattern. Future work will focus on compositional reasoning to prove system requirements.

4.5.2 PALS Pattern

The purpose of the PALS pattern is to make portions of a distributed asynchronous system operate in virtual synchrony. This allows portions of the system logic to be designed and verified as though they will be executed on a synchronous platform, and then deployed in the asynchronous system with the same guaranteed behavior. As a result, design and verification activities are greatly simplified.

The PALS pattern was originally developed to address the need for synchronization in Integrated Modular Avionics (IMA) systems. In typical IMA architectures, each processing resource is driven by its own clock. While these clocks may have the same period, they execute asynchronously relative to each other with their own offset, drift, and jitter. This results in an architecture in which synchronous components execute asynchronously relative to each other. Such designs are often referred to as Globally Asynchronous/Locally Synchronous (GALS) architectures.

When system functionality is distributed to achieve a high level of reliability, the individual components usually still need to agree on some part of the global system state, such as which side of the aircraft (pilot or copilot) is the currently active side or which node is the current leader in a decision-making algorithm. Developing protocols to achieve such agreement in an asynchronous environment can be extremely difficult. Great care must be taken to establish the necessary coordination between the distributed components to avoid race and deadlock conditions and to implement the correct behavior. The PALS pattern provides a simplified design approach that helps guarantee correct implementation of these coordination algorithms.

The PALS pattern assumes each node has access to the global time within some small error, but does not require tight synchronization of its nodes as in a time-triggered architecture. The underlying intuition of the PALS design pattern is quite simple and is illustrated in Figure 35. Periodic computations performed on the distributed nodes may vary in time by some bounded amount. By enforcing appropriate bounds on when messages can be exchanged between nodes, it is possible to guarantee that all nodes will receive a consistent set of messages at each execution step.

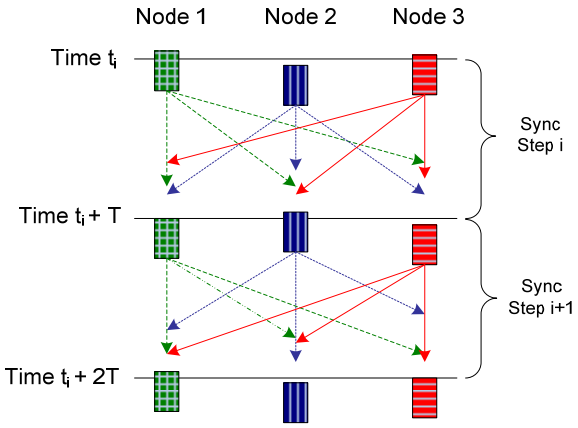


Figure 35 – PALS design pattern

4.5.2.1 Pattern instantiation

To use the pattern, a group of nodes (systems) is selected that are to execute at approximately the same time at period T. Some outputs (ports) are designated that are to be received by other nodes in the group such that all nodes will receive the same values at each execution step.

The pattern does not add any new data connections to the model, but assumes that the required connections already exist.

4.5.2.1.1 Pattern arguments:

1. Set of system blocks to synchronize
2. PALS synchronization period T
3. Name of PALS group

4.5.2.1.2 Assumptions

The PALS assumptions are conditions that the system design model must satisfy, either when the pattern is instantiated or possibly at a later stage of system design.

1. *Bounded Local Clock Error* - Each node i has access to an approximation of the true global time t via a local clock c_j, where the maximum error (called either jitter or skew) of each local clock is ε, i.e.,
    a. $| C_i - t | < \varepsilon$.
2. *Monotonic Local Clocks* - The value of each local clock C_i is monotonically increasing. Each node may adjust its local clock rate, but it may never decrease the value of its local clock.

3. *Bounded Computation Time* – The computation of a node's new local state and outputs completes within a specified time. Typically this is the periodic scheduling deadline for a thread α_{max}.
4. *Bounded Message Delivery* – Messages are reliably delivered to their destinations with latency μ, where $\mu_{min} \leq \mu \leq \mu_{max}$. Depending on the system fault assumptions, this may require thus use of a fault-tolerant network.
5. *Node fault assumptions* – A failed node must not be able to send extra messages during a PALS period. This could result in nodes receiving different messages, even though each was delivered correctly by the network.

In addition, there are some constraints relating system parameters that must be satisfied. These should appear in the model as "assumed" properties that must be verified. Two important constraints expressed in terms of AADL properties are:

1. *Causality constraint* – Messages cannot be sent too early.

   Min(Output time) $\geq 2\varepsilon - \mu_{min}$
2. *PALS period constraint* – Messages cannot be sent too late.

   Max(Output time) $\leq T - \mu_{max} - 2\varepsilon$

4.5.2.1.3 Guarantees:

1. The synchronization logic for each node executes with period T at approximately the same global time as every other node.
2. Messages generated during synchronization step i are consumed by their destination nodes in synchronization step i+1. For two nodes A and B where A sends data to B, this may be expressed (depending on analysis tool syntax) as

   B.in(i) = A.out(i-1), or next(B.in) = A.out.
3. In our model, this is expressed by a new AADL property called Synchronous_Communication which, when assigned the value "one_step_delay" will be used by system verification tools to generate an appropriate verification model. It may also be used to satisfy an assumption for Synchronous_Communication associated with another pattern (e.g., Leader Selection).
4. All nodes that have common external inputs start each synchronization step with identical values for those inputs. (This feature is not implemented in the current version of the pattern.)

4.5.2.2 Verification

Figure 36 shows a timeline of the computation and communication associated with a PALS node. Logical clock step i is assumed to begin at time t_i. For node j, this can be denoted by $\uparrow(C_j = i) = t_i$. Since each local clock may exhibit maximum error ε, computation during this step may actually start between true time $t_i - \varepsilon$ and $t_i + \varepsilon$. The local thread being synchronized by use of the PALS pattern may run any time during this logical period. In fact, its start may be delayed by the execution of other threads scheduled with the same or higher priority. A scheduling deadline α_{max} is specified for the thread, so it must be scheduled to complete execution by $t_i + \varepsilon + \alpha_{max}$.

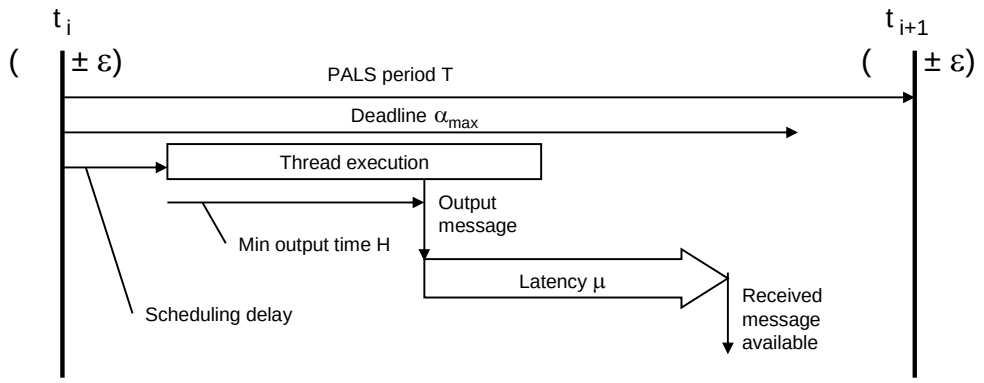


Figure 36 – PALS timeline

An output message to another node may be sent no earlier than H (output hold time) after the start of the thread. Error in the local clock means that the actual sending may occur as early as $t_i + H - \varepsilon$ (if the thread is scheduled to execute immediately) and as late as the thread's latest scheduled completion at $t_i + \varepsilon + \alpha_{max}$. Arrival of this message at its destination node then occurs between time $t_i + H - \varepsilon + \mu_{min}$ and $t_i + \alpha_{max} + \varepsilon + \mu_{max}$.

Finally, the start of the next computation step i+1 occurs at time t_{i+1}. Since the PALS clock period is T, this occurs between time $t_i + T - \varepsilon$ and $t_i + T + \varepsilon$.

4.5.2.2.1 PALS Constraints

Two constraints on H and T must be met to satisfy the requirement that messages generated during synchronization step i are consumed by their destination nodes in synchronization step i+1. These constraints are described in [46]

PALS Causality Constraint – A message created by a node in step i must not be delivered to its destination before the end of step i-1 on the destination (see Figure 37). Since the logical period in the consuming node for step i may begin as late as $t_i + \varepsilon$, this is condition is satisfied if

$$t_i + H - \varepsilon + \mu_{min} \geq t_i + \varepsilon$$

which means that the earliest output time must satisfy

$$H \geq 2\varepsilon - \mu_{min}.$$

In fact, communication latency μ may be very small, so we say

$$H \geq \max (2\varepsilon - \mu_{min,} 0)$$

Typically, H = 0 and messages can be sent as soon as they are computed.

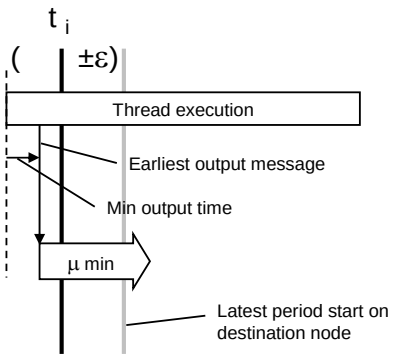


Figure 37 – PALS causality constraint

PALS Clock Period Constraint – A message created by a node in step i must be delivered to its destination before in step i so that it can be consumed in step i+1 (see Figure 38). Since the logical period in the consuming node for step i+1 may begin as early as $t_i + T - \varepsilon$, this is ensured if

$$t_i + T - \varepsilon \geq t_i + \alpha_{max} + \varepsilon + \mu_{max}$$

or simplifying

$$T \geq \alpha_{max} + 2\varepsilon + \mu_{max}$$

The deadline for periodic threads is often just the period itself. If the desired PALS period T is given as a requirement, an earlier deadline may be required. In this case, the thread deadline α_{max} must satisfy

$$\alpha_{max} \leq T - \mu_{max} - 2\varepsilon.$$

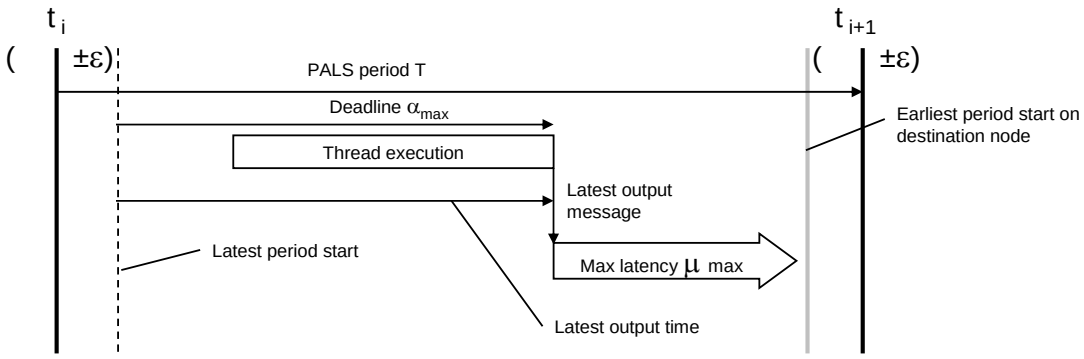


Figure 38 – PALS period constraint

As can be seen by the derivation of these constraints, PALS provides the optimal performance for synchronization of a distributed computation that can be supported the by the communications network and the local processors.

4.5.2.2.2 Summary of formal proof

A formal presentation of the PALS design pattern and a proof of its correct implementation of synchronous computation can be found in [50]. The key definitions and arguments are summarized in this section.

We first define the global computation model for the distributed system. Next, we examine the properties of global computations driven by perfectly synchronized clocks with period T. We then show that these behaviors are logically identical to that of the same system driven by clocks with bounded skews and period T under the PALS protocol, in the sense that they have the identical state transitions and identical inputs and outputs.

G1: Local Clocks for Global Computation

Each state machine M_i engaged in global computation is driven by a local clock C_i with the same period T. The global clock time is denoted as t. Clock C_i is said to be at its jth period, denoted as $C_i = j$, if the global time t satisfies the constraint

$$\uparrow(C_i = j) \leq t < \uparrow(C_i = j + 1)$$

where $\uparrow(C_i = j)$ denotes the time of the rising edge of clock C_i, when it just enters its jth period.

All the local clocks used for global computation are synchronized with the perfect global clock with clock skews of at most ε. If $\varepsilon = 0$, all clocks are perfectly synchronized. A system driven by perfectly synchronized clocks and the same system driven by the perfect global clock have the same behaviors.

Note that is important to ensure that the clock values are not only monotonic but also avoid large jumps. When a clock is ahead/behind, it should be corrected by decreasing/increasing its rate of progress. A clock value that goes backwards or has large jumps can result in serious errors in the computation of physical quantities such as velocity and acceleration.

Finally, in addition to global computation, a node can also perform local computations. Local computations are modeled by different state machines. The clocks used for local computations do not need to be synchronized with the clocks for global computation. In practice, it is convenient to synchronize a set of local clocks at some rate and then derive all other clock values. For example, we may choose to synchronize all the 100 Hz real time clocks and then derive all the other clock rates from this 100 Hz clock, including those used for global computation.

G2: Real Time Network

The network has a transmission delay μ bounded by $0 < \mu_{min} \leq \mu \leq \mu_{max}$. This delay may include any local queuing delays q on the sending/receiving nodes, as well as any switching delays in the network.

G3: Real Time Machine

Messages arriving at real time machine M_i during its jth clock period are buffered. At $\uparrow(C_i = j + 1)$, M_i reads the messages from the buffer, carries out the computation, transitions to the next state, and sends output messages. The task completion time α, including real time scheduling, computation, and I/O is bounded by $0 \leq \alpha \leq \alpha_{max}$. Since state transitions are driven by the clock, we say that a state machine M_i is at its jth state, when its clock is at its jth period.

To simplify notation, we assume that a (global computation) state machine sends and receives messages from and to every machine at each state. When there is no physical message, then it is modeled as sending/receiving messages with the "null" value that has no effect on the computation.

A system may interact with the external environment. When the environment sends a message to two replicated state machines, the network delays may be different. This can result in one machine receiving the data at clock period j, while the other receives it at clock period j + 1, even if their clocks are perfectly synchronized. The inconsistent views may lead to the divergence of the state machines. To avoid this problem, we need an environment message I/O synchronizer.

G4: Environment Message I/O Synchronizer

Let the input synchronizer, M^{I_sync}, be a real time machine as defined by G3. Messages from the environment are sent to the input synchronizer. Messages arriving at M^{I_sync} during $\uparrow(C^{I\_sync} = j) \leq t < \uparrow(C^{I\_sync} = j + 1)$ are buffered. At $\uparrow(C^{I\_sync} = j + 1)$, the input synchronizer reads buffered messages and forwards them to their destinations. When machines need to send messages to the external environment, they send them to the output synchronizer. Similarly, the output synchronizer, M^{O_sync}, reads messages at the rising edge of its clock tick and forwards them to the environment. The output synchronizer allows an external observer to have a synchronous view of the distributed states of a global computation.

We note that a networked computer is typically shared by state machines for both global computation and local computation. Local computation state machines can perform their local I/O independently of the PALS protocol. For example, a local servo controller reads the states of the local physical device, compares them to the setpoint provided by the supervisory controller, computes the control commands, and sends them to the device at a rate that is typically higher than the PALS clock rate used for supervisory control.

G5: The Period of Perfectly Synchronized Clocks

Giving a set of perfectly synchronized clocks used to drive global computation, the clock period T should satisfy the constraint $T > \alpha_{max} + \mu_{max}$.

We now state the properties of a distributed real time system driven by perfectly synchronized clocks modeled by G1, G2, G3, G4 and G5.

Fact 1. Under a set of perfectly synchronized clocks C_i: $1 \leq i \leq N$, defined by G5, when a machine M_s sends a message during period $C_s = j$, this message will reach all the N receiving machines M_r: $1 \leq r \leq N$, before their next clock ticks at $\uparrow(C_r = j + 1)$: $1 \leq r \leq N$. That is, a message sent during the jth period of the sender's local clock will be received when receiving machines are still in their jth period.

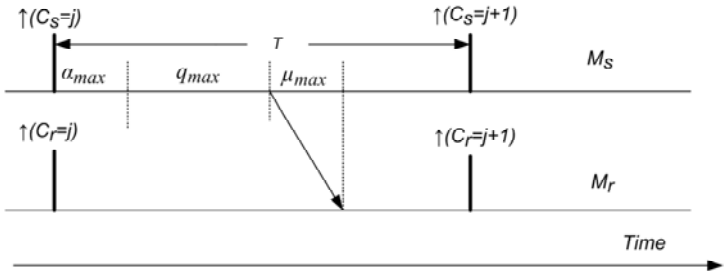


Figure 39 – A System Using Perfectly Synchronized Clocks

Proof. As illustrated by Figure 39, by G5 any pair of sender M_s and receiver M_r, the distance between sender's rising clock edge at period j and any receiver's rising clock edge at next period j + 1 is T. That is, $\uparrow (C_r = j + 1) - \uparrow (C_s = j) = T$. Since $T > \alpha_{max} + \mu_{max}$, Fact 1 follows.

Fact 2. Under a set of perfectly synchronized clocks C_i: $1 \le i \le N$, with their period T defined by G5, any message from external environment received by input synchronizer during jth period, will reach each of the N receiver machines M_r: $1 \le r \le N$, during the (j + 1)th period.

Proof. By G4, any message from environment must be sent to the input synchronizer. The messages arriving at time $\uparrow (C^{I\_sync} = j) \le t < \uparrow(C^{I\_sync} = j + 1)$ will be buffered at the synchronizer M^{I_sync}. By G4, these messages will be forwarded at $t = \uparrow (C^{I\_sync} = j + 1)$. By Fact 1, the message will reach all the N receiver machines by $\uparrow (C_r = j + 2)$: $1 \le r \le N$. Fact 2 follows.

We now examine a distributed real-time system where global computations are driven by clocks with bounded skews.

PALS Clocks. All the local clocks used by the PALS protocol for global computation are synchronized with the global clock with skews of at most ε.

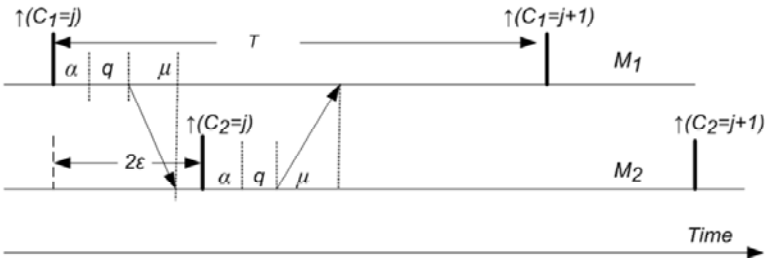


Figure 40 – Logical Equivalence to Causality Violation

We now examine the effect of clock skews. In Figure 40, M_1 is a replica of M_2. First, M_1 sends a message to M_2 at time $t = \uparrow(C_1 = j) + \alpha + q$. At global time $t = \uparrow(C_1 = j)$, M_2's local time is at $\uparrow(C_2 = j) - 2\varepsilon$. Suppose that the end to end delay from M_1 to M_2 is very short. That is, $\alpha + q + \mu < 2\varepsilon$. Under this condition, the M_1's message transmitted during clock period j may reach M_2 when M_2 is still at clock period j - 1. If the clocks were perfectly synchronized, this could only happen through a violation of causality. Since M_1 is a replica of M_2, this simulates sending a message to one's own past. Finally, note that M_2 also sends a message to M_1 at its jth period at $t = \uparrow(C_2 = j) + \alpha + q$. This message is received by M_1 at its jth period. Inconsistent views between replicated machines may lead to state divergence.

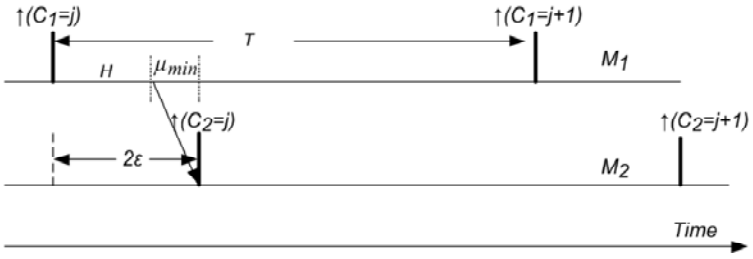


Figure 41 – Clock C_1 leads Clock C_2

To make sure that M_1's message sent during its jth period will arrive at M_2 no earlier than $\uparrow(C_2 = j)$, we introduce a minimal output hold time H. As we can see in Figure 41, if M_1 sends its message at or after $\uparrow(C_1 = j) + H$, where $H = 2\varepsilon - \mu_{min}$, then the message will arrive at M_2 no

earlier than ↑(C_2 = j). So the first rule of the PALS protocol, called the PALS Causality Constraint, is to require that machine M_i at its local PALS clock period j can transmit a message no earlier than ↑(C_1 = j) + H. Since the time lag between any pair of machines is less than or equal to 2ε, under the PALS Causality Constraint when a message sent from a machine M_s during its jth period cannot reach a receiving machine M_r earlier than ↑(C_r = j).

G6: PALS Causality Constraint.

A machine M_i at (PALS) clock period j cannot send a message earlier than ↑(C_i = j) + H, where H = 2ε - μ_{min}.

We now examine the case where machine M_2 leads machine M_1 by 2ε, as illustrated in Figure 4. Since clock C_1 now lags clock C_2, we need to ensure that a message sent by M_1 at jth period will reach M_2 before ↑(C_2 = j + 1).

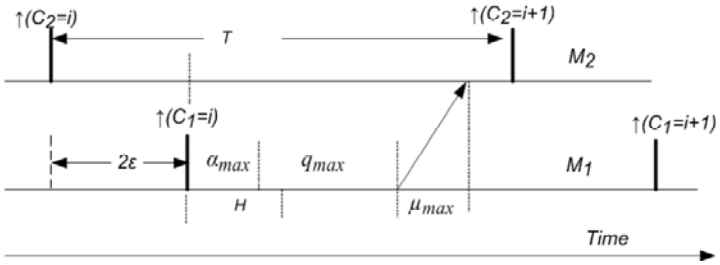


Figure 42 – Clock C_2 leads Clock C_1

As illustrated in Figure 42, the latest instant at which M_1 can transmit its messages on the network is max((↑(C_1 = j) + H), (↑(C_1 = j) + μ_{max} + q_{max})). The maximal network transmission delay is μ_{max}. Hence, it is necessary that the clock period satisfy

$$T > 2\varepsilon + \max(\mu_{max} + q_{max}, H) + \mu_{max}.$$

We now define the PALS clock period constraint.

G7: PALS Clock Period Constraint.

PALS clock period $T > 2\varepsilon + \max(\mu_{max} + q_{max}, H) + \mu_{max}$

The definitions so far now allow us to define a PALS system.

PALS Definition. A PALS system consists of state machines, environment input synchronizer, environment output synchronizer, and PALS clocks defined by rules G1, G2, G3, G4, G6, and G7.

Fact 3. Under PALS constraints, a message sent during sender's jth clock period will be received by other machines when they are still in their jth clock period.

Proof. Suppose that Fact 3 is false. There are two possible cases.

Case 1. Assume that there exists a pair of machines, where M_1's message sent during its clock C_1's jth period reaches M_2, during M_2's clock C_2's (j - 1)th period.

Proof of Case 1. As illustrated in Figure 41, in order for machine M_2 to receive M_1's period j message at M_2's clock period j - 1, M_2's clock must lag M_1's clock and the maximal lag is 2ε.

Due to the PALS Causality Constraint (G6), the earliest possible message arrival time at M_2, t_{arr}, is

$$t_{arr} = \uparrow(C_1 = j) + H + \mu_{min}$$
$$= \uparrow(C_1 = j) + (2\varepsilon - \mu_{min}) + \mu_{min}$$
$$= \uparrow(C_1 = j) + 2\varepsilon$$

However, M_2 lags M_1 at most 2ε. It follows that

$$(\uparrow(C_2 = j) - \uparrow(C_1 = j)) \leq 2\varepsilon$$

Substituting, we have $\uparrow(C_2 = j) - \uparrow(C_1 = j) \leq t_{arr} - \uparrow(C_1 = j)$

Hence, $\uparrow(C_2 = j) \leq t_{arr}$. This contradicts the Case 1 assumption.

Case 2. Assume that there exists a pair of machines, where a message from M_1 sent during C_1's jth period reaches M_2 during C_2's (j + 1)th period.

Proof of Case 2. As illustrated in Figure 42, to maximize the chance of machine M_2 receiving M_1's jth message at M_2's (j+1)th clock period, C_1 should lag C_2 by the maximum 2ε. The latest message arrival time at machine M_2, t_{arr} is:

$$t_{arr} = \uparrow(C_1 = j) + \max(\mu_{max} + q_{max}, H) + \mu_{max}$$

The starting time of machine M_2's clock period j + 1 is

$$\uparrow(C_2 = j + 1) = \uparrow(C_2 = j) + T$$
$$= (\uparrow(C_1 = j) - 2\varepsilon) + T$$

Since $T > \max(\mu_{max} + q_{max}, H) + \mu_{max} + 2\varepsilon$

$$\uparrow(C_2 = j + 1) > (\uparrow(C1 = j) - 2\varepsilon) + (2\varepsilon + \max(\mu_{max} + q_{max}, H) + \mu_{max})$$
$$= ((\uparrow(C1 = j) + \max(\mu_{max} + q_{max}, H) + \mu_{max})$$

Subtracting, we have $\uparrow(C_2 = j+1) - t_{arr} > 0$ That is, the message arrives at machine M_2 before $\uparrow(C_2 = j + 1)$. This contradicts the assumption of Case 2. By the proofs of Case 1 and Case 2, Fact 3 follows.

Fact 4. Under the PALS protocol, messages from the environment buffered by the Input Synchronizer during clock period j will reach all N destination machines at time t, where $\uparrow(C_r = j + 1) \leq t < \uparrow(C_r = j + 2)$: $1 \leq r \leq N$.

Proof. Similar to the proof of Fact 3.

4.5.2.3 Formalization and Correctness of PALS

A complete formal specification of PALS can be found in [45]. In this paper, PALS is specified using Real-Time Maude as a formal model transformation that maps a synchronous design, together with a set of performance bounds of the underlying infrastructure, to a formal distributed real-time system specification that is semantically equivalent to the synchronous design. This semantic equivalence is proved, showing that the formal verification of temporal logic properties of the distributed system in CTL^* can be reduced to their verification on the much simpler synchronous design.

The PALS formalization presented in [45] includes the following topics:

1. A formal model in rewriting logic [44] of the PALS transformation, expressed in the Real-Time Maude formal specification language [46], including precise requirements about the allowable synchronous designs to which PALS can be applied and about the real-time bounds assumed for the network and clock synchronization infrastructures.

2. A precise derivation of the PALS period based on the formal model, as well as a proof of its optimality, showing that it is shortest possible under the given assumptions about the asynchronous implementation, message format, and network and clock synchronization infrastructures.

3. A bisimulation theorem, showing that the original synchronous design and the so-called stable states of the corresponding PALS asynchronous design constitute bisimilar systems.

4. A mathematical justification of a method that reduces the formal verification of temporal logic properties in CTL* of an asynchronous PALS design – typically infeasible due to state space explosion – to the model checking verification of its much simpler synchronous counterpart.

5. An avionics case study illustrating the usefulness of the PALS pattern for formal verification purposes.

4.5.3 Leader Selection Pattern

The purpose of the Leader Selection pattern is to coordinate a group of nodes so that a single node is agreed upon as the 'leader' at any given time. The nodes typically correspond to replicated computations hosted on distributed computing resources, and are used as part of a fault-tolerance mechanism. If a replicated node fails, this allows a non-failed node to be selected as the one which will interact with the rest of the system.

4.5.3.1 Pattern instantiation

To use the pattern, a group of N nodes (systems or processes) is identified that are to select a leader from among themselves. The leader selection pattern will insert new leader selection threads into each of the systems/processes which are to participate in leader selection. Each thread will have a unique identifier (an integer) to determine its priority in selecting a leader. Connections will be added so that all leader selection threads are able to communicate with each other (N input ports, 1 output port). In addition, each leader selection thread will have an input port from which it determines (from other local systems) if it is failed, and an output port which will say if it is the leader. These two ports are initially left unconnected.

4.5.3.1.1 Pattern arguments

1. Set of nodes to select a leader from
2. Leader priority for each node

4.5.3.1.2 Assumptions

1. The leader selection nodes must communicate synchronously with a one-step delay.
2. At least one node is functional (non-failed) at any given time.

4.5.3.1.3 Guarantees

1. All non-failed nodes shall agree on who is the leader.

2. If a node fails, leadership is transferred to a non-failed node in the next step.
3. If non-failed nodes exist, then in the next step one of them will be the leader.
4. A non-failed leader node shall remain leader as long as no user request to change leadership to a different non-failed node has been made.

4.5.3.2 Verification

The leader selection pattern selects a leader from a group of systems. It can be used to support automatic failover among a set of redundant computing resources (nodes). This failover capability can part of a fault-tolerant design, provided that the node failures are independent. Thus, the leader selection pattern provides a useful building block for constructing fault-tolerant architectures, especially if formal guarantees can be provided for the leader selection algorithm.

4.5.3.2.1 Idealized Leader Selection Pattern Requirements

The main requirements for a leader selection algorithm involve consistency and bounded leader transition times. An idealized leader selection algorithm would have the following properties (requirements):

R1. All non-failed nodes agree on who is the leader

R2. If a node fails, leadership is immediately transferred to a non-failed node

R3. If any non-failed nodes exist, then one of them will be the leader.

There are several additional requirements that can be levied on a leader selection algorithm, depending on the desired level of sophistication. For example:

R4. A non-failed leader node shall remain leader as long as no user request to change to a different non-failed node has been made.

R5. An external actor shall be able to cause leadership to be transferred to a specific node, as long as that node is non-failed.

4.5.3.2.2 Implementable Leader Selection Pattern Requirements

The idealized requirements, as specified, cannot be met in a realistic system because they require delay-free communication between nodes. Therefore, we must relax R1-R3 to allow for communication delays:

R1. All non-failed nodes shall agree on who is the leader within some maximum time delta D_1.

R2. If a node fails, leadership is transferred to a non-failed node within some maximum time delta D_2

R3. If any non-failed nodes exist, one of them will be chosen as the leader within some maximum time delta D_3.

These requirements, given appropriate time bounds D_1, D_2, and D_3, provide requirements that can be used to specify a leader selection pattern on a real system.

Creating leader selection algorithms for asynchronous systems is a complex problem that presents a significant formal analysis challenge. Approaches such as [37][38] have analyzed similar problems but make fairly conservative failure assumptions. We would like to leverage our pattern-based approach to simplify the analysis task and to allow for less conservative failure

assumptions. The PALS pattern provides a basis for allowing us to treat the communication between the nodes as synchronous, allowing a significant simplification of the problem and the complexity of the formal analysis. Therefore, we assume that the nodes in the leader selection problem can communicate synchronously with a single step delay. With the assumption of synchrony the requirements become:

R1. All non-failed nodes shall agree on who is the leader.

R2. If a node fails, leadership is transferred to a non-failed node in the next step.

R3. If non-failed nodes exist, then in the next step one of them will be the leader.

4.5.3.2.3 Creating a Synchronous Implementation in C++

The implementation of the leader selection algorithm, assuming synchronous communication, is straightforward in C++ and is shown in Figure 43. The C code is generic in the sense that it can be generalized to any number of nodes, but it makes several assumptions about the failure model and the communications layer between the nodes.

1. Each node within the leader selection group has a unque ID (stored as IDX in Figure 43). All nodes agree on the IDs of the other nodes.
2. Each node communicates synchronously with all other nodes with a single-step delay. The values of each node's status are communicated to all other nodes through the status array. This array contains the step-delayed status of all other nodes (including the 'self' node).
3. Node failures can only occur in two ways: an internally-detected error (represented by the Boolean 'fail' variable) or an externally detectable error, such as a hardware failure. These external errors are detected by the other nodes through erroneous data values or by the failure to send data during a step.
4. The initial status values (prior to any messages being received) for all other nodes are set to failed (device_ok is FALSE).
5. In the first step after a failure, the 'init' flag will be set to TRUE by the communication infrastructure.

Given these assumptions, the behavior of the node algorithm is defined by five cases labeled by (1) through (5) in Figure 43.

- In case (1) the node has declared itself failed. In this case, it notifies the other nodes by changing its device_ok variable to false. Note that the device_ok status can also be set indirectly by the communication infrastructure in the absence of a message.
- In case (2), we are not re-initializing the node and the previous leader has not failed, so the current leader is unchanged.
- Case (3) handles initializing the node (or reinitializing in case of a failure). The node will attempt to use the current leader from a non-failed node, if any.
- Because of the single-step delay, the reported leader from a non-failed node may have in fact failed in the previous step. In this case, all the non-failed nodes which have already started will use case (4) to choose a new leader in the current step, so this node will do likewise. In case (4), we choose a new node by simply choosing the first non-failed node in sequential order based on node ID.

- If we reach case (5), there is no non-failed node in the previous step. In this case, it is not possible to safely choose a new leader because there is insufficient information about the system state, so we have to choose 'no leader.'

The behavior of the group can be inferred from the nodes in a straightforward way. During stable operation, no nodes fail, and all nodes execute case (2). For initialization, the re-entering nodes will first try to choose a leader that matches the other nodes, if it is possible (i.e., if the 'leader' node has not failed in the previous step). Finally, if a failure has occurred in the current leader in the preceding step, all nodes choose a new leader simply by picking the first non-failed node in the array.

```
const int NO_LEADER = -1 ;
struct status_info {
   bool device_ok ;
   int leader ;
};

class Node {
   // status of all nodes; assumes a PALS clique
   status_info *status;
   int max_elem ;

   // information about this node
   int IDX ;

   Node(int ID, int MAX, status_info *all_status) {
      IDX = ID;
      max_elem = MAX;
      status = all_status ;
      status[IDX].device_ok = false;
      status[IDX].leader = NO_LEADER ;
   }

   void step(bool init, bool failed) {

      // if internally-detected failure is noticed, notify other nodes
(1)   if (failed) {
         status[IDX].device_ok = false;
         return ;
      }
      status[IDX].device_ok = true;

      // if current leader is o.k, we're done.
(2)   if (!init && status[IDX].leader != NO_LEADER &&
            status[status[IDX].leader].device_ok) {
         return;
      }

      //  if reinitializing, check the leader from the
      // first o.k. device (they will all be the same).
      // If this leader is still an o.k. device, then choose it.
(3)   if (init) {
         for (int i = 0; i <= max_elem; i++) {
            if (status[i].device_ok && status[i].leader != NO_LEADER &&
              i != IDX &&
              status[status[i].leader].device_ok) {
                 status[IDX].leader = status[i].leader ;
                 return ;
            }
         }
      }
      // choose a new leader, if one is available, because of leader failure
(4)   for (int i = 0; i <= max_elem; i++) {
         if (status[i].device_ok) {
            status[IDX].leader = i ;
            return ;
         }
      }
      // default; no (safe) leader possible
(5)   status[IDX].leader = NO_LEADER ;
   }
} ;
```

Figure 43 – Synchronous Leader Selection algorithm in C++

4.5.3.2.4 Formalizing Requirements for Leader Selection.

In our implementation, we desire to satisfy requirements R1-R4. We specify these requirements using the Property Specification Language (PSL) [40]. PSL is an IEEE standard that extends the standard temporal logic Linear Temporal Logic (LTL) with several kinds of syntactic sugar to make properties much more straightforward to write and understand.

PSL treats the execution of a program as a sequence of steps. Each step corresponds to one round in which all of the nodes read inputs, compute their next state and output a result. Properties in PSL contain temporal operators that describe allowable paths, where a path is a sequence of steps. The only temporal operators used in the properties below are G and X. The 'G' operator states that the property must be *globally true*. The *X* operator states that the property must be true in the *next step*.

To capture a Boolean system *invariant* that should always be true in every execution of the system, one often writes properties of the following form:

```
PSLSPEC G(<invariant>) ;
```

Another common type of requirements is specifies that in any state where a given *precondition* is true, then in the next state, the *postcondition* is always true. This is captured by the following form:

```
PSLSPEC G(<precondition> → X(<postcondition>)) ;
```

The leader selection requirements are formalized as follows.

R1: All non-failed nodes shall agree on who is the leader.

This can be specified formally in PSL as

```
PSLSPEC
   forall i in {0:MAX_ELEM} : forall j in {0:MAX_ELEM} :
      G((status[i].device_ok & status[j].device_ok) ->
         status[i].leader = status[j].leader);
```

Transliterated, this property reads: for all nodes *i,j*, in all execution steps (G), if both devices are non-failed, then the leader of node *i* is equal to the leader of node *j*.

R2: If a node fails, leadership is transferred to a non-failed node in the next step.

This can be formalized in PSL as

```
PSLSPEC
   forall i in {0:MAX_ELEM} : forall j in {0:MAX_ELEM} :
   G(!status[i].device_ok ->
         X(status[j].device_ok -> status[j].leader != i)) ;
```

This property can be read as follows: for all nodes *i,j*, in all execution steps (G) if a device *i* has failed, then in the next step (X) it shall not be the leader for any non-failed device *j*.

R3: If non-failed nodes exist, then in the next step, one of them will be chosen as the leader.

This can be formalized in PSL as

```
PSLSPEC
   G( (!(forall i in {0:MAX_ELEM} : !status[i].device_ok)) ->
      X(forall j in {0:MAX_ELEM} : (status[j].device_ok ->
         status[j].leader != NO_LEADER)));
```

This property is slightly more difficult to read, as it involves double negation. The first antecedent:

```
!(forall i in {0:MAX_ELEM} : !status[i].device_ok)
```

can be read: not all devices *i* have been declared failed (!status[i].device_ok), so the property as a whole can be read: in all execution steps (G) if not all devices *i* have been declared failed, then in the next step (X), any non-failed process *j* shall have a leader.

R4: A non-failed leader node shall remain leader as long as no user request to change to a different non-failed node has been made.

This can be formalized in PSL as

```
PSLSPEC
   forall i in {0:MAX_ELEM} : forall j in {0:MAX_ELEM} :
      G((status[i].device_ok &
      status[i].leader != NO_LEADER &
      status[i].leader = j &
         status[status[i].leader].device_ok) ->
            X(status[i].device_ok -> status[i].leader = j));
```

The PSL version of this property is more complex than the English equivalent simply because we have to examine each node in the group. The property can be read: for any node *i*, it is globally (G) the case that if:

- the node is non-failed: `status[i].device_ok`
- the node has a leader: `status[i].leader != NO_LEADER`
- the node's leader is equal to *j*: `status[i].leader = j`
- the node's leader is o.k.: `status[status[i].leader].device_ok`

then in the next state (X), if the node *i* is still non-failed, it shall have the same leader.

Given the C program in Figure 43, it is possible to create an equivalent model in a model-based development notation such as Simulink [39] or a formal analysis tool (such as NuSMV [43]) and prove that requirements R1-R4 are met for an architecture consisting of *N* nodes.

In the next section, we re-implement the node program in NuSMV and use it to check that a synchronous architecture consisting of *N* nodes for $N = 2..10$ meets the requirements of the leader selection pattern.

4.5.3.2.5 Modeling and Checking Requirements in NuSMV

We are interested in formally establishing the properties that we formalized in the previous section. While model checking approaches that directly check programming languages have advanced significantly over the last decade [42] it is more straightforward to reason about models of program behavior, even if those models are functionally equivalent to the program.

For our analysis, we re-implement the C program from Figure 43 in NuSMV and use it to demonstrate that the architecture that we have created is sound.

The description of a node in NuSMV (called a *device*) is shown in Figure 44. NuSMV doesn't have a notion of looping, so the algorithm becomes somewhat more cumbersome to write, and less general. We have to enumerate every element of the array separately, and of course we have to fix the number of elements in the clique. In Figure 44 we are creating a node for a group of four nodes.

NuSMV does not have a particularly rich type system so we create two arrays to represent the 'status' array from Figure 43, one each for *device_ok* and *leader*. There is no way to 'halt' execution of a NuSMV process in the sense of a hardware failure, so instead we allow the device_ok variable to be non-deterministically assigned. Other than these changes, the cases within the NuSMV code matche the structure of the code in the C program.

```
MODULE CONSTS
DEFINE
   NO_LEADER := -1 ;

MODULE device(ARRAY_MAX, IDX, device_ok, leader)
VAR Constants: CONSTS
DEFINE
   NO_LEADER := -1 ;
   other_leader :=
   case
      device_ok[0] & IDX != 0 : leader[0] ;
      device_ok[1] & IDX != 1 : leader[1] ;
      device_ok[2] & IDX != 2 : leader[2] ;
      device_ok[3] & IDX != 3 : leader[3] ;
      TRUE : Constants.NO_LEADER ;
   esac ;

   other_leader_ok :=
   other_leader = NO_LEADER ? FALSE : device_ok[other_leader] ;

ASSIGN
  -- device_ok[IDX] is allowed to float; it can be true or false.

   init(leader[IDX]) := Constants.NO_LEADER;
   next(leader[IDX]) :=
      case
      -- device failed
      !next(device_ok[IDX]) : Constants.NO_LEADER ;

      -- not re-initializing and previous leader still o.k.
      device_ok[leader[IDX]] & leader[IDX] != Constants.NO_LEADER :
            leader[IDX] ;

      -- initializing...need to sync up with rest of 'good' clique, as
      -- long as 'leader' node is still o.k.
      next(device_ok[IDX]) & !device_ok[IDX] & other_leader_ok :
            other_leader ;

      -- choose first 'good' leader.
      device_ok[0] : 0 ;
      device_ok[1] : 1 ;
      device_ok[2] : 2 ;
      device_ok[3] : 3 ;

      -- no leader available
      TRUE : Constants.NO_LEADER ;
      esac;
```

Figure 44 – NuSMV implementation of Node (called device)

In order to tie the nodes together and check the properties, we create a main program as shown in Figure 45. We assume that the nodes communicate with one another via a one-step delay. For the purposes of analysis, it is sufficient to connect all of the nodes together through shared arrays (*device_ok* and *leader* in Figure 45). In NuSMV, one expresses the property 'inside' the module that is to be checked, so we add the PSL properties directly into the model.

```
MODULE main

DEFINE
   ARRAY_MAX := 3 ;

VAR
   Constants : CONSTS ;
   device_ok : array 0..ARRAY_MAX of boolean;
   leader : array 0..ARRAY_MAX of Constants.NO_LEADER .. ARRAY_MAX ;
   device0 : device(ARRAY_MAX, 0, device_ok, leader);
   device1 : device(ARRAY_MAX, 1, device_ok, leader);
   device2 : device(ARRAY_MAX, 2, device_ok, leader);
   device3 : device(ARRAY_MAX, 3, device_ok, leader);

-- Note: use of named constants for quantifiers not yet supported
-- in NuSMV for PSL properties

-- devices agree on a leader
PSLSPEC
   forall i in {0:3} : forall j in {0:3} :
     G((device_ok[i] & device_ok[j]) -> leader[i] = leader[j]);

-- if a node was leader and has not failed, it is still leader.
PSLSPEC
   forall i in {0:3} : forall j in {0:3} :
      G((device_ok[i] &
                leader[i] != NO_LEADER &
                leader[i] = j
         device_ok[leader[i]]) ->
           X(device_ok[i] -> leader[i] = j));

-- there exists a non-failed node in the previous step,
-- and node 'j' not failed, then the leader should not be -1.
PSLSPEC
   G( (!(forall i in {0:3} : !device_ok[i])) ->
      X(forall j in {0:3} : (device_ok[j] -> leader[j] != -1)));

-- If a node is declared failed, it
-- shall not be leader in the next step.
PSLSPEC
   forall i in {0:3} : forall j in {0:3} :
      G(!device_ok[j] -> X(leader[i] != j)) ;
```

Figure 45 – Main NuSMV Module

We can now check the requirements of the NuSMV model. The time required to check the properties grows geometrically with the number of nodes in the group; however it is still possible to check systems with less than ten nodes in a reasonably short amount of time. Table 6 shows the time required to perform the analysis. The analysis was performed with the following command line:

```
nusmv –dynamic leader-elect-new-k.smv
```

where *k* is the size of the group to be analyzed. The analysis was performed on a Core 2 duo laptop running at 3.06 GHz with 4GB RAM using NuSMV 2.5.2 and Windows 7.

Table 6 – Analysis times to check requirements R1–R4 on group of *N* nodes

Number of Nodes	Analysis Time
2	0.12s
3	0.12s
4	0.34s
5	0.63s
6	2.76s
7	20.29s
8	2m 39s
9	6m 9s
10	> 3 hours

Our implementation of leader selection is quite straightforward, and allows fairly strong correctness guarantees to be proved quickly. The reason that we are able to use a straightforward algorithm is due to the assumption of synchrony that is provided by the PALS pattern. This demonstrates the power of creating complex system architectures using relatively simple pieces through the composition of formally verifiable patterns.

4.5.4 Replication Pattern

The purpose of the Replication pattern is to create identical copies of portions of the system. This is typically used to implement fault tolerance by assigning the copies to execute on separate hardware platforms with independent failure modes.

4.5.4.1 Pattern Instantiation

To use the pattern, one or more nodes (systems) are selected and the number of copies to create is specified. Optional arguments determine how ports and their connections are handled in the replication process. Each new system and port created is given a unique name. When multiple outputs are created they may be merged by the addition of a new system block to select, average, or vote the outputs. This is accomplished using the Fusion pattern.

4.5.4.1.1 Pattern arguments

1. Set of system components to replicate
2. N, the desired number of copies, including the original component(s)
3. For each new output port in the set of replicas, choose:
   a. replicate the destination of the data connection (default)
   b. add a new system block (Select, Vote, or Average) to merge the new outputs and connect to the destination of the original data connection
4. For each new input port in the set of replicas, choose:
   a. replicate source of the data connection (default)
   b. fan out the data connection from the source of the original input data connection

4.5.4.1.2 Assumptions

Replicated systems will be hosted on platform hardware with independent failure modes. This can be specified through use of the AADL property `Not_Collocated`.

4.5.4.1.3 Guarantees

If hardware failures are independent, and there are less than N failures, then at least one replica will be functional at all times. Guarantees associated with the use of various voting algorithms will be addressed separately, as part of the definition of voting design patterns.

4.5.4.2 Verification

When the replicas are bound to hardware, the `Not_Collocated` property can be checked easily by examination of the model structure. The guaranteed behavior for the replication pattern follows directly.

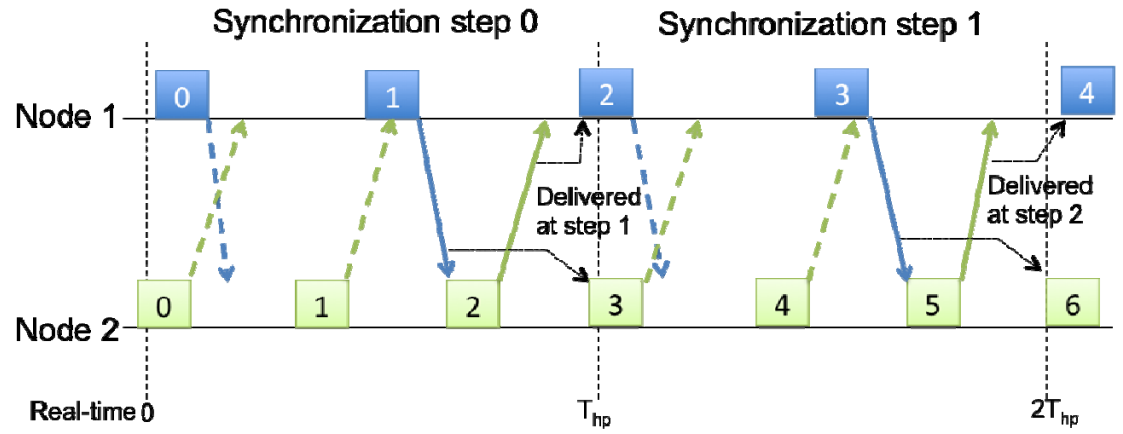
4.5.5 Multi-Rate PALS Pattern

The multi-rate PALS pattern is designed to support the synchronization pattern between multi-rate distributed computations in hierarchical control systems. The pattern guarantees that the design and verification complexities for distributed synchronization in the coordination of multi-rate systems remains tractable and is equivalent to the corresponding single-rate synchronous system.

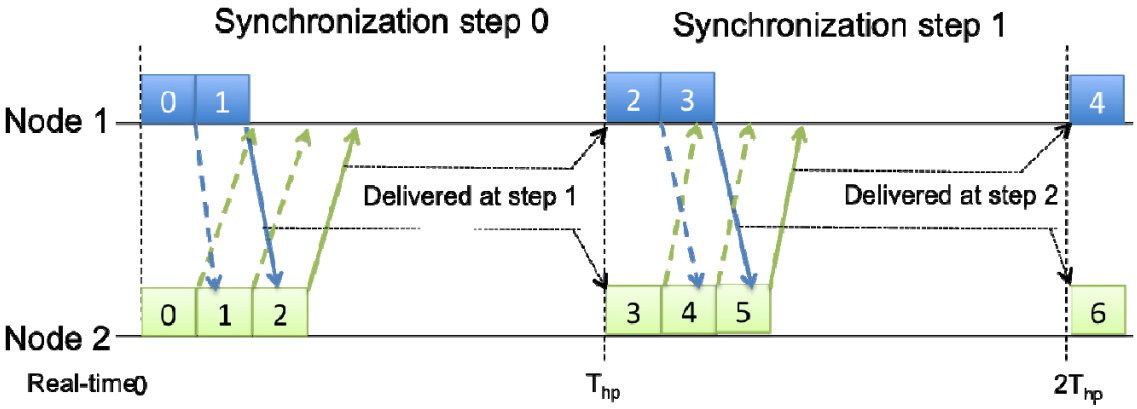
The period of the corresponding synchronous computations is fixed and equal to the hyper-period, which is the least common multiple (LCM) of the periods of the multi-rate computations. This is based on the fact that the hyper-period is perfectly divisible by the periods; thus synchronized state transitions always occur at the hyper-period boundary of the equivalent synchronous system. Other intermediate state transitions of the multi-rate systems between two consecutive hyper-period boundaries can be easily abstracted in the synchronous system. We also note that any other synchronization period that is not equal to an integer multiple of the hyper-period either creates more complex system in terms of verification or is infeasible to produce equivalent fixed periodic synchronous executions.

4.5.5.1 Basic Concept of the Multi-Rate PALS Pattern

The underlying intuition of the multi-rate PALS design pattern is illustrated in Figure 46. A group of multi-rate systems (Figure 46a) is equivalent a single-rate synchronous system (Figure 46b). The pattern abstracts away the intermediate node and event interleavings in the multi-rate asynchronous system so that a mapping with the synchronous system can be achieved. In each synchronization interval T_{sync}, which is equal to the LCM of the periods, there are $n_i = T_{sync}/T_i$ executions of $Node_i$ (period= T_i) in the multi-rate system. In the synchronous design, these n_i executions are sequentially executed. This could be achieved by invoking the thread function in a loop n_i times in the synchronous design. Based on the assumptions on T_i and output hold time in the asynchronous system, the pattern guarantees that the destination node receives n_i outputs from $Node_i$ in each synchronization interval T_{sync}. The multi-rate synchronizer at each node in the asynchronous system (inserted by the pattern) then deterministically chooses the output to be delivered to the computation node. In the currently proposed design, it chooses the last of the n_i outputs, which is also the output of the combined execution in the equivalent synchronous system. Based on this equivalence, the synchronous design greatly simplifies design and verification of these multi-rate systems.



a. Asynchronous multi-rate system



b. Synchronous abstraction

Figure 46 – Multi-rate PALS design pattern

4.5.5.2 Pattern instantiation

To use the pattern, a group of *n* nodes (systems) is selected that are to execute at different periods T_i, i = 1…N. In order to guarantee virtual synchronization, the pattern adds a synchronization interface, called the *multi-rate synchronizer,* at each node. The synchronizer executes at a period equal to T = LCM(T_1, …, T_n). The pattern guarantees that other nodes in the group receive the outputs of a node at approximately the same global time.

The type of the inserted synchronizer component (system, process, or thread) is determined by the user. The pattern assumes it will be collocated with the receiver node, i.e. it executes on the same processor. If it is an AADL thread component, the pattern then adds an AADL thread subcomponent for the synchronizer in the same process as the synchronization logic, i.e. the PALS computation thread, of each node. It propagates the message through an AADL *immediate* connection. The pattern also defines the communication and scheduling characteristics between the synchronizer and the receiving computation logic.

4.5.5.2.1 Arguments

1. A group of n nodes. In AADL, this corresponds to a set of AADL thread elements distributed across defined process elements.
2. Period of each computation, T_i
3. Name of the multi-rate PALS group.
4. Set of (output port, input port) pairs used in the multi-rate synchronization.

4.5.5.2.2 Assumptions

Assumptions (1-5) of the multi-rate PALS pattern are similar to those of the original PALS pattern. The rest are specific to the multi-rate synchronizer component.

1. *Bounded Local Clock Error* - Each node *j* has access to an approximation of the true global time t via a local clock c_j, where the maximum error (called either jitter or skew) of each local clock is ε, i.e.,

   $| C_j - t | < \varepsilon$.

2. *Monotonic Local Clocks* - The value of each local clock C_j is monotonically increasing. Each node may adjust its local clock rate, but it may never decrease the value of its local clock.
3. *Bounded Computation Time* – The computation of a node's new local state and outputs completes within a specified time. Typically this is the periodic scheduling deadline for a thread α^i_{max}.
4. *Bounded Message Delivery* – Messages are reliably delivered to their destinations with latency μ, where $\mu_{min} \leq \mu \leq \mu_{max}$. Depending on the system fault assumptions, this may require thus use of a fault-tolerant network.
5. *Node Fault Assumptions* – A failed node must not be able to send extra messages (more than one) during a PALS period. This could result in nodes receiving different messages, even though the network delivered each correctly. By default, we assume that the output of a failed node is '*null.*'
6. *Collocated Multi-Rate Synchronizer* – We assume that multi-rate synchronizer is executed on the same processor as the synchronized computation logic so that they have identical hardware failure characteristics.
7. *Output Assumptions of Multi-Rate Synchronizer* – Currently the pattern assumes that in each step, multi-rate synchronizer only propagates the last message it received during its previous period. (This assumption can be relaxed so that the synchronizer provides a vector of received messages.). Based on the node fault assumption, if the sender fails, the synchronizer propagates a 'null' message.
8. *First Execution of Multi-Rate Synchronizer* –The first dispatch time of both multi-rate synchronizer and the receiving computation logic are same.

Similar to the original PALS pattern, the computation logic at each node must also satisfy the following constraints relating the system parameters of the associated computation logic *i*:

1. *Causality constraint* – Messages cannot be sent too early.

   $\text{Min}(\text{Output time}_i) + \text{Min}(\text{Output time}_{sync}) \geq 2\varepsilon - \mu_{min}$

2. *Computation period constraint* – Messages cannot be sent too late.

   $\text{Max}(\text{Output time}_i) \leq T_i - \mu_{max} - 2\varepsilon$

Additionally, the following constraints on the period of the multi-rate synchronizer must be verified.

3. Period of multi-rate synchronizer = LCM(T1, …, Tn); LCM = Least common multiple.
4. Priority of multi-rate synchronizer > Priority of the receiving computation logic.

4.5.5.2.3 Guarantees

1. Synchronization between multi-rate computations happens only at the hyper-period boundary. This hyper-period, denoted by T_{hp}, is equal to the LCM (Least common multiple) of the computation logics.
   a. The source node sends its messages to the multi-rate synchronizers at the receiving nodes. The synchronizers execute at a period equal to $T_{sync} = T_{hp}$.
   b. In each period, a multi-rate synchronizer receives $n_i = T_{sync}/T_i$ number of inputs from a source node with period T_i.
   c. Of the n_i messages received during the multi-rate synchronizer period j, the synchronizer only propagates the last message and makes it available to the receiving computation logic during its period j+1.

   Suppose that A (Period T_A) sends messages to B (period T_B). Then, there are $n_A = T_{sync}/T_A$ and $n_B = T_{sync}/T_B$ executions of A and B in an interval T_{sync}. With the multi-rate synchronizer, these nodes interact only at every T_{sync} interval such that at the multi-rate synchronization period j,

   $A.in(j.n_A+k) = B.out((j.n_B -1); k = 0…n_A-1$

   The pattern guarantees that the interaction of these nodes is equivalent to a group of perfectly synchronized nodes executing at period T_{sync}. If the corresponding equivalent nodes of A and B are A' and B', then

   $A'.in(j) = B'.out(j-1)$

2. Similar to the original PALS pattern, external inputs are passed through environment input synchronizer. If the external inputs are sent to multiple computations with different periods, then the environment input synchronizer operates at the period equal to the LCM of their periods.

4.5.5.3 Verification

Figure 47 shows a timeline of the computation and communication associated with a node. Logical synchronization step j is assumed to begin at time $t_{j,0}$. Dispatch of j^{th} execution of $Node_i$ multi-rate synchronizer and $j.n_i$ th executions of $Node_i$ computation logic coincide at this time. These can be denoted by $\uparrow(C_{i,sync} = j) = t_{j,0}$ and $\uparrow(C_i = j.n_i) = t_{j,0}$ respectively. Since each local clock may exhibit maximum error ε, computation during these executions may actually dispatch between the true time $[j.T_{hp} – \varepsilon, j.T_{hp} + \varepsilon)$. The other n_i-1 executions of the computation logic subsequently dispatch at $\uparrow(C_i = j.n_i+k) = t_{j,k}$; k=1..$n_i$-1. In true time, these executions occur between $[j.T_{hp}+ k.T_i – \varepsilon, j.T_{hp}+ k.T_i + \varepsilon)$.

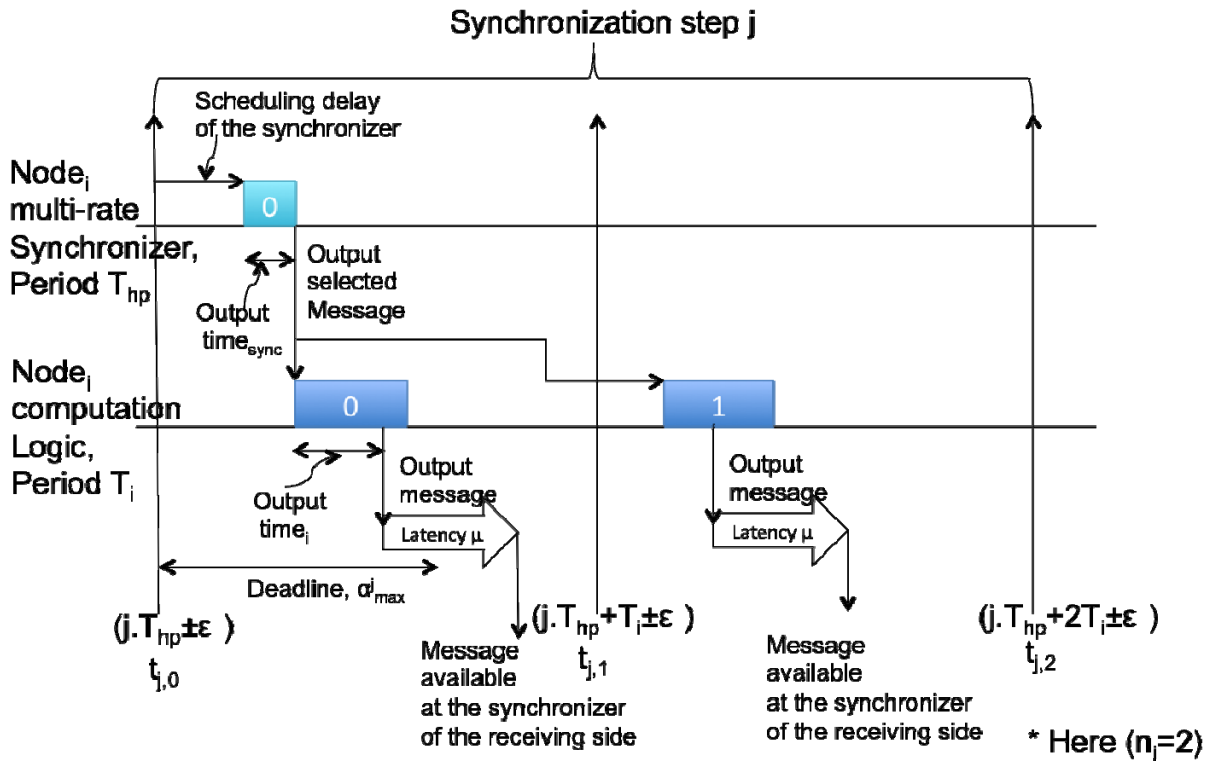


Figure 47 – Multi-rate PALS timeline

The deadline of the computation logic is given by α^i_{max}; thus it must be scheduled to complete its execution by $t_{j,k} + \alpha^i_{max}$; for k=0.. n_i-1.

Additionally, the output message is delivered at least after an interval of H = Output time$_{sync}$ + Output time$_i$. Output of this thread is propagated to the multi-rate synchronizer at the receiving side after a latency of μ. A message generated by the execution at $t_{j,k}$ is expected to arrive at least after $t_{j,k} + H + \mu$.

4.5.5.3.1 Multi-Rate PALS Timing Constraints

Similar to the PALS pattern, the multi-rate PALS pattern assumes two timing restrictions on the clock period (T_i) and output time (H) of the computation logic at each node. These constraints must be met to satisfy the requirement that messages generated during the logical synchronization step j are consumed by their destination nodes in synchronization step j+1.

Multi-rate PALS Causality Constraint – The first message generated in a synchronization interval by the execution at $t_{j,0}$ is expected to arrive at least after $t_{j,0} + H + \mu$. In true time, this can happen as early as $j.T_{hp} + H + \mu - \varepsilon$ in true time. Given the minimum latency, μ_{min}, the earliest message arrival time in a synchronization step j is $j.T_{hp} + H + \mu_{min} - \varepsilon$.

At the receiving node, the multi-rate synchronizer can be dispatched as late as $j.T_{hp} + \varepsilon$ during the synchronization step j. Thus, in order to guarantee that this message is not received before this time,

$$j.T_{hp} + H + \mu_{min} - \varepsilon \geq j.T_{hp} + \varepsilon$$

which means that the earliest output time must satisfy

Equation 1: $H \geq 2\varepsilon - \mu_{min}$

The only difference to the PALS pattern is that in case of multi-rate PALS pattern, H comprises of the output time of both multi-rate synchronizer and the computation logic.

Multi-rate Computation Period Constraint – The n_i^{th} execution, i.e. the last execution, of a node $Node_i$ in the synchronization step j occurs at $t_{j,ni-1}$ or in true time between $j.T_{hp} + (n_i-1).T_i \pm \varepsilon$.

The logical period in the consuming node for step j+1 may begin as early as $(j+1).T_{hp} - \varepsilon = (j+1).n_i.T_i - \varepsilon$

The restriction on period (T_i) and deadline (α^i_{max}) depends on the requirement that the output of last execution also has to arrive before the synchronization step j+1 of the consuming node. The latest output arrival time is given by $j.T_{hp} + (n_i-1).T_i + \varepsilon + \alpha^i_{max} + \mu_{max}$. Therefore,

$$j.T_{hp} + (n_i-1).T_i + \varepsilon + \alpha^i_{max} + \mu_{max} \geq (j+1).T_{hp} - \varepsilon$$

Since $T_{hp} = n_i.T_i$, this simplifies into

Equation 2: $T_i \geq 2\varepsilon + \alpha^i_{max} + \mu_{max}$ or,

$\alpha^i_{max} \leq T_i - 2\varepsilon - \mu_{max}$

4.5.5.3.2 Summary of Formal Proof

The proof of the pattern's synchronization guarantee is divided into following parts:

1. We prove that based on the constraints on the periods and output timing assumptions of each computation, the periodic computation logic at each node sends a *fixed* number of identical outputs in each synchronization interval that are to be consumed in the next synchronization step (Lemma 1).
2. Next, we prove that the equivalence between the multi-rate asynchronous system and the corresponding synchronous system. (Theorem 1). This proof is done in two steps: First we prove that a multi-rate asynchronous system is equivalent to a single rate PALS system (Lemma 2). Then, based on the properties of the PALS pattern, the multi-rate asynchronous system is equivalent to a single-rate synchronous system.

We use a state machine model to define both multi-rate asynchronous system (along with multi-rate synchronizers) and synchronous system and prove the logical equivalence between multi-rate asynchronous system and the corresponding synchronous system

Definition 1 (State machine model of a multi-rate system): The multi-rate system consists of N distributed state machines $M_1,\ldots,M_N$. The state transitions of these machines are periodically driven by the local clock at a period T_i for i=1…N. At each node i, there is a multi-rate synchronizer $M_{i,sync}$ driven the same local clock but at a period $T_{sync} = T_{hp}$. The timing bounds on clock skew, deadline, and period are based on the pattern assumption.

The state transition function of M_i is defined by the functions,

$next\text{-}state_i$: $input \times state \rightarrow state$, and

$next\text{-}output_i$: $input \times state \rightarrow output$.

Similarly, the next-state and next-output functions of the multi-rate synchronizer $M_{i,sync}$ are given by

next-state$_{i,sync}$: input × *state* → *state*, and

next-output$_{i,sync}$: input × *state* → *output*

Additionally, the system consists of an environment input synchronizer M_{in_syn}, and an environment output synchronizer, M_{out_syn} to synchronize the interaction with external environment at period T_{hp}. Note that the environment input/output synchronizers also satisfy the same timing constraints of Equation 1 and Equation 2.

Definition 2 (State machine model of a synchronous system): The synchronous system consists of N distributed state machines $M'_1,\ldots,M'_N$. The state transitions of these machines are periodically driven by the perfectly synchronized local clock at a period T_{hp}. The state transition function is defined by the functions,

next-state$_i$': input × *state* → *state*, and

next-output$_i$': input × *state* → *output*.

Additionally, the system consists of an environment input synchronizer M_{in_syn}, and an environment output synchronizer, M_{out_syn} to synchronize the interaction with external environment at period T_{hp}.

In our proposed design, for each node i, the relations between next-state$_i$, next-state$_i$', next-output$_i$, and next-output$_i$' are as follows:

next-state$_i$'(in, s) = next-state$_i$ (in, s'), and

next-output$_i$'(in, s) = next-output$_i$ (in, s')

where s' = next-state$_i$ (in, …, next-state$_i$ (in, s)) with next-state$_i$ function being applied n_i-1 times ($n_i=T_{hp}/T_i$).

Lemma 1: In the multi-rate asynchronous system, when a state machine M_s sends its messages to multi-rate synchronizers $M_{r,sync}$ of receiving state machine M_r, these synchronization interfaces receive exactly $n_s = T_{hp}/T_s$ messages in each synchronization step j if the multi-rate timing constraints $\alpha^i_{max} \leq T_i - 2\varepsilon - \mu_{max}$ (Equation 2) and output time, $H \geq 2\varepsilon - \mu_{min}$ are satisfied.

Proof: There are n_s executions of M_s in each synchronization step j. There are two possible cases: n_s=1 and n_s>1. Note that the case of n_s=1 is similar to the PALS pattern. Thus, it is trivially proven from the timing constraints.

We prove for the case of n_s>1.

Based on the assumption $H \geq 2\varepsilon - \mu_{min}$, the 1st output of the M_s at each synchronization step j arrives after the latest dispatch of the receiving synchronizer $M_{r,sync}$ at synchronization step j. Similarly, the restriction on the deadline of the computation logics enforces that the n_s^{th} output is received before the state transition of $M_{r,sync}$ at $t_{j+1,0}$. The lemma immediate follows since the outputs of the remaining (n_i-2) executions are also in FIFO order between the 1st and n_i^{th} outputs.

Fact: Since the receiving multi-rate synchronizers receive the same set of outputs at synchronization step and the logic of the synchronizers are same at the receiving nodes, the receiving computation logic reads identical inputs in the multi-rate systems during a synchronization step.

Design of Multi-Rate Synchronizer

Multi-rate synchronizer can have different logic based on the user requirement or selection criteria. One possible design for the multi-rate synchronizer is based on selecting the last received input, i.e. n_s^{th} output from M_s in each step. By Lemma 1, the multi-rate synchronizer, $M_{r,sync}$, of the receiving state machine M_r, receives a fixed number of $n_s = T_{hp}/T_s$ inputs from the sending state machine M_s. Since only the last received input is propagated, the next-output of $M_{r,sync}$ in synchronization step j is given by

next-output$_{r,sync}$((in_s^1, in_s^2, …, in_s^{ns}), s) = in_s^{ns}; where (in_s^1, in_s^2, …, in_s^{ns}) are the received input in synchronization step j-1.

For this design of multi-rate synchronizer, it has a single state and remains unchanged during its execution.

Definition 3 (Synchronous composition of multi-rate synchronizer, $M_{i,sync}$ and M_i):

M_i and $M_{i,sync}$ are collocated and driven by the same physical clock. Since there is a fixed execution order of these state machines in a synchronization step, the state transition of $M_{i,sync}$ and n_i state transitions of M_i can be composed as a single state transition of an equivalent state machine M^c_i which executes at a period of T_{hp} using the same clock. The next-state and next-output function of M^c_i are given as follows:

- next-state^c_i((in_s^1, in_s^2, …, in_s^{ns}), s) = next-state$_i$(in_s^{ns}, s'),

where s' = next-state$_i$ (in_s^{ns}, …, next-state$_i$ (in_s^{ns}, s)) with next-state$_i$ being applied n_i-1 times. (in_s^1, in_s^2, …, in_s^{ns}) is the vector of input received from the sending state machine M_s. From this vector, only the last element in_s^{ns} is used in the state transition.

- next-output^c_i((in_s^1, in_s^2, …, in_s^{ns}), s) = (out_i^1, out_i^2, …, out_i^{ni})

where out_i^1, out_i^2, …, out_i^{ni} are intermediate outputs produced during the execution of M^c_i and are equal to the outputs of n_i executions of M_i.

Here, the deadline of M^c_i is given by the deadline of the n_i^{th} execution of M_i, which is equal to $(n_i\text{-}1).T_i + \alpha^i_{max}$. Output time is given by the sum of the output time of M_i and $M_{i,sync}$. Similar to the multi-rate system, the state machines' local clock skews and communication delays are bounded by ε and μ_{max} respectively.

Lemma 2: A multi-rate asynchronous system defined in Definition 1 is equivalent to a single rate PALS system of PALS period equal to T_{hp} which consist of n state machines M^c_i, an environment input synchronizer, M_{in_syn}, and an environment output synchronizer, M_{out_syn}. In this case, M^c_i is the synchronous composition of $M_{i,sync}$ and M_i as defined in Definition 3. M_{in_syn} and M_{out_syn}, are the same environment input, output synchronizers used in the multi-rate system.

Proof: It is sufficient to prove that the system consisting of M^c_i**,** M_{in_syn} and M_{out_syn} (i=1…N) satisfies the constraints of a PALS system.

In these state machines, the minimum output time trivially satisfies the PALS causality constraint based on the assumption given in Equation 1.

Similarly, the deadlines of these state machines also satisfy the PALS period constraint. The deadline of M^c_i is given by $(n_i\text{-}1).T_i + \alpha^i_{max}$. Since $\alpha^i_{max} \leq T_i - 2\varepsilon - \mu_{max}$,

Deadline of $M^c_i = (n_i\text{-}1).T_i + \alpha^i_{max} \leq n_i.T_i - 2\varepsilon - \mu_{max} \leq T_{hp} - 2\varepsilon - \mu_{max}$

Similarly, environment input/output synchronizers also satisfy these constraints and given identical environment inputs are received during a synchronization step, the multi-rate system is equivalent to the single-rate PALS system. □

Theorem 1: A multi-rate asynchronous system defined in Definition 1 is equivalent to a single rate perfectly synchronized system consisting of N state machines $M^c_1,\ldots, M^c_N$, an environment input synchronizer, M_{in_syn} and an environment output synchronizer M_{out_syn} where each state machine is driven by perfectly synchronized clocks at a period of T_{hp}.

Proof: The proof this theorem immediately follows from Lemma 2 and the guarantee provided by the PALS pattern. The theorem completes the proof that the multi-rate PALS pattern guarantees the virtual synchrony. □

For the verification purpose, we can further simplify the synchronous design in which instead of propagating the vector of ouputs in each synchronization step, each state machine sends only the output of last element of the vector to the receiving state machines. Thus we can reduce the synchronous system into the synchronous sytem defined in Definion 2, such that

- $\text{next-state}'_i(in_s^{ns}, s) = \text{next-state}_i(in_s^{ns}, s')$, and
- $\text{next-output}'_i(in_s^{ns}, s) = \text{next-output}_i(in_s^{ns}, s')$,

where $s' = \text{next-state}_i\ (in_s^{ns}, \ldots, \text{next-state}_i\ (in_s^{ns}, s))$ with next-state_i being applied n_i-1 times.

4.6 System Verification

This section describes the compositional verification of system level properties. Our idea is to partition the formal analysis of a complex system architecture into a series of verification tasks that correspond to the decomposition of the architecture. By partitioning the verification effort into proofs about each subsystem within the architecture, the analysis will scale to handle large system designs. Additionally, the approach naturally supports an architecture-based notion of requirements refinement: the properties of subcomponents necessary to prove a system-level property in effect define the requirements for those subcomponents. We call the tool that we have created for managing these proof obligations *AGREE*: Assume Guarantee Reasoning Environment.

There were two goals in creating this verification approach. The first goal was to reuse the verification already performed on the components and design patterns. The idea is that much of the complexity of the system architecture is contained in the proofs of the design patterns; if these facts must be re-proved on each system architectural instance, the approach will not be able to scale to real systems. Additionally, we have developed sophisticated tools for analyzing component-level models, and we want to be able to bring these component-level results to bear on the system architectural analysis.

The second goal was to enable distributed development by establishing the formal requirements of subcomponents that are used to assemble a system architecture. If we are able to establish a system property of interest using the contracts of its subcomponents, then we in fact have a means for performing *virtual integration* of components. We can use the contracts of each of the subcomponents as a specification for a subcontractor and have a great deal of confidence that if the subcontractor meets the specification, the integrated system will work properly.

Our approach is based on compositional assume-guarantee reasoning [53]. That is, for each component within the architecture, we define a set of *assumptions* that the component expects to always be true of the external environment and a set of *guarantees* that the component will always satisfy. In other words, the *guarantees* are the component functional requirements, and the *assumptions* define the context in which we can use the component. Together, the assumptions and guarantees for a component form a *contract* for that component. If a component is composite (that is, defined by other subcomponents), then we use the assumptions and guarantees of the subcomponents, as well as *facts* known about the architecture by the instantiation of patterns, to prove that the component satisfies its guarantees. To specify contracts we use the PSL notation [19], an IEEE standard that is widely used in hardware verification. Our initial framework assumes that components communicate *synchronously* with a single-step delay between each component.

4.6.1 An Example: Transient Response

In this section we describe our compositional verification approach using the avionics system example model.

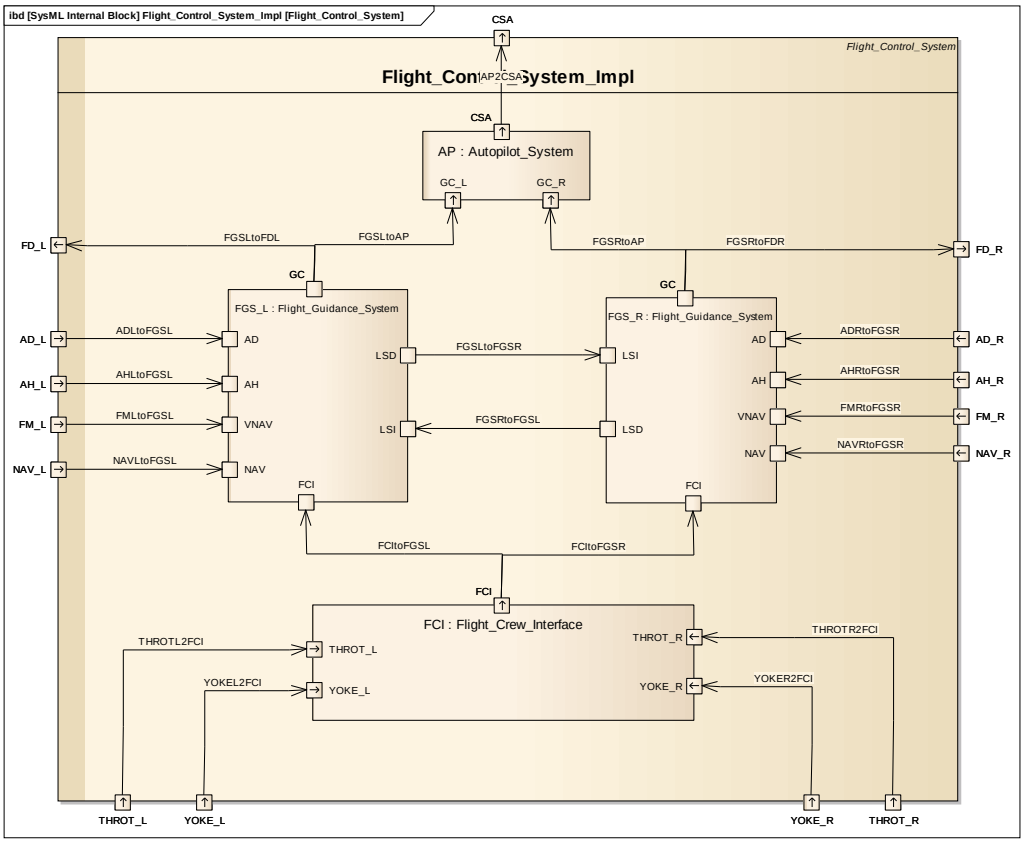


Figure 48 – Final FCS System Architecture

One of the typical requirements levied on a flight control system has to do with transients in the actuator commands. For passenger comfort and safety, a limit is placed on the forces that would be experienced by the passengers during normal operation. For example, the automation should not command a sharp change in the pitch of the aircraft, even in the presence of component failures.

In our system architecture, this property becomes a constraint on the control surface actuator (CSA) output of the system. We would like the commanded pitch to be bounded both in terms of the both the actuator angle and its rate of change. In our notation, we can write these properties as follows:

```
transient_response_1 : assert
  true -> abs(CSA.CSA_Pitch_Delta) < CSA_MAX_PITCH_DELTA ;
transient_response_2 : assert
  true -> abs(CSA.CSA_Pitch_Delta - prev(CSA.CSA_Pitch_Delta, 0.0))
     < CSA_MAX_PITCH_DELTA_STEP ;
```

The "`true` **->**" portion of each property states the property is initially true. The remainder of the first property states that the absolute value of the commanded pitch (CSA_Pitch_Delta) is less than some constant (CSA_MAX_PITCH_DELTA). The second property is similar, but states that the difference between the current pitch and the previously commanded pitch is less than some constant (CSA_MAX_PITCH_DELTA_STEP).

We will use these properties to demonstrate the capabilities of our assume guarantee framework. In order to prove these properties, we will have to define assumptions about the environment in which the system operates and use facts that we have previously proven about our architectural patterns.

4.6.2 Contract Syntax

The syntax for contracts is derived from the Property Specification Language (PSL), specialized for use with the Lustre language [52]. PSL defines a family of languages each based on some host notation, such as Verilog, SystemVerilog, RTL, and VHDL. In the PSL documentation, these are called "flavors" of the language. The flavors define a syntax for local definitions such as constants and variables and the set of basic expressions that can be manipulated in temporal expressions. We have created a Lustre "flavor" for PSL since it is the input language of the Kind model checking tool, and it has a convenient syntax for creating local definitions.

The contract for the Flight Control System is shown in Figure 49. The syntax for contracts is somewhat rich, and allows for local definitions of functions, variables, and constants (1) to simplify writing properties. *Properties* (2) are reusable fragments of temporal logic, similar to macros. It is possible to parameterize properties (analogous to Boolean functions), though we do not do so in this example. The component assumptions (3) describe constraints that are expected to hold on the external environment and assertions (4) describe the guarantees that will be provided by the component.

In this contract we have defined some constants to specify the limits on pitch transients both for inputs and outputs. Then we define a set of properties that we will use to specify the system assumptions.

```
fun abs(x: real) : real = if (x > 0) then x else -x ;                          (1)
const ADS_MAX_PITCH_DELTA: real = 3.0 ;
const FCS_MAX_PITCH_SIDE_DELTA: real = 2.0 ;
const CSA_MAX_PITCH_DELTA: real = 5.0 ;
const CSA_MAX_PITCH_DELTA_STEP: real = 5.0 ;

property AD_L_Pitch_Step_Delta_Valid =                                         (2)
  true ->
    abs(AD_L.pitch.val - prev(AD_L.pitch.val, 0.0)) <
    ADS_MAX_PITCH_DELTA ;
property AD_R_Pitch_Step_Delta_Valid =
  true ->
    abs(AD_R.pitch.val - prev(AD_R.pitch.val, 0.0)) <
    ADS_MAX_PITCH_DELTA ;
property Pitch_lr_ok =
  abs(AD_L.pitch.val - AD_R.pitch.val) < FCS_MAX_PITCH_SIDE_DELTA ;
property some_fgs_active =
  (FD_L.mds.active or FD_R.mds.active) ;

active_assumption: assume some_fgs_active ;                                    (3)

transient_assumption :
  assume AD_L_Pitch_Step_Delta_Valid and
         AD_R_Pitch_Step_Delta_Valid and Pitch_lr_ok ;                         (4)
transient_response_1 :
  assert true -> abs(CSA.CSA_Pitch_Delta) < CSA_MAX_PITCH_DELTA ;
transient_response_2 :
  assert true ->
      abs(CSA.CSA_Pitch_Delta - prev(CSA.CSA_Pitch_Delta, 0.0)) <
      CSA_MAX_PITCH_DELTA_STEP ;
```

Figure 49 – Contract for Flight Control System

In the FCS architecture in Figure 48, the current of the aircraft pitch is sensed using the air data system. The air data system is replicated, so there are inputs for the left and right air data system (AD_L and AD_R). The assumptions for the FCS describe constraints on the sensed pitch values, both in terms of their rate of change (`AD_L_Pitch_Step_Delta_Valid` and `AD_R_Pitch_Step_Delta_Valid`) and also the discrepancy between the left and right AD values (`Pitch_lr_ok`). Finally, we make an assumption about simultaneous failures: the assumption `some_fgs_active` states that it is not the case that both sides have failed simultaneously. The guarantees of the contract have already been discussed.

A fragment of the grammar for AGREE is shown in Figure 50. The grammar is formatted in EBNF form and is parsable by the ANTLR tool. The grammar supports the entire LTL temporal logic fragment of PSL[1].

[1] If an invariant-based model checker is used, liveness operators (F, U) will be interpreted optimistically over finite paths [51].

```
spec_stmt
 : (ID ':')? 'assume' expr ';'
 | (ID ':')? 'assert' expr ';'
 | 'property' ID '=' expr ';'
 | 'const' ID ':' type '=' expr ';'
 | 'eq' ID ':' type '=' expr ';'
 | 'parameter' ID ':' type ';'
 | 'fun' ID '(' arg (',' arg)* ')' ':' type '=' expr ';'
 ;
facts : pattern_instance*;
pattern_instance : 'pattern_instance' ID ':' spec_stmt*
   'end' 'pattern_instance' ID ';' ;
arg : ID ':' type ;
expr : arrow_expr ;
arrow_expr : a=implies_expr ('->' arrow_expr)? ;
implies_expr : equiv_expr ('=>' b=implies_expr)? ;
equiv_expr : bin_temporal_op ('<=>' b=bin_temporal_op)? ;
bin_temporal_op : ( 'U' uny_temporal_op | 'V' uny_temporal_op )* ;

uny_temporal_op
 : ( 'X' uny_temporal_op
   | 'G' uny_temporal_op
   | 'F' uny_temporal_op
   | 'always' uny_temporal_op
   | 'never' uny_temporal_op
   )
 | if_then_else_expr ;
if_then_else_expr
  : 'if' if_then_else_expr 'then' if_then_else_expr 'else' if_then_else_expr
  | or_bool_op ;
or_bool_op : and_bool_op ('or' and_bool_op )* ;
and_bool_op : not_bool_op ('and' not_bool_op )* ;
not_bool_op
  : 'not' not_bool_op;
  | relational_op;
relational_op : ...
```

Figure 50 – AGREE PSL Fragment

4.6.3 Facts

For architectural patterns, we want to automatically import and instantiate the verification results that were previously proven about the pattern. In AGREE, we introduce the verification results as *facts*, that is, guarantees about the system that we do not have to prove again. An example of facts for the Leader Select pattern specialized for the FCS system architecture is shown in Figure 51. The generic properties that were proven in Section 4.2.1.3 for Leader Select are, at the time of pattern application, instantiated over the architectural components in the model that the pattern affects. More specifically, the properties proved were quantified over a "generic" set of components. The generic set is instantiated to match the architectural elements and the quantifiers are expanded out of the formulas.

```
pattern_instance Leader_Select_1 :

  -- sync single-step delay between elements
  assume single_step_delay_comm(FGS_L, FGS_R);
  assume single_step_delay_comm(FGS_R, FGS_L);

 -- All non-failed nodes agree on who is the leader
 leader_agreement:
    assert (FGS_L.LSO.Valid and FGS_R.LSO.Valid) =>
      FGS_L.LSO.Leader = FGS_R.LSO.Leader;

 -- If a node fails, leadership is transferred to a non-failed node
 leader_transfer_1:
   assert (prev(not(FGS_L.LSO.Valid), false) =>
    (FGS_R.LSO.Valid =>
      FGS_R.LSO.Leader != Get_Property(FGS_L, Leader_Select_ID)));

 leader_transfer_2:
   assert prev(not(FGS_R.LSO.Valid), false) =>
     (FGS_L.LSO.Valid =>
        FGS_L.LSO.Leader != Get_Property(FGS_R, Leader_Select_ID));

  -- If any non-failed nodes exist, one of them will be the leader
  leader_existence:
   assert (prev(FGS_L.LSO.Valid or FGS_R.LSO.Valid, false)) =>
     (( FGS_L.LSO.Valid => (FGS_L.LSO.Leader >= 1 and FGS_L.LSO.Leader <= 2)) and
      ( FGS_R.LSO.Valid => (FGS_R.LSO.Leader >= 1 and FGS_R.LSO.Leader <= 2)));

  -- If the leader does not fail, it shall remain the leader.
  leader_persistence_1: assert
    (prev(FGS_L.LSO.Valid and
     FGS_L.LSO.Leader = Get_Property(FGS_L, Leader_Select_ID), false)) =>
       (FGS_L.LSO.Valid =>
          FGS_L.LSO.Leader = Get_Property(FGS_L, Leader_Select_ID));

  leader_persistence_2: assert
   (prev(FGS_R.LSO.Valid and
    FGS_R.LSO.Leader = Get_Property(FGS_R, Leader_Select_ID), false)) =>
      (FGS_R.LSO.Valid =>
         FGS_R.LSO.Leader = Get_Property(FGS_R, Leader_Select_ID));
end pattern_instance Leader_Select_1 ;
```

Figure 51 – Facts for Leader Select implemented in FCS

Note that facts are "conditional": it is possible for a pattern to have assumptions about the environment in which it is instantiated. These assumptions must be discharged (like any other assumptions) for the guarantees that the pattern provides to hold. In the case of the FCS, for the algorithm to work properly we need an assumption that the components involved in Leader Selection communicate synchronously. This is expressed by:

```
-- sync single-step delay between elements
assume single_step_delay_comm(FGS_L, FGS_R);
assume single_step_delay_comm(FGS_R, FGS_L);
```

In this example, this assumption can be discharged through our use of the PALS pattern, but this is not yet integrated into the AGREE tool. The goal is to build what we call an *evidence graph* that describes all remaining assumptions that have not been discharged by proof. This would allow patterns to discharge assumptions of other patterns, and display to the user a ledger that describes any outstanding proof obligations.

4.6.4 The Proof Process

The proof system uses induction over time to ensure that conclusions derived from analysis of a component soundly follow from system assumptions, guarantees provided by sub-components, and pattern guarantees (facts). The justification of the reasoning process is similar to the one provided by McMillan [53].

The AGREE tool examines the data dependencies associated with system components and builds a series of proof obligations based on these dependencies. For example, in the FCS architecture, the dependencies are shown in Figure 52. In this case, the system inputs feed the flight guidance systems, which feed the autopilot, which ultimately feeds the CSA output.

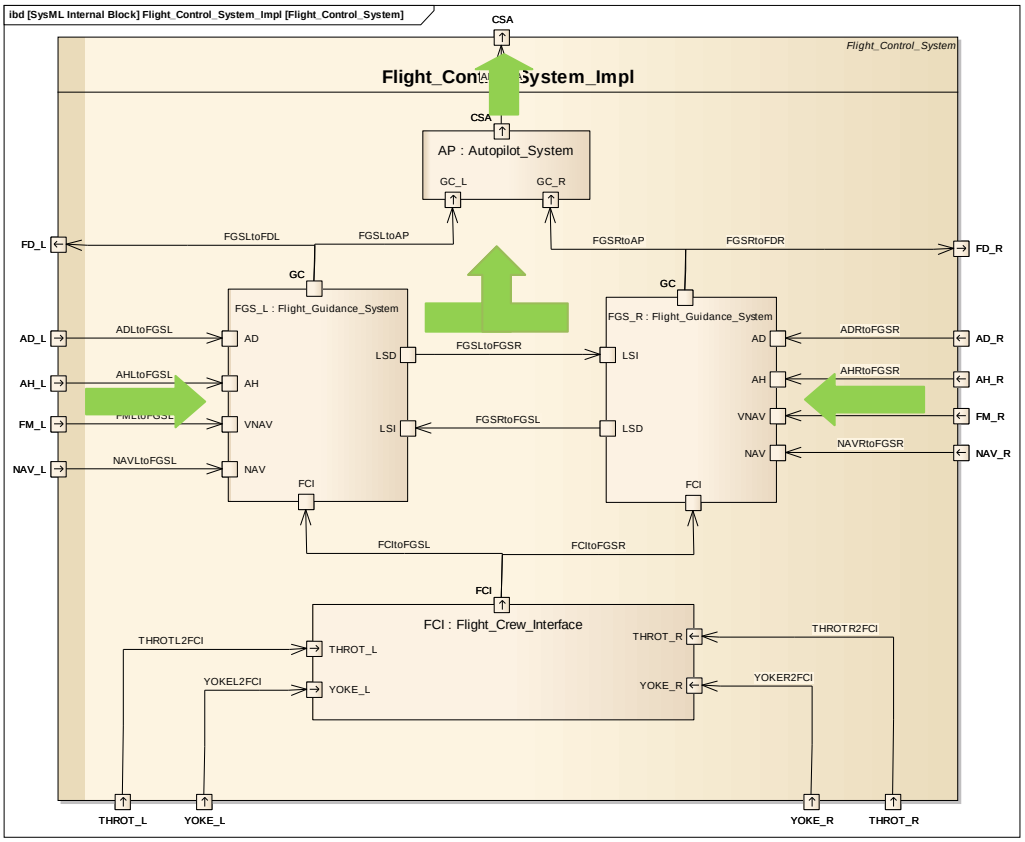


Figure 52 – Architecture Data Dependencies

Based on the data dependencies, we establish a series of proof obligations. The dependencies discovered provide an ordering principle by which we can construct the proof obligations. Intuitively, any component "later" in the order can use guarantees from "earlier" components within the order and also the system assumptions. In our example, this means that five proof obligations are created:

- The FCS system assumptions ⇒ the FGS_L component assumptions
- The FCS system assumptions ⇒ the FGS_R component assumptions
- The FCS system assumptions ⇒ the FCI component assumptions
- The FCS system assumptions and the FGS_L and FGS_R component guarantees ⇒ the AP component assumptions
- The FCS system assumptions, the FGS_L and FGS_R component guarantees and AP component guarantees ⇒ the FCS system guarantees.

The formal system justifies the soundness of the approach. Note that the FGS_L and FGS_R systems communicate back and forth. In our example, it was not necessary to account for these *circular* communications within the proof. However, many times it is necessary to reason about circular communications, which if handled naïvely can lead to unsoundness in the proof system. In these cases, the apparent circularity in reasoning can be broken by using induction over time. That is, if we have two circular components {A, B} and we use the guarantees provided by B in the current time step to prove the assumptions of A, then we can only use the guarantees of A in the *previous time step* to establish the assumptions of B. To support this style of reasoning, the tool allows the user to specify how to break the cycle between components to support inductive reasoning.

To prove the system guarantee, the component guarantees, taken together, must be sufficiently strong to establish it. This means for the FCS example, we must allocate functionality to the flight guidance systems and the autopilot. The contracts for these components are shown in Figure 53 and Figure 54.

```
  parameter Leader_Select_ID : int ;

  property this_side_in_control =
   GC.mds.active;

  property active_implies_valid =
    this_side_in_control => LSO.Valid ;

  property leader_implies_active =
    (LSO.Leader = Leader_Select_ID) => GC.mds.active ;

  const GC_MAX_PITCH_DELTA_STEP: real = 1.0 ;
  const GC_MAX_PITCH_DELTA: real = 5.0 ;

  property gc_ok =(true ->
    (abs(GC.cmds.pitch_delta - prev(GC.cmds.pitch_delta, 0.0)) <
     GC_MAX_PITCH_DELTA_STEP)) and
    (abs(GC.cmds.pitch_delta) < GC_MAX_PITCH_DELTA);

  assert active_implies_valid ;
  assert leader_implies_active ;
  assert LSO.Valid = AD.pitch.valid ;

  this_side_control_correct: assert
    this_side_in_control => gc_ok ;

  assert this_side_in_control => AD.Pitch.Val = GC.cmds.pitch_delta ;
```

Figure 53 – FGS contract

```
const CSA_MAX_PITCH_DELTA: real = 5.0 ;
const CSA_MAX_PITCH_DELTA_STEP: real = 5.0 ;

eq leader_pitch_delta : real =
   if (GC_L.mds.active) then GC_L.cmds.Pitch_Delta
   else if (GC_R.mds.active) then GC_R.cmds.Pitch_Delta
   else (prev(leader_pitch_delta, 0.0)) ;

-- assertion just defines CSA pitch delta in terms of the leader pitch delta.
assert (leader_pitch_delta > 0.0 => (CSA.CSA_Pitch_Delta > 0.0 and
        CSA.CSA_Pitch_Delta <= leader_pitch_delta)) and
       (leader_pitch_delta <= 0.0 => (CSA.CSA_Pitch_Delta <= 0.0 and
        CSA.CSA_Pitch_Delta >= leader_pitch_delta)) ;
```

Figure 54 – AP Contract

4.6.5 The AGREE Tool

The AGREE tool is implemented as a plug-in in the Eclipse environment. It uses the property set PSL_Properties to add support for compositional reasoning to AADL models using custom AADL properties. The PSL_Properties property set is defined as follows:

```
property set PSL_Properties is
  Contract: aadlstring applies to (system, process, thread);
  Facts: aadlstring applies to (system, process, thread);
end PSL_Properties;
```

That is, it supports contracts and facts on systems, processes, and threads specified as AADL strings. Verification of AADL models is performed through the translation of the AADL structure and subcomponent assumptions and guarantees into a form suitable for model checking. Currently the KIND model checker is supported, but it would be straightforward to add support for additional model checkers and theorem provers.

In our initial implementation, subcomponents are assumed to operate synchronously with a one-step communication delay between connected subcomponents. This makes the analysis tractable and creates a sound approximation of the behavior of the system. Any error found during verification corresponds to an error in the actual system. The approximation is complete in the case of synchronous systems (e.g. systems using the PALS pattern), and incomplete in the general case. Incompleteness means that the absence of verification errors does not ensure that the system is correct.

To execute the AGREE tool, we open the AAXL file of a system which we would like to verify, select the system implementation, and choose “Verify with Kind,” as shown in Figure 55. This will cause the plug in to examine the data dependencies of the model, construct proof obligations, and submit them to the Kind model checker. The results are shown in Figure 56.

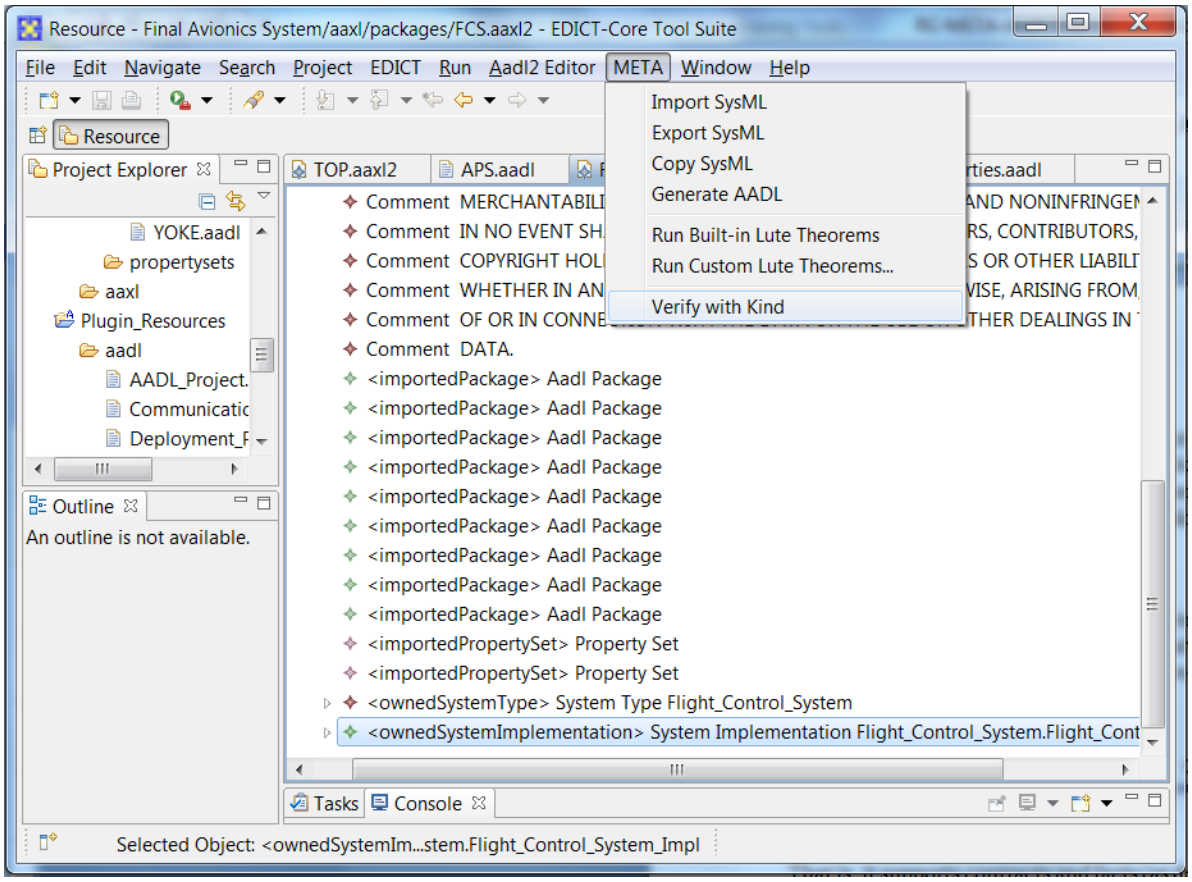


Figure 55 – AGREE Plug-In

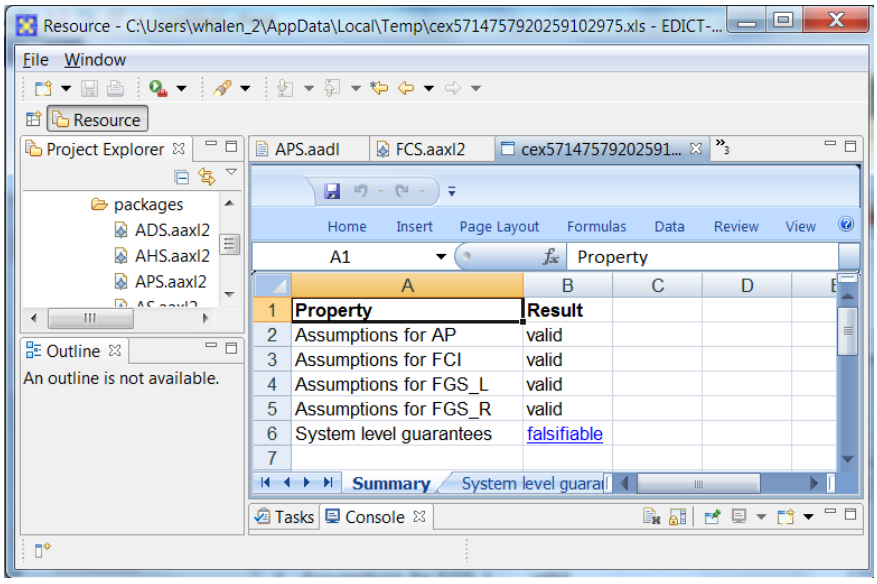
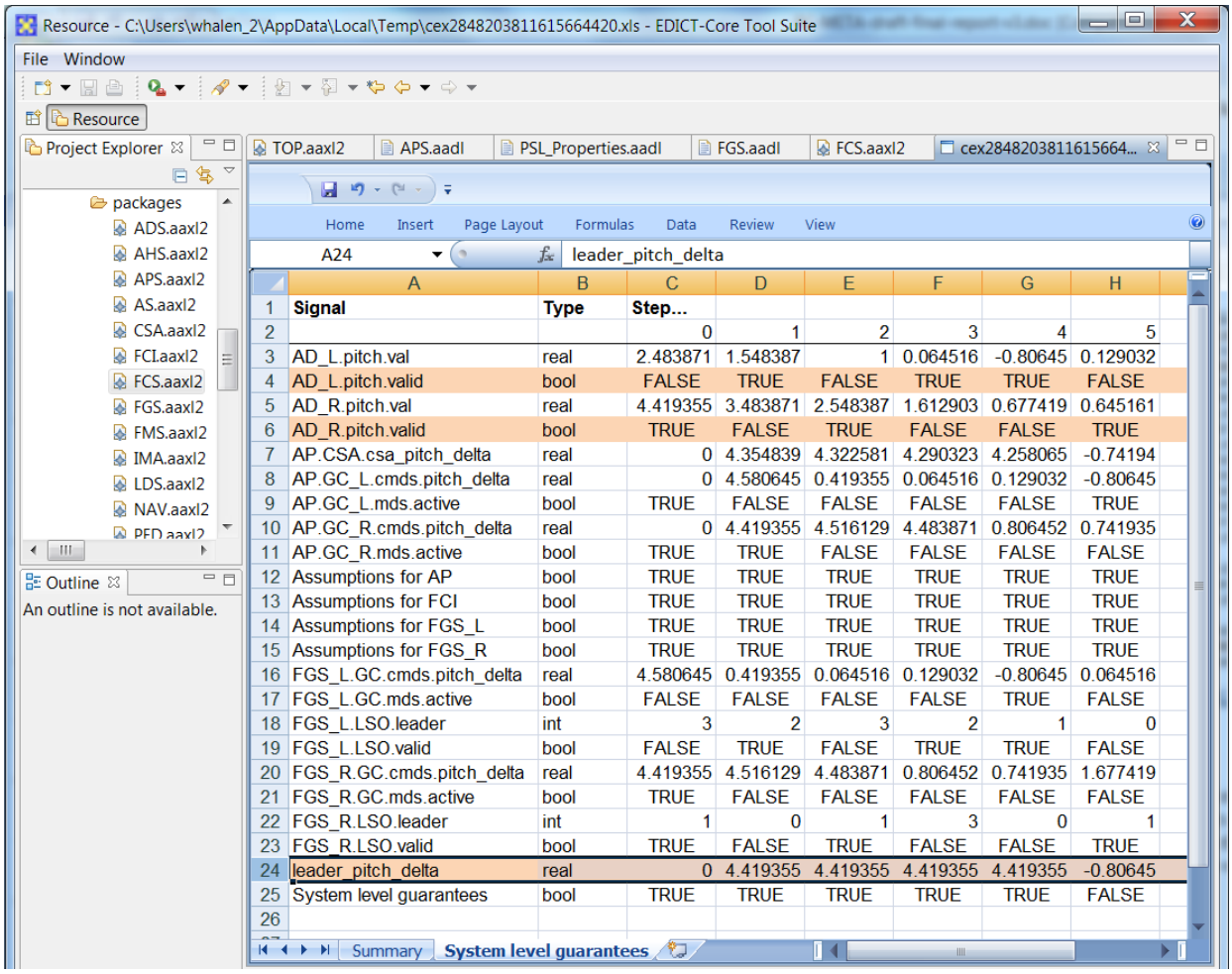


Figure 56 – Verification Results

Note that the final proof failed: the system-level guarantee was not provable using the facts from leader selection and the component level guarantees. This is something that needs to be fixed, so we can click on the link to see a *counterexample*, which is a test case that demonstrates a case in

which the component guarantees were satisfied but the system property was not satisfied. The counterexample to this property is shown in Figure 57.



	A	B	C	D	E	F	G	H
1	**Signal**	**Type**	**Step...**					
2			0	1	2	3	4	5
3	AD_L.pitch.val	real	2.483871	1.548387	1	0.064516	-0.80645	0.129032
4	AD_L.pitch.valid	bool	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE
5	AD_R.pitch.val	real	4.419355	3.483871	2.548387	1.612903	0.677419	0.645161
6	AD_R.pitch.valid	bool	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE
7	AP.CSA.csa_pitch_delta	real	0	4.354839	4.322581	4.290323	4.258065	-0.74194
8	AP.GC_L.cmds.pitch_delta	real	0	4.580645	0.419355	0.064516	0.129032	-0.80645
9	AP.GC_L.mds.active	bool	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE
10	AP.GC_R.cmds.pitch_delta	real	0	4.419355	4.516129	4.483871	0.806452	0.741935
11	AP.GC_R.mds.active	bool	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
12	Assumptions for AP	bool	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
13	Assumptions for FCI	bool	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
14	Assumptions for FGS_L	bool	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
15	Assumptions for FGS_R	bool	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
16	FGS_L.GC.cmds.pitch_delta	real	4.580645	0.419355	0.064516	0.129032	-0.80645	0.064516
17	FGS_L.GC.mds.active	bool	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
18	FGS_L.LSO.leader	int	3	2	3	2	1	0
19	FGS_L.LSO.valid	bool	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE
20	FGS_R.GC.cmds.pitch_delta	real	4.419355	4.516129	4.483871	0.806452	0.741935	1.677419
21	FGS_R.GC.mds.active	bool	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
22	FGS_R.LSO.leader	int	1	0	1	3	0	1
23	FGS_R.LSO.valid	bool	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE
24	leader_pitch_delta	real	0	4.419355	4.419355	4.419355	4.419355	-0.80645
25	System level guarantees	bool	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
26								

Figure 57 – Counterexample

If we examine the counterexample, what we see is that the left and right sides have a "ping-pong" failure where the left side is valid for one step, then the right side, then the left side, etc. Although the leader select algorithm is working as intended, when the autopilot is unsure of which side is the leader, it holds its previous value until it is able to establish one side as the leader. In this case, it does not "trust" either side until it has been active for two consecutive steps. In this case, because there was a significant delay since the last "trusted" value, there is a large output transient that is generated.

We decide that this "ping-pong" failure is not a reasonable operating environment for the system, and add additional assumptions to the model that state that if a component is becomes valid, then it will stay valid for at least two steps. After adding the assumption, we prove the property, as shown in Figure 58.

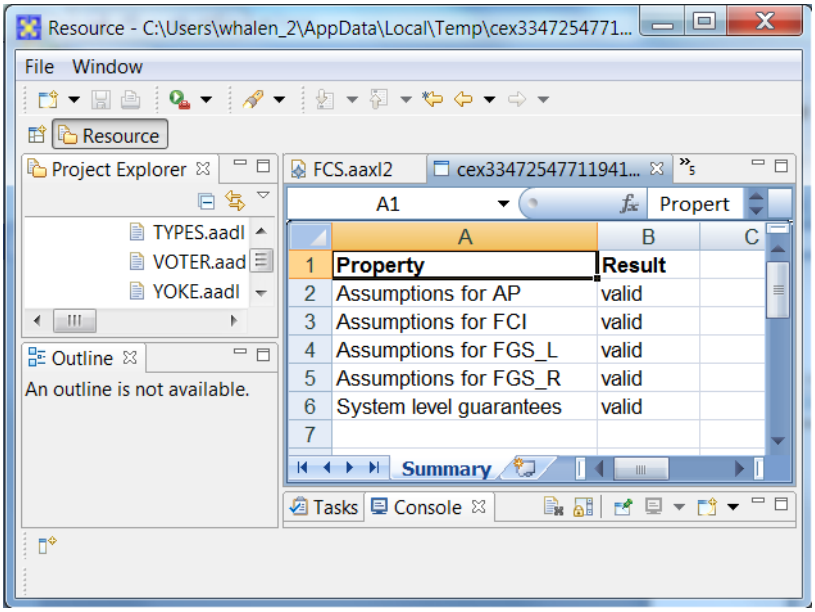


Figure 58 – Final Proof Result

5.0 CONCLUSIONS

The impact of this project is illustrated in Figure 59. The traditional development process is often pictured as a 'V' with design steps proceeding down the left side and verification or testing steps proceeding up the right side. Verification failure at any step leads to rework to identify and correct the error, with the consequence of failure growing larger the later in the process it occurs.

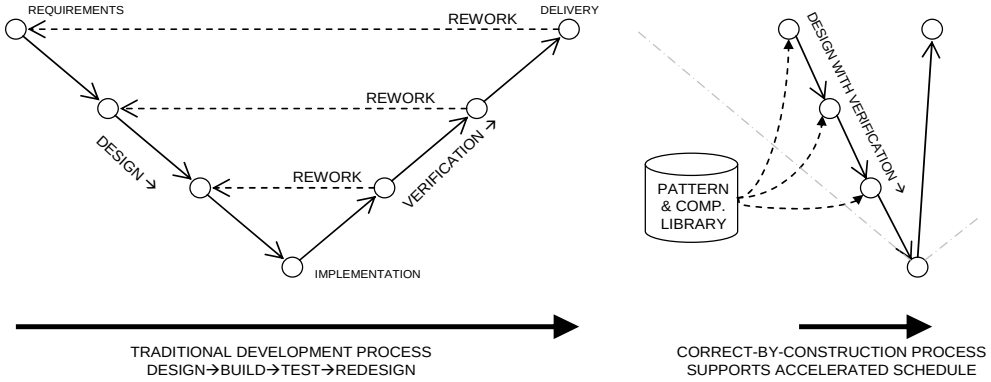


Figure 59 – Impact of correct-by-construction development process

The work described here provides improvements on both sides of the 'V'. First and most important, the use of verified system design patterns that integrate components having verified properties yields system models at each level of refinement that are correct by construction. These patterns and component models are drawn from a library that specifies, in addition to their functionality, the properties that they guarantee and the constraints under which they may be used. Note that this is not so much about software or model reuse, as it is about *verification reuse*. As a result of this correct-by-construction process, the resulting implementation can be expected to operate as specified. A confirmatory system-level test should be run prior to delivery, but no errors should be found and the entire right side of the 'V' should be greatly reduced.

The left side of the 'V' is improved as well. The use of a standard collection of domain-relevant models along with automated software tools for handling composition and analysis speeds the design process. Complexity-reducing design patterns result in designs that are simpler to manage and understand. The resulting process is a "half-V" or "backslash" development process.

6.0 REFERENCES

Design Problems

[1] AFE #58 Summary Final Report, System Architecture Virtual Integration (SAVI) Program, Aerospace Vehicles Systems Institute, SAVI-58-00-001, October 8, 2009.

[2] Allan, N. S., "A Practical Reliability Evaluation of Embedded Avionic Software," Proceedings of the 1987 IEEE Southern Tier Technical Conference, pp. 238-243, April 29, 1987. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=716399.

[3] Boehm, Barry, Ricardo Valerdi, Jo Ann Lane and A. Winsor Brown., "COCOMO Suite Methodology and Evolution," Cross Talk, The Journal of Defense Software Engineering, April, 2005, p.20-25.

[4] Bradford, R., Interview with Principal Software Engineer, Rockwell Collins Commercial Systems, October 25, 2010.

[5] Bradford, R., Teleconference with Rockwell Collins Enterprise Architects, October 25, 2010.

[6] Bradford, R., Interview with Senior Systems Engineer, Rockwell Collins Commercial Systems, October 26, 2010.

[7] Burchell, Bill, "Untangling No Fault Found," Aviation Week & Space Technology, February 9, 2007. http://www.aviationweek.com/aw/generic/story_generic.jsp?channel=om&id=news/om207cvr.xml.

[8] Burns, Stephanie, Chris Hansen, and Steve Maher, "Clean Sheet Designs Root Cause Analysis," Rockwell Collins Inc. Presentation, December 18, 2009

[9] Burns, Stephanie, Chris Hansen., and Steve Maher, "Common Themes for Clean Sheet Designs," Rockwell Collins Inc. Presentation.

[10] Burns, Stephanie, Chris Hansen., and Steve Maher, "Analysis of Complex System Development Efforts," Rockwell Collins Inc. Presentation.

[11] Dabney, J. B. and K. Costello, "Return on Investment for Software IV&V," Third Annual NASA project Management Conference, March 2006.

[12] Helton, S. B., J. Hansson, and D.A. Redman, "SAVI ROI Analysis," produced under the System Architecture Virtual Integration (SAVI) Proof-Of-Concept Program AFE #58, proprietary document of the Aerospace Vehicles Systems Institute, SAVI-58-03-004, August 24, 2009.

[13] Hooks, I. and K. Farry, "Customer Centered Products: Creating Successful Products Through Smart Requirements Management," AMACOM American Management Association, New York, New York, 2001.

[14] King, T. and J. Marasco, "What is the Cost of a Requirements Error?" http://www.stickyminds.com/sitewide.asp?ObjectId=12529&Function=edetail.

[15] Lempia, David L. and Steven P. Miller. *Requirements Engineering Management Handbook*, FAA Contractor Report AR-08-32, June 2009.

[16] Miller, Steven P., Darren D. Cofer, Lui Sha, Jose Meseguer, and Abdullah Al-Nayeem, "Implementing Logical Synchrony in Integrated Modular Avionics," 28th Digital Avionics Systems Conference (DASC 2009), Orlando, Florida, Oct 25-29, 2009.

[17] *The Economic Impacts of Inadequate Infrastructure for Software Testing*, NIST Planning Report 02-3, May 2002.

[18] Potocki De Montalk, J.P., "Computer Software in Civil Aircraft," Sixth Annual Conference on Computer Assurance (COMPASS '91), Gaithersburg, MD, June 24-27, 1991.

Design Flow

[19] IEEE Std 1850-2005, IEEE Standard for Property Specification Language (PSL).

[20] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[21] E. Clarke, O. Grumberg, and P. Peled, *Model Checking*, The MIT Press, Cambridge, Massachusetts, 2001.

[22] O. Gilles, J. Hugues, Expressing and Enforcing User-Defined Constraints of AADL Models, *Engineering of Complex Computer Systems (ICECCS),* pp.337-342, March 22-26, 2010.

[23] G. Hagen and C. Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD'08)*, Portland, Oregon. IEEE, 2008.

[24] T. Henzinger, The Theory of Hybrid Automata, *Verification of Digital and Hybrid Systems* (M.K. Inan, R.P. Kurshan, eds.), NATO ASI Series F: Computer and Systems Sciences, Vol. 170, Springer-Verlag, 2000, pp. 265-292.

[25] M. Kaufmann, P. Manolios, and J.S. Moore, *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers, June, 2000. (ISBN 0-7923-7744-3).

[26] H. Kopetz. The Time-Triggered Architecture, *Proceedings of the IEEE*, 91(1), January, 2003.

[27] N. A. Lynch and F. W. Vaandrager. Action Transducers and Timed Automata. *Formal Aspects of Computing*, 8(5):499–538, 1996.

[28] S. Miller, D. Cofer, L. Sha, A. Al-nayeem, Implementing Logical Synchrony in Integrated Modular Avionics, Proceedings of the 28th Digital Avionics Systems Conference, 2009.

[29] X. Nicollin and J. Sifakis. The Algebra of Timed Processes, *ATP: Theory and Application. Information and Computation*, 114(1):131–178, 1994.

[30] IRST, http://nusmv.irst.itc.it/ The NuSMV Model Checker, IRST, Trento Italy.

[31] L. Sha, A. Al-nayeem, M. Sun, J. Meseguer, P. Ãlveczky and W.M.Y. Nam and P. Feiler, PALS: Physically Asynchronous Logically Synchronous Systems, University of Illinois Urbana Champaign Technical Report, 2009 (http://hdl.handle.net/2142/11897).

[32] S. Schneider. Concurrent and Real-time Systems. John Wiley and Sons, 2000.

[33] SRI International, http://pvs.csl.sri.com , The PVS Specification and Verification System, SRI International.

[34] SRI, SAL Home Page, http://www.csl.sri.com/projects/sal/.

[35] University of Iowa, KIND Home Page, http://clc.cs.uiowa.edu/Kind/.

[36] W. Yi, CCS + Time = An Interleaving Model for Real Time Systems. In *18th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 510 of Lecture Notes in Computer Science, pages 217–228. Springer, 1991.

Pattern Verification

[37] G. Brown and L. Pike, "Easy" parameterized verification of cross clock domain protocols. In *Designing Correct Circuits (DCC'06)* (Participants' Proceedings), 2006.

[38] G. Brown and L. Pike. Easy parameterized verification of biphase and 8N1 protocols. In *12th International Conference on Tools and Algorithms for the Construction and Analysis of Algorithms (TACAS'06)*, volume 3920 of LNCS, pages 58--72, 2006. Springer.

[39] J. Dabney and T. Harmon, *Mastering Simulink*, Pearson Prentice Hall: Upper Saddle River, NJ, 2004.

[40] C. Eisner and D. Fisman, *A Practical Introduction to PSL*. Springer Verlag, 2006.

[41] O. Gilles and J. Hugues. Validating requirements at model-level. *Proceedings of the 4th workshop on Model-Oriented Engineering (IDM'08)*, June 2008.

[42] K. Havelund and T. Pressburger, Model Checking JAVA Programs using Java Pathfinder, *International Journal on Software Tools for Technology Transfer (STTT)*, Springer Verlag, 2000.

[43] IRST: http://nusmv.irst.itc.it/ The NuSMV Model Checker, IRST, Trento Italy

[44] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, *Theoretical Computer Science* 96 (1992) 73-155.

[45] J. Meseguer and P. C. Ölveczky. 2010. Formalization and correctness of the PALS architectural pattern for distributed real-time systems. In *Proceedings of the 12th international conference on Formal engineering methods and software engineering* (ICFEM'10), Jin Song Dong and Huibiao Zhu (Eds.). Springer-Verlag, Berlin, Heidelberg, 303-320. (www.ideals.illinois.edu/handle/2142/17089)

[46] P.C. Ölveczky, J. Meseguer, Semantics and pragmatics of Real-Time Maude, *Higher-Order and Symbolic Computation* 20 (2007) 161-196.

[47] S. Miller et. al, Implementing Logical Synchrony in Integrated Modular Avionics, *Digital Avionics System Conference (DASC)*, December 2009, IEEE Press.

[48] Min-Young Nam; Pellizzoni, R.; Lui Sha; Bradford, R.M., ASIIST: Application Specific I/O Integration Support Tool for Real-Time Bus Architecture Designs, *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, June 2009.

[49] SRI International: http://sal.csl.sri.com The Symbolic Analysis Laboratory, SRI International

[50] L. Sha, A. Abdullah, M. Sun, J. Meseguer, P. Olveczky, PALS: Physically Asynchronous Logically Synchronous Systems, May, 2009 (www.ideals.illinois.edu/handle/2142/11897).

Pattern Verification

[51] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with Temporal Logic on Truncated Paths, in *Proceedings of Computer Aided Verification (CAV) 2003,* pp. 27-39, 2003.

[52] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. In *Proceedings of the IEEE*, Volume 79, #9, pp. 1305-20, September 1991.

[53] K. McMillan. Circular Compositional Reasoning about Liveness, *Technical Report 1999-02,* Cadence Berkeley Labs, Berkeley, CA 94709, 1999.

APPENDIX A AADL MODELS OF AVIONICS SYSTEM

A.1 Complete Avionics System (TOP)

```
package TOP
public

  with AS;
  with IMA;

  system Complete_Avionics_System
  end Complete_Avionics_System;

  system implementation Complete_Avionics_System.
                              Complete_Avionics_System_Impl
    subcomponents
      SW: system AS::Avionics_System.Avionics_System_Impl;
      HW: system IMA::IMA_Platform.IMA_Platform_Impl;

    properties
      Actual_Connection_Binding =>
            (reference (HW.IMA_Bus)) applies to SW.FCS.FGSRtoFGSL;
      Actual_Connection_Binding =>
            (reference (HW.IMA_Bus)) applies to SW.FCS.FGSLtoFGSR;
      Actual_Processor_Binding =>
            (reference (HW.B.PRC)) applies to SW.FCS.FGS_R;
      Actual_Processor_Binding =>
            (reference (HW.A.PRC)) applies to SW.FCS.FGS_L;

  end Complete_Avionics_System.Complete_Avionics_System_Impl;

end TOP;
```

A.2 Avionics System (AS)

```
package AS
public
  with CSA, YOKE, THROTTLE, FCS, PFD, ADS, AHS, FMS, NAV;

  system Avionics_System
  end Avionics_System;

  system implementation Avionics_System.Avionics_System_Impl
    subcomponents
      CSA: device CSA::Control_Surface_Actuators;
      YOKE_R: device YOKE::Yoke;
      THROT_L: device THROTTLE::Throttle;
      YOKE_L: device YOKE::Yoke;
      THROT_R: device THROTTLE::Throttle;
      FCS: system
            FCS::Flight_Control_System.Flight_Control_System_Impl;
      PFD_L: system
            PFD::Primary_Flight_Display.Primary_Flight_Display_Impl;
      ADS_L: system
            ADS::Air_Data_System.Air_Data_System_Impl;
      AHS_L: system
```

```
            AHS::Attitude_Heading_System.Attitude_Heading_System_Impl;
      FMS_L: system
      FMS::Flight_Management_System.Flight_Management_System_Impl;
      NAV_L: system
            NAV::Navigation_System.Navigation_System_Impl;
      PFD_R: system
      PFD::Primary_Flight_Display.Primary_Flight_Display_Impl;
      ADS_R: system
            ADS::Air_Data_System.Air_Data_System_Impl;
      AHS_R: system
      AHS::Attitude_Heading_System.Attitude_Heading_System_Impl;
      FMS_R: system
      FMS::Flight_Management_System.Flight_Management_System_Impl;
      NAV_R: system
            NAV::Navigation_System.Navigation_System_Impl;
    connections
      FCS2PFDL: port FCS.FD_L -> PFD_L.FD;
      AHSL2FCS: port AHS_L.AH -> FCS.AH_L;
      ADSL2FCS: port ADS_L.AD -> FCS.AD_L;
      FMSL2FCS: port FMS_L.FM -> FCS.FM_L;
      NAVL2FCS: port NAV_L.NAV -> FCS.NAV_L;
      THROTL2FCS: port THROT_L.THROT -> FCS.THROT_L;
      YOKEL2FCS: port YOKE_L.YOKE -> FCS.YOKE_L;
      FCS2PFDR: port FCS.FD_R -> PFD_R.FD;
      ADSR2FCS: port ADS_R.AD -> FCS.AD_R;
      AHSR2FCS: port AHS_R.AH -> FCS.AH_R;
      FMSR2FCS: port FMS_R.FM -> FCS.FM_R;
      NAVR2FCS: port NAV_R.NAV -> FCS.NAV_R;
      THROTR2FCS: port THROT_R.THROT -> FCS.THROT_R;
      YOKER2FCS: port YOKE_R.YOKE -> FCS.YOKE_R;
      FCS2CSA: port FCS.CSA -> CSA.CMD;
  end Avionics_System.Avionics_System_Impl;
end AS;
```

A.3 Flight Control System (FCS)

```
package FCS
public

  with FGS, ADS, AHS, FMS, NAV, THROTTLE, YOKE, APS, FCI, META_Properties;

  system Flight_Control_System
    features
      FD_L: out data port FGS::Guidance_Data;
      AD_L: in data port ADS::Air_Data;
      FD_R: out data port FGS::Guidance_Data;
      AD_R: in data port ADS::Air_Data;
      AH_L: in data port AHS::Attitude_Heading_Data;
      AH_R: in data port AHS::Attitude_Heading_Data;
      FM_L: in data port FMS::Flight_Management_Data;
      FM_R: in data port FMS::Flight_Management_Data;
      NAV_L: in data port NAV::Navigation_Data;
      NAV_R: in data port NAV::Navigation_Data;
      THROT_L: in data port THROTTLE::Throttle_Data;
      THROT_R: in data port THROTTLE::Throttle_Data;
      YOKE_L: in data port YOKE::Yoke_Data;
      YOKE_R: in data port YOKE::Yoke_Data;
```

```
    CSA: out data port APS::Control_Surface_Actuator_Data;

  end Flight_Control_System;

  system implementation Flight_Control_System.Flight_Control_System_Impl
    subcomponents
      AP: system APS::Autopilot_System.Autopilot_System_Impl;
      FCI: system FCI::Flight_Crew_Interface.Flight_Crew_Interface_Impl;
      FGS_L: system
      FGS::Flight_Guidance_System.Flight_Guidance_System_Impl;
      FGS_R: system
      FGS::Flight_Guidance_System.Flight_Guidance_System_Impl;

    connections
      FGSLtoAP: port FGS_L.GC -> AP.GC_L;
      FGSRtoAP: port FGS_R.GC -> AP.GC_R;
      AP2CSA: port AP.CSA -> CSA;
      THROTR2FCI: port THROT_R -> FCI.THROT_R;
      YOKEL2FCI: port YOKE_L -> FCI.YOKE_L;
      YOKER2FCI: port YOKE_R -> FCI.YOKE_R;
      FCItoFGSL: port FCI.FCI -> FGS_L.FCI;
      FCItoFGSR: port FCI.FCI -> FGS_R.FCI;
      THROTL2FCI: port THROT_L -> FCI.THROT_L;
      ADLtoFGSL: port AD_L -> FGS_L.AD;
      AHLtoFGSL: port AH_L -> FGS_L.AH;
      FMLtoFGSL: port FM_L -> FGS_L.VNAV;
      NAVLtoFGSL: port NAV_L -> FGS_L.NAV;
      FGSRtoFGSL: port FGS_R.LSD -> FGS_L.LSI;
      FGSLtoFDL: port FGS_L.GC -> FD_L;
      FGSLtoFGSR: port FGS_L.LSD -> FGS_R.LSI ;
      AHRtoFGSR: port AH_R -> FGS_R.AH;
      FMRtoFGSR: port FM_R -> FGS_R.VNAV;
      NAVRtoFGSR: port NAV_R -> FGS_R.NAV;
      FGSRtoFDR: port FGS_R.GC -> FD_R;
      ADRtoFGSR: port AD_R -> FGS_R.AD;

    properties
      Latency => 5 ms .. 8 ms applies to FGSLtoFGSR;
      Latency => 5 ms .. 8 ms applies to FGSRtoFGSL;
      META_Properties::Not_Collocated =>
            (reference (FGS_L)) applies to FGS_R;
      META_Properties::Not_Collocated =>
            (reference (FGS_R)) applies to FGS_L;

  end Flight_Control_System.Flight_Control_System_Impl;

end FCS;
```

A.4 Flight Guidance System (FGS)

```
package FGS
public

  with FMS, ADS, AHS, NAV, FCI, LDS, TYPES, IMA;
  with META_Properties, PALS_Properties;
```

```
  data Capture_Conditions
  end Capture_Conditions;

  data Flight_Modes
  end Flight_Modes;

  data Guidance_Commands
  end Guidance_Commands;

  data Guidance_Data
  end Guidance_Data;

  data Lateral_Modes
  end Lateral_Modes;

  data Vertical_Modes
  end Vertical_Modes;

  process Flight_Guidance_Process
    features
      VNAV: in data port FMS::Flight_Management_Data;
      AD: in data port ADS::Air_Data;
      AH: in data port AHS::Attitude_Heading_Data;
      NAV: in data port NAV::Navigation_Data;
      FCI: in data port FCI::Flight_Crew_Interface_Data;
      LSI: in data port LDS::Leader_Selection_Data;
      LSO: out data port LDS::Leader_Selection_Data;
      GC: out data port Guidance_Data;
  end Flight_Guidance_Process;

  system Flight_Guidance_System
    features
      FCI: in data port FCI::Flight_Crew_Interface_Data;
      AD: in data port ADS::Air_Data;
      AH: in data port AHS::Attitude_Heading_Data;
      VNAV: in data port FMS::Flight_Management_Data;
      NAV: in data port NAV::Navigation_Data;
      GC: out data port Guidance_Data;
      LSD: out data port LDS::Leader_Selection_Data;
      LSI: in data port LDS::Leader_Selection_Data;
  end Flight_Guidance_System;

thread Control_Laws
    features
      AH: in data port AHS::Attitude_Heading_Data;
      AD: in data port ADS::Air_Data;
      VNAV: in data port FMS::Flight_Management_Data;
      NAV: in data port NAV::Navigation_Data;
      CC: out data port Capture_Conditions;
      MD: in data port Flight_Modes;
      GC: out data port Guidance_Data;
  end Control_Laws;

  thread Mode_Logic
    features
      LSR: out data port LDS::Leader_Selection_Rank;
      LSA: in data port LDS::Leader_Selection_Active;
```

```
      FCI: in data port FCI::Flight_Crew_Interface_Data;
      CC: in data port Capture_Conditions;
      MD: out data port Flight_Modes;
  end Mode_Logic;

  data implementation Flight_Modes.Flight_Modes_Impl
    subcomponents
      active: data TYPES::Boolean;
      lat: data Lateral_Modes.Lateral_Modes_Impl;
      ver: data Vertical_Modes.Vertical_Modes_Impl;
  end Flight_Modes.Flight_Modes_Impl;

  data implementation Guidance_Commands.Guidance_Commands_Impl
  end Guidance_Commands.Guidance_Commands_Impl;

  data implementation Guidance_Data.Guidance_Data_Impl
    subcomponents
      mds: data Flight_Modes.Flight_Modes_Impl;
      cmds: data Guidance_Commands.Guidance_Commands_Impl;
  end Guidance_Data.Guidance_Data_Impl;

  data implementation Lateral_Modes.Lateral_Modes_Impl
    subcomponents
      ROLL_active: data TYPES::Boolean;
      HDG_active: data TYPES::Boolean;
  end Lateral_Modes.Lateral_Modes_Impl;

  data implementation Vertical_Modes.Vertical_Modes_Impl
  end Vertical_Modes.Vertical_Modes_Impl;

process implementation  Flight_Guidance_Process.Flight_Guidance_Process_Impl
    subcomponents
      LS: thread LDS::Leader_Selection.Leader_Selection_Impl
            { Deadline => 20 ms in binding (IMA::PowerPC_350Mhz);
              Deadline => 30 ms in binding (IMA::PowerPC_250Mhz);
              META_Properties::Output_Delay => 10 ms
                  in binding (IMA::PowerPC_250Mhz);
              META_Properties::Output_Delay => 7 ms
                  in binding (IMA::PowerPC_350Mhz);
              PALS_Properties::PALS_Id => "Leader_Selection";
              PALS_Properties::PALS_Period => 40 ms;
              Period => 40 ms;
            };
      ML: thread Mode_Logic.Mode_Logic_Impl;
      CL: thread Control_Laws.Control_Laws_Impl;

    connections
      LSItoLS: port LSI -> LS.LSI;
      LStoLSO: port LS.LSO -> LSO;
      LStoML: port LS.LSA -> ML.LSA;
      MLtoLS: port ML.LSR -> LS.LSR;
      FCItoML: port FCI -> ML.FCI;
      CCtoML: port CL.CC -> ML.CC;
      MLtoCL: port ML.MD -> CL.MD;
      AHtoCL: port AH -> CL.AH;
      ADtoCL: port AD -> CL.AD;
      VNAVtoCL: port VNAV -> CL.VNAV;
```

```
      NAVtoCL: port NAV -> CL.NAV;
      CLtoGC: port CL.GC -> GC;
  end Flight_Guidance_Process.Flight_Guidance_Process_Impl;

  system implementation Flight_Guidance_System.Flight_Guidance_System_Impl
    subcomponents
      FGP: process Flight_Guidance_Process.Flight_Guidance_Process_Impl;

    connections
      VNAVtoFGP: port VNAV -> FGP.VNAV;
      ADtoFGP: port AD -> FGP.AD;
      AHtoFGP: port AH -> FGP.AH;
      NAVtoFGP: port NAV -> FGP.NAV;
      FCItoFGP: port FCI -> FGP.FCI;
      LSItoFGP: port LSI -> FGP.LSI;
      FGPtoLSO: port FGP.LSO -> LSD;
      FGPtoGC: port FGP.GC -> GC;
  end Flight_Guidance_System.Flight_Guidance_System_Impl;

  thread implementation Control_Laws.Control_Laws_Impl
  end Control_Laws.Control_Laws_Impl;

  thread implementation Mode_Logic.Mode_Logic_Impl
  end Mode_Logic.Mode_Logic_Impl;

end FGS;
```

A.5 Autopilot System (APS)

```
package APS
public

  with FGS;

  data Control_Surface_Actuator_Data
  end Control_Surface_Actuator_Data;

  process AutoPilot_Process
    features
      CSA: out data port Control_Surface_Actuator_Data;
      GC_L: in data port FGS::Guidance_Data;
      GC_R: in data port FGS::Guidance_Data;

  end AutoPilot_Process;

  system Autopilot_System
    features
      GC_L: in data port FGS::Guidance_Data;
      GC_R: in data port FGS::Guidance_Data;
      CSA: out data port Control_Surface_Actuator_Data;

  end Autopilot_System;

  thread Autopilot_Thread
    features
      CSA: out data port Control_Surface_Actuator_Data;
      GC: in data port FGS::Guidance_Data;
```

```
  end Autopilot_Thread;

  thread Guidance_Selector
    features
      IN1: in data port FGS::Guidance_Data;
      IN2: in data port FGS::Guidance_Data;
      OUTPUT: out data port FGS::Guidance_Data;

  end Guidance_Selector;

  process implementation AutoPilot_Process.Autopilot_Process_Impl
    subcomponents
      SELT: thread Guidance_Selector.Guidance_Selector_Impl;
      APT: thread Autopilot_Thread.Autopilot_Thread_Impl;

    connections
      GCLtoSELT: port GC_L -> SELT.IN1;
      GCRtoSELT: port GC_R -> SELT.IN2;
      SELTtoAPT: port SELT.OUTPUT -> APT.GC;
      APT2CSA: port APT.CSA -> CSA;
  end AutoPilot_Process.Autopilot_Process_Impl;

  system implementation Autopilot_System.Autopilot_System_Impl
    subcomponents
      APP: process AutoPilot_Process.Autopilot_Process_Impl;

    connections
      APP2CSA: port APP.CSA -> CSA;
      GCL2APP: port GC_L -> APP.GC_L;
      GCR2APP: port GC_R -> APP.GC_R;
  end Autopilot_System.Autopilot_System_Impl;

  thread implementation Autopilot_Thread.Autopilot_Thread_Impl
  end Autopilot_Thread.Autopilot_Thread_Impl;

  thread implementation Guidance_Selector.Guidance_Selector_Impl
  end Guidance_Selector.Guidance_Selector_Impl;

end APS;
```

A.6 Flight Control Interface (FCI)

```
package FCI
public

  with THROTTLE;
  with YOKE;

  data Flight_Crew_Interface_Data
  end Flight_Crew_Interface_Data;

  process Flight_Crew_Interface_Process
    features
      THROT_L: in data port THROTTLE::Throttle_Data;
      THROT_R: in data port THROTTLE::Throttle_Data;
      YOKE_L: in data port YOKE::Yoke_Data;
```

```
      YOKE_R: in data port YOKE::Yoke_Data;
      FCI: out data port Flight_Crew_Interface_Data;
  end Flight_Crew_Interface_Process;

  system Flight_Crew_Interface
    features
      THROT_L: in data port THROTTLE::Throttle_Data;
      THROT_R: in data port THROTTLE::Throttle_Data;
      YOKE_L: in data port YOKE::Yoke_Data;
      YOKE_R: in data port YOKE::Yoke_Data;
      FCI: out data port Flight_Crew_Interface_Data;
  end Flight_Crew_Interface;

  thread Flight_Crew_Interface_Thread
    features
      YOKE_L: in data port YOKE::Yoke_Data;
      YOKE_R: in data port YOKE::Yoke_Data;
      THROT_L: in data port THROTTLE::Throttle_Data;
      THROT_R: in data port THROTTLE::Throttle_Data;
      FCI: out data port Flight_Crew_Interface_Data;
  end Flight_Crew_Interface_Thread;

  process implementation
      Flight_Crew_Interface_Process.Flight_Crew_Interface_Process_Impl
    subcomponents
      FCIT: thread
      Flight_Crew_Interface_Thread.Flight_Control_Interface_Thread_Impl;

    connections
      YOKEL2FCIT: port YOKE_L -> FCIT.YOKE_L;
      YOKER2FCIT: port YOKE_R -> FCIT.YOKE_R;
      THROTL2FCIT: port THROT_L -> FCIT.THROT_L;
      THROTR2FCIT: port THROT_R -> FCIT.THROT_R;
      FCIT2FCI: port FCIT.FCI -> FCI;
  end Flight_Crew_Interface_Process.Flight_Crew_Interface_Process_Impl;

  system implementation Flight_Crew_Interface.Flight_Crew_Interface_Impl
    subcomponents
      FCIP: process
      Flight_Crew_Interface_Process.Flight_Crew_Interface_Process_Impl;

    connections
      THROTL2FCIP: port THROT_L -> FCIP.THROT_L;
      THROTR2FCIP: port THROT_R -> FCIP.THROT_R;
      YOKEL2FCIP: port YOKE_L -> FCIP.YOKE_L;
      YOKER2FCIP: port YOKE_R -> FCIP.YOKE_R;
      FCIP2FCI: port FCIP.FCI -> FCI;
  end Flight_Crew_Interface.Flight_Crew_Interface_Impl;

  thread implementation
      Flight_Crew_Interface_Thread.Flight_Control_Interface_Thread_Impl
  end Flight_Crew_Interface_Thread.Flight_Control_Interface_Thread_Impl;

end FCI;
```

A.7 Control Surface Actuators (CSA)

```
package CSA
public

  with APS;

  device Control_Surface_Actuators
    features
      CMD: in data port APS::Control_Surface_Actuator_Data;

  end Control_Surface_Actuators;

end CSA;
```

A.8 Air Data System (ADS)

```
package ADS
public

  with TYPES;

  data Air_Data
  end Air_Data;

  data Airspeed
  end Airspeed;

  device Airspeed_Sensor
    features
      AS: out data port Airspeed;
  end Airspeed_Sensor;

  process Air_Data_Process
    features
      AD: out data port Air_Data;
      AS2: in data port Airspeed;
      AS3: in data port Airspeed;
      AS1: in data port Airspeed;
  end Air_Data_Process;

  system Air_Data_System
    features
      AD: out data port Air_Data;
  end Air_Data_System;

  thread Air_Data_Thread
    features
      AD: out data port Air_Data;
      AS: in data port Airspeed;
  end Air_Data_Thread;

  thread Airspeed_Voter
    features
      IN1: in data port Airspeed;
      IN2: in data port Airspeed;
      IN3: in data port Airspeed;
```

```
    OUTPUT: out data port Airspeed;
  end Airspeed_Voter;

  data implementation Air_Data.Air_Data_Impl
    subcomponents
      AirSpeed: data Airspeed.Airspeed_Impl;
  end Air_Data.Air_Data_Impl;

  data implementation Airspeed.Airspeed_Impl
    subcomponents
      Val: data TYPES::Real;
      Valid: data TYPES::Boolean;
  end Airspeed.Airspeed_Impl;

  process implementation Air_Data_Process.Air_Data_Process_Impl
    subcomponents
      ADT: thread Air_Data_Thread.Air_Data_Thread_Impl;
      ASVT: thread Airspeed_Voter.Airspeed_Voter_Impl;

    connections
      ADTtoADP: port ADT.AD -> AD;
      ASVTtoADT: port ASVT.OUTPUT -> ADT.AS;
      AS1toASVT: port AS1 -> ASVT.IN1;
      AS2toASVT: port AS2 -> ASVT.IN2;
      AS3toASVT: port AS3 -> ASVT.IN3;
  end Air_Data_Process.Air_Data_Process_Impl;

  system implementation Air_Data_System.Air_Data_System_Impl
    subcomponents
      AS1: device Airspeed_Sensor;
      AS2: device Airspeed_Sensor;
      AS3: device Airspeed_Sensor;
      ADP: process Air_Data_Process.Air_Data_Process_Impl;

    connections
      ADPtoADS: port ADP.AD -> AD;
      AS3toADP: port AS3.AS -> ADP.AS1;
      AS2toADP: port AS2.AS -> ADP.AS2;
      AS1toADP: port AS1.AS -> ADP.AS3;
  end Air_Data_System.Air_Data_System_Impl;

  thread implementation Air_Data_Thread.Air_Data_Thread_Impl
  end Air_Data_Thread.Air_Data_Thread_Impl;

  thread implementation Airspeed_Voter.Airspeed_Voter_Impl
  end Airspeed_Voter.Airspeed_Voter_Impl;

end ADS;
```

A.9 Attitude Heading System (AHS)

```
package AHS
public

  data Attitude_Heading_Data
  end Attitude_Heading_Data;
```

```
  process Attitude_Heading_Process
    features
      AH: out data port Attitude_Heading_Data;
  end Attitude_Heading_Process;

  system Attitude_Heading_System
    features
      AH: out data port Attitude_Heading_Data;
  end Attitude_Heading_System;

  thread Attitude_Heading_Thread
    features
      AH: out data port Attitude_Heading_Data;
  end Attitude_Heading_Thread;

  process implementation
      Attitude_Heading_Process.Atttude_Heading_Process_Impl
    subcomponents
      AHT: thread Attitude_Heading_Thread.Attitude_Heading_Thread_Impl;

    connections
      AHTtoAHP: port AHT.AH -> AH;
  end Attitude_Heading_Process.Atttude_Heading_Process_Impl;

  system implementation
      Attitude_Heading_System.Attitude_Heading_System_Impl
    subcomponents
      AHP: process Attitude_Heading_Process.Atttude_Heading_Process_Impl;

    connections
      AHPtoAHS: port AHP.AH -> AH;
  end Attitude_Heading_System.Attitude_Heading_System_Impl;

  thread implementation
      Attitude_Heading_Thread.Attitude_Heading_Thread_Impl
  end Attitude_Heading_Thread.Attitude_Heading_Thread_Impl;

end AHS;
```

A.10 Flight Management System (FMS)

```
package FMS
public

  data Flight_Management_Data
  end Flight_Management_Data;

  process Flight_Management_Process
    features
      FM: out data port Flight_Management_Data;
  end Flight_Management_Process;

  system Flight_Management_System
    features
      FM: out data port Flight_Management_Data;
  end Flight_Management_System;
```

```
  thread Flight_Management_Thread
    features
      FlowPort: out data port Flight_Management_Data;
  end Flight_Management_Thread;

  process implementation
      Flight_Management_Process.Flight_Management_Process_Impl
    subcomponents
      FMT: thread Flight_Management_Thread.Flight_Mangement_Thread_Impl;

    connections
      FMT2FM: port FMT.FlowPort -> FM;
  end Flight_Management_Process.Flight_Management_Process_Impl;

  system implementation
      Flight_Management_System.Flight_Management_System_Impl
    subcomponents
      FMP: process
      Flight_Management_Process.Flight_Management_Process_Impl;

    connections
      FMP2FM: port FMP.FM -> FM;
  end Flight_Management_System.Flight_Management_System_Impl;

  thread implementation
      Flight_Management_Thread.Flight_Mangement_Thread_Impl
  end Flight_Management_Thread.Flight_Mangement_Thread_Impl;

end FMS;
```

A.11 Navigation System (NAV)

```
package NAV
public

  data Navigation_Data
  end Navigation_Data;

  process Navigation_Process
    features
      NAV: out data port Navigation_Data;
  end Navigation_Process;

  system Navigation_System
    features
      NAV: out data port Navigation_Data;
  end Navigation_System;

  thread Navigation_Thread
    features
      NAV: out data port Navigation_Data;
  end Navigation_Thread;

  process implementation Navigation_Process.Navigation_Process_Impl
    subcomponents
      NAVT: thread Navigation_Thread.Navigation_Thread_Impl;
```

```
    connections
      NAVTtoNAVP: port NAVT.NAV -> NAV;
  end Navigation_Process.Navigation_Process_Impl;

  system implementation Navigation_System.Navigation_System_Impl
    subcomponents
      NAVP: process Navigation_Process.Navigation_Process_Impl;

    connections
      NAVPtoNAV: port NAVP.NAV -> NAV;
  end Navigation_System.Navigation_System_Impl;

  thread implementation Navigation_Thread.Navigation_Thread_Impl
  end Navigation_Thread.Navigation_Thread_Impl;

end NAV;
```

A.12 Primary Flight Display (PFD)

```
package PFD
public

  with FGS;

  process Primary_Flight_Display_Process
    features
      FD: in data port FGS::Guidance_Data;
  end Primary_Flight_Display_Process;

  system Primary_Flight_Display
    features
      FD: in data port FGS::Guidance_Data;
  end Primary_Flight_Display;

  thread Primary_Flight_Display_Thread
    features
      FD: in data port FGS::Guidance_Data;
  end Primary_Flight_Display_Thread;

  process implementation
      Primary_Flight_Display_Process.Primary_Flight_Display_Process_Impl
    subcomponents
      PFDT: thread
        Primary_Flight_Display_Thread.Primary_Flight_Display_Thread_Impl;

    connections
      FD2PFDT: port FD -> PFDT.FD;
  end Primary_Flight_Display_Process.Primary_Flight_Display_Process_Impl;

  system implementation Primary_Flight_Display.Primary_Flight_Display_Impl
    subcomponents
      PFDP: process
      Primary_Flight_Display_Process.Primary_Flight_Display_Process_Impl;

    connections
      FD2PFDP: port FD -> PFDP.FD;
```

```
  end Primary_Flight_Display.Primary_Flight_Display_Impl;

  thread implementation
      Primary_Flight_Display_Thread.Primary_Flight_Display_Thread_Impl
  end Primary_Flight_Display_Thread.Primary_Flight_Display_Thread_Impl;

end PFD;
```

A.13 Throttles (THROTTLES)

```
package THROTTLE
public

  data Throttle_Data
  end Throttle_Data;

  device Throttle
    features
      THROT: out data port Throttle_Data;

  end Throttle;

end THROTTLE;
```

A.14 Yokes (YOKES)

```
package YOKE
public

  data Yoke_Data
  end Yoke_Data;

  device Yoke
    features
      YOKE: out data port Yoke_Data;

  end Yoke;

end YOKE;
```

A.15 Leader Selection (LDS)

```
package LDS
public

  data Leader_Selection_Active
  end Leader_Selection_Active;

  data Leader_Selection_Data
  end Leader_Selection_Data;

  data Leader_Selection_Rank
  end Leader_Selection_Rank;

  thread Leader_Selection
    features
```

```
      LSI: in data port Leader_Selection_Data;
      LSO: out data port Leader_Selection_Data;
      LSA: out data port Leader_Selection_Active;
      LSR: in data port Leader_Selection_Rank;
  end Leader_Selection;

  thread implementation Leader_Selection.Leader_Selection_Impl
  end Leader_Selection.Leader_Selection_Impl;

end LDS;
```

A.16 IMA Platform (IMA)

```
package IMA
public

  bus IMA_BUS
    properties
      Latency => 5 ms .. 8 ms;
  end IMA_BUS;

  data IMA_Data
  end IMA_Data;

  processor PowerPC_250Mhz
    features
      ES: requires bus access;
    properties
      Clock_Jitter => 10 us;
  end PowerPC_250Mhz;

  processor PowerPC_350Mhz
    features
      ES: requires bus access;
    properties
      Clock_Jitter => 10 us;
  end PowerPC_350Mhz;

  system CCM_Fast
    features
      ES: requires bus access;
  end CCM_Fast;

  system CCM_Slow
    features
      ES: requires bus access;
  end CCM_Slow;

  system IMA_Platform
  end IMA_Platform;

  bus implementation IMA_BUS.IMA_BUS_Impl
  end IMA_BUS.IMA_BUS_Impl;

  processor implementation PowerPC_250Mhz.PowerPC_250Mhz_Impl
  end PowerPC_250Mhz.PowerPC_250Mhz_Impl;
```

```
  processor implementation PowerPC_350Mhz.PowerPC_350Mhz_Impl
  end PowerPC_350Mhz.PowerPC_350Mhz_Impl;

  system implementation CCM_Fast.CCM_Fast_Impl
    subcomponents
      PRC: processor PowerPC_350Mhz.PowerPC_350Mhz_Impl;

    connections
      EStoPRC: access ES -> PRC.ES;
  end CCM_Fast.CCM_Fast_Impl;

  system implementation CCM_Slow.CCM_Slow_Impl
    subcomponents
      PRC: processor PowerPC_250Mhz.PowerPC_250Mhz_Impl;

    connections
      EStoPRC: access ES -> PRC.ES;
  end CCM_Slow.CCM_Slow_Impl;

  system implementation IMA_Platform.IMA_Platform_Impl
    subcomponents
      IMA_Bus: bus IMA_BUS.IMA_BUS_Impl;
      A: system CCM_Fast.CCM_Fast_Impl;
      B: system CCM_Fast.CCM_Fast_Impl;
      C: system CCM_Slow.CCM_Slow_Impl;

    connections
      BUStoA: access IMA_Bus -> A.ES;
      BUStoB: access IMA_Bus -> B.ES;
      BUS2C: access IMA_Bus -> C.ES;
  end IMA_Platform.IMA_Platform_Impl;

end IMA;
```

A.17 META Property Set

```
property set META_Properties is
  Not_Collocated: list of reference (system) applies to (system);
  Output_Delay: inherit Time applies to (system, process, thread);

end META_Properties;
```

A.18 PALS Property Set

```
property set PALS_Properties is
  PALS_Id     : inherit aadlstring applies to (system, process, thread);
  PALS_Period : inherit Time       applies to (system, process, thread);

end PALS_Properties;
```

APPENDIX B Mode Logic Overview

The Mode Logic (ML) is a component of the Flight Guidance System (FGS) described in Section 4.4.1.4. As such, it is not described in detail in the FCS system architecture. However, since so many aspects of the FCS depend on the mode logic, this appendix has been included to provide more information about it.

The mode logic determines which lateral and vertical control laws are active and armed. The lateral modes control the behavior of the aircraft about the longitudinal, or roll, axis, while the vertical modes control the behavior of the aircraft about the pitch axis. For example, the Heading Hold (HDG) mode holds the aircraft to a selected heading while the Vertical Speed (VS) mode which holds the aircraft to selected vertical speed.

A mode is said to be *selected* if it has been manually requested by the flight crew or if it has been automatically requested by a subsystem such as the FMS. The simplest modes have only two states, *cleared* and *selected*, as shown in Figure 60. Such a mode becomes active immediately upon selection with its associated flight control law providing guidance commands to the flight director and, if engaged, the autopilot. When cleared, a mode's associated flight control law is non-operational, i.e., it does not generate any outputs.

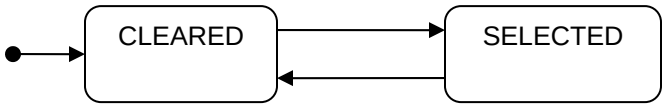


Figure 60 – A Simple Mode

Some modes can be armed to become active when a criterion is met, such as the acquisition of a navigation source or proximity to a target reference such as a desired altitude. Such modes have three states as shown in Figure 61. The two states *armed* and *active* are sub-states of the *selected* state, i.e., when the mode is armed or active, it is also said to be selected. While in the armed state, the mode's flight control law is not generating guidance commands for the flight director or the autopilot, but it may be accepting inputs, accumulating state information, and helping to determine if the criterion for becoming active is met. Once the criterion is met, the mode transitions to the active state and its flight control law begins generating guidance for the flight director and autopilot. Note that the only way to exit the active state is to deselect the mode, i.e., it is not usually possible to revert directly from the active state to the armed state.

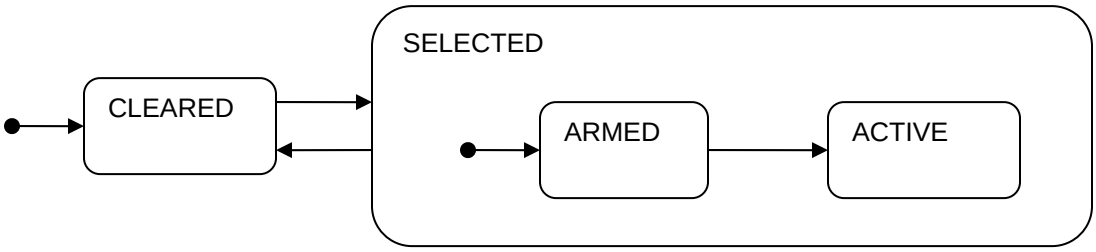


Figure 61 – An Arming Mode

Some modes also distinguish between capturing and tracking of the target reference or navigation source. Such a mode is shown in Figure 62. Once in the active state, such a mode's flight control law first *captures* the target by maneuvering the aircraft to align it with the navigation source or reference. Once correctly aligned, the mode transitions to the *tracking* state in which it holds the aircraft on the target. Both the *capture* and *track* states are sub-states of the *active* state and the mode's flight control law is active in both states, i.e., generating guidance commands for the flight director and autopilot. Note that the only way to exit the active (capture or track) state is to deselect the mode, i.e., it is not possible to revert directly from the track state to the capture state or from the active state to the armed state.

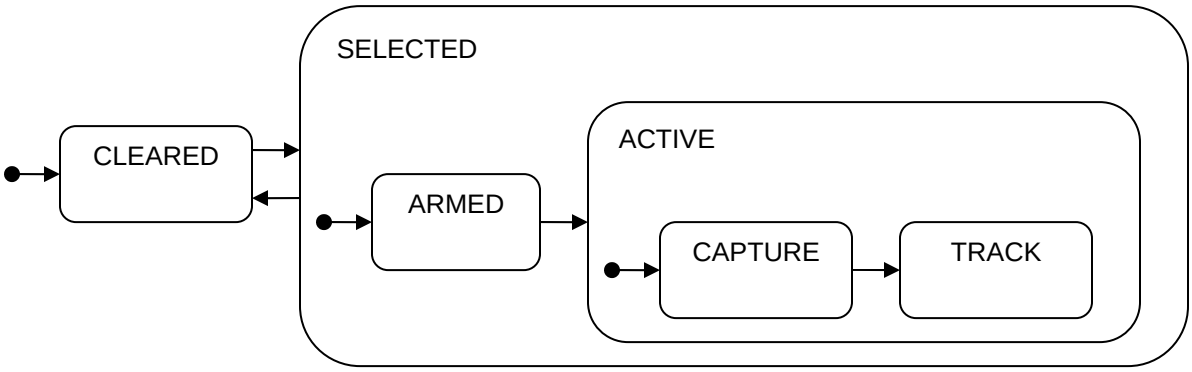


Figure 62 – A Capture/Track Mode

The *mode logic* consists of all the available modes and the rules for transitioning between them. Typical lateral modes include Roll Hold, Heading Hold, Navigation, Lateral Approach, and Lateral Go Around. Typical vertical modes include Pitch, Vertical Speed, Altitude Hold, Altitude Select, Vertical Approach, and Vertical Go Around.

In order to provide effective guidance of the aircraft, these modes are tightly synchronized so that only a small portion of their total state space is actually reachable. For example, to ensure that meaningful guidance is provided to the flight director and autopilot, only one lateral and one vertical mode can be active at any time. For the same reason, if the autopilot is engaged or the flight director is turned on, at least one lateral and one vertical mode must be active. Other constraints enforce sequencing of modes that are dictated by the characteristics of the aircraft and the airspace. For example, vertical approach mode is not usually allowed to become active until lateral approach mode has become active to ensure that the aircraft is horizontally centered on the localizer before tracking the glideslope. These constraints are clearly important to safe flight and can become quite complex.

LIST OF ACRONYMS

AADL	Architecture Analysis and Design Language
ACL2	A Computational Logic for Applicative Common LISP (theorem proving environment)
ADS	Air Data System
AHS	Attitude and Heading System
AP	Autopilot
ATA	Air Transportation Association
BDD	Binary Decision Diagram
CCM	Common Computing Module
CL	Control Logic
CSA	Control Surface Actuator
CSP	Communicating Sequential Processes
CTL	Computation Tree Logic
EDICT	System dependability tool suite developed by WWTG
FCI	Flight Control Interface
FCS	Flight Control System
FD	Flight Director
FGS	Flight Guidance System
FMS	Flight Management System
GUI	Graphical User Interface
IMA	Integrated Modular Avionics
KIND	K-Induction (model checker developed by University of Iowa)
LCM	Least Common Multiple
ML	Mode Logic
MTBF	Mean Time Between Failures
NFF	No Fault Found
NuSMV	Symbolic model checker developed by IRST, Carnegie Mellon, and University of Trento
OSATE	Open Source Architectural Tool Environment
PALS	Physically Asynchronous Logically Synchronous
PFD	Primary Flight Display
PSL	Property Specification Language
PVS	Prototype Verification System (theorem prover developed by SRI International)

REM	Requirements Engineering Management
RTOS	Real Time Operating System
SMT	Satisfiability Modulo Theories
SysML	System Modeling Language