Software Engineering Institute

A Virtual Upgrade Validation Method for Software-Reliant Systems

Dio de Niz
Peter Feiler
David P. Gluch
Lutz Wrage

June 2012

TECHNICAL REPORT
CMU/SEI-2012-TR-005
ESC-TR-2012-005

Research, Technology, and System Solutions Program
ttp://www.sei.cmu.edu

CarnegieMellon

This material is based upon work funded and supported by the U.S. Army Program Executive Office, Aviation of the U.S. Aviation and Missile Research, Development, and Engineering Center of the U.S. Department of Defense and under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Army Program Executive Office, Aviation of the U.S. Aviation and Missile Research, Development, and Engineering Center or the United States Department of Defense.

This report was prepared for the

SEI Administrative Agent
ESC/CAA
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

.

Table of Contents

List of Figures

List of Tables

Executive Summary

The work presented in this report was performed by the Carnegie Mellon® Software Engineering Institute (SEI) for the Army Strategic Software Improvement Program (ASSIP) and sponsored by the Army Program Executive Office (PEO) Aviation. This report presents a Virtual Upgrade Validation (VUV) approach to improving design quality and confidence in qualification through testing for military systems impacted by computer platform upgrades. This approach uses architecture-centric, model-based analysis to identify system-level problems early in the upgrade process to complement established test qualification techniques. For purposes of this report, the authors focus on changes to the computer platform consisting of processor, network, operating system, or runtime infrastructure.

Helicopters and airplanes in military use today are operational well beyond their original life spans and typically are facing multiple platform upgrades as part of technology refresh cycles. Changes to the computer platform tend to be particularly risky because the embedded software makes many assumptions about the computer system. For example, software may have been developed for a federated architecture in which each software component is assumed to run on a dedicated, special processor using a cyclic executive as its runtime executive. The static nature of the task execution order may not be guaranteed on other computer platforms, affecting the execution order and timing. The emergence of the Integrated Modular Avionics (IMA) architecture provides the benefit of increased flexibility for growth of mission capability by utilizing a distributed computer system as a common computing platform. However, migration to this computer resource can have side effects not anticipated by the original embedded software application. For example, applications originally scheduled using a cyclic executive may now execute based on preemptive scheduling paradigms. As a result, the various control systems in the aircraft may encounter latency jitter and race conditions, due to nondeterministic sampling, that are difficult to detect through testing techniques. In one such case, the pilot experienced random blurring of the tracking symbol on his display screen due to latency jitter, which was traceable to nondeterministic sampling under certain processor load conditions [Feiler 1998]. This example illustrates that even planned upgrades to well-known standards-based architectures, such as Aeronautical Radio Incorporated (ARINC)653, can have impactful, unintended side effects.

The U.S. Army has traditionally qualified systems and components by similarity, analysis, test, demonstration, or examination. Furthermore, current test approaches to achieving confidence in systems' airworthiness for the U.S. Army are based on traditional federated avionics systems [Boydston 2009]. The most common approach to dealing with platform change today is to port the code to the new platform and regression test exhaustively. Testers hope that the regression tests provide sufficient coverage for discovering time-sensitive faults as observable defects.

The migration to IMA architectures, the exponential growth in software size and complexity, the increased role of software in the system, and the increasing pace of changes have introduced new hazards [Leveson 2004] that the current "build then test" approach struggles to detect—leading to testing until budgets are exhausted or the testers have run out of testing time. While extensive

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

regression testing is a key part of the verification process for platform changes, regression testing alone is not effective in achieving the desired level of confidence when embedded software is executing in an integrated system.

As a recent study of best practices and the state of the art has concluded, advances in architecture research offer a way of addressing this problem [Feiler 2008b]. The VUV method, presented in this report, uses an industry standard modeling notation, the SAE International[1] Architecture Analysis and Design Language (AADL), and leverages its well-defined semantics in a model-based approach to analyze the impact of platform-related changes [SAE 2004].

The value of the VUV approach is that it helps military programs improve design quality and testing efficiency, and it enables the early discovery of problems for platform-related software changes. The VUV approach improves the quality of the design because it models the *specific* changes related to the platform change, rather than involving testers who run a suite of regression tests until they feel they have probably covered everything. This approach increases confidence in the design changes and prevents unintended side effects from popping up in the test phase. The VUV method improves the efficiency of qualification testing in this way: an architecture model that is developed for analysis and evaluation early in the software life cycle incrementally evolves to reflect platform upgrades. Developers can thus analyze models' upgrade alternatives at design time, rather than waiting for an implementation to be tested on the new platform. Finding problems early reduces rework, which shortens the testing qualification cycle and reduces development cost and schedule [Feiler 2009a].

The VUV method has been applied in a pilot project to analyze the impact of a platform upgrade on the software for an Apache helicopter. The application of the VUV method and the findings of this pilot project will be the subject of a separate report.

[1] SAE international was formerly known as the Society of Automotive Engineers.

Abstract

This report presents a Virtual Upgrade Validation (VUV) method to improve design quality and confidence in qualification through testing for military systems impacted by computer platform changes. This approach uses of architecture-centric, model-based analysis to identify system-level problems early in the upgrade process to complement established test qualification techniques. For purposes of this report, the authors focus on changes to the computer platform consisting of processor, network, operating system, and runtime infrastructure. They describe the VUV method steps and introduce the Architectural Dependencies Catalog, which provides guidance for modelers on which aspects of the system to model and how to model them. The report also provides a history and overview of the Architecture Analysis and Design Language standard, which is used with the VUV method.

1 Background

This section provides background on the SAE International[2] Architecture Analysis and Design Language (AADL) standard, which is the standard of the proposed Virtual Upgrade Validation (VUV) method. This section provides a brief history of the standard, an overview of the standard, some research work by the Carnegie Mellon® Software Engineering Institute (SEI) that feeds into the proposed method, and industry initiatives that are utilizing the AADL standard.

In the 1990s, the Defense Advanced Research Projects Agency (DARPA) invested in software architecture research because it recognized the need to understand increasing system complexity in terms of software and hardware component interactions. During that time, several architecture description languages emerged; these had well-defined semantics to support quantitative analysis of a range of operational quality attributes.[3] In particular, MetaH was developed by the Honeywell Technology Center for use on avionics systems and for the first time applied to a missile guidance system at the Army Missile Research, Development, and Engineering Center (AMRDEC) in 1994. The success of this research, combined with the need felt by the Avionics/Aerospace industry, resulted in the development of the SAE AADL as an international industry standard. Following its initial publication in November 2004, the standard underwent revision in January 2009 and January 2011, based on feedback from several industrial pilot projects [SAE 2004, 2009, 2011].

The focus of the AADL standard is to support architecture modeling, quantitative analysis, and validation of embedded software systems. In particular, AADL defines a set of concepts with well-defined semantics for describing the task and communication architecture of the embedded software, the computer system platform, and the interface to the system and its environment, such that the operational quality attributes that are crucial for embedded real-time system (such as timing, throughput, safety, and reliability) can be analyzed. Furthermore, the standard suite includes a standardized interchange format that supports the interfacing of various analysis tools and exchange of AADL models between different development teams such as the system integrator and its suppliers. Because AADL is an embedded-system-specific architecture description standard with well-defined semantics, developers do not interpret AADL models differently and thus avoid communication mismatches.

Due to continued interest in learning the reasons for system failure, in 2007 and 2008 the SEI investigated the following question: “Why do system-level failures still occur despite our best design methods and fault tolerance techniques being deployed in systems?” In this study, the SEI examined several system-level failures in a variety of safety-critical systems in the aviation and space domain. Feiler identified four architectural root cause areas for these failures [Feiler 2009b]:

1. end-to-end flow of time-sensitive data

[2] SAE International was formerly known as the Society of Automotive Engineers.

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

[3] Operational quality attributes (OQAs) describe nonfunctional properties of the software system, such as performance, safety, and reliability, as they relate to runtime concerns. OQAs will be discussed later in this report.

2. distributed communicating of state machines
3. virtualized resources
4. resource availability

The study also investigated an appropriate architecture-modeling notation and analytical frameworks that support discovery of such problems through an architecture-centric, model-based approach early in the development and upgrade process [Feiler 2010, Feiler 2008a, de Niz 2008b, Hansson 2008, Mohan 2009]. The information gained in these studies was central to developing the method documented in this report.

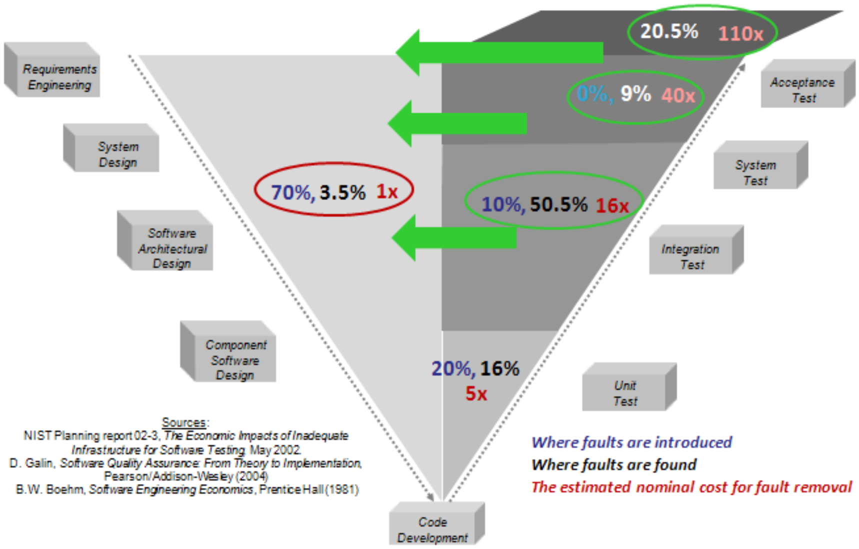
Recent studies confirm that a paradigm shift towards analysis and formal validation at the architecture level to complement testing must occur to meet the challenges of time-sensitive, software-reliant systems with high safety and reliability demands:

- General Accounting Office (GAO) Space-Based Software Study highlighting the reality of more testing than planned (exhausting versus exhaustive testing) due to the increasingly complex interactions between system components [GAO 2008]
- NASA Software Complexity Study on flight software growth and complexity, and the need for integration of fault prevention, detection, and containment with nominal system operation [NASA 2009a]
- Leveson Study on the role of software in spacecraft accidents [Leveson 2004]
- National Research Council (NRC) Study by the Committee for Certifiably Dependable Software Systems addressing the issue of sufficient evidence for software for dependable systems through analysis and formal validation [Jackson 2007]

The Army Strategic Software Improvement Program (ASSIP) organization became interested in AADL in 2007. ASSIP formed an Integrated Product Team to research problems and solutions for real-time, safety-critical, embedded (RTSCE) systems and invited the SEI to give a presentation on the benefits of AADL. In that presentation, Peter Feiler stated that AADL modeling could help in finding problems early in complex, embedded software.

The SEI then followed with an ASSIP-funded study of the state of best practice and emerging technology to support model-based engineering of embedded systems [Feiler 2008b]. Recognizing the need for predictive architectural analysis, a number of industry initiatives using AADL have invested in the development of an architecture-centric, model-based approach to engineering their software-reliant systems. This trend has culminated in an aerospace industry-wide, four-phase practice improvement initiative called System Architecture Virtual Integration (SAVI). The U.S. Army has been a participant in this initiative, which in 2009 completed the proof of concept phase [Feiler 2009a]. In addition, NASA funded an SEI AADL IV&V study that provides support artifacts such as templates for analysis reports and process scripts [NASA 2009b].

Data from member companies of SAVI, such as Boeing, Airbus, and Lockheed Martin, show that the size of source lines of code (SLOC) has doubled every four years and it is predicted that by 2014 the cost of 27 million SLOC of software will exceed $10 billion. Industry data shows that 70% of faults are introduced early in the life cycle, while 80% of faults are not caught until integration test or later. These faults carry a repair cost of 110 times or more at system operation than if they had been caught early in the life cycle. If we can discover a portion of these late system-level faults earlier in the development process, we can expect considerable cost savings [Feiler 2009a]. Figure 1 shows percentages of fault introduction, discovery, and cost of repair [Feiler 2009a].



Sources:
NIST Planning report 02-3, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, May 2002.
D. Galin, *Software Quality Assurance: From Theory to Implementation*, Pearson/Addison-Wesley (2004)
B.W. Boehm, *Software Engineering Economics*, Prentice Hall (1981)

Figure 1-1: Late Discovery of System-Level Problems

The benefits described above, along with findings from other workshops and reports, ultimately led to ASSIP's decision to support this work on behalf of the Army Program Executive Office Aviation (PEO-AVN). The work consists of the development of the VUV method, the subject of this report, and its pilot application to an Army aviation system.

The Apache Block Upgrade III (AB3) was the chosen candidate for an initial VUV case study. This study illustrates the value of model-based engineering (MBE) with AADL in the context of architecture assessments and evaluations; models developed during the MBE activities of an Architecture Tradeoff Analysis Method® (ATAM®) assessment of AB3 were reused in the VUV case study. This MBE/ATAM case study is the subject of the SEI special report SEI-SR-021-2008 to the Army, titled *An AADL-Based Analysis of Apache*, in December 2008.

The AADL, along with compatible MBE tool suites such as the SEI Open-Source AADL Tool Environment (OSATE), provides the means to investigate ATAM risk themes through realistic modeling and analyses of complex architectures and the means to explore and evaluate various risk reduction/mitigation paths and multiple solutions to prioritized stakeholder scenarios. AADL MBE provides the system project office or prime contractor with powerful and cost/schedule-effective methods to model and evaluate multiple design solutions early in the design—reducing technical risk, high cost, and schedule delay due to late discovery of system-level problems.

The VUV method focuses the AADL-based MBE approach used during the ATAM specifically to investigate potential issues arising from a migration to a new computing platform during a technology refresh. The case study application of VUV to AB3 allows us to illustrate the ability to

® Architecture Tradeoff Analysis Method and ATAM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

discover, through analysis of an AADL model, system-level problems that are traditionally discovered through prototype implementation at much higher cost. In addition, both the AB3 Program Office and its prime contractor have been utilizing the AB3 AADL model, study data, and available MBE tool suites to perform their own studies of the AB3 architecture.

2 Introduction

This report is the first in a series of three reports developed by the SEI for the ASSIP and sponsored by the Army PEO AVN. This first report introduces readers to the VUV method, which is an approach to analyzing the impact on software due to platform-related changes. The second report is a case study that applies the method described in this report to a successfully fielded Army avionics system. The third report summarizes the outcome of the case study and provides an assessment of the value of the VUV method to the program.

The purpose of this report is to document a method for analyzing the impact of changes to software due to platform changes. This is a serious concern for Army avionics in large part because aircraft remain in use for a very long time, often much longer than anticipated. Therefore, it is likely that over a 20+-year life span, part of the platform (hardware, network, or underlying architecture) will require replacement multiple times. For the purposes of this report, the authors define platform changes as changes to hardware, network, and operating system (i.e., changes in the computational infrastructure).

The migration to Integrated Modular Avionics (IMA) architectures, the exponential growth in software size and complexity, the increased role of software in systems, and the increasing pace of changes have introduced new hazards [Leveson 2004] that the current “build then test” approach struggles to detect before operation—leading to testing until budgets are exhausted [Boydston 2009].

This “build then test” approach to dealing with the impact of platform change today is to port the code to the new platform, compile it, fix any problems caused by the hardware replacement, and then regression-test exhaustively. After the code is ported, testers run extensive suites of regression tests in an attempt to identify problems during the test phase rather than in flight. What the testers are really trying to do is to see if they can hammer the system with test cases in hopes that internal problems show up as observable defects. While widespread in practice, this approach is extremely resource-intensive and struggles to discover sources of system-level failure, such as race conditions, unintended resource contention, and unexpected latency jitter. The result can be rework or, worse, undetected problems that unpredictably show up in operational flight.

One of our goals for the VUV method is to discover errors that testing might miss. Rather than calling for a suite of regression tests to be run until testers feel they have probably covered everything, the VUV approach models the *specific* architectural changes related to the platform change. The use of modeling increases confidence in the design and prevents unintended side effects from popping up in the test phase.

Another benefit from the VUV method that improves design quality for embedded systems changes is that the AADL allows for very precise modeling of the environment on an extremely fine-grained scale, as needed to examine root cause areas. The VUV method uses the AADL to develop models that describe the dependencies of the embedded application software on the computer system with well-defined semantic meaning that enables automated analysis. The semantics of the language guide the designer to refine architectural abstractions as necessary to identify omissions and mismatches in these dependencies. Both the automated analysis and the precise semantics enabled by AADL models have been recognized by the academic research community [Hugues 2008, Delange 2008, Sokolsky 2006, 2009] and by industry circles [Feiler 2009a, LaCerte 2007, Conquet 2008, Casteres 2008] as key strategies in reducing errors. Precision in modeling is essential for analyzing changes at the embedded software layer because small changes can have significant ramifications.

Another problem with relying on regression testing alone to analyze the impact of platform-related change is that testers have to wait until the system is built to test it. This is too late, according to engineers at the U.S. Army AMRDEC Aviation Engineering Directorate (AED). Boydston describes the situation [Boydston 2009].

> *Waiting until Preliminary Design Review (PDR) is too late to start addressing these considerations and may require redesign later or onerous testing during qualification that is costly in monetary and schedule terms.*

The model-based approach of VUV not only addresses the impact of upgrades analytically, but also leads to continued use of the resulting system architectural models. The idea is that once you have invested in developing a model, not only is it available early in the software life cycle, at design time, so you can find problems earlier, it is also reusable over multiple platform upgrades. The objective is to lower the cost and schedule impact of errors by reducing rework and the likelihood of schedule overruns, thereby minimizing the number of times tests have to be applied to the upgrade.

The three key elements of the VUV method are (1) a description of the VUV method steps, (2) an Architectural Dependencies Catalog (ADC) and, (3) strategies for modeling the existing application system and computer platform as well as the new platform.

This report has the following structure. Section 3 provides the context for introducing the method, including an overview of root cause of system faults. Section 4 provides an overview of the VUV method. Section 5 illustrates the use of the VUV method by applying it to an example and elaborating its steps. Section 6 explains the purpose of the ADC as an important tool to the method and presents its initial content. Section 7 discusses modeling strategies for applying the method in modeling the application architecture, addressing key operational quality attributes in the model, and determining resource availability in a computer platform. Section 8 outlines broader applicability of the VUV method.

3 Challenges in Software-Reliant System Upgrades

In this section, we explore issues pertaining to software-reliant system upgrades. We begin with an overview of embedded system architecture. We follow this overview with a discussion of the importance of understanding and managing the impact of change on OQAs. Next, we discuss how mismatched assumptions—stemming from how an engineer believes a system works versus how it actually works—can lead to disastrous results. Because assumption mismatch can have such devastating ramifications, we focus the VUV method to address these areas of mismatched assumptions.

We wrap up Section 3 with a summary of four root cause areas of mismatched assumptions and the AADL concepts available to represent them. The root cause areas are incorporated into the VUV method to help modelers identify and prevent common assumption-related system failures through the ADC.

3.1 Brief Introduction to Software-Reliant System Architecture

Large-scale, software-reliant, mission-critical systems (such as aircraft, transport vehicles, or robotic systems) are distributed real-time embedded systems. They consist of three major architectural components: (1) the mission system and the environment in which it operates; (2) the computer platform in the form of networked processors; and (3) the embedded application software. Figure 3-1 illustrates these elements.

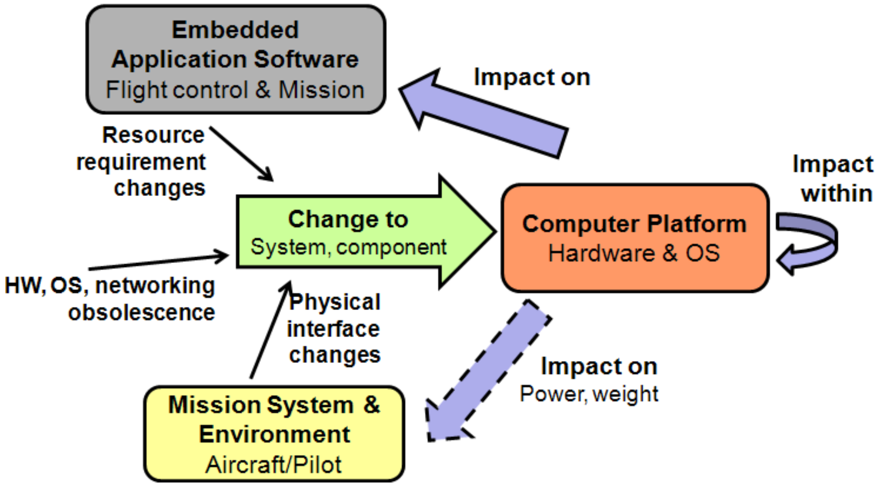

Embedded Application Software
Flight control & Mission
Impact on
Resource requirement changes
Impact within
Change to
System, component
Computer Platform
Hardware & OS
HW, OS, networking obsolescence
Physical interface changes
Impact on
Power, weight
Mission System & Environment
Aircraft/Pilot

Figure 3-1: Three Elements of Software-Reliant Mission-Critical Systems

Figure 3-1 also illustrates the focus of this report: the impact of the computer platform changes (upgrades) on the embedded application software and on the mission system. The mission system has an architecture that system engineers develop to meet mission requirements. Typically, computer engineers design the computer platform architecture to interface with the mission system through sensors and actuators and to provide the resources to execute the embedded mission application software. The embedded application software is responsible for processing observations of the environment and controlling the operation of the physical mission system, including management of physical system component failures. The embedded software system is deployed on the computer platform in ways that allow it to fulfill its requirements, such as performance, safety, and reliability, despite possible computer hardware failures.

3.2 Importance of Operational Quality Attributes (OQAs)

In this section, we discuss the importance of understanding the impact of platform changes on current and desired system qualities (e.g., performance, fault tolerance, and safety). In software engineering, we capture requirements in terms of functional requirements and quality attribute requirements (also often referred to as *nonfunctional requirements*). Functional requirements describe the work a software system does, such as computing a value. Quality attribute requirements describe qualities of the software system such as performance, safety, and reliability. We refer to quality attribute requirements focused on runtime concerns as *operational quality attributes* or OQAs, as explained in Section 1. OQAs are central to the VUV method because we must understand the impact of proposed platform changes on the OQAs.

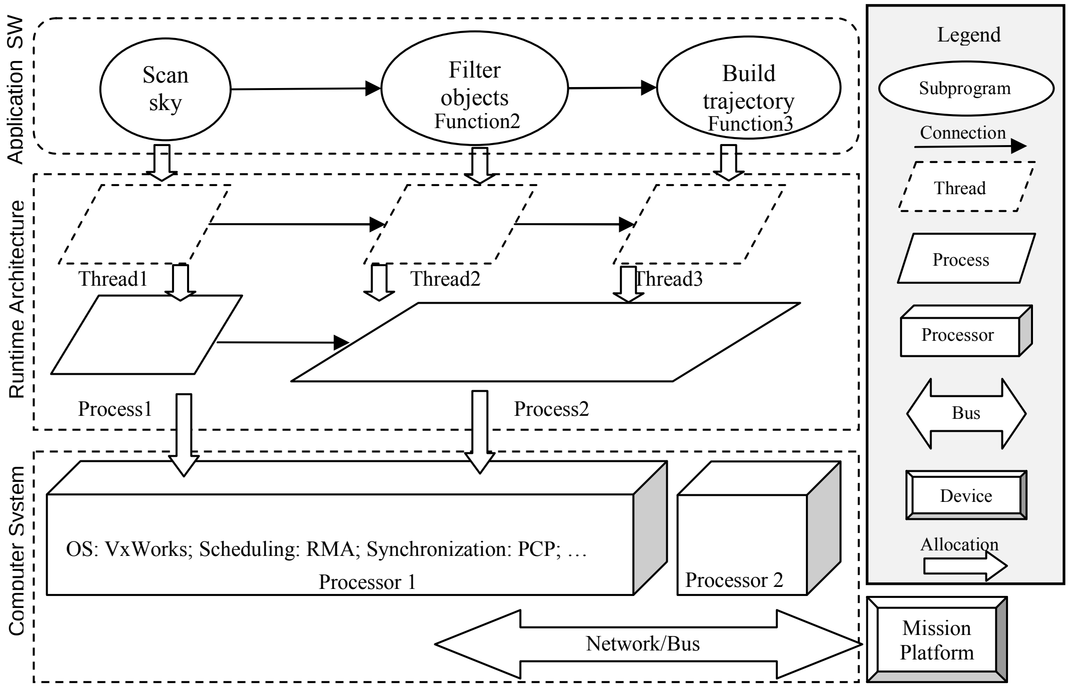


Figure 3-2: Example of an Embedded Software System

The graphic in Figure 3-2 represents a radar system that must provide updates to the position of objects in the sky with enough frequency to allow a timely reaction to them. This graphic represents the application software functions as three threads that are grouped into two separate processes in the system’s runtime environment and shows how these software elements are bound to (executing on) a single processor. For the system to meet its timeliness requirements, the threads must complete their execution by their specified deadline and not exceed their allotted processor execution time. The scheduling algorithm used in the operating system determines the particular schedulability analysis technique. For example, the rate-monotonic analysis theory

[Klein 1993] can analytically determine whether all threads meet their deadline under a fixed priority preemptive scheduler, assuming the thread priority is assigned according to the execution rate.

The radar system example illustrates the importance of using an architecture notation that supports component concepts specific to embedded systems with well-defined semantics and properties to support the validation of OQAs through analysis of an architecture model. To validate OQAs at a specific point in the system development, the architecture model of the computer hardware and software designed for the fielded system (i.e., the operational architecture) must be precise enough to permit quantitative assessment. This quantitative assessment is refined as better data describing the architecture and performance measures are provided.

Modeling approaches for embedded systems must also allow for modeling at different levels of abstraction. The individual characteristics of the different parts of the runtime architecture (e.g., worst-case execution time [WCET] of tasks) are combined with characteristics of the computer system (e.g., processor speed and scheduling protocol) to provide a system-wide OQA (e.g., in this case, the schedulability of the task set). At the same time, the software functions (software development units) are built to, or assume specific services are provided by, the computer platform. For instance, when the software sends messages between processors, the programmer may assume a reliable communication mechanism. In the final system, these assumptions are typically implicit and do not form part of the description in the final software artifacts (e.g., source code). Analysis approaches must allow for analysis of interactions of software components in terms of the software's task and communication architecture as it is deployed on the computer platform and interfaces with the physical mission system to verify requirements and validate assumptions.

The bottom line is that a good understanding of OQAs is important because violated OQAs are a key contributor to system failure or design-related problems. For this reason, OQAs are central to the VUV method.

3.3 Implications of Mismatched Assumptions

Another key contributor to system failure is mismatched assumptions. Mismatched assumptions can lead to disastrous circumstances and are therefore a central target of the VUV method. Mismatched assumption failures occur when engineers make certain assumptions about how a system will operate or be used and one or more of the assumptions are wrong. It is impossible for one engineer to understand a large, complex system in precise detail. So different types of engineers focus on different aspects of the system, often making assumptions about how other aspects of the system impact those the engineer is focused upon.

Figure 3-3 illustrates the different perspectives and focus areas for the various types of engineers involved in developing embedded systems. A system engineer focuses on the capabilities of the system to be built and its interactions with its environment, including the system user or operator, as well as its decomposition into major subsystems. The system engineer makes assumptions about how the operator interacts with the system (e.g., an assumption that the driver is inside the train when the doors are closing). A control engineer focuses on how the physical system is to be controlled (i.e., on the interaction between the system under control and the control system) in order to achieve objectives and may make assumptions about the response of the physical system to control commands. Application developers translate the control algorithms into software implementations. In writing the software, the application developer makes assumptions about the size of data values when choosing an 8- or 16-bit representation for variables, or the measurement unit associated with the data value. An embedded software system engineer makes decisions about the runtime architecture of the embedded software in terms of concurrently executing tasks and their way of communicating with each other as well as decisions about the distribution of these tasks on a networked computer platform. In the process, assumptions are made about concurrent execution and mutually exclusive access to data or hardware and about software executing on physically separate hardware to achieve redundancy. In Section 3.4, we identify four root cause areas of system-level faults caused by mismatched assumptions in the embedded software.

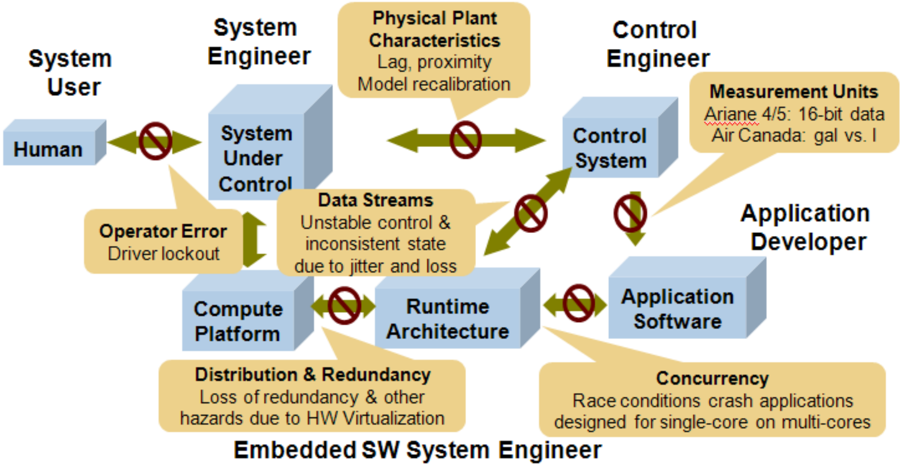


Figure 3-3: Implications of Mismatched Assumptions

In its research, the AADL team made a key finding: mismatched assumptions between the mission platform, the computer system, and the embedded application software contribute significantly to increased failures and delays in system delivery. To illustrate, we present several examples of mismatched assumptions from an earlier report [Feiler 2009a].

After years of development of the F/A-22 fighter plane, flight tests began in late 1997, but the aircraft still experienced serious avionics instability problems as late as 2003. According to testimony from the GAO, “The Air Force told us avionics have failed or shut down during numerous tests of F/A-22 aircraft due to software problems. The shutdowns have occurred when the pilot attempts to use the radar, communication, navigation, identification, and electronic warfare systems concurrently” [Li 2003]. The shutdowns were due to the use of asynchronous clocks for processors, insufficient control of state, and the resulting set of inconsistent states across the processors.

In another example, the flight software for a fighter was migrated to an IMA architecture. The application software was originally implemented as a cyclic executive with periodic sampling tasks. When the application was ported to a rate-monotonic, fixed-priority, preemptive scheduler, the display showing tracked objects randomly blurred. The transfer of target data from the sensor to the display, which predictably took four frames in the original system, now varied between four and eight frames due to preemption and depending on the workload. The display of time-sensitive data was affected by a change in the scheduling protocol and the use of a nondeterministic sampling communication scheme. This showed itself as an oscillating target symbol of the tracked object [Feiler 1998].

In 2008, a Qantas flight unexpectedly dropped as much as 650 feet multiple times within a few minutes [Wikipedia 2008]. A fault in one of three Air Data Inertial Reference Units (ADIRUs) caused the unit to supply incorrect data to other aircraft systems and led to automatic disengagement of the autopilot, false stall warnings, and loss of altitude information on the pilot display. With the autopilot off, the primary flight control computer still received false data two minutes later from an ADIRU and commanded a major pitch downward. A failure in one component of a triple-redundant unit caused an operational mode change and operational response to a data stream by another subsystem with no recognition of its faulty nature (i.e., the subsystem assumed a correct data stream due to the redundant nature of the source).

3.4 Root Cause Areas of System-Level Faults

Analysis of the problem examples described in the previous section and additional cases led us to the identification of four root cause areas of system-level faults [Feiler 2009b, 2010]:

1. end-to-end flows of data streams
2. distributed communicating state machines
3. virtualized resources
4. resource availability

In this section, we describe the elements of each of the four areas and explain the AADL concepts that support the capture and analysis of relevant system information for assessing each of these areas in an architecture model. The elements of each root cause area have been incorporated into the ADC, which is part of the VUV method. We describe the ADC in Section 6.

3.4.1 End-to-End Flow of Data Streams

One root cause of system-level faults is that engineers often look only at a subset of the system. For systems such as control systems, engineers must also look at the flow of data through the system from end to end. A typical example of an application pattern that should be modeled end-to-end is a control system that performs periodic sampled processing of a data stream of sensor readings. Control loops are sensitive to time and respond with unstable behavior if unexpected jitter occurs in the latency of the processed data. Another example of where we recommend end-to-end analysis is in a mission system that processes information about the mission environment to provide reasonably accurate situational awareness. In this case, messages must be processed in a time-consistent manner to present a cohesive whole.

To make these end-to-end flows ready for analysis, we must explicitly represent them in an architecture model of the system as the flow of information between components along connections. This involves identifying all the components involved in the flow and their connections. For such analysis to be performed early in the life cycle, it is essential that flows through components can be represented abstractly by flow specifications from input to output without necessitating looking inside the implementation of the component. The AADL provides constructs to represent both end-to-end flows and component flow specifications annotated with properties for analysis at different levels of fidelity.

Each processing step can affect the handling of such data streams, and the runtime architecture of the embedded software can greatly affect the timing of data processing and transfer. Therefore, it is essential to document assumptions made about such data stream characteristics, which fall into the following categories:

- characteristics of data such as the application data type (e.g., external temperature), its base type (e.g., 8-bit or 16-bit signed integer to represent temperature values), and measurement unit (e.g., temperature expressed in terms of degrees Celsius or Fahrenheit). Two units exchanging such data might mistakenly make different assumptions about the data representation.

  Support for analysis with AADL: These characteristics are affected by changes and additions to mission capabilities. An example of a mission capability change is the addition of a device that can provide position information with higher precision. However, this information must be propagated throughout the system in order to draw benefits. These characteristics are represented as property values on the data types associated with ports and are checked along port connections. The OSATE toolset supports such consistency checking [SEI 2009b]. Its utility has been demonstrated in the Aerospace Vehicle Systems Institute (AVSI) System Architecture Virtual Integration (SAVI) proof of concept project [Feiler 2009a].

- data stream characteristics such as lack of missing stream elements, complete transmission of all stream elements, and acceptable limits in value changes between elements of the stream. The characteristics of data streams are affected by

  - whether the application performs sampled processing of input or message processing driven by the arrival of messages
  - faults in the processing steps
  - faults in the protocols used in communicating the data between the processing steps

  Support for analysis with AADL: Since data and information streams flow through port connections, the data stream characteristics can naturally be represented by properties on data and event data (message) ports. Incoming ports record assumptions they make about data they receive, while outgoing ports specify the output pattern of the data stream they generate. The OSATE toolset includes a simple analysis capability that identifies mismatches between data stream characteristics [SEI 2009b].

- time sensitivity of the data in the form of age, response time, and latency jitter. Contributors to these related terms are not only the transfer time through the physical device and wire, processing time, queuing delay, and sampling delay, but also elements of the software runtime system such as preemptive task scheduling, rate group optimization, concurrency, partition scheduling, and protocol execution semantics.

  Support for analysis with AADL: The mapping of the application software into a runtime architecture in terms of threads and port connections and its binding to computer hardware and runtime infrastructure determine the contributors to latency and response time variation. Flow specifications and end-to-end flows can have property values to reflect requirements, estimates, and actual data. The OSATE toolset includes an end-to-end latency analysis capability that takes into account latency contributions by various runtime system mechanisms [Feiler 2007a, 2008a].

3.4.2 Distributed Communicating State Machines

A second root cause problem related to system-level failure is that engineers make assumptions about their runtime systems and computer hardware when implementing communicating state machines. Examples of such distributed state machine communication are the replicated logic to manage the reconfiguration of redundant systems, the coordination of operational modes in different subsystems of a mission-critical system, and application protocols, such as hand shaking, to manage the release of mission assets such as weapons. We have identified the following types of state machine interactions:

- coordination of state machines. Replicated state machines must show identical state behavior, mirrored state behavior, or coordinated state behavior. Implementations of such state behavior coordination range from communication of state transition events through a message system or alarm handler to periodic sampling of transmitted state. It is essential to ensure that the state machine logic of a single state machine preserves its behavior when it is replicated and distributed. Different implementations of the state coordination respond differently to different runtime system mechanisms and hardware characteristics. For example, change in scheduling protocol from cyclic executive to preemptive scheduling based on rate-monotonic analysis (RMA) may introduce race conditions. Therefore, it is critical that the assumptions made by the application software are satisfied by the execution platform and still hold after a platform upgrade.

  Support for analysis with AADL: Rockwell Collins and the SEI have demonstrated the ability to validate redundancy logic through model checking. The SEI has prototyped such an analysis capability in the context of the OSATE toolset and included a demonstration in the AVSI SAVI proof of concept (POC) project [de Niz 2009, Feiler 2009a].

- event observations. For control systems, state behavior modeling in Simulink StateFlow often leads to an implementation that samples exchanged state. State changes are observed by sampling state in periodically executing control system threads. Sampling of these time-sensitive states may result in event observation loss and protocol lockup when operating on different processor and network platforms. State logic that works for nominal operation in a synchronous system setting without failure may not hold in actual deployment on a distributed set of processors operating on separate clocks with transient failures of the communication network. For example, message loss in a communication protocol can have catastrophic results when transition events are communicated. Similarly, it is imperative to ensure that variation in time due to jitter or clock drift does not affect the intended transmission of state transition events.

  Support for analysis with AADL: AADL provides well-defined semantics or queuing event and message ports as well as sampling data ports to represent different port-based communication. Binding of threads as well as their communication via connections to elements of the computer platform allow us to check for incompatibility with the services and protocols provided by processors and networks. This allows us to analyze the system architecture to detect potential nondeterminism and loss of information—both through model checking [de Niz 2009] and by identifying the need for double buffering in sampled port implementations [Feiler 2008c].

3.4.3 Virtualized Resources

A third root cause area related to system-level failure is that engineers do not understand the possibly negative side effects that occur when processor, network, and memory resources are virtualized. System architectures use these virtual resources and assume certain guarantees about the physical resource that virtual resources make.

Virtualization of resources takes on many forms. Multiple threads (tasks) executing on the same processor share processor cycles, and one thread’s execution can preempt other task executions and affect their start and completion time. As a result, a task being migrated from a dedicated processor to a shared processor may sample its input at different times rather than the beginning of the sampling period. Migration from a federated architecture to an IMA architecture introduces time and space partitioning where a partition represents a virtual processor. Threads within a partition are scheduled as if they were on a processor, but their start and completion time is affected by the time slot a partition is assigned on a processor. Again the thread sampling time is affected, in this case also by the allocation of partitions to partition schedule time slots.

The AADL virtual bus concept represents virtual channels and protocols. The AADL virtual processor concept can represent time partitions as well as hierarchical schedulers. The assumptions in this root cause area fall into the following areas:

- resource isolation guarantee. Virtual resource concepts of processor partitioning (e.g., Aeronautical Radio Incorporated [ARINC]653) and virtual channels represent apportionments of physical resources as well as information access and fault propagation boundaries that must be validated and enforced.

  Support for analysis with AADL: The AADL process concept represents desired address space protection within a processor. A processor may be decomposed into virtual machines or partitions using the AADL virtual processor concept. It provides both time and space partitioning of the physical processor [SAE 2009]. A processor may not support such address space protection at runtime, in which case dedicated hardware is a common solution.
- redundancy guarantee. Virtualization turns physical redundancy into logical redundancy. In order to ensure reliability and availability, we must guarantee that the deployment of the virtualized resources provides physical redundancy where required.

  Support for analysis with AADL: AADL models include deployment binding of application threads to virtual processors and to processors through binding properties. Similarly, connections are bound to appropriate virtual buses and buses representing virtual channels and physical networks. For application components, the modeler can indicate whether redundant components require deployment on separate physical components. In addition, the need for physical redundancy can be recorded through the Error Model Annex of AADL [SAE 2006] and drive hazard, reliability, or availability analysis [Feiler 2007b].
- fair resource use guarantee. Mixed-criticality applications, such as periodic and event-driven processing, scheduling priorities and load scheduling priorities, multiple security layers,

safety-criticality levels, and redundancy requirements, must use shared resources consistently despite conflicting demands.

Support for analysis with AADL: Resource capacities and resource budgets can be associated with physical resources as well as logical resources and application components. The OSATE toolset includes a resource analysis capability for processor, memory, and network (bus) resources. In the case of safety and security concerns, modeling and tool support ranges from simple safety- and security-level consistency checking along port connections to full-scale security analysis based on the Bell LaPadula model [Hansson 2008], including use of virtual processors and virtual buses [Delange 2009a].

- reference time guarantee. Sampling of data on a dedicated processor is performed relative to the processor clock, while the same software executing as threads in partitions sample the data in terms of virtual time (i.e., the time the application code actually executes relative to other threads and partitions). Similarly, applications may process time-sensitive data by time stamping that assumes a common reference time despite potentially multiple time sources.

  Support for analysis with AADL: Virtual time is implicitly reflected in virtual processor and thread components executing on a shared processor resource according to a scheduling protocol. AADL provides property support for characterizing different scheduling protocols, including ARINC653 partition scheduling. AADL also supports modeling of the fact that different parts of a computer platform may execute on separate clocks through multiple reference time components. Their characterization includes bounds on clock drift [SAE 2009].

3.4.4 Resource Availability

A fourth root cause is that the computer resources as well as physical resources are shared, and concurrent use can lead to resource contention. Therefore, it is necessary to validate assumptions about availability of resources and resource guarantees.

AADL supports modeling of physical and logical resources as discussed in the previous section. Those resources have properties to indicate coarse-grained resource capacities as well as detailed forms of resource capacity specification. For example, a processor has properties for a MIPS[4] budget, processor cycle time, and context switch times between threads and between processes.

The assumptions for resource availability can be categorized as follows:

- undocumented direct and indirect demands on all logical and physical resources by all application and infrastructure units for peak demands

  Support for analysis with AADL: AADL provides component categories of processor, memory, and bus to represent physical computer architectures. Similarly, virtual resources can be represented in AADL models to both require resources as well as provide resource capacity. Application-level resource demands on processors, memory, and networks can be derived from deployment bindings, as demonstrated in the AVSI SAVI POC project [Feiler 2009a].

[4] MIPS stands for Microprocessor without Interlocked Pipeline Stages.

- impedance mismatch of resource demand and capacity, such as high-volume transfers over a bus flooding a low-speed processor with interrupts to handle the traffic, resulting in denial of service and lower-than-expected processor speed

  Support for analysis with AADL: In AADL models, computer hardware components are interconnected via buses through bus access connections. Bus access features of processor, memory, bus, and device components can have properties that indicate maximum contributions to network traffic.

- lack of resource guarantees of shared hardware resources such as delay of low-volume data transfer due to high-volume traffic by a direct memory access (DMA) transfer

  Support for analysis with AADL: An AADL-based analysis capability has recently been developed to extend the analysis capabilities in OSATE to address this resource contention issue [Mohan 2009].

3.4.5 Other Quality Dimensions and Root Cause Areas

The VUV method is not limited to analysis of the four identified root cause areas. As new technologies are deployed in execution platforms, they may uncover additional implicit assumptions by the embedded software application that may be violated. Similarly, new hazards may be introduced that must be addressed in terms of reliability, safety, or security. The VUV method leverages the extensibility of the AADL to permit the addition of properties and sublanguages to accommodate new analytical frameworks. At the same time, the Eclipse-based implementation of the OSATE toolset allows the community to rapidly prototype new analysis capabilities, with over 50 research groups contributing such analysis capabilities.

In addition, some OQAs may be identified as critical, such that their value should be preserved or improved. For certain OQAs, the AADL-model-based engineering community has associated analytical frameworks and prototyped implementations (http://www.aadl.info). Two examples are security [Hansson 2008] and redundancy in support of safety-critical and mission-critical systems [Feiler 2004].

4 Virtual Upgrade Validation (VUV): Method Overview

The objective of applying the VUV method is to understand the impact a computer system upgrade can have on the OQAs of interest for an embedded software system and the relevant system properties associated with those OQAs (i.e., QA concerns[5]). In the VUV method, we discover potential problems by virtually validating an architecture model of the upgraded system before the upgrade is performed. We evaluate solutions to these problems by revising the model of the upgraded system. In effect, we use modeling to identify and mitigate technical risks associated with the upgrade of an embedded, real-time system *before* a single change is made.

The VUV method focuses on the impact of changes to the computer platform. We can expand it to cover the impact of changes to components of the embedded application software and to changes in the physical mission platform (e.g., airframe dynamics) through interaction with system engineering models.

4.1 The VUV Method and the Architecture Dependencies Catalog (ADC)

The modeling effort in VUV helps to keep the cost of upgrade as low as possible by assessing an upgrade's impact before the upgrade begins. The Architecture Dependencies Catalog (ADC) is a collection of tables that provides modeling guidance to aid focus on potential upgrade problem areas (see Section 6). The ADC supports the evaluation of potential impact. The ADC allows the modeler to determine the relevant modeling requirements and assumptions that must be taken into account

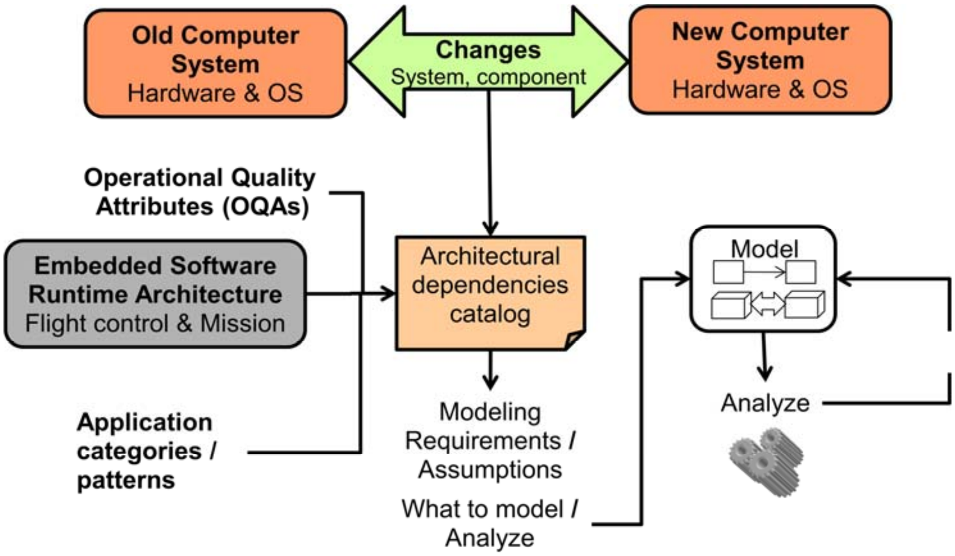

Old Computer System
Hardware & OS
Changes
System, component
New Computer System
Hardware & OS
Operational Quality Attributes (OQAs)
Embedded Software Runtime Architecture
Flight control & Mission
Architectural dependencies catalog
Model
Application categories / patterns
Modeling Requirements / Assumptions
What to model / Analyze
Analyze

Figure 4-1: Focused Modeling via Architectural Dependencies Catalog

[5] Quality attribute (QA) concerns are the properties of a system by which attributes of a system are judged, specified, and measured [Barbacci 1995]. For example, concerns for the performance OQAs include latency, jitter, and throughput.

when evaluating an architectural change from two perspectives: the application category and the OQAs of concern. The modeling requirements help modelers to determine which parts of the system they need to model in the evaluation. Figure 4-1 illustrates this modeling process. The ADC helps identify the prevalent application patterns for each application category. For each application pattern, the ADC allows the modeler to identify the intended component interactions, assumptions made by the pattern, and modeling requirements for the application and platform. The modeling requirements are expressed in terms of the application intent and in terms of possible alternate implementations. For example, events such as operator button pushes can be communicated as queued events and processed on demand as events arrive, or events can be communicated through a state variable that is sampled by the recipient at a high enough rate to allow recognition of state changes as events.

The ADC allows the modeler to identify the most relevant OQAs and related system properties that can be affected by changes to the computer platform for each of the application categories. For each system property, the ADC indicates the system components, assumptions, and modeling requirements on the application and platform in order to perform the relevant analyses.

4.2 Steps and Artifacts of the VUV Method

The VUV process for discovering and evaluating potential risks of computer system upgrades consists of seven steps, which are listed in Table 4-1.

Table 4-1: Steps of the VUV Method

Step		Description
1	*Describe the Upgrade.*	Explain the modifications, reasons, and objectives of the upgrade.
2	*Describe Relevant Operational Quality Attributes and System Properties.*	Describe and prioritize the OQAs and relevant system properties (OQA concerns) that must be preserved across the changes.
3	*Identify Changes in Computer System.*	Identify changes in the architecture of the old computer system with a focus on the parts that are modified, replaced, or added.
4	*Identify Architectural Dependencies.*	Identify architectural dependencies by utilizing the ADC.
5	*Model and Analyze the Original System.*	Create and analyze a model of the original embedded system architecture with respect to the identified OQAs and concerns.
6	*Model and Analyze the Upgraded System.*	Create and analyze a model of the new embedded system architecture.
7	*Revise the System Upgrade.*	Provide the opportunity to explore alternatives to correct any shortcomings of the upgrade.

The steps as well as the artifacts created in the execution of the VUV method are shown in Figure 4-2. In the figure, the steps are represented as gray rounded rectangles and outputs are represented as blue rectangular icons with a truncated upper-right corner. Requirements, ADC, ATAM results, and other supporting documents used in the method are represented as a single rectangle.

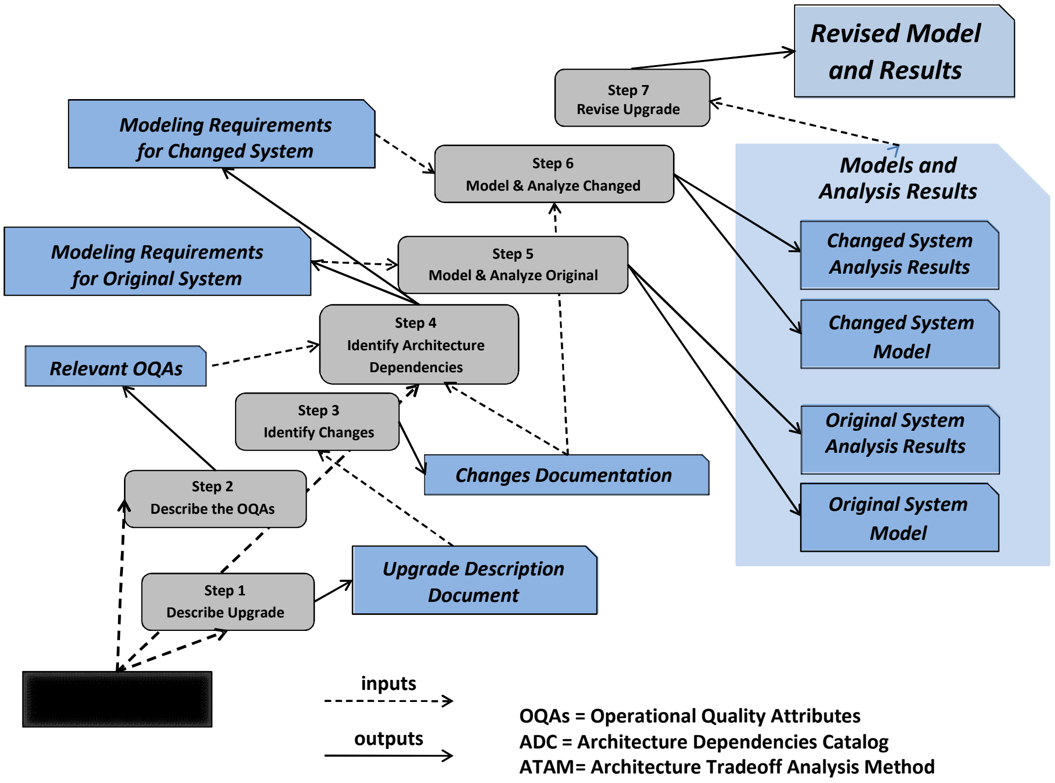

Revised Model and Results
Step 7
Revise Upgrade
Modeling Requirements for Changed System
Step 6
Model & Analyze Changed
Models and Analysis Results
Modeling Requirements for Original System
Step 5
Model & Analyze Original
Changed System Analysis Results
Changed System Model
Step 4
Identify Architecture Dependencies
Relevant OQAs
Step 3
Identify Changes
Original System Analysis Results
Changes Documentation
Step 2
Describe the OQAs
Original System Model
Upgrade Description Document
Step 1
Describe Upgrade
[redacted]
inputs
outputs
OQAs = Operational Quality Attributes
ADC = Architecture Dependencies Catalog
ATAM= Architecture Tradeoff Analysis Method

Figure 4-2: Steps and Artifacts of the VUV Method

4.3 Summary of the Steps of the VUV Method

In this section, we summarize the purpose and outcomes of each of the steps of the VUV method.

Step 1 explains the modifications of the upgrade (e.g., addition of processors or change of operating system), the reasons for the upgrade (e.g., obsolescence, vendor issues, or need for increased processing capacity), and the objectives of the upgrade (e.g., increased performance, reliability, and/or reduced weight). Input to this step takes the form of a description of the upgrade (e.g., a requirements document). The outcome of this step is an *Upgrade Description Document* or similar document that includes the description, rationale, and objectives of the upgrade.

Step 2 describes the OQA concerns and values of system properties that must be preserved across the change as well as those that must be improved (e.g., same end-to-end deadlines, desired target values, and prioritization of system properties to facilitate tradeoff decisions when resources are limited). A quality attribute utility tree from the ATAM method [Clements 2001] is useful for prioritizing the OQAs. Figure 4-3 displays a sample utility tree with quantified OQA measures and their priority levels shown as the leaves of the tree. The leaves of a utility tree typically represent use case scenarios. We can regard a computer system upgrade as a growth scenario. Such growth scenarios as well as other use scenarios are reflected in the architecture model through properties on architecture components, connections, and flows. The outcome of this step is a summary of the required OQAs and values of the relevant system properties that the upgrade must satisfy.

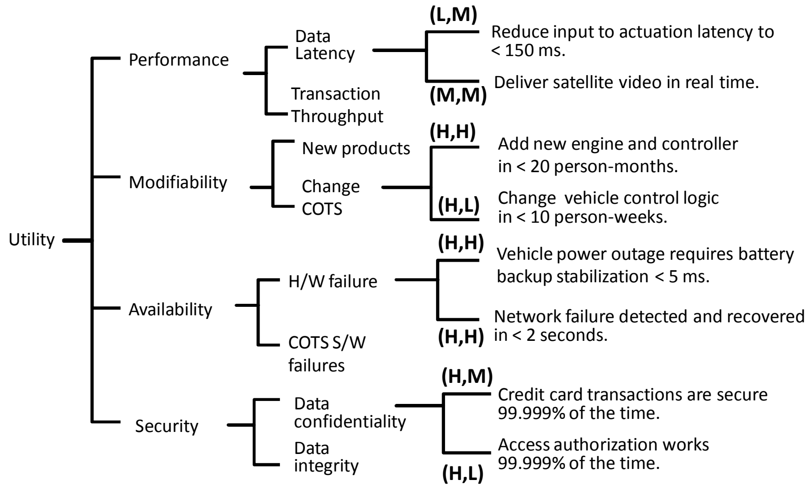

Utility
Performance
Data Latency
Transaction Throughput
(L,M)
Reduce input to actuation latency to < 150 ms.
(M,M)
Deliver satellite video in real time.
Modifiability
New products
Change COTS
(H,H)
Add new engine and controller in < 20 person-months.
(H,L)
Change vehicle control logic in < 10 person-weeks.
Availability
H/W failure
COTS S/W failures
(H,H)
Vehicle power outage requires battery backup stabilization < 5 ms.
(H,H)
Network failure detected and recovered in < 2 seconds.
Security
Data confidentiality
Data integrity
(H,M)
Credit card transactions are secure 99.999% of the time.
(H,L)
Access authorization works 99.999% of the time.

Figure 4-3: A Sample Utility Tree

Step 3 identifies changes in the architecture of the computer platform with a focus on the parts of the architecture that are added, modified, or removed. These parts can be schedulers, protocols, processors, networks, or memory. The outcome of this step is the documentation of those changes in the computer that are involved in the upgrade.

Step 4 identifies architectural dependencies that provide the modeling and analysis requirements for the original and changed systems. This critical step is supported by the ADC, which is presented in Section 6. This catalog reflects insights from the root cause areas discussed in Section 3.4. It guides a designer in defining the modeling and analysis required to assess potential impact on OQAs and identifies relevant system properties as well as impacted assumptions. The outcome of this step is the documentation of the modeling and analysis requirements for both the original and new systems.

The ADC consists of four tables that are defined in Section 6. These tables guide a designer from a general application domain categorization of the system to specific modeling and analysis requirements and associated assumptions that facilitate evaluating potential problems in an upgrade. Figure 4-4 shows the five-substep procedure employing the ADC with the four tables shown on the left.

- In the first two substeps, a designer (1) identifies the relevant application patterns using the *Application Category to Patterns* table (Table 6-1 on page 34) and (2) establishes modeling and analysis requirements for those patterns, using the *Modeling Requirements for Application Patterns* table (Table 6-3 on page 35).
- In the next two substeps, the designer (3) identifies the relevant OQAs and relevant system properties using the *Application Category to OQA-Related System Properties* table (Table 6-2 on page 35) and (4) establishes the modeling and analysis requirements for the relevant system properties using the *Modeling Requirements of Relevant System Properties* table (Table 6-4 on page 37).
- Finally, the designer (5) compiles a combined listing of recommended modeling and analysis requirements for the system. Modeling and analysis requirements describe what to model, the

analyses to conduct, and the assumptions associated with the relevant application patterns and relevant system properties.

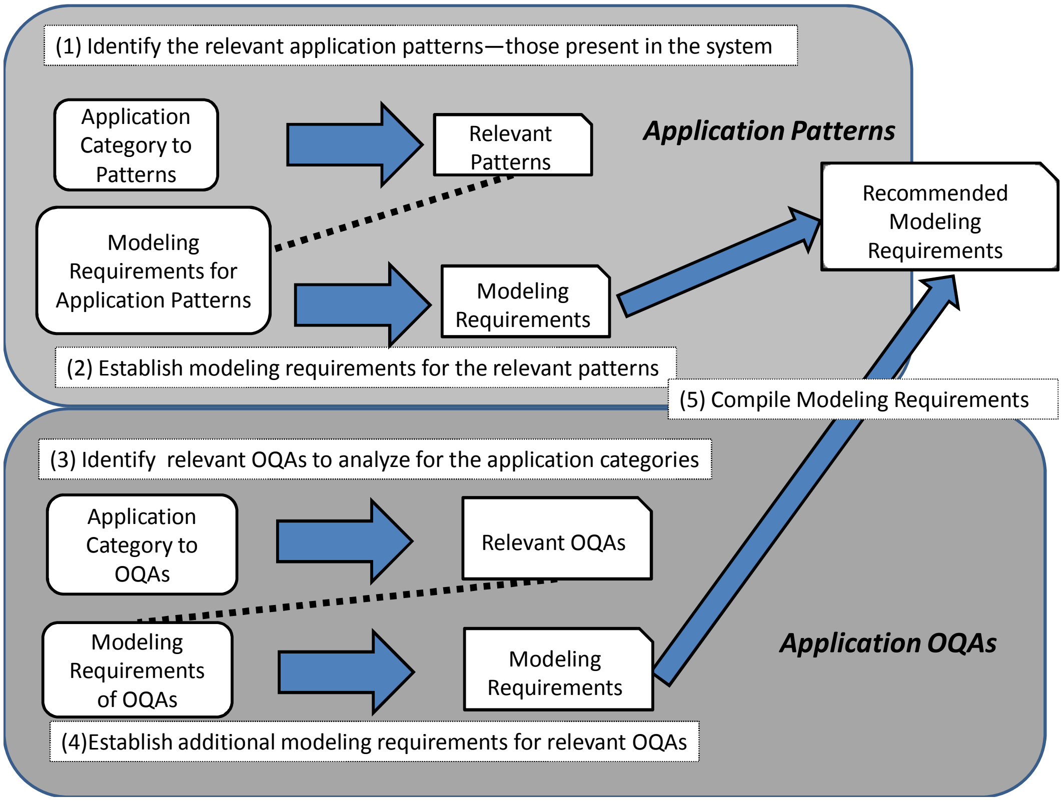

(1) Identify the relevant application patterns—those present in the system
Application Category to Patterns
Relevant Patterns
Application Patterns
Modeling Requirements for Application Patterns
Modeling Requirements
Recommended Modeling Requirements
(2) Establish modeling requirements for the relevant patterns
(5) Compile Modeling Requirements
(3) Identify relevant OQAs to analyze for the application categories
Application Category to OQAs
Relevant OQAs
Modeling Requirements of OQAs
Modeling Requirements
Application OQAs
(4)Establish additional modeling requirements for relevant OQAs

Figure 4-4: Step 4 ADC Procedure

This procedure is completed for the original system in support of developing a model of the original in *Step 5* and for the modified (upgraded) system in support of developing a Modified (Upgrade) Model in *Step 6*.

Step 5 creates and analyzes a model of the original embedded system architecture, as defined by the modeling and analysis requirements identified in *Step 4*. If a model of the original system is not available, the designer creates one. Otherwise, the designer uses the relevant elements of an existing model. The model of the original system consists of the application software architecture and a model of the existing execution platform. The focus of the models is on the upgrade changes. This focus results in models that have some parts of the architecture elaborated (those most relevant to the changes), while other parts are represented more abstractly. For example, when part of a system is having its backbone network replaced, we may represent a multi-core processor as a black box. The results of *Step 4* provide guidance on the information that we must capture to perform OQA-specific analysis. Section 7.1 (beginning on page 39) provides additional modeling and analysis guidance both for representing application patterns and for supporting analysis of specific OQAs. The outcome of this step is the model of the original system and the results of the analyses of that model.

Step 6 creates and analyzes a model of the new embedded system architecture, as defined by the modeling and analysis requirements identified in *Step 4*. In this case, the focus is primarily on capturing the new execution platform and annotating those parts that have changed and are relevant to the OQA analyses. The analyses will identify whether the OQAs meet their desired values. The outcome of this step is the model of the change system and the results of the analyses of that model.

Step 7 provides the opportunity to explore alternatives to correct any shortcoming of the upgrade. This may involve refinements to the computer system, refinements to the runtime architecture of the application to change the assumptions on the computer system, or possible tuning of the application functionality to reduce the dependency on certain runtime architecture properties. The outcomes of this step are revised models and supporting analysis results.

Figure 4-5 shows an analysis framework for software assurance developed by the SEI for NASA [NASA 2009b]. The steps of the VUV method fit into this framework as follows. Steps 1, 2, 3, and 4 support the *Focus* activity. Steps 5 and 6 support the activities of the *Build* and *Analyze* phases. *Step 7* is represented in the process flow by the feedback loop between the *Build* and the *Analyze* phases, as shown by the dual-colored flow from *Analyze Models*. In this case, a modeler iterates between revising the upgraded system model and analyzing the impact of the change on the OQA concerns.

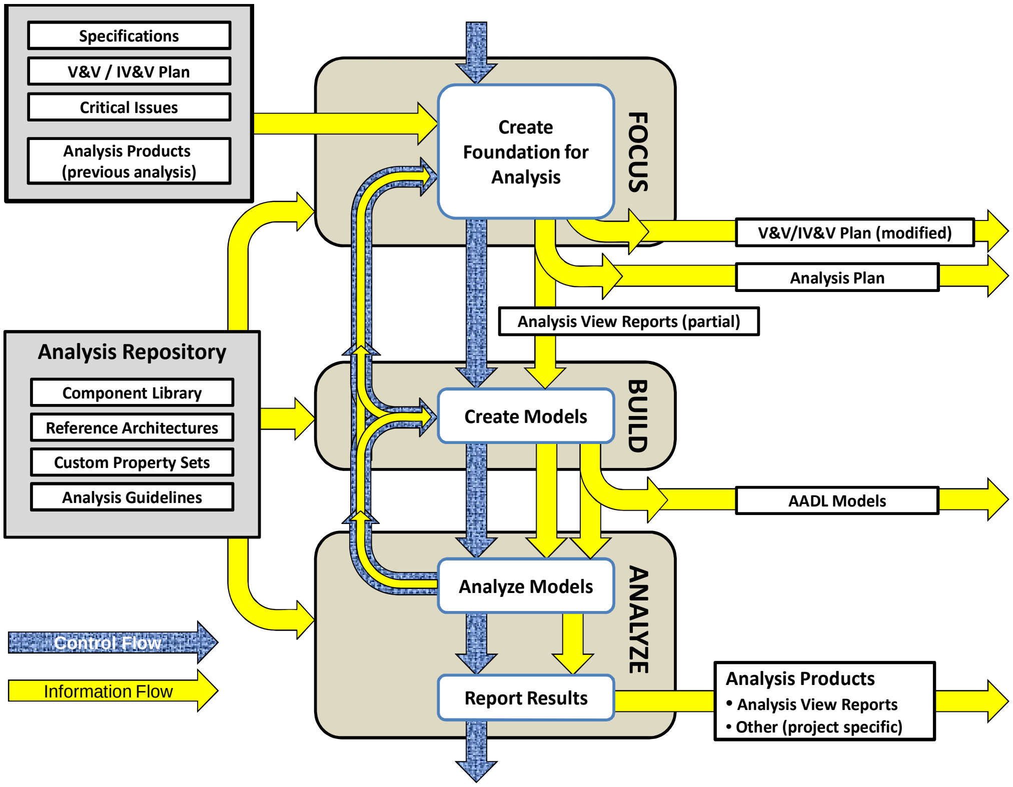


Figure 4-5: Analysis Practice Framework Summary

5 Application of VUV by Example

This section elaborates on the VUV method steps. We describe the actions and outcomes for each step in a simplified fly-by-wire system such as the one in Figure 5-1. In this system, a pilot uses a sidestick[6] to command the aircraft. The sidestick output goes to a fly-by-wire control computer that calculates and sends the requisite commands to control surface actuators that move the surfaces. Sensors associated with the control surfaces send position-related information to the fly-by-wire control computer via individual sensor lines. **Note that this system is a simplified example used to illustrate the method. It does not represent any specific fly-by-wire implementation.**

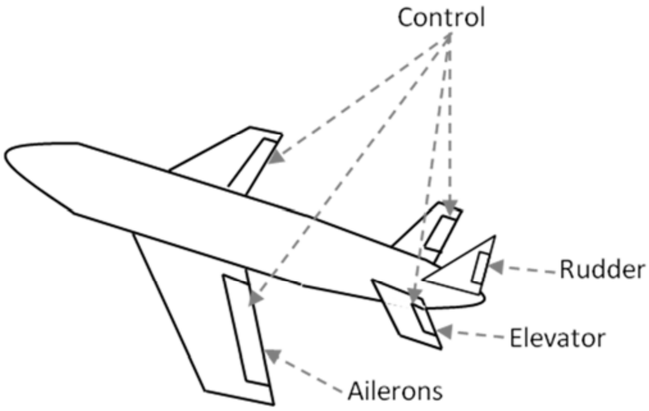


Figure 5-1: Control Surfaces on an Aircraft

In subsequent sections, we describe the application of the VUV method to this simplified example.

5.1 Step 1: Describe the Upgrade

This step provides the reason and objective for the upgrade. The reason must state the situation that triggered the decision to upgrade the execution platform. The objective should state the desired result of the upgrade; it is paired with a description of what must be done, at a high level, to accomplish the objective.

In our example, the *reason* for the upgrade is to eliminate noise[7] in the signals received from the sensors. This noise is due to the lengthy runs from the sensors to the control computer. The upgrade *objective* is to reduce the distance between the sensors and the `Sense Plane Status` function to less than 10 meters. This objective implies that this function will be separated from the other functions and put into a special computer closer to the wings.

[6] A *sidestick* is a device that translates the motion of the pilot’s hand into control commands for the airplane.

[7] Signal noise is due to random fluctuation in an electrical circuit.

Figure 5-2 illustrates the upgrade change. The upper portion shows the original system that consists of application tasks that execute on a single processor. The lower part shows the same set of application tasks distributed across two processors that interconnect via a network bus. In this figure, the solid lines with black arrowheads connecting the software components represent the communication of data between those components. The solid lines with white arrowheads define the connections between processors and the network bus. The double-lined arrows, extending from each system boundary to a processor and from a cross-system connection to the network bus, represent the binding of the software to a specific processor and communication connection to a specific network.

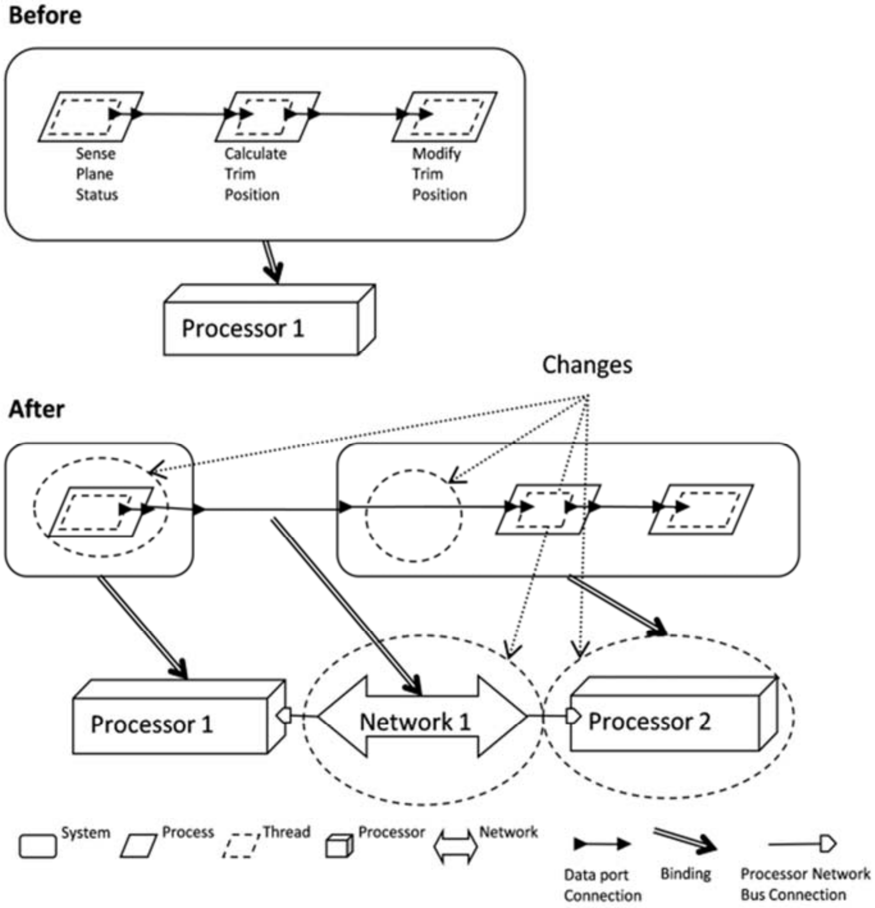


Figure 5-2: An Example of Execution Platform Changes

5.2 Step 2: Describe Relevant Operational Quality Attributes and System Properties

Three scenarios of system properties relevant to OQA concerns must be documented: system properties whose values must meet requirements in the original system, system properties whose values must meet modified requirements, and possibly system properties whose values must meet new requirements. When we document these system properties, the parts of the system involved in achieving them must be included. The documentation will guide our investigation of which parts and characteristics of the old execution platform affect the relevant system properties.

To provide further detail regarding the system properties to be maintained, we expand the explanation of the system. The computer continuously senses the status of the aircraft to evaluate changes in the environment (e.g., disturbance) and adjusts the control surfaces accordingly. This continuous sensing and correction of the control surfaces must be performed frequently enough to avoid variations in the environment that are too large to correct. A system property of interest is the frequency required to sense the environment and aircraft position and adjust the trim of the control surfaces. (For the Airbus A320, this sensing-adjustment cycle is completed every 30 seconds.[8]) This is modeled as a periodic thread that executes the functions in sequence. A schedulability analysis verifies that this task together with other tasks executing on the same processor can finish by the required individual end-to-end deadlines.

To understand better the architectural impact of the platform change on the relevant system properties, consider the new embedded system architecture depicted in Figure 5-3.

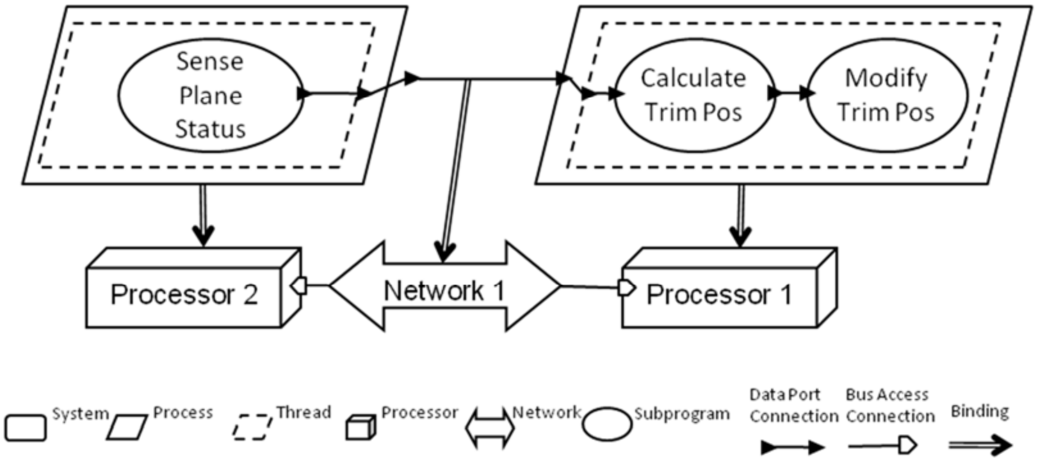


Figure 5-3: New Runtime Architecture for Automatic Trimming

The change in this figure consists of the movement of the module `Sense Plane Status` to a new processor, *Processor 2,* that is communicating with the old processor, *Processor 1,* through a new network, *Network 1*. While these changes seem to increase the processor cycles available for scheduling the different modules in the automatic trim subsystem, the interprocessor communication protocol, the scheduling of the network and processor, and the speed of the new processor and network, among other things, can affect the OQA concerns of interest. To understand the logic behind this change, let us consider the elements of the change (i.e., the network bandwidth, the communication protocol, and the processor scheduling).

- A direct comparison against the required communication bandwidth shows the network bandwidth effect on the changes in Figure 5-3. In particular, let us imagine that the plane status is contained in a data structure of 30 KB, which is communicated between the `Sense Plane Status` and the `Calculate Trim Pos` modules. Communicating every 30 seconds, then, requires 1KB/s of bandwidth capacity (an additional latency due to the communication delay will occur). Obviously, if the network has a bandwidth capacity of less than 1KB/s, this OQA requirement will not be satisfied.

[8] While this period seems long, it is the one presented in the reference literature [Apollo 2002]. Hence, we will use it to exemplify that problems can emerge even at a slow rate.

- The selected communication protocol plays a role in how the application is communicating data. In this case, to reduce the required communication bandwidth, we can change the application to send only those parts of the plane position data structure that have changed—a *state change* or *state transition communication* application pattern (see Table 6-3). However, if the communication protocol does not guarantee message delivery and a message is lost, then the `Calculate Trim Pos` module acts on a position of the plane that is incorrect due to the loss of incremental updates.
- Processor scheduling involves the timing and organization of the synchronous execution of the different threads. While a fixed-priority scheduler offers multiple advantages, if the original system is using static timeline scheduling, the designers of the original system may have assumed specific characteristics of this scheduling policy (e.g., a specific unchanging order of execution of the threads). In our example, the functions `Calculate Trim Pos` and `Modify Trim Pos` execute in a single thread, which implies that they are always executed in the same order. If we put them in separate threads and change the scheduler in *Processor 1* from a time-division multiplexing schedule (fixed execution timeslots over a time frame) to a preemptive fixed-priority scheduler (using rate monotonic scheduling [Klein 1993]), then it can no longer be assumed that the thread `Calculate Trim Pos` will always run before thread `Modify Trim Pos`. That is, it may be possible for `Modify Trim Pos` to run before `Calculate Trim Pos`.

In summary, the move to a two-processor system and the change in scheduling policy affect the information flow, in particular its latency and the age of data. The chosen scheduling policy, network protocol, and hardware are potential contributors to latency and latency jitter. There is also a potential for race conditions, especially if the implementation of inter-thread communication is through shared data buffers.

5.3 Step 3: Identify Changes in the Computer System

Along with the description of the OQAs, the affected parts of the system are documented. This documentation is not a full description but an identification of the areas to explore further in search of potential problems. In our example, the changes include the addition of a processor, a network, and associated communication protocol and the selection (and changes) to the processors' schedulers. These changes can violate application assumptions for communication across the network and the execution sequence of the modules `Calculate Trim Pos` and `Modify Trim Pos`, among other areas.

5.4 Step 4: Identify Architectural Dependencies

This key step helps define the potential risks and the amount of modeling needed to reduce uncertainty about those risks. In this step, we take the OQA concerns in terms of relevant system properties and the architectural changes identified in the previous two steps and evaluate problems in terms of

- potential disruptions to system property values due to changes
- potential assumption mismatches related to architectural changes

The identification of both types of potential problems utilizes the catalog of architectural dependencies that contains relevant system properties and assumption dependencies (discussed in Section 6). The catalog lists the required runtime patterns upon which each system property depends and the application assumptions affected by the upgrade. In addition, for each application assumption, the catalog provides a list of application patterns and categories sensitive to the assumption.

Using the ADC, we relate changes to affected system properties and modeling requirements in order to identify elements that affect these system properties in the existing runtime architecture and the new runtime architecture. Similarly, taking the view of the application assumptions affected by the changes, we verify whether our application contains the categories and architecture interaction patterns sensitive to these assumptions. Where this is the case, we model the parts of the application belonging to the category.

As an example, let us reconsider the automatic trimming example of Figure 5-3. It contains both relevant system properties and application patterns sensitive to changes to specific parts of the runtime architecture. On the one hand, a system property of interest is the end-to-end deadline that determines the rate required to complete the control cycle from the position sensing to the trim position modification. The flow follows the path of the control loop pattern, whose assumption is deterministic sampling of the data stream. On the other hand, the state change (state transition communication) pattern (Table 6-3) applies because changes in data state are being transmitted. From the state transition communication pattern, we identify the assumption of reliable communication that needs to be modeled in both the old and new runtime architectures. For the computer platform, we identify the schedulers, the communication hardware and protocol, and the timing parameters of the threads related to the changed communication path (between `Sense Plane Position` and `Calculate Trim Pos`) as elements that can affect the assumptions.

5.5 Step 5: Model and Analyze Original System

Using the application category (Control System), we can obtain both application patterns (from Table 6-1) and related OQAs (from Table 6-2). Then from the patterns and OQAs, we can obtain the modeling needs related to both (Table 6-3 and Table 6-4, respectively). These modeling needs state the system elements and properties relevant to the analysis of a particular OQA.

In this modeling step, we express—using AADL semantics—how the current runtime architecture achieves the system properties of interest and complies with the application assumptions. The model must contain enough information to enable analyses that support these assessments. We call this model the base model. In our automatic trimming example, the system properties of interest that we model are the processor together with the application threads.

The second goal is to identify and model the parts of the original architecture that support the application assumptions. In our example, this implies modeling the shared memory structure that represents the plane position and how this structure is updated and queried. Figure 5-4 depicts this model, with the list of component properties shown in Table 5-1 and the computer platform-level properties shown in

Table 5-2. Figure 5-4 shows the thread operating at a 30-second period.

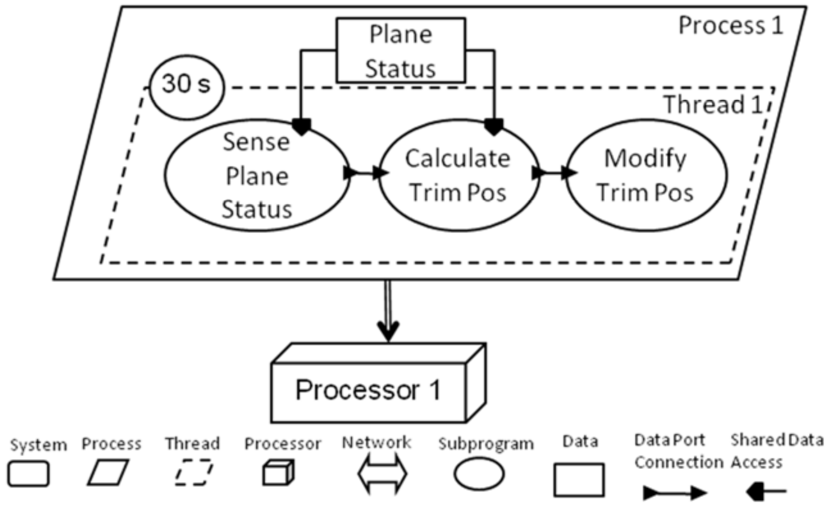


Figure 5-4: Base Model

Table 5-1: Application Component Properties of the Base Model

Component	Property	Value
Processor 1	Processor Speed	1 GHZ
	Scheduling Policy	Rate-Monotonic
Process 1	Memory Requirement	5 MB
Thread 1	Activation Pattern	Periodic
	Period	30 s
Plane Status	Size of Memory	30 KB
Sense Plane Status	Worst-Case Execution Time (WCET)	1 s
Calculate Trim Pos	WCET	1 s
Modify Trim Pos	WCET	1 s

Table 5-2: Computer Platform Properties

Component	Property	Value
Processor 1	Utilization	10%
Thread 1	Deadlines Met	True
System	Failure Mode	All or Nothing

5.6 Step 6: Model and Analyze Changed System

Using the base model, we next develop a modified model that includes the new computer system. The design of this new model should be guided by the potential disruptions to relevant system properties identified in Table 5-2. It should contain enough information to conduct analysis of these system properties.

We need the same level of modeling to describe the features that support the architectural assumptions. In our example, this involves the new processor and network, the schedulers in both processors and the network, the tasks run by the processors and the flows going through the network, and the communication protocol. The modeling of these elements must contain the proper characteristics that enable us to (a) verify the system property of interest and (b) evaluate whether the assumptions from the state transition communication are honored or not. The architecture is presented in Figure 5-5 and its properties in Table 5-3.

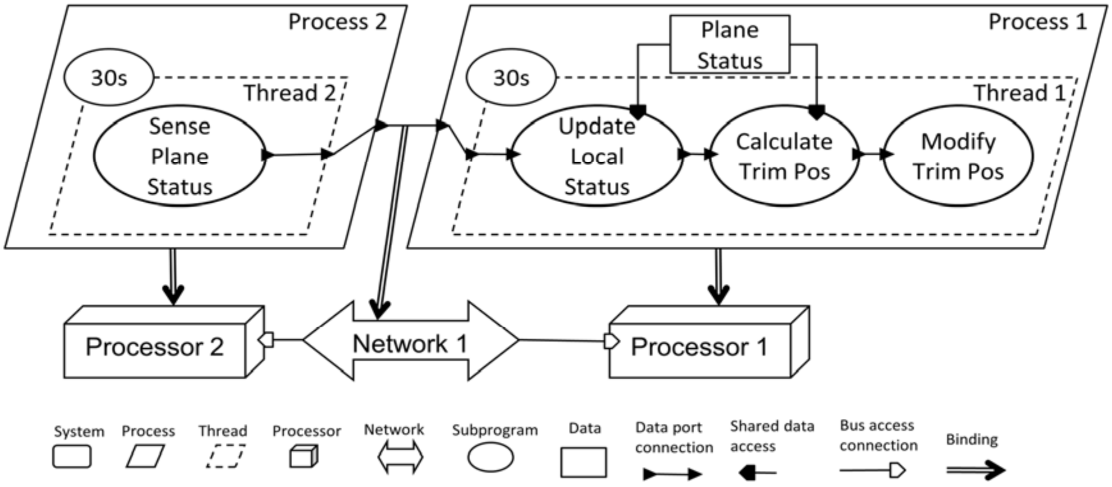


Figure 5-5: Modified Model Using New Computer System

Table 5-3: System Component Properties of the Modified Architecture

Component	Property	Value
Processor 1	Processor Speed	1 GHz
	Scheduling Policy	Rate-Monotonic
Processor 2	Processor Speed	1 GHz
	Scheduling Policy	Rate-Monotonic
Network 1	Bandwidth Capacity	512 Bytes
	Message Scheduler	CANBus
	Reliability	Unreliable
Process 1	Memory Requirement	3 MB
Process 2	Memory Requirement	1 MB
Thread 1	Activation Pattern	Periodic
	Period	30 s
Thread 2	Activation Pattern	Periodic
	Period	30 s
Plane Status	Size of Memory	30 KB
Sense Plane Status	WCET	1 s
Calculate Trim Pos	WCET	1 s
Modify Trim Pos	WCET	1 s
Update Local Status	WCET	1 s
Sense Plane Status → Update Local Status	Synchronization	Sampled
	Reliability	Unreliable
	Message Size	128 Bytes

5.7 Step 7: Revise the System Upgrade

In this step, we conduct a model-based analysis of the system properties of interest and application assumptions affected by the change. The problems discovered in this analysis guide model corrections to achieve the desired system property values. As we make these corrections, new interactions with other system properties can occur and new corrections will be triggered. The corrections to the model can be of two types: (1) corrections to the computer system to support the desired system property values or (2) corrections to the runtime architecture of the embedded software application to adapt to the new computer system. The decision of what kind of corrections are desired and the potential cost of these corrections are part of the tradeoffs needed to be made before any firm decision takes place in terms of platform acquisition, code changes, and the like.

In our example, one of these tradeoffs is that we transmit not the complete plane state every 30 seconds but only the delta, which is reflected in the state change/transition communication pattern. As shown in Table 6-3, this pattern is sensitive to other architectural characteristics (e.g., reliability of communication). Hence, as we explore this option, the service guarantees (no message loss, ordered delivery) of the communication must be added to the model to enable exploring the tradeoffs.

In addition, one assumption implicit in the base model is that functions execute in order: calculation of trim corrections always occurs after update of the plane status. The destination process of the connection in the modified model samples its input independently of the sender, which results in potentially nondeterministic sampling of the input stream. This sampling is indicated by the type of connection between the two threads. For time-sensitive data, this can lead to jitter, as is identified by the end-to-end latency analysis. For discrete-event data transmission through sampling rather than queuing, such as state changes or events, this can lead to a missed change or event, if the jitter exceeds the rate at which changes or events are sampled. In our example, this means that the plane status may or may not be updated before the calculation of the trim correction starts.

Our first refinement resolves this nondeterministic communication across processors by marking the connection between the processes as an immediate connection (i.e., a connection that assures deterministic sampling). Figure 5-6 depicts this architecture and Table 5-4 depicts its properties. This change implies that *Thread 1* delays its execution until *Thread 2* is finished. This delay may affect the execution of other threads that execute on *Processor 1* (i.e., the schedulability of threads on that processor).

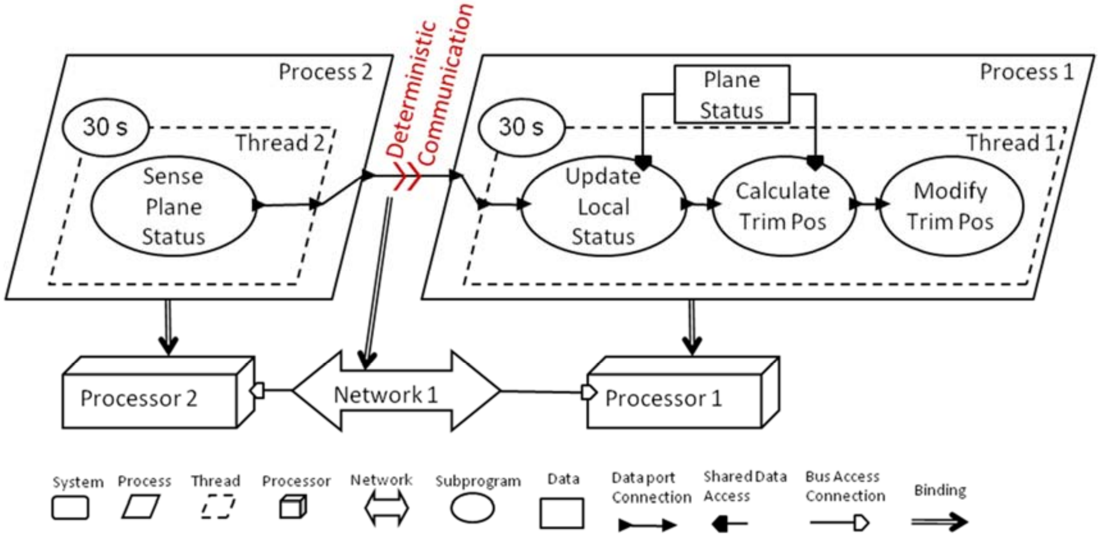


Figure 5-6: Deterministic Communication

Table 5-4: Properties of First Refinement

Component	Property	Value
Processor 1	Processor Speed	1 GHz
	Scheduling Policy	Rate-Monotonic
Processor 2	Processor Speed	1 GHz
	Scheduling Policy	Rate-Monotonic
Network 1	Bandwidth Capacity	512 Bytes
	Message Scheduler	CANBus
	Reliability	Unreliable
Process 1	Memory Requirement	3 MB
Process 2	Memory Requirement	1 MB
Thread 1	Activation Pattern	Periodic
	Period	30 s
Thread 2	Activation Pattern	Periodic
	Period	30 s
Plane Status	Size of Memory	30 KB
Sense Plane Status	WCET	1 s
Calculate Trim Pos	WCET	1 s
Modify Trim Pos	WCET	1 s
Update Local Status	WCET	1 s
Sense Plane Status → Update Local Status	Synchronization	Synchronous
	Reliability	Unreliable
	Message Size	128 Bytes

The final refinement introduces reliable communication needed for the state transition pattern. We achieve this by making the network protocol reliable, such that no messages can be lost between the two processors. If the immediate connection between the two processors is bound to a reliable network, no messages that flow over the connections will be lost. Figure 5-7 presents the architecture for this refinement and Table 5-5 presents its properties.

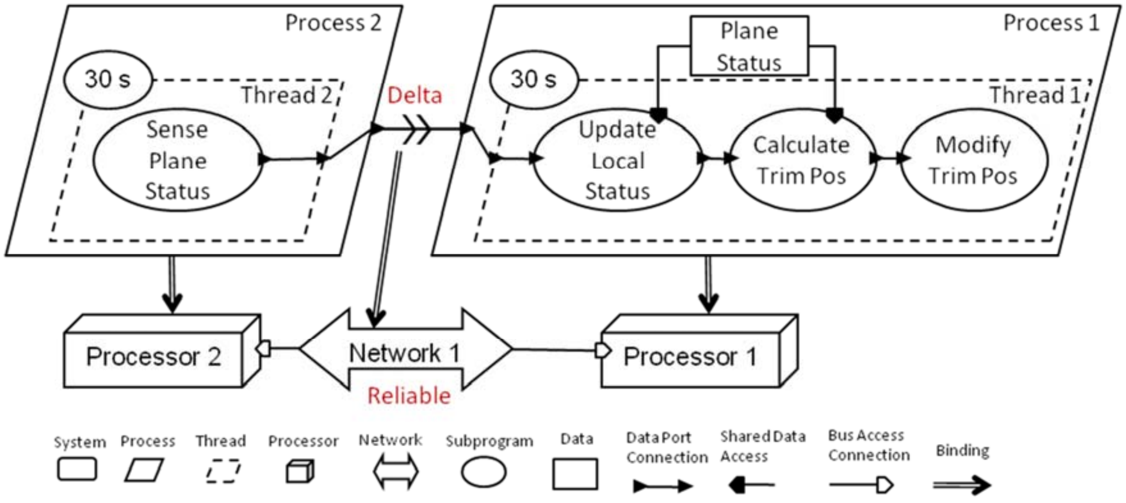


Figure 5-7: Final Refinement

Table 5-5: Properties of the Final Refinement

Component	Property	Value
Processor 1	Processor Speed	1 GHz
	Scheduling Policy	Rate-Monotonic
Processor 2	Processor Speed	1 GHz
	Scheduling Policy	Rate-Monotonic
Network 1	Bandwidth Capacity	512 Bytes
	Message Scheduler	CANBus
	Reliability	Reliable
Process 1	Memory Requirement	3 MB
Process 2	Memory Requirement	1 MB
Thread 1	Activation Pattern	Periodic
	Period	30 s
Thread 2	Activation Pattern	Periodic
	Period	30 s
Plane Status	Size of Memory	30 KB
Sense Plane Status	WCET	1 s
Calculate Trim Pos	WCET	1 s
Modify Trim Pos	WCET	1 s
Update Local Status	WCET	1 s
Sense Plane Status → Update Local Status	Synchronization	Synchronous
	Reliability	Reliable
	Message Size	128 Bytes

6 Architectural Dependencies Catalog

The purpose of the ADC is to support a modeler in characterizing the runtime architecture of an embedded software system and identifying the assumptions related to the computer system associated with that architecture.

The root cause areas identified in Section 3.4 are reflected in this catalog through application patterns, computer platform components, and deployment bindings as follows:

- *End-to-end flow of data streams* is a generalization of several application patterns such as the control loop, message processing, and sensor/signal fusion.
- *Distributed communicating state machines* is a generalization of several application patterns such as hybrid control in control loops, redundancy management logic of replicated distributed system components, and state change/transition communication such as application hand-shaking protocols.
- *Virtualized resources* is an abstraction of several runtime architecture concepts such as scheduling of concurrent thread execution, time and space partitioning of processors and networks, multi-processor and multi-core computing platforms, and globally synchronous execution semantics versus hardware running asynchronously through separate clocks.
- *Resource availability* deals with the capacity of processor, network/bus, and memory resources actually available for the execution of the embedded software application system. The resource demand of the application is mapped onto the resource capacity of the computer platform through deployment bindings.

The ADC is organized in four tables that guide the designer from a general categorization of the application domain to specific modeling requirements to evaluate potential problems in a planned upgrade. The first table (Table 6-1) contains a mapping of application categories to common application patterns used in these categories. The second table (Table 6-2) contains a mapping of application categories to common OQA-specific system properties that are important to honor in these categories. From these categories then, we use the last two tables to map each of the application patterns (Table 6-3) and system properties (Table 6-4) to the modeling and analysis requirements.

We use the tables of the catalog as follows. First, using Table 6-1, the designer should identify the application category to which the system being upgraded belongs. For instance, for the automatic trimming adjustment in the example in Section 5, we would choose “Control System.” We use the application patterns column as a checklist to verify whether any of these common patterns are present in the system. For our example, we would choose “Control loops.”

Second, using Table 6-2, we identify the OQAs we would need to check, such as hard deadlines, utilization, latency jitter, end-to-end deadlines, and reliability.

Third, using Table 6-3, we take the application patterns that were present in our system (control loops and time-triggered activities) and identify the common assumptions that these patterns make and the recommended implementation pattern. In our example, the assumptions would be minimal sample processing jitter, and the implementation patterns would be periodic real-time tasks. Note that the rows in this table also list what we need to model to be able to explore the OQAs of interest.

Fourth, using Table 6-4 and taking the OQAs identified in Table 6-2, we get additional modeling requirements for each OQA. This procedure would then give us the recommended modeling details that we need to include and the OQAs we need to check.

Based on the identified information from the ADC, we create the AADL model. Section 7 provides guidance on what to model in terms of AADL concepts and properties and which analysis to run.

6.1 Application Categories to Application Patterns

Table 6-1: Application Category to Patterns

Application Category	Application Patterns
Control System: *Continuous time control and modal hybrid control*	• Control loops • State change/transition communication • Replication
Signal Processing: *Multiple data streams and operational modes*	• Sensor/Signal fusion • State change/transition communication • Replication
Multimedia: *Multiple streams and Quality-of-Service levels and concurrent processing*	• Sensor/Signal fusion • State change/transition communication • Replication
Mission Control: *Message fusion into common blackboard with security concerns and operational modes and processing capacity*	• Message processing/fusion • State change/transition communication • Shared data communication • System partitioning • Replication

This table shows the patterns of each of the application categories in underlined text. For example, the primary patterns of a *control system* are control loops to represent continuous time-sampled processing of sensor data to generate actuator control signals and the state change/transition communication pattern to represent coordination of discrete hybrid control states. In the case of *signal processing*, we have sensor/signal fusion to represent handling of multiple interacting data streams and state change/transition communication to represent coordination of operational mode state machines. The *multimedia* application category deals with multiple media streams that require time synchronization (sensor/signal fusion) and operate in several discrete QoS levels (state change/transition communication). The category *mission control* does message processing and fusion to maintain a common operational picture (shared data communication as blackboard implementation) with operational modes (state change/transition communication) and system partitioning to address security concerns with respect to the information.

All four application categories may use the replication pattern (shown without underlining) for the following reasons: redundancy for fault tolerance and extra capacity to improve throughput or reduce response time. The replication pattern uses state change/transition communication to implement its redundancy management logic.

6.2 Application Categories to OQA-Related System Properties

Table 6-2: Application Category to OQA-Related System Properties

Application Category	OQA-Related System Properties
Control System	• Hard deadlines • Latency jitter • End-to-end deadlines • Utilization • Reliability
Signal Processing	• Soft deadlines • Response time as soft end-to-end deadline • Throughput and utilization
Multimedia	• Soft deadlines • Response time as soft end-to-end deadline • Latency jitter • Throughput and utilization
Mission Control	• Hard deadlines • Soft deadlines • Response time as soft end-to-end deadline • Latency jitter • Utilization • Availability (mean time to failure) • Security (hazard leakage)

Many of the OQA-related system properties in the above table focus on performance, which is affected by variability of the workload and availability of the computer resources and is measured in terms of an end-to-end data flow. Similarly, security in the form of confidentiality must validate security properties along end-to-end information flows [Hansson 2008]. OQAs such as reliability or availability lead to the use of both

- the Error Model Annex [SAE 2006] for dependability modeling for stochastic predictions [Feiler 2007b]
- replication patterns to achieve fault tolerance through physical redundancy in computer hardware and logical redundancy in application software [Feiler 2004]

6.3 Modeling Requirements for Application Patterns

Table 6-3 outlines for each application pattern the intent of the application pattern and the assumptions the pattern makes. Furthermore, the table suggests the most appropriate model representation to capture the application's intent. In addition, it indicates model representations of possible alternate application implementations that we may encounter in actual systems. If the application intent and its implementation differ, there is a potential for the implementation to not correctly realize the intent or only do so assuming a particular computer platform. Any change to the platform may result in incorrect application behavior, typically timing-related misbehavior.

Table 6-3: Modeling Requirements for Application Patterns

Application Pattern	**Application Intent**	**Assumptions**	**Modeling Requirements**
Control loops	Sampled processing of input by periodic real-time tasks	Deterministic sampling of data stream at given rate Sampling jitter is minimal	Application intent: periodic threads, data flow via data ports and flow specifications Alternate implementations: time-triggered activities with port-based or shared data communication Platform: sync/async platform, scheduling protocol, communication protocol and network Analysis: schedulability, response time, jitter
State change/transition communication	Coordination state change/transition between state machines Replication and distribution of state machines	Reliable ordered transfer of transitions Transfer of most recent state	Application intent: queued delivery of ordered transition event or state change stream vs. sampling of transferred state Alternate implementations: event observation by sampling of state Platform: sync/async platform, delivery guarantees of network/protocol Analysis: transition observation misses due to latency jitter, transient inconsistency due to latency
Sensor/signal fusion	Time-consistent fusion of periodic data by managed latency and time-stamped messages	Common time reference in time-stamped data Latency jitter is minimal	Application intent: periodic threads, data flow via data ports or event data ports and flow specifications Alternate implementations: event/message-triggered activities with port queues, or data sampling with application queuing Platform: sync/async platform and time source, scheduling protocol, communication protocol and network Analysis: schedulability, response time, jitter, latency delta at fusion points
Message processing and fusion	Timely message processing Time-consistent fusion of data by time-stamped messages	Common time reference in time-stamped data	Application intent: aperiodic threads, data flow via event data ports and flow specifications Alternate implementations: data sampling (data port or shared data) with application queuing Platform: sync/async platform and time source, scheduling protocol, communication protocol and network Analysis: schedulability, response time, jitter, time delta at fusion points
Replication	Centralized vs. distributed redundancy logic	Physical redundancy to address hardware failures	Application intent: Distributed state transition event coordination of time and event-triggered tasks Alternate implementations: see *State transition communication.* Platform: see *State transition communication.* Analysis: see *State transition communication*, logic validation under asynchronous execution, logic failures, communication losses.
Shared data communication	Coordinated shared state updates	Large data volume Infrequent or partial access	Application intent: synchronized, shared data access by independently executing tasks Alternate implementations: sampled periodic data processing or distributed remote data server Platform: sync/async platform and time source, scheduling protocol, communication protocol and network Analysis: schedulability, blocking times, and priority inversion

Application Pattern	Application Intent	Assumptions	Modeling Requirements
System partitioning	Resource virtualization and partitioning, fault management, multiple security levels	Fault and hazard isolation	Application intent: space and time partitioning via logical processor and network Alternate implementations: Dedicated processor and network Platform: Processor and virtual processor, network and virtual bus (channel) Analysis: resource enforcement, fault propagation, data integrity (security)

6.4 Modeling Requirements of Relevant System Properties

Table 6-4 specifies for each system property the system components that may influence the property values, assumptions made when specifying values for these properties, and modeling requirements in terms of the application and platform as well as suggested analyses.

Table 6-4: Modeling Requirements of Relevant System Properties

System Property	System Components	Assumptions	Modeling Requirements
Hard deadlines	Processor, network as scheduled resource	All processor and network resource demands are known. All resource capacities are known.	Application: periodic and aperiodic tasks, shared data, immediate and delayed port connections Platform: processor with scheduler, network with communication protocol, hardware capacities, deployment on given processor Analysis: schedulability and response time, blocking times and priority inversion
Soft deadlines	Processor, network as scheduled resource	Known variability of resource demand and capacity	Application: periodic and aperiodic tasks, shared data, immediate and delayed port connections, (sampling) data and (queued) event data ports Platform: processor with scheduler, network with communication protocol, hardware capacities, deployment on given processor Analysis: schedulability and response time, blocking times and priority inversion, percentage of missed deadlines, queuing
End-to-end deadlines	Computer platform (processors and networks) as scheduled resource	Full control of processor and communication scheduling	Application: periodic and aperiodic tasks, shared data and port connections Platform: processor with scheduler, network with communication protocol, hardware capacities, deployment on platform Analysis: schedulability and response time, flow latency
Latency jitter	Computer platform (processors and networks) as scheduled resource	Known min/max end-to-end deadline contributors	Application: periodic and aperiodic tasks, shared data and port connections Platform: processor with scheduler, network with communication protocol, hardware capacities, deployment on platform Analysis: schedulability and response time, flow latency
Throughput and Utilization	Computer platform (processors and	Known peak loads Guaranteed upper	Application: periodic and aperiodic tasks, shared data and port connections

System Property	System Components	Assumptions	Modeling Requirements
	networks) as scheduled resource	bound in processing time	Platform: processor with scheduler, network with communication protocol, hardware capacities, deployment on platform Analysis: schedulability and response time, WCET inflation
Reliability and availability	Managed replication of computer platform components	Physical redundancy	Application: logical replication of tasks and connections, redundancy management logic Platform: physical replication of processors and networks, deployment on platform with explicit collocation, noncollocation of application units, component fault behavior and probability Analysis: availability and reliability, fail-over logic consistency, hazard analysis, Failure Mode and Effects Analysis
Security	Physical and logical security boundaries through partitions	Encryption and authentication infrastructure	Application: tasks and data interchange via ports and data sharing, security levels, and categories Platform: processor and network with logical partitioning, data communication security mechanisms, security policies, deployment on platform Analysis: secrecy, confidentiality, sanitization

7 AADL Modeling and Analysis Strategies

In this section, we describe the specific strategies used to develop the models required by the ADC. We separate these into the application pattern modeling strategies and the OQA modeling strategies. In addition, we provide some guidance in determining resource availability in a computer platform, which may be less than the available raw capacity.

It is worth noting that the purpose of this section is not to teach how to model in AADL. Rather, we convey the strategies to model, analyze, and understand the result of the analysis to avoid the potential problems associated with upgrading the execution platform of an embedded, real-time system. As a result, we assume the reader is familiar with AADL concepts (including those introduced in Version 2 of the standard),[9] which are summarized in the Appendix of this report. We will not dwell on the syntax and semantics of the language, but refer the reader to other documentation [Feiler 2006, SEI 2009a].[10]

Note as well that there is not a single, unified analysis that can be run to discover the entire realm of potential problems. We must choose from a collection of analyses to explore areas of problems. The purpose of this method is to guide the designer to make these choices.

7.1 Application Pattern Modeling Strategies

Central to our approach for developing AADL models representing application patterns is that we understand the application pattern's intent and compare it to the implementation chosen by the developer. One example when the intent of the application pattern is to communicate random events, each of which is expected to be processed by the recipient, but the implementation represents events in state variables that are periodically communicated and sampled by the recipient. Such an implementation is sensitive to data transfer timing, input sampling, and execution timing. Under some circumstances, the recipient may miss an event.

7.1.1 Control Loops

The purpose of control loops is to control continuous time systems (i.e., systems that follow the laws of physics). They are designed as sense-computation-actuate loops that periodically sample sensor input and compute actuator output in order for the controlled system to reach a desired state. The control algorithm assumes that the latency between a sensor reading and output to the actuator is known and has little variation (jitter). Sensor readings may be available at a higher rate, the control algorithm may down-sample the sensor data stream to a lower rate, and the actuator may require its data at a different rate.

When the data stream is sampled, the control engineer assumes that sampling occurs deterministically at well-defined time intervals. This means that a processing step samples every

[9] We will refer to version 2 of the AADL standard as AADL V2 from this point forward in this report.

[10] Feiler, Peter H. & Gluch, David. *MBE Essentials: An Introduction to the SAE Architecture Analysis & Design Language (AADL)*. To be published by Addison-Wesley.

data stream element or that the recipient samples every other element when down-sampling. In some cases, the sampled data is passed to the next processing step within the same execution frame (mid-frame)—that is, the recipient processes its input within the same frame after the sender completes its action. In other cases, the control loop assumes that the data is received by a processing step in the next frame (frame-delayed); a control system simulation environment such as Simulink provides these execution semantics to the control engineer.

Studies have shown that control loops are sensitive to latency jitter. Control system stability can be affected by differences in the way application software components exchange data or in the way the underlying runtime system schedules application tasks [Cervin 2006, Feiler 2008a]. Therefore, it is beneficial to represent the expected execution and communication timing semantics explicitly in the runtime architecture model of a control system.

The intent of the control loop pattern is best captured in AADL by using periodic threads for processing steps and using devices with periodic processing behavior for sensors and actuators. The data is communicated through data ports. The semantics of AADL specify that, by default, input is sampled at dispatch time (i.e., threads sample their data port input at the beginning of the frame). Flow of data between the sensor, the processing steps, and the actuator is specified through data port connections. Data port connections can be declared always to communicate data mid-frame (immediate connection) or frame-delayed (delayed connection), such that sampling by the recipient occurs deterministically. Data ports are annotated with a data type, and AADL ensures that both ends of a port connection have matching types. The data type representation may include properties to indicate the base type, the expected range of values, and the measurement unit associated with the data. In addition, properties on the port may indicate data stream characteristics, such as the expected data stream rate and the capability to handle missing elements in the data stream. End-to-end flow declarations are used to specify control loop flows of interest, for which latency and latency jitter must be determined.

Feiler discusses a number of software contributors to latency and latency jitter, such as the scheduling protocol, the communication protocol, or the use of partitions as virtual machine abstraction in the runtime system [Feiler 2008a]. Similarly, variations in the computer hardware, such as processor speed, network speed, and distribution of tasks across processors affect latency and latency jitter. Therefore, we model those aspects of the computer platform in AADL and specify the binding of the software to the hardware.

With this modeling information, the analyses that can be used to verify the OQA include flow latency [Feiler 2007a]; schedulability and jitter [Singhoff 2005]; near-optimal allocation options with bin packing algorithms [de Niz 2006] that can be performed with OSATE [SEI 2009b]; and buffer allocation and end-to-end deadlines [Thiele 2000]. If response time and latency are of concern, see Feiler's identification of latency contributors by the runtime architecture and runtime system services [Feiler 2007a]. Furthermore, a computational trace of the latency calculation is available in an Excel spreadsheet format to help in understanding the actual time contribution of each step in the end-to-end flow [Feiler 2009a].

These resources provide guidance on how to refine the upgraded system to reduce end-to-end latency. For example, in the case of migration to a partitioned architecture, the overrun may be primarily driven by cross-partition communication; thus, reduction of cross-partition communication steps is key to resolving the problem. In the case of missed deadlines, alternative near-optimal resource allocation (thread deployment) options can be explored using the bin packing algorithms provided with OSATE [de Niz 2006, de Niz 2008a]. The problem might also be due to nondeterministic sampling of the data stream that may result from the implementation of the communication through shared variables or insufficient double buffering of port-based communication. Such sampling may work well under a static timeline schedule but result in a send/receive order race condition under preemptive scheduling or concurrent execution on multiple processors or cores. Feiler provides guidance in identifying insufficient buffering and user-level send-and-receive service calls as contributors to the problem [Feiler 2008b].

The VUV method takes advantage of the semantics associated with AADL modeling concepts for a precise specification of the system. By default, AADL assumes that input through data ports is sampled at dispatch time (i.e., at the beginning of the frame). Dispatch time sampling cannot be guaranteed if the sampling is initiated by the application code. If each processing step performs its own sampling, input will be sampled at the time the task executing the processing steps actually executes on the processor. Even if sampling of input is consolidated into an input/output (I/O) task executing at highest priority, sampling may not occur at the beginning of the frame. For example, if the application tasks are mapped into a partition of a partitioned runtime system such as ARINC653, then the I/O task executes at the beginning of the window slot within the frame, which may not be the first slot. AADL gives us the ability to specify input sampling times other than dispatch time. Partitions can be modeled as virtual processors, including the specification of partition allocation to window slots. This information can then be used to determine whether changes in the computer system affect end-to-end latency and latency jitter (and thus, the stability of the control system).

When examining the actual system, we may encounter an implementation that differs from the application intent described above. One common implementation approach is to maintain the control data in a common data area and have processing steps read and write this data area directly. Data received from a different processor or a device is placed into the common data area by a high-priority I/O task at the beginning of each frame. The same task is also responsible for providing data from the common data area to devices and to applications on other processors. Such implementations are typically combined with a cyclic executive scheduling protocol (i.e., a periodic task that executes different processing steps on the same processor in a fixed order). The execution order of the processing steps determines whether data is passed mid-frame or frame-delayed. Note that a change to a preemptive scheduling protocol affects the execution order. Similarly, allocation of tasks to different cores of a multi-core processor results in concurrent execution of those tasks. This results in frame-level changes to latency and latency jitter.

AADL supports modeling of shared data between multiple threads. Thus, we can model the implementation as is. We use data access features on threads or subprograms to indicate which data element in the common data area is accessed. Using access rights, we can indicate whether a thread or subprogram writes or reads the data. Typically, only one processing step writes data into a particular data element, and one or more processing steps read it. They may read it within the same frame if they execute after the writer or in the next frame if they execute before the writer. We can record the expected execution order in order to achieve the appropriate data flow in terms of mid-frame or frame-delayed transfer. We can do so through either

- a call sequence for subprograms as processing steps or

- an additional property on threads that indicates which threads are expected to execute later in the same frame, effectively documenting mid-frame communication

This model can then be mapped into a data-port-based model with port connections and input sampling during execution to explicitly document the intended data flow and support end-to-end latency analysis. The analysis can take into account the scheduling protocol, thread binding to processors, and sampling or queuing delays of communication protocols to determine whether the desired latency and jitter are achieved [Feiler 2008c].

7.1.2 State Transition Communication

Many embedded systems are stateful, and exchange of state information is common practice. These state behaviors take on different forms. A state may represent an operational mode, and a state machine with transitions between different mode states reflects expected mode changes. Multiple subsystems or system components may have mode-dependent behavior with respect to the same mode. In this case, the mode state must be made accessible to them. These subsystems or components may operate differently according to the current mode, or they may take special action every time a mode transition occurs. In the former case, the recipient is interested in the most recent state value, while in the latter case the recipient must be informed of every transition event.

We may also have the situation where different subsystems have their own operational modes, and the challenge is to coordinate these modes and their transitions. For instance, the electronic stability control (ESC) of a car involves braking individual wheels to change the rolling forces of the car in a curve. As part of this process, it is important to coordinate with the cruise control (CC) system. Hence, when the ESC transitions into the braking mode, it needs to inform the CC system. If the transition communication is lost, bad things can happen, such as the braking of a wheel by the ESC system while the CC system accelerates.

Other forms of discrete system states exist in hybrid control systems, in subsystems and components that offer different QoS levels, in replicated stateful system components that require coordination of state, in the redundancy logic of fault tolerant systems, and in application-level interaction protocols, to name a few. In all these cases, it is important to identify whether the objective is to communicate the most recent state or for the recipient to have critical awareness of every state transition as a separate event.

State transition communication is an application pattern that falls into the *distributed communicating state machines* root cause area of system-level faults. Section 3.4.2 provides some insights on possible analyses to use to identify upgrade impact on the embedded application software for this pattern. We can use the AADL port communication semantics and properties to document explicitly the intended communication of state or state transition events.

If the most recent state is of primary interest, the state managed by a component can be made accessible to others through a data port. The recipients can sample the state at their leisure, and port connections identify all interested parties. The recipient may miss an intermediate state if its sampling rate is slow or if the data transfer is over an unreliable communication channel, but it will always operate on the most recent state it received. As an alternative to using data ports, we can use data access features to a shared data component representing the state variable. The port-based model is more amenable to distributed computer platforms, while the shared-data component model assumes a computer system with physically or logically shared memory.

If state transitions are important, the application intent of responding to every state transition event is best reflected in event ports or event data ports with queues. This ensures that at the application level all events are passed on as long as the communication medium ensures guaranteed and ordered delivery of events or messages. In other words, we want to annotate the AADL bus and virtual bus components of the underlying computer platform with the respective QoS property values.

It is a common practice to implement state communication by transferring the current value of a state variable even though the state transitions are of importance. This is particularly the case if the application primarily performs periodically sampled processing. In this case, the recipient samples the state value at a given rate and deduces state transition events by comparing the received state value against the previous value. For example, a button pushed in an operator interface is mapped into a Boolean state variable. The push event is effectively transformed into a "pulse" signal, setting the value to true for a limited time before being reset to false. If we model such an implementation in AADL, using data port sampling or shared data component access, we would have to augment the model with a property to reflect the length of this pulse in order to ensure that the pulse is not missed by the sampling recipient.

For communication of state transition events, we must investigate potential causes of loss. One possible cause is the reliability and service guarantees of the network or protocol, which can result in the loss of part of the transmission involved in the state communication. Another potential cause is the timing of the communication and the sampling by the recipient, effectively a potentially nondeterministic race condition between the transfer and the input sampling. Both causes of loss must be evaluated along with the consequences of such a loss. An example of an analysis technique that can help in this process can be found in a paper by de Niz [de Niz 2008a]. The problem might also be in the implementation of the communication through shared variables or insufficient double buffering of port-based communication. Feiler provides guidance in identifying buffering and user-level send-and-receive service calls as contributors to the problem [Feiler 2008c].

7.1.3 Sensor/Signal Fusion

Sensor/signal fusion is sensitive to the sampling times of the fused data. That is, if two signals from two different sensors are being fused in a particular thread, the sampling times of these two signals must be close enough to each other that they can be considered simultaneous readings. As a result, we model end-to-end flows from the sensor to the fusion thread and evaluate their flow latency. The domain expert (control system engineer or signal processing engineer) must evaluate whether differences in the flow latency between the data streams being fused are within the allowable range. The modeling elements of the control loop are sufficient for this analysis.

Sensor/signal information represents some physical system state. If the state information is sizable and communication bandwidth is low, state changes are often communicated, a situation that is effectively equivalent to the state-transition-communication issue discussed in Section 7.1.2. For example, a radar system may track objects and communicate changes in the form of track updates. Every state change is important and must be communicated and processed in order for the system to maintain its consistency and data integrity. Therefore, we want to reflect in the AADL model whether the sensor/signal data represents state or state changes and then examine whether the particular implementation of sensor/signal state ensures delivery of state changes.

7.1.4 Message Processing and Fusion

Message processing and fusion may deal with time-sensitive fused data. That is, if two messages from two different sources are being fused in a particular thread, the time stamp of the two messages must be close enough to each other so that they can be considered the same observation. We define end-to-end flows from each source to the fusion point and evaluate their flow latency. The domain expert (mission system engineer or mission expert) must evaluate then whether the time shift between the message streams is within the allowable range.

We model the intent of the application through event data port communication with queues. End-to-end latency analysis takes into account queuing latency. This allows us to determine how long messages from different streams must be held in queues (i.e., the size of these queues, in order to achieve time-consistent fusion).

We explicitly model the concept of time stamping by a time server component or by utilizing the time service on a processor. If the system is asynchronous (i.e., if different processors operate with separate clocks), we use the synchronization domain notion introduced in AADL V2 through the `Reference_Time` property.

We also want to reflect, as a property of the message stream (i.e., the ports through which the message streams are communicated), whether the fusion algorithm assumes that every message arrives or can compensate for missing messages on one or more of the data streams. We can then apply the same analysis as for state transition communication to ensure that the application implementation meets the intent, even when changes are made to the computer platform or application deployment binding.

7.1.5 Replication

Replication is a common technique for achieving redundancy in order to improve the reliability of a system. When modeling a replication pattern, we must consider three aspects of the pattern. First, the replicated component itself has state, and we have to ensure that this replicated state is maintained consistently between the two replicates. Maintaining state consistency between replicates is typically achieved at switch-over when the primary component fails, or state is periodically exchanged when the replicates are active. In the former case, we specify data port connections between the replicates and indicate that they are active only during a mode transition. In the latter case, we represent the periodic interchange through ports and specify the rate of data exchange, which may be different from the processing rate of the replicated component, as a port property.

The second aspect of the replication pattern is the redundancy logic that manages the replicated components. The logic reflects whether the pattern uses hot or cold standby in a primary/backup configuration [Budhiraja 1992] or whether all replicates are active and a voter or observer examines their output. The logic may also include the ability of an operator to control whether to operate in critical mode with all active replicates or in noncritical mode with a primary/backup configuration. In non-critical mode, the operator has the ability to choose which replicate is the primary (e.g., in a dual-flight guidance system) [Miller 2005].

The redundancy logic can be specified in AADL as modes, complemented with Behavior Annex subclauses, and validated for correctness under nominal operation and under various failure conditions. For instance, in a primary/backup replication, the failure of the communication channel between the replicates can lead to having two primaries or no primary at all. The redundancy logic itself may be replicated and distributed across redundant processors together with the replicated application software component. Effectively, we have a state-transition-communication pattern with two identical state machines reflecting the current (critical/non-critical) operational mode, and we have two state machines that are the mirror image of each other reflecting which instance of the primary/backup replication is active. Incorrect coordination logic or incorrect implementation of the coordination can lead to loss of transition events and safety hazards. Examples of analysis of these situations can be found in work by Miller and by de Niz [Miller 2005, de Niz 2006].

The third aspect of replication patterns to be considered is the monitoring of output of replicated components to detect possible faulty component behavior. This monitoring may take the form of an observer pattern with the output being monitored in parallel with its being sent out or of a guard pattern in which the output passes through a voter before becoming available as output. The two variants of the redundancy pattern, illustrated in Figure 7-1, differ in that the first has transient fault propagation (i.e., bad data output is sent before being detected by the observer) and the second results in longer latency. We want to record these effects for each pattern to ensure that components receiving the output can handle transient bad data or increased latency.

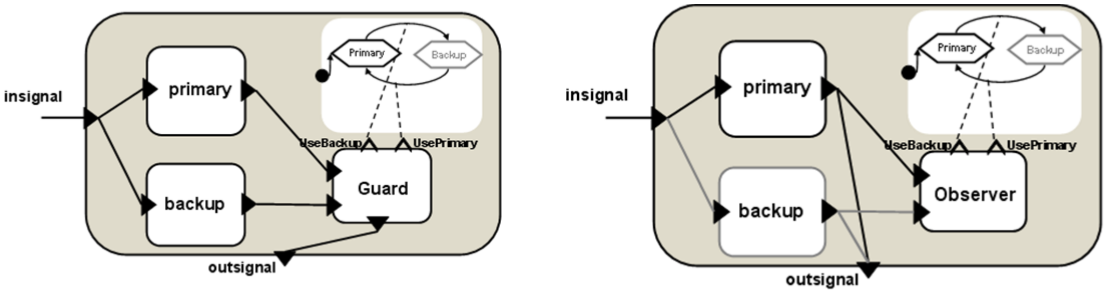


Figure 7-1: Observer and Guard Redundancy Pattern

7.1.6 Shared Data Communication

The shared data communication pattern may represent a particular application implementation (e.g., communication between threads via shared variables such as the communication semantics of Simulink blocks). We have already addressed this issue in previous sections. Here we focus on application architectures, whose intent is to provide shared data communication. Examples of such architectures are blackboard architectures (global common data area), database systems, and systems maintaining situational awareness.

The application intent of such architectures is to provide read-and-write access to a shared data component for a number of application tasks by coordinating concurrent access to ensure data integrity. In AADL, we express such an architecture via data components; connections from data access features are modeled by application components such as threads. An access right property of the data-access feature indicates the desired read-and-write access. A concurrency control protocol property on the data component indicates the mechanism used for managing concurrent access. Some implementations may rely on a particular scheduling protocol, such as non-preemptive scheduling and data access from a single processor to ensure mutually exclusive access. We need to ensure that such assumptions about the runtime system are valid and that the application source code actually uses the specified mechanism.

In distributed systems, a common way to implement a shared data communication architecture is through a server. AADL supports modeling of server architectures through subprogram access features and connections. The server is defined as a process with threads that provide access to the services (`provides` subprogram access). Users of the service require subprogram access, and access connections connect the two. Properties are used to indicate whether a server is multi-threaded in that it can service multiple service requests simultaneously (a standard property in AADL V2).

7.1.7 System Partitioning

System partitioning is an architectural pattern that provides isolation between application components through processor and memory virtualization concepts in the runtime system. System partitioning can be used to meet safety and security requirements. There are two aspects to modeling in this architectural approach: modeling of desired isolation regions in the application and modeling partitions as logical resource partitions. The ARINC653 Annex of the AADL standard provides guidance on how to represent ARINC653-compliant system architectures in AADL [SAE 2011].

The process concept in AADL represents a protected address space that is expected to be enforced at runtime by the hardware and operating system. In other words, threads operating within a process cannot affect data in other processes except through explicitly declared communication channels, such as ports and access to shared data. The computer platform is represented by processor, memory, and bus components that are tagged with the security or safety level for which they have been approved. In addition, we can use the AADL V2 virtual processor and virtual bus concepts to represent logical partitions of processors and buses. A virtual processor represents a virtual machine that enforces both space and time partitioning on a processor. A virtual bus represents a virtual channel that ensures bandwidth and no cross-channel data leakage. When an application with security and safety requirements is bound to the computer platform, we can validate that the safety and security requirements expressed at the application level are ensured by the computer platform. Further modeling guidance can be found in work by Delange, Pautet, and Feiler [Delange 2009a].

For the purpose of reliability and safety analysis, the Error Model Annex of the AADL standard [SAE 2006] provides a way of capturing hazard and fault information as annotations to the AADL architecture model of a system [Feiler 2007b]. From this annotated model, appropriate analysis models are generated to perform these different analyses consistently with the architecture.

For the purpose of security analysis, the SEI has defined a property set to represent security models, such as Bell-LaPadula, in AADL models [Hansson 2008]. Security levels and data categories are used to identify the degree of protection desired for data communicated through ports and the credentials of tasks operating on the data. Similarly, application components can be characterized with different safety levels under which they are expected to operate. Given this information, we can establish whether the way the application components interact violates security and safety rules at the logical level.

7.2 OQA Modeling Strategies

For each of the OQAs presented in Table 6-4, we discuss what kind of modeling is necessary, how to perform this modeling, and what kind of analysis is needed.

7.2.1 Hard Deadlines

Some applications require that processing be completed before a well-defined time—the deadline. Otherwise, the resulting output is of little value. For that purpose, we need to model the relevant tasks, specify their dispatch protocol as time-triggered (periodic) or as event/data-triggered (aperiodic, sporadic, or timed), their deadline relative to their dispatch time, and their resource demands on the processor in terms of execution time. In the case of event/data-triggered tasks, we also specify arrival rates with an upper bound. Once the tasks are bound to a processor with a specified scheduling protocol, we can use an appropriate scheduling analysis tool to determine schedulability (i.e., whether all deadlines are met).

Schedulability analysis assumes that task WCETs are not exceeded and that there are no additional resource demands on the processor not reflected in the model. Furthermore, it is desirable to perform such resource analysis early in the development, at which time execution times may be initial estimates or task details may not even be known. We can address uncertainty in the specified execution time of tasks by performing sensitivity analysis with respect to schedulability (i.e., examine by how much execution times can vary before the system becomes unschedulable). If we do not have the details of a task architecture yet, we may represent budgeted resource demands of subsystems (i.e., the rate at which such subsystems intend to operate) and compare them against the available resources on a processor. This provides an early insight as to whether exceeding resource demand will result in missed deadlines.

A similar resource scheduling analysis can be performed for networks. The resource demand for communication is determined by the port connections between subsystems and tasks that are bound to particular buses. The bus protocol (i.e., a sampling protocol such as 1553B or CAN-bus or queuing message protocols) determines the type of analysis to be performed to meet communication deadlines.

7.2.2 Soft Deadlines

Soft deadlines are similar to hard deadlines; one difference is the consequence of missing the deadline. Two scenarios are possible here. One scenario is that a missed deadline is acceptable but is treated by the recipient of the data stream as a missing data element. This is sometimes referred to as *firm deadline*. Schedulability analysis can determine at what rate missed deadlines are encountered. This rate can then be compared against the miss-rate that is acceptable to the recipients, as indicated by a property on their incoming ports.

The second soft deadline scenario is that completion time has a certain distribution that may be specified or determined through queuing analysis. Some completion times later than the specified deadline are acceptable as long as they are bounded. In this case, the recipient will still process the data, but again may specify an acceptable upper bound or an acceptable average for the age of received data. Again, scheduling analysis can provide results that can be validated against the expectations of the recipient.

7.2.3 End-to-End Deadlines

End-to-end deadlines can be analyzed for end-to-end flows. For these, then, it is important to model the end-to-end flows of interest with an associated end-to-end deadline. With this model, the flow latency analysis can be run to verify that the latency is within bounds [Feiler 2007a, Feiler 2008a]. In addition, various analysis tools (e.g., Cheddar [Singhoff 2005] and Rapid RMA [Tri-Pacific 2011]) can be used for analysis more specific to the scheduling policy and network protocols of interest in our architecture.

7.2.4 Latency Jitter

Jitter can be evaluated for end-to-end flows. We determine jitter bounds through end-to-end response time or latency analysis, taking into account best-case execution time and WCET, as well as variation in other contributors to latency and response time, such as communication and queued processing or preemption [Feiler 2007a].

7.2.5 Throughput and Utilization

Throughput refers to the amount of processing supported by a system, typically in terms of number of processed items per unit of time. Utilization refers to the degree to which computer platform resources are used to perform the tasks.

Throughput analysis can be supported in a stochastic manner if distributions of arrival rates and processing times and rates are provided. These can be expressed through properties on tasks and ports and used as input to a stochastic analysis tool. In the case of communication throughput, these would be property values indicating output rates and data sizes at the application level, as well as protocol overhead and transmission capacities of the network components of the computer platform. Early in the development process, such data may not be available and the details of the task architecture may not be known. In this case, we can assign rate and processing budgets to application subsystems and available processing capacity to computer platform components and use those to calculate early throughput estimates. Note that the model has a record of the assumptions used in the calculation of these estimates.

Utilization is determined as part of scheduling analysis. In the case of stochastic workloads, the workload may be reflected as a statistical distribution. Early in the development process, we can use resource budget estimates associated with the system architecture model at the level of detail available at the time (e.g., subsystem or task-level model) to determine the estimated utilization.

7.2.6 Reliability and Availability

Reliability and availability reflect the fact that system components may fail and, as a result, the system as a whole may not be able to provide service. AADL has an Error Model Annex [SAE 2006] that allows us to associate probability of fault occurrence in system components and probabilistic error propagation, taking into account error masking and repair. Detailed guidance to dependability modeling with the Error Model Annex is provided in *Dependability Modeling with* *the Architecture Analysis and Design Language (AADL)* [Feiler 2007b]. This error modeling capability supports hazard analysis, failure mode and effects analysis (FMEA), fault tree analysis (FTA), and stochastic reliability and availability analysis.

Reliability and availability are improved with redundancy. Physical components such as sensors and other devices as well as computer platform hardware components are replicated such that a backup component can fill the void when a component fails. Software that runs on failing computer hardware must be replicated and bound to the hardware replicate in order not to lose software functionality. In other words, we must ensure that different copies of the same software are deployed on different instances of computer hardware. This requirement can be expressed in AADL by a property indicating that a replicated component must not be collocated. This constraint can then be validated for a given deployment binding, or an analysis tool determining a deployment binding may take this constraint into account [de Niz 2008b].

Reliability and availability analysis makes a set of assumptions about the system, the lack of collocated software replicates being one example. Other assumptions are the consistency and correctness of the redundancy management logic and a source code implementation that complies with the logic specification. The validation of this assumption has been discussed in Section 7.1.5.

7.2.7 Security

While there can be multiple security concerns that could be modeled in AADL, we focus on confidentiality with three aspects: confidentiality, integrity, and sanitization. Confidentiality addresses concerns that sensitive data should only be disclosed to or accessed by authorized users (i.e., enforcing prevention of unauthorized disclosure of information). Data integrity is closely related, as it concerns prevention of unauthorized modifications of data.

To validate the confidentiality of a system, we ensure that a modeled system conforms to a set of common conditions that support system confidentiality independent of a specific reasoning framework for security. We map concepts of subjects operating on objects by permissible access (read, execute, append, and write), found in the Bell-LaPadula model, into components and ports in the AADL model, enabling us to model and validate security at both the software and hardware levels [Hansson 2008]. For that purpose, we have defined a set of properties to represent security levels and information categories to identify the degree of protection desired for data communicated through ports and the credentials of tasks operating on the data. Application components represent the subject with permitted security-level and information categories. Annotated ports represent the object (i.e., the data type and the security level and category of the data). Connections in the AADL that represent information flows and sanitation steps are indicated as part of a flow specification.

With this data, it is possible to discover incorrect communication of confidential data (to a component that does not have the privilege to access it) or incorrect sanitization structures. The security plug-in in OSATE can be used to analyze the security problems in the architecture [Hansson 2008].

In addition, we can model the system partitioning in the runtime system to ensure that the computer platform supports information isolation and the application deployment binding is consistent with the partitions as discussed in Section 7.1.7.

7.3 Computer System Resource Management

The actual performance of processors available to the application is affected by the overhead of the operating system and other infrastructure services, as well as by resource contention between applications and system services sharing resources. The operating system overhead of an application can be reflected in net processor cycles available to the application, or it can be included in the execution time demands of the application thread. We can also model the operating system services as a second runtime architecture layer with its own thread model to capture concurrent processing activity at that level. We can do so using virtual processors and virtual buses or by creating a separate runtime architecture model of the operating system layer and associating it with the `Implemented As` property to the computer system abstraction of the original model.

Resource contention affects resource availability to the application. In this section, we review three important issues that affect real-time performance:

1. bounds on priority inversion
2. bounds on cross partition interference under IMA
3. rate group schedulability margin

Although we use avionics as an example, the techniques discussed here are also generally applicable to other real-time embedded systems.

7.3.1 Bounds on Priority Inversion

In a static priority scheduling setting based on RMA, priority inversion can occur when a high-priority task is delayed by one or more lower priority tasks. Bounds on priority inversions, if not calculated correctly, will render an RMA analysis invalid, leading to unexpected timing failures during integration or deployment. Bounds on priority inversion must be computed for each type of shared resource, especially bounds on the duration of priority inversion on

- CPU sharing
- I/O interfaces
- each communication switch

The basic concept of priority inversion is now well known. However, a significant number of engineers focus only on priority inversion in the CPU and fail to analyze priority inversion in complex I/O interfaces such as a Peripheral Control Interface (PCI) bus or a network switch. As a result, real-time performance failures during system integration or deployment occur in systems with a heavy I/O or communication load.

The experimental investigation of priority inversion bounds must be guided by the actual system architecture and deployment configuration rather than a benchmark configuration. For example, a PCI bus has many different physical configuration and bus transaction types. However, the bound on priority inversion is specific to these physical configuration and selected bus transaction types. As another example, priority inversion for application tasks depends on the specific real-time operating system and the real-time synchronization protocol that it implements.

7.3.2 Bound on CPU Stall Induced Worst-Case Execution Time (-) Inflation

RMA uses each task's WCET as part of the required inputs for schedulability analysis. Many developers assume that the WCET of a task remains the same when it runs alone or runs together with other tasks. Unfortunately, this is not true.

As illustrated in Figure 7-2, modern "smart" I/O devices can be independent bus masters. If a task's cache was first invalidated by prior tasks, the current task will try to reload the cache with instructions across the front side bus. If there is an ongoing bus transaction on behalf of other tasks, the filling of the cache can be significantly slowed because a typical bus master uses a round-robin sequence for competing bus transactions. This bus contention results in a significant slowdown of the task execution. When a PCI bus is used, the execution time of the task has been found to increase as much as 37% in laboratory experiments [Nam 2009]. Indeed, this a key reason for frame overruns that frequently occur when there is heavy I/O.

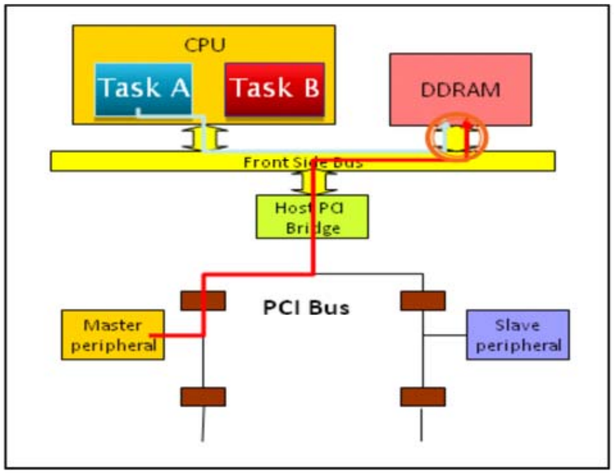


Figure 7-2: Resource Contention on PCI Bus

CPU stall-induced worst-case execution time (SWCET) has ominous implications for modern IMA architectures because many users of IMA architectures mistakenly believe that the CPU cycles allocated to each ARINC653 partition are isolated from those in other partitions. When tasks in the one partition invalidate the cache of a later partition, tasks in the later partition must reload their cache via the front side bus. The bus is subjected to the interference of direct memory access (DMA) from other partitions. Thus, a partition dedicated to a safety-critical, real-time application can be adversely affected by I/O for non-safety-critical applications in other partitions. The solution for this problem is to have an integrated CPU and I/O real-time architecture, which is, however, outside the scope of this report. We are concerned with calculating the bound on SWCET, especially in the context of IMA because most avionics systems have widely adopted ARINC653.

To measure the SWCET for IMA systems, we must do the following:

1. Flush the cache before the application in the next partition starts.
2. Conduct heavy DMA transfers to account for worst-case DMA workload on the front side bus as permitted by the existing design.

3. Measure the increase of WCET as compared with the case in which there is no DMA on the front side bus.
4. Add SWCET to WCET in the RMA analysis for all of the hard-real-time tasks.

Schedulability analysis and bus delay analysis is done using a tool called ASIIST (Application Specific I/O Integration Support Tool) [Nam 2009]. An analysis plug-in to OSATE, ASIIST reads in AADL models and can perform schedulability analysis with I/O cache fetch interference and I/O bus delay analysis for an IMA system configuration.

7.3.3 Rate Group Schedulability Margin

In a real-time system using fixed-priority scheduling or rate group scheduling, a task can miss its deadline with a workload considerably less than 100%. A rate group schedulability margin can be estimated by using exact schedulability analysis to compute the worst-case margin for each rate group (margin = deadline - worst-case completion time). The system peak load is the maximum utilization of the system given the minimal margin over all of the rate groups. However, margin and peak load computation is correct only if the estimation of WCET, the bound on priority inversion, and the bound on SWCET are all valid.

Note that this computation for peak load is very different from the peak load measurement generally reported for most computer systems. Most peak load values are generated by measuring the utilization of a background task or by summing the measured loads of each rate group. These methods for computing peak load lead to highly optimistic views of the worst-case system performance.

During system integration, an experimental measure of rate group schedulability margin and peak load should be conducted to guard against inadvertent mistakes in parameter estimation, as follows:

1. Replace all tasks not yet written with dummy tasks using busy loops and dummy I/O.
2. Run all the tasks under stress scenarios, including lower priority tasks. Running low-priority tasks is important to check for priority inversions that may have been overlooked in analysis.
3. As permitted by the design, the engineer should run tests with heavy I/O workload and heavy application CPU workload concurrently.
4. Program the logic analyzer and capture the minimum schedulability margin for each rate group task (margin = deadline - completion time).
5. Plot the schedulability margin for each task: (system peak load =1.0 - the minimum of the task schedulability margins).

IMA is relatively new, and many engineers and even system architects are not discovering until system integration test that interpartition interference can be as high as 30-40%. Keeping interpartition interference to a minimum requires the following steps:

1. Partition structures with low-priority inversion and low interpartition interference.
2. Measure, validate, and track bounds of priority inversion and interpartition interference.
3. Develop a schedulability model.
4. Estimate, measure, and track schedulability margin and peak load.

8 Broader Applicability

Because the key stakeholder for this report is the Army PEO for Aviation, this report is intentionally focused toward avionics platform upgrades. However, these practices are applicable to other organizations that build and manage embedded systems. Examples of systems with embedded software that could benefit include satellite flight software, tactical communication systems, and terrestrial war-fighting vehicles. In addition, the VUV method is applicable for a wide range of problems related to obsolescence of hardware, technology refresh, and requirements for new capability.

To support the assertion that there is broad applicability for this work, we have included a few additional examples of mismatched assumptions that might have been avoided if architecture-centric, model-based engineering practices had been applied. One example of mismatched assumptions is the translation of functionality into application software causing the destruction of Ariane 5 during the rocket's maiden flight. Overflow of a 16-bit signed integer variable in reused Ariane 4 software to perform a function that was "not required for Ariane 5" was not caught due to a disabled handler [Wikipedia 2010]. The reason for the overflow was the representation of vertical velocity, which exceeded the 16-bit signed integer range due to a different altered flight path. A consistency check of value ranges between the value range of the variable and the expected value range of the domain parameter could have discovered this inconsistency.

In the late 1990s, a well-intentioned attempt to improve performance of a satellite ground station mission software had unplanned side effects. The subsystem tracking objects close to a spacecraft had originally sent a complete map of the objects to the command and control subsystem. In order to reduce the load on the network, a change was made to communicate only changes to the map. Unfortunately, this communication occurred over a network protocol that drops packets under overload conditions. As result, during integration testing, it was discovered that state changes randomly were not delivered. In other words, the data representation of the communicated data stream assumed guaranteed delivery. A consistency check between the communication QoS assumption of the application and the QoS provided by the protocol could have discovered this inconsistency.

When laptops with dual-core processors came out, iTunes crashed randomly when ripping a music CD [Apple 2005]. iTunes was designed as a multi-threaded application, with one thread determining the decibel level of tracks while the second thread converted the audio. A single-processor system executed first one thread, then the second thread. On a dual-core processor, the two concurrently executing threads were attempting to update the same music catalog without explicit synchronization. In other words, the original implementation assumed sequential execution of tasks to assure mutually exclusive access without using a synchronization mechanism. Similarly, as existing avionics code migrates from a federated system using a cyclic executive to an IMA architecture, concurrency race conditions may be encountered that did not exist in a federated setting due to strict execution order and dedicated processors. Testing for such race conditions may be very difficult. A consistency check of the assumed mutual exclusion mechanism for shared data access through sequential execution against the multi-core task deployment with concurrent task execution could have discovered this issue.

When system components become virtualized, assumptions about physical redundancy will be violated. In 1986, the internet, then ARPA-net, was accidentally split into two networks [Zakon 1993]. All seven New England trunk lines, which had previously been separate physical lines, were severed when AT&T suffered a fiber optic cable break that lasted 11 hours. When AT&T had converted to fiber optic cable, these physical trunk lines became logical trunk lines on this much higher bandwidth connection, losing all physical redundancy [HOC 2001]. The same virtualization occurs when embedded software is migrated to ARINC653 partitions or hard drives are partitioned into multiple logical drives [Wikipedia 2006]. A collocation consistency check could have discovered this inconsistency.

These additional examples of system-level faults have been reflected in the root cause areas of system-level faults identified in Section 3.4. In other words, we see broad applicability for architecture-centric, model-based engineering and the VUV method across many application areas in the Army and beyond.

9 Conclusion

In this report, we introduced readers to the VUV method for analyzing the impact of platform change on embedded systems. We described the VUV method steps and introduced the readers to the ADC, which provides guidance for modelers on what problems to model and how to model them, based on hands-on experience and real-world research.

We provided a history of the AADL standard, an overview of what the standard is, and some background regarding SEI and Army ASSIP work related to the AADL standard. We concluded the report with a discussion of the broad applicability for architecture-centric engineering supported by AADL models and the VUV method across the Army and beyond. We provided examples that demonstrate the wide range of domains impacted by system failure due to architectural mismatch.

In a pilot case study, we will use the VUV method to analyze the impact of a platform upgrade to the Apache helicopter. We will summarize the outcome of the case study and provide an assessment of the value of applying the VUV method to analyze the impact of the Apache platform upgrade in a separate report.

Appendix: Modeling with the SAE AADL

The SAE AADL standard provides formal modeling concepts for the description and analysis of an application system architecture in terms of distinct components and their interactions [Feiler 2006]. The AADL includes software, hardware, and system component abstractions to specify and analyze real-time embedded systems, complex systems of systems, and specialized performance capability systems and to map software onto computational hardware elements. The AADL is especially effective for model-based analysis and specification of complex real-time embedded systems.

In model-based analysis, the AADL supports the development of a comprehensive architecture model of a system. This single representation drives diverse analyses in support of software assurance practices throughout the development life cycle. These analyses address top-level concerns including requirements, system design, and software architecture. In addition, we can use AADL models to specify and validate compliance between the model and the source code.

SAE AADL: The Language

The core AADL modeling elements are organized into Components, Interactions, and Properties. Components represent distinct logical and physical entities that compose a system. Their external interfaces are defined as features of a component type. Their internal structure is defined using subcomponent declarations within a component implementation. Modes enable the modeling of dynamic reconfigurations of a system, including alternative properties, features, subcomponents, calls, and connections.

Interactions among components are established using explicit connections and software call declarations. These connections represent the transfer of control and data through ports; explicitly define access to data, buses, and subprograms; and define the exchange of data through subprogram parameters. Other component interactions, including the binding of software to hardware, are declared explicitly by assigning values to binding properties.

Properties, applied to various modeling elements, define the characteristics required for a complete architectural description, an executable implementation, and comprehensive analyses of a system. Values assigned to properties establish specific attributes of components and their interactions (e.g., the dispatch protocols supported by a processor), the binding of software components to hardware components (e.g., threads bound to processors), and the logistical aspects of deploying software (e.g., the name of a software component's source code file). There are standard (predeclared) properties defined within the language. For example, standard properties include the execution time for a thread, the speed of a processor, and the smallest independently readable and writeable unit of storage of a memory component. In addition, we can define new properties and associate them with AADL elements.

Component Abstractions

Within the AADL, a component is characterized by its identity (a unique name and runtime essence), possible interfaces with other components, distinguishing properties (critical characteristics of a component within its architectural context), and subcomponents and their interactions. In defining an architecture, we organize components into hierarchical runtime structures that can include dynamic reconfiguration using operational modes and mode transitions.

In addition to interfaces and internal structural elements, we can define other abstractions for component and system architectures. For example, abstract flows can be identified and associated with specific components and interconnections to perform flow analysis. These additional elements can be included through core AADL language capabilities and the specification of a supplemental annex language. The component abstractions of the AADL are separated into three categories: application software, execution platform (hardware), and composite. We have summarized these component categories in Table 9-1. Note that the execution platform category represented as "Computer System and Mission Platform."

Table 9-1: Component Categories

Application Software	
`thread`	Active component that can execute concurrently and be organized into thread groups
`thread group`	Abstraction for logically organizing thread, data, and thread group components within a process
`process`	Protected address space whose boundaries are enforced at runtime
`data`	Data types and static data in source text
`subprogram`	Concepts such as call-return and calls-on methods (modeled using a subprogram component that represents a callable piece of source code)
Computer System and Mission Platform	
`processor`	Schedules and executes threads
`memory`	Stores code and data
`device`	Sensors, actuators, or other components that interface with the external environment
`bus`	Interconnects processors, memory, and devices
`virtual processor`	Virtual machines, partitions, and hierarchical schedulers
`virtual bus`	Virtual channels and protocols
Composite	
`system`	Composite components that can consist of software, computer system, and mission platform components
`abstract`	Generic component that can be refined into any of the above component categories

Interactions

The AADL standard includes runtime semantics for component interactions including data sampling via data ports, message passing via event data ports, event passing via event ports, synchronized access to shared components via data access features, and remote service calls via subprogram access. Connections define relationships between ports, access features, subprogram parameters, and feature groups. Calls define relationships between calling components (threads and subprograms) and subprogram interfaces. Bindings of software components to hardware components are declared through property associations. We have summarized interactions and component features in Table 9-2.

Table 9-2: Interactions and Component Features

Connections	
`port`	Directional exchange of sampled data, events, and messages among components between ports or between feature groups of components
`access`	Relationships that enable multiple components to access a common `data` or `bus` component
`parameter`	Relationships among data elements associated with `subprogram calls`
`feature group`	Interaction between a collection of features represented as a single relationship
Calls	
`subprogram`	Relationships within component implementations that enable synchronous call/return access to `subprograms`
Bindings	
`binding`	Relationships that define the mapping of software components, connections, and subprogram calls to hardware components

Properties

Properties define characteristics for the elements that compose an AADL model. Each property has a name and a type. A property type establishes the values that can be assigned to a property.

There are AADL standard (predeclared) properties and property types. Collectively, these standard properties and property types encompass common attributes for the elements of the language. For example, a standard property of a port is `Required_Connection`, which is of standard type **aadlboolean** and has a default value of *true*. You can use a property association to assign the value *false* to this property for a port, allowing that port to be unconnected.

The AADL also permits users to define additional properties and property types. These are defined in property sets. For example, a new property for system components is declared with a name and a type for the property, and that property is applied to all system components in a model. The type declared for a new property may be a standard type (e.g., **aadlinteger**), or a new type that has been declared using a property type declaration.

AADL Annexes

We can use AADL properties and language extensions to create new and focused architecture analyses. Property sets can be declared that enable specialized analyses such as resource utilization. With the extension capabilities of the language, we can add additional models and properties.

Several such annexes have been defined as SAE AADL Annex standards. The Error Model Annex standard [SAE 2006] allows error sources, component error behavior, and error propagation to be associated with AADL models. These annotations can serve as a source for Functional Hazard Analysis (FHA), Fault Mode and Effects Analysis (FMEA), and Fault Tree Analysis (FTA), as well as reliability and availability analysis. The Behavior Annex standard [SAE 2011] supports state-based specifications to be associated with components to characterize their interaction behavior, concurrency control behavior, and functional behavior. The ARINC653 Annex standard [SAE 2011] supports specification of ARINC653 standard compliant architectures. The Data Model Annex standard [SAE 2011] supports association of data models expressed in other notations into an AADL architecture model.

Modeling Application Components

When modeling an embedded application, we can take two approaches: capturing its architecture as depicted in a design document or representing those system elements relevant to evaluating specific use scenarios. In the former case, we translate architecture diagrams into an AADL model using the graphical editor. The challenge in this case is to pick the appropriate AADL concepts to represent the architecture.

In the following, we outline some factors for consideration in modeling subprograms, threads, ports, and processes.

1. subprograms with the following characteristics
   a. execution time—specifies the time it takes to execute the program running by itself as a range between best-case and worst-case
   b. internal flows—specify the control and data flows that go from an input parameter to an output parameter of the subprograms
   c. input and output parameters—the parameters passed into the function
   d. input and output events—the parameters where the result of the computation is expected
2. threads with the following characteristics
   a. input/output ports—the input control and data into and out of the threads. These ports are then connected to other threads' ports or to the containing process ports.
   b. activation pattern—specifies whether the thread is activated periodically or only activated when an event arrives
   c. periodicity—must be specified if the thread is periodic
   d. deadlines—specify the maximum time that the thread can take to complete its execution starting from its activation
   e. subprogram calls—the sequence of calls that are executed every time the thread is activated
   f. connections to/from calls—the mapping from the thread ports to the parameters of the calls. This mapping is contained in each call.
   g. flows—specify the control and data flow from the thread ports to the subprogram calls and between the subprogram calls
   h. assignment of threads to processors—defines where the threads run and how to schedule them
3. ports with the following characteristics
   a. data port—represents communication of most recent state value; it is typically used for sampled processing
   b. event port—represents queued communication of events. This port is used for triggering thread execution or mode transitions as well as for sampled processing of alarms
   c. event data port—represents queued communication of data. This port is used for processing of complete message/data streams.

4. processes with the following characteristics
   a. input/output ports—connected either to other processes' ports or to the internal threads' ports. These ports parallel the input/output ports for threads.
   b. flows—represent the control and data flows between the process' internal threads or between the threads and the process' ports
   c. threads—the internal threads of the process
   d. connections to threads—the connections between the threads' ports and the thread and processes' ports. Timing constraints on sampled communication by periodic threads is specified as immediate, delayed, or sampled connection. *Immediate* means that data is always passed within the same frame by delaying the execution start of the recipient thread. *Delayed* means that passing of the data is always delayed to the next frame. *Sampled* means that the recipient samples data at dispatch time independently of the execution of the sending thread.

Modeling the Computer Platform

AADL offers the component categories of processor, memory, bus, and their groupings into systems to represent a wide range of computer system architectures, as has been demonstrated effectively in *Computer Structures: Readings and Examples* [Bell 1971]. Users can introduce different processor, bus, and memory types as abstractions of computer system components. For example, a processor type may represent bare processor hardware of an Intel X86 CPU, or it may include the operating system software (e.g., Linux). Similarly, bus types represent communication hardware and services such as PCI bus or Ethernet with or without different protocol stacks. These interconnected components are modeled by bus access connections between the respective hardware component and a bus using the system component.

When defining a processor type, we specify its scheduling policy and its processor speed and speed scaling factor relative to a reference processor. We also specify various performance parameters such as thread and process context switch time, supported priority levels, memory requirements for operating system software, protocol support, and reference to a detailed hardware description in a notation such as very high-speed integrated circuit (VHSIC) Hardware Design Language (VHDL). When defining a bus type, we specify its bandwidth, transmission time parameters, supported hardware component connectivity, provided QoS guarantees, and supported protocols.

Application software is bound to the computer system by binding source code and application data to memory. Similarly, application threads are bound to different processors, and port connections are bound to different buses. This provides the means to determine the workload on memory, buses, and processors based on application data (execution times, memory footprint, and communicated data volume).

We use the virtual processor concept to represent virtual machines such as ARINC653 partitions as well as hierarchical schedulers. Partitions also represent fault isolation boundaries. We use the virtual bus concept to represent virtual channels and communication protocols. It allows us to capture protocol stacks and record QoS properties such as guaranteed delivery, ordered delivery, and secure delivery of data. Assumptions about protocols and schedulers, as well as deployment on networks and processors made by the application, are recorded as properties on runtime architecture components such as threads and connections. A combination of virtual processor and virtual bus can be used in representing security levels and regions. Additional modeling guidance can be found in work by Delange [Delange 2009a, 2009b].

Glossary of Acronyms

Acronym	Definition
AADL	Architecture Analysis and Design Language
AB3	Apache Block Upgrade III
ADC	Architectural Dependencies Catalog
ADIRU	Air Data Inertial Reference Units
AED	Aviation Engineering Directorate
AMRDEC	Aviation and Missile Research Development and Engineering Center
ARINC	Aeronautical Radio Incorporated
ASIIST	Application Specific I/O Integration Support Tool
ASSIP	Army Strategic Software Improvement Program
ATAM	Architecture Tradeoff Analysis Method
AVSI	Aerospace Vehicle Systems Institute
CC	cruise control
CD	compact disk
COTS	commercial off-the-shelf
CPU	central processing unit
DARPA	Defense Advanced Research Project Agency
DMA	direct memory access
ESC	electronic stability control
FHA	Functional Hazard Analysis
FTA	Fault-Tree Analysis
GAO	General Accounting Office
I/O	input and output
IMA	Integrated Modular Avionics
IV&V	Independent Verification and Validation
MBE	model-based engineering
MIPS	Microprocessor without Interlocked Pipeline Stages
NASA	National Aeronautics and Space Administration
NIST	National Institute for Standards and Technology
OQA	operational quality attribute
OSATE	Open Source AADL Tool Environment
PCI	Peripheral Control Interface
PEO AVN	Program Executive Office Aviation
POC	proof of concept
RMA	Rate Monotonic Analysis
RT	real time
RTSCE	real-time, safety-critical, embedded
SAE	Society of Automotive Engineers
SAVI	System Architecture Virtual Integration
SEI	Software Engineering Institute
SLOC	source lines of code
SWCET	stall-induced worst-case execution time

Acronym	Definition
VHDL	VHSC Hardware Design Language
VHSIC	Very High Speed Integrated Circuit
VUV	Virtual Upgrade Validation
WCET	worst-case execution time

Bibliography

URLs are valid as of the publication date of this document.

[Apollo n.d.]
Apollo Software Publishing. *AirBus Fly-by-wire – How it Really Works.* n.d.
http:// www.apollosoftware.com/products/FlyByWire/FlyByWire_english.pdf
(Accessed on November 11, 2011.)

[Apple 2005]
Apple Support Communities, jazzman40. *iTunes Crashes When Ripping*.
http://discussions.apple.com/thread.jspa?messageID=1235236& (2005).

[Bell 1971]
Bell, C. G. & Newell, A. *Computer Structures: Readings and Examples*. McGraw-Hill Book Company, 1971.

[Barbacci 1995]
Barbacci, Mario, Klein, Mark H., Longstaff, Thomas A., & Weinstock, Charles B. *Quality Attributes* (CMU/SEI-95-TR-021). Software Engineering Institute, Carnegie Mellon University, 1995. http://www.sei.cmu.edu/library/abstracts/reports/95tr021.cfm

[Boydston 2009]
Boydston, Alex & Lewis, William. "Qualification and Reliability of Complex Electronic Rotorcraft Systems." *Army Helicopter Society System Engineering Meeting*, U.S. Army Aviation Engineering Directorate (AED), Aviation and Missiles Research, Development and Engineering Center (AMRDEC), October 2009.

[Budhiraja 1992]
Budhiraja, N., Marzullo, K., Schneider, F. B., & Toueg, S. "Primary-Backup Protocols: Lower Bounds and Optimal Implementation," 187-198. *Proceedings of the 3rd IFIP Conference on Dependable Computing for Critical Applications*. Mondello, Italy, September 1992. Springer-Verlag, 1992.

[Casteres 2008]
Casteres, Jean, Callaud, Jean-Marie, & Gaudaire, Stéphane. "Technology Evolution of Aircraft Simulator for Real Equipments Validation," 5. *Proceedings of the 4th European Congress on Embedded Real-time Software (ERTS 2008)*. Toulouse, France, January–February, 2008. Societe des Ingenieurs de l'Automobile, 2008. Available through
http://www.sia.fr/dyn/publications_detail.asp?codepublication=R-2008-01-3B02

[Cervin 2006]
Cervin, A., Årzén, K.-E. & Henriksson, D. "Control Loop Timing Analysis Using TrueTime and Jitterbug," 1194-1199. *Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design (CACSD)*. Munich, Germany, October 2006. IEEE, 2006.

[Clements 2001]

Clements, Paul, Kazman, Rick, & Klein, Mark. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2001.

[Conquet 2008]

Conquet, E. "ASSERT: A Step Towards Reliable and Scientific System and Software Engineering." *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS 2008)*. Toulouse, France, January–February 2008. Available through "Proceedings" at http://www.erts2008.org/

[de Niz 2006]

de Niz, Dionisio & Rajkumar, Raj. "Partitioning Bin-Packing Algorithms for Distributed Real-Time Systems." *International Journal of Embedded Systems: Special Issue on Design and Verification of Real-Time Embedded Software 2,* 3/4 (2006), 196-208.

[de Niz 2008a]

de Niz, Dionisio. "Architectural Concurrency Equivalence with Chaotic Models," 57-67. *Proceedings of the 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES2008)*. Budapest, Hungary, April 2008. IEEE, 2008. http://doi.ieeecomputersociety.org/10.1109/MOMPES.2008.10

[de Niz 2008b]

de Niz, Dionisio & Feiler, Peter. "On Resource Allocation in Architectural Models," 291-297. *Proceedings of the 11th IEEE International Symposium on Object/Service-Oriented Real-Time Distributed Computing.* Orlando, FL (USA), May 2008. IEEE, 2008.

[de Niz 2009]

de Niz, Dionisio & Feiler, Peter. "Verification of Replication Architectures in AADL," 365-370. *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS09), UML&AADL Workshop*. Potsdam, Germany, June 2009. IEEE, 2009. http://doi.ieeecomputersociety.org/10.1109/ICECCS.2009.18

[Delange 2008]

Delange, J., Hugues, J., Pautet, L., & Zalila, B. "Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain." *Proceedings of the 4th European Congress on Embedded Real-time Software (ERTS 2008)*. Toulouse, France, January–February, 2008. Available through "Proceedings" at http://www.erts2008.org/

[Delange 2009a]

Delange, Julien, Pautet, Laurent, & Feiler, Peter. "Validating Safety and Security Requirements for Partitioned Architectures," *Proceedings 14th International Conference on Reliable Software Technologies (RTS2009)—Ada Europe*. Brest, France, June 2009. Lecture Notes in Computer Science 5570, Springer, 2009. http://julien.gunnm.org/data/publications/article-dpf-rst09.pdf

[Delange 2009b]

Delange, Julien, Pautet, Laurent, Plantec, Alan, Kerboeuf, Yves, Singhoff, Frank, & Kordon, Fabrice. "Validate, Simulate, and Implement ARINC653 Systems," 31-44. *Proceedings of the*

ACM SIGAda Annual International Conference on Ada and Related Technologies. St. Petersburg, FL (USA), November 2009. ACM, 2009.

[Douglass 2003]

Douglass, Bruce Powel. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley, 2003.

[Feiler 1998]

Feiler, Peter H. *Upgrading Avionics Systems: A Case Study*. DARPA EDCS Project Report, June 1998.

[Feiler 2004]

Feiler, Peter H., Gluch, David P., Hudak, John, & Lewis, Bruce A. “Pattern-Based Analysis of an Embedded Real-time System Architecture.” *Proceedings of IFIP World Computer Congress - Workshop on Architecture Description Languages (WADL04)*, *176/2005*. Toulouse, France, August 2004. Springer, 2004.

[Feiler 2006]

Feiler, Peter H., Gluch, David, & Hudak, John. *The Architecture Analysis and Design Language (AADL): An Introduction* (CMU/SEI-2006-TN-011). Software Engineering Institute, Carnegie Mellon University, 2006. http://www.sei.cmu.edu/library/abstracts/reports/06tn011.cfm

[Feiler 2007a]

Feiler, Peter H. & Hansson, Jőrgen. *Flow Latency Analysis with the Architecture Analysis and Design Language (AADL)* (CMU/SEI-2007-TN-010). Software Engineering Institute, Carnegie Mellon University, 2007. http://www.sei.cmu.edu/library/abstracts/reports/07tn010.cfm

[Feiler 2007b]

Feiler, Peter H. & Rugina, Ana. *Dependability Modeling with the Architecture Analysis and Design Language (AADL)* (CMU/SEI-2007-TN-043). Software Engineering Institute, Carnegie Mellon University, 2007. http://www.sei.cmu.edu/library/abstracts/reports/07tn043.cfm

[Feiler 2008a]

Feiler, Peter H. & Hansson, Jőrgen. “Impact of Runtime Architectures on Control System Stability.” *Proceedings of the 4th European Congress on Embedded Real-time Software (ERTS 2008)*. Toulouse, France, January–February, 2008. Available through “Proceedings” at http://www.erts2008.org/

[Feiler 2008b]

Feiler, Peter H. & de Niz, Dionisio. *ASSIP Study of Real-Time Safety-Critical Embedded Software-Intensive System Engineering Practices* (CMU/SEI-2008-SR-001). Software Engineering Institute, Carnegie Mellon University, 2008. http://www.sei.cmu.edu/library/abstracts/reports/08sr001.cfm

[Feiler 2008c]

Feiler, Peter H. “Efficient Embedded Runtime Systems through Port Communication Optimization.” *Proceedings of the 13th IEEE International Conference on Engineering of*

Complex Computer Systems (ICECCS08), UML&AADL Workshop. Belfast, Northern Ireland, April 2008. IEEE, 2008. http://doi.ieeecomputersociety.org/10.1109/ICECCS.2008.20

[Feiler 2009a]

Feiler, P. H., Hansson J., de Niz, D., & Wrage L. *System Architecture Virtual Integration: An Industrial Case Study* (CMU/SEI-2009-TR-017). Software Engineering Institute, Carnegie Mellon University, 2009. http://www.sei.cmu.edu/library/abstracts/reports/09tr017.cfm

[Feiler 2009b]

Feiler, Peter H. "Challenges in Validating Safety-Critical Embedded Systems." *Proceedings of SAE International AeroTech Congress*. Seattle, WA (USA), November 2009. SAE International, 2009. http://papers.sae.org/2009-01-3284/. Also a most outstanding SAE Technical Paper of 2010 in *SAE International Journal of Aerospace 3*, 1 (December 2010):109-116. Available through http://saeaero.saejournals.org/content/3/1/109.full.pdf+html

[Feiler 2010]

Feiler, Peter H. "Model-Based Validation of Safety-Critical Embedded Systems," 1-10. *Proceedings of IEEE Aerospace Conference*. Big Sky, MT (USA), March 2010. IEEE, 2010. http://dx.doi.org/10.1109/AERO.2010.5446809

[GAO 2008]

General Accounting Office. Report to Congressional Committees. "DOD's Goals for Resolving Space Based Infrared System Software Problems Are Ambitious" (GAO-08-1073). September 2008. http://www.gao.gov/new.items/d081073.pdf

[Hansson 2008]

Hansson, Jőrgen, Feiler, Peter H., & Morley, John. "Building Secure Systems Using Model-Based Engineering and Architectural Models." *CrossTalk: The Journal of Defense Software Engineering* (September 2008): 10-14.

[HOC 2001]

"The History of Computing Project." *History of the Internet*. http://www.thocp.net/reference/internet/internet2.htm (2001).

[Hugues 2008]

Hugues, J., Pautet, L., Zalila, B., Dissaux, P., & Perrotin, M. "Using AADL to Build Critical Real-Time Systems: Experiments in the IST-ASSERT project." *Proceedings of the 4th European Congress on Embedded Real-Time Software (ERTS 2008)*. Toulouse, France, January–February, 2008. Available through "Proceedings" at http://www.erts2008.org/

[Jackson 2007]

Jackson, Daniel, Thomas, Martyn, & Millet, Lynette I., eds. *Software for Dependable Systems: Sufficient Evidence?* Committee on Certifiably Dependable Software Systems, National Research Council. National Academic Press, 2007. ISBN: 0-309-10857-8.

[Klein 1993]
Klein, Mark, Ralya, Thomas, Pollak, Bill, Obenza, Ray, & Gonzalez Harbour, Michael. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems*. Springer, 1993.

[LaCerte 2007]
LaCerte, Yves. "FPGAs: High Assurance through Model Based Design." *AADL Workshop*. January 2007. Rockwell-Collins, 2007. http://aadl.sei.cmu.edu/aadl/documents/Rockwell_FPGA-AADL_Jan2007.pdf

[Leveson 2004]
Leveson, Nancy. "The Role of Software in Spacecraft Accidents." *AIAA Journal of Spacecraft and Rockets, 41*, 4 (July 2004): 564-575. http://sunnyday.mit.edu/papers/jsr.pdf

[Li 2003]
Li, Allen. "Testimony by Allen Li, Director, Acquisition and Sourcing Management, U.S. General Accounting Office, to the House Subcommittee on Tactical Air and Land Forces, Committee on Armed Services." U.S. General Accounting Office, April 2003. http://www.gao.gov/new.items/d03603t.pdf

[Miller 2005]
Miller, Steve P., Wallen, Mike W., O'Brien, Dan, Heimdahl, Mats P., & Joshi, Anjali. *A Methodology for the Design and Verifications of Globally Asynchronous/Locally Synchronous Architectures* (Technical Report NASA/CR-2005-213912). NASA, 2005.

[Mohan 2009]
Mohan, Sibin, Nam, Min-Young, Pellizoni, Rodolfo, Sha, Lui, Bradford, Richard, & Fliginger Shana. "Rapid Early-Phase Virtual Integration," 33-44. *Proceedings of 30th IEEE Real-Time Systems Symposium*. Washington, DC, December 2009. IEEE, 2009.

[Nam 2009]
Nam, M.-Y., Pellizzoni, R., Sha, L., & Bradford, R. M. "ASIIST: Application Specific I/O Integration Support Tool for Real-Time Bus Architecture Designs," 11-22. *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009)*. Potsdam, Germany, June 2009. IEEE, 2009. http://dx.doi.org/10.1109/ICECCS.2009.31

[NASA 2009a]
Dvorak, Daniel L. *NASA Study on Flight Software Complexity* (Technical Report NASA/CR-2005-213912). NASA Office of Chief Engineer Technical Excellence Program, March 2009.

[NASA 2009b]
Gluch, Dave & Feiler, Peter. *A Practice Framework for Model-based Analysis with AADL* (Technical Report MAC-T IVV-09-020). NASA, February 2009.

[SAE 1996]
SAE International Aerospace Division. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment* (Aerospace Recommended Practice ARP4761). SAE International, 1996. http://standards.sae.org/arp4761

[SAE 2004]
SAE International. *Architecture Analysis and Design Language 1* (SAE Document AS-5506). SAE International, 2004. (See v2.0 listing, [SAE 2009])

[SAE 2006]
SAE International. *Architecture Analysis & Design Language (AADL) Annex 1.0* (SAE Document AS-5506/1). SAE International, 2006. http://standards.sae.org/as5506/1

[SAE 2009]
SAE International. *Architecture Analysis and Design Language 2.0* (SAE Document AS-5506A). SAE International, 2009. http://standards.sae.org/as5506a

[SAE 2011]
SAE International. *Architecture Analysis & Design Language (AADL) Annex 2* (SAE Document AS-5506/2), SAE International, 2011. http://standards.sae.org/as5506/2

[SEI 2009a]
Software Engineering Institute. *Training Course: Modeling System Architectures Using AADL.* Carnegie Mellon University, 2009. http://www.sei.cmu.edu/training/p72.cfm

[SEI 2009b]
Software Engineering Institute. *Open Source AADL Tool Environment.* Carnegie Mellon University, 2009. http://www.aadl.info

[Singhoff 2005]
Singhoff, F., Legrand, J., Nana, L., & Marcé, L. "Scheduling and Memory Requirement Analysis with AADL." *ACM SIGAda Ada Letters 25*, 4 (November 2005): 1-10. ISSN:1094-3641.

[Sokolsky 2006]
Sokolsky, Oleg, Lee, Insup, & Clarke, Duncan. "Schedulability Analysis of AADL Models." *20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006)*. Rhodes Island, Greece, April 2006. IEEE, 2006. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1639421

[Sokolsky 2009]
Sokolsky, Oleg, Lee, Insup, & Clarke, Duncan. "Process-Algebraic Interpretation of AADL Models," 222-236. *14th International Conference on Reliable Software Technologies - Ada-Europe'2009 (Ada-Europe 2009)*. Brest, France, June 2009. ACM, 2009.

[Thiele 2000]
Thiele, L., Chakraborty, S., & Naedele, M. "Real-Time Calculus for Scheduling Hard Real-Time Systems," 101-104. *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems (ICSAS 2000)*. Geneva, Switzerland, May 2000. IEEE, 2000. http://dx.doi.org/10.1109/ISCAS.2000.858698

[Tri-Pacific 2011]
Tri-Pacific Software, Inc. *RAPID RMA: The Art of Modeling Real-Time Systems*, 2011. http://www.tripac.com/rapid-rma

[Wikipedia 2006]

Wikipedia. *ARINC653 (Avionics Application Standard Software Interface)*. http://en.wikipedia.org/wiki/ARINC_653 (2006).

[Wikipedia 2008]

Wikipedia. *Qantas Flight 72*. http://en.wikipedia.org/wiki/ Qantas_Flight_72 (2008).

[Wikipedia 2010]

Wikipedia. *Ariane 5 Flight 501*. http://en.wikipedia.org/wiki/ Ariane_5_Flight_501 (2010).

[Zakon 1993]

Zakon, Robert. *Hobbes' Internet Timeline 10.1*, 1993.http://www.zakon.org/robert/internet/timeline/

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. **AGENCY USE ONLY**
   (Leave Blank)

2. **REPORT DATE**
   June 2012

3. **REPORT TYPE AND DATES COVERED**
   Final

4. **TITLE AND SUBTITLE**
   A Virtual Upgrade Validation Method for Software-Reliant Systems

5. **FUNDING NUMBERS**
   FA8721-05-C-0003

6. **AUTHOR(S)**
   Dio de Niz, Peter Feiler, David P. Gluch, Lutz Wrage

7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
   Software Engineering Institute
   Carnegie Mellon University
   Pittsburgh, PA 15213

8. **PERFORMING ORGANIZATION REPORT NUMBER**
   CMU/SEI-2012-TR-005

9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
   HQ ESC/XPK
   5 Eglin Street
   Hanscom AFB, MA 01731-2116

10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**
    ESC-TR-2012-005

11. **SUPPLEMENTARY NOTES**

12A **DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified/Unlimited, DTIC, NTIS

12B **DISTRIBUTION CODE**

13. **ABSTRACT (MAXIMUM 200 WORDS)**

This report presents a Virtual Upgrade Validation (VUV) method to improve design quality and confidence in qualification through testing for military systems impacted by computer platform changes. This approach uses architecture-centric, model-based analysis to identify system-level problems early in the upgrade process to complement established test qualification techniques. For purposes of this report, the authors focus on changes to the computer platform consisting of processors, network, operating system, and runtime infrastructure. They describe the VUV method steps and introduce the Architectural Dependencies Catalog that provides guidance for modelers on which aspects of the system to model and how to model them. The report also provides a history and overview of the Architecture Analysis and Design Language standard, which is used with the VUV method.

14. **SUBJECT TERMS**
    virtual integration, AADL, embedded system architecture, real-time, safety-critical

15. **NUMBER OF PAGES**
    84

16. **PRICE CODE**

17. **SECURITY CLASSIFICATION OF REPORT**
    Unclassified

18. **SECURITY CLASSIFICATION OF THIS PAGE**
    Unclassified

19. **SECURITY CLASSIFICATION OF ABSTRACT**
    Unclassified

20. **LIMITATION OF ABSTRACT**
    UL