AFRL-IF-WP-TR-2001-1506

INCREMENTAL SOFTWARE EVOLUTION FOR REAL-TIME SYSTEMS (INSERT)

J. LEHOCZKY
P. FEILER
B. KROGH
T. MARZ
R. RAJKUMAR
B. CALLONI
J. PRESTON

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Lockheed Martin
Tactical Aircraft Systems
Mail Zone 2445
P.O Box 748
Fort Worth, TX 76101

JANUARY 2001

FINAL REPORT FOR PERIOD 01 SEPTEMBER 1997 – 30 DECEMBER 2000

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

NOTICE

USING GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA INCLUDED IN THIS DOCUMENT FOR ANY PURPOSE OTHER THAN GOVERNMENT PROCUREMENT DOES NOT IN ANY WAY OBLIGATE THE US GOVERNMENT. THE FACT THAT THE GOVERNMENT FORMULATED OR SUPPLIED THE DRAWINGS, SPECIFICATIONS, OR OTHER DATA DOES NOT LICENSE THE HOLDER OR ANY OTHER PERSON OR CORPORATION; OR CONVEY ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY RELATE TO THEM.

THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

KENNETH LITTLEJOHN
Project Engineer

JAMES S. WILLIAMSON, Chief
Embedded Info Sys Engineering Branch
Information Technology Division
Information Directorate

WALTER B. HARTMAN
Acting Wright Site Coordinator
Information Directorate

Do not return copies of this report unless contractual obligations or notice on a specific document require its return.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

	January 2001	Final Report, 09/01/1997 – 12/30/2000

INCREMENTAL SOFTWARE EVOLUTION FOR REAL-TIME SYSTEMS (INSERT)

J. LEHOCZKY, P. FEILER, B. KROGH, T. MARZ, R. RAJKUMAR, B. CALLONI, AND J. PRESTON

C: F33615-97-C-1012
PE: 62302E
PR: 3090
TA: 01
WU: 13

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Lockheed Martin
Tactical Aircraft Systems
Mail Zone 2445
P.O Box 748
Fort Worth, TX 76101

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AFB, OH 45433-7334
POC: Kenneth Littlejohn, AFRL/IFTA, (937) 255-6548 x3587

AFRL-IF-WP-TR-2001-1506

Approved for public release; distribution unlimited.

INSERT is a capability package designed to support safe on-line upgrades of software components in real-time systems and the safe insertion of new capabilities into those systems. High reliability is guaranteed through the use of run-time monitoring and switching. The INSERT run-time monitor can detect and overcome semantic, data, and system errors. The run-time system is complemented with tools for off-line analysis and design to support development and implementation of INSERT-protected systems. This report documents the architecture and the associated middleware. In addition, the capability package contains methods for verification of the INSERT switching rules and Analytic Redundancy Component (ARC) based verification methods (which are also referred to as dependency tracking methods). The report documents a major experiment in which the INSERT architecture was implemented in the Lockheed Martin F-16 ground-based simulators. The Automated Maneuvering and Attack System (AMAS) algorithm was then installed. The INSERT architecture successfully protected the system against residual software faults. A Lockheed Martin cost estimation process concluded that the INSERT architecture could result in a reduction of 20% in labor hours in a real-time safety critical system.

Real-time systems, Safety-critical systems, Software upgrades, Simplex architecture, F-16, AMAS algorithm

110

Unclassified	Unclassified	Unclassified	SAR

TABLE OF CONTENTS

List of Figures

List of Tables

1. Introduction

This document presents the final report of the INcremental Software Evolution for Real-Time Systems (INSERT) Project. The project was carried out during 1996-2000 by Carnegie Mellon University and Lockheed Martin Tactical Aircraft Systems (LMTAS) with the support of the Defense Advanced Research Projects Agency (DARPA) Evolutionary Design of Complex Software (EDCS) Program and the Air Force Research Laboratory (Wright-Patterson) under contract F33615-97-C-1012.

The INSERT project is a capability package designed to support safe on-line upgrades of software components in real-time systems and the safe insertion of new capabilities into those systems. High reliability is guaranteed through the use of run-time monitoring and switching. The INSERT run-time monitor can detect and overcome semantic, data, and system errors. The run-time system is complemented with tools for off-line analysis and design to support development and implementation of INSERT-protected systems.

The INSERT package aids in the construction of upgradable, explicitly fault-tolerant systems. Several upgrade strategies can be supported, including the following:

- upgrading legacy systems
- inserting new functionality
- enhancing existing functionality (incremental evolution)
- constructing virtual federated systems.

The protected, analytically redundant systems constructed with INSERT can contribute to several other aspects of the software development process, including the following:

- early live system testing (i.e., permitting system testing in the live environment without compromising safety)
- nonstop testing (i.e., recovering and continuing testing though errors that ordinarily stop testing occur, a feature that is especially valuable in flight testing)
- factorial experimentation (i.e., several experiments can be scheduled with INSERT managing the switching between cases and protection from faults).

The INSERT capability package consists of five major components, described in the following subsections.

1.1 Application-Independent Run-Time Services (Middleware)

The INSERT run-time services support dynamic binding, monitoring, fault detection and switching. The dynamic binding services permit the interconnection of related subsystems during run-time processing, while preserving the execution time and address space partitioning of the system. The activities of dynamic binding services are done with the support of a real-time, publisher/subscriber interprocess communication messaging service. This service may also be externally managed, allowing system operators to add and delete system components, as well as to enable or disable the Interprocess Communication (IPC) links of a system component. System visualization and control tools support system management using an application-independent facility.

A prototype analytically redundant application environment is also provided in the INSERT package, allowing system developers to construct analytically redundant execution environments

quickly and easily. Application-independent methods of constructing monitoring and control functions have been developed, and are being evaluated.

These middleware services can play a role in many phases of the software development cycle, including testing, implementation, deployment, run time and maintenance.

1.2 Data Fusion Integrity Processes (DFIP)

The INSERT software architecture uses standard signal conditioning methods for sensor input and both DFIP-in (detection of input data errors) and DFIP-out (detection of output data errors caused possibly by semantic errors in the software) to detect and/or correct or compensate for data errors.

The DFIP capabilities contribute to run-time safety and hence are a part of the deployment phase of the software development cycle.

1.3 Dependency Tracking Tools

Previously hidden dependencies between system components (especially application-specific semantic and time-sensitive dependencies) are captured in an architectural system model. These hidden dependencies can be identified by examination of maintenance error logs and results of root cause analysis. The system model can then be incrementally enriched to reduce the hidden dependencies. Using such system models and static analysis tools, one can identify configuration inconsistencies in multivariant components and determine version constraints to semantic configuration consistency to be checked at run time. Analysis tools can also identify the set of components that depend on the changed component whose assumptions and constraints might be violated by the change. Determining the set of affected components allows developers to better estimate development efforts, and helps keep validation and testing efforts in proportion with the scope of change.

Dependency tracking tools contribute to the design, testing, implementation and maintenance phases of the software development cycle. They also identify sets of changes that must be made together to ensure correct system behavior.

1.4 Analysis Tools for Application-Dependent Switching Rules

The application-dependent switching rules in an INSERT analytically redundant unit (ARU) may be derived using heuristics and past experience. To certify that the switching rules will provide the desired protection against software faults, a new analytic method has been developed to analyze the reachability of systems with both discrete dynamics (the INSERT switching logic) and continuous dynamics (the environment and continuous actions of a given variant within the ARU). Properties of finite-state models generated automatically from the complete system dynamics are analyzed to verify the correctness and completeness of the switching rules. The verification tools are implemented in Matlab with a graphical user interface (GUI).

The INSERT tools for switching rule verification fall into the test phase of the software development cycle, which includes analysis tools.

1.5 Application-Dependent Switching Rules for High-Performance Avionics Systems

A key capability of INSERT is the ability to protect the system from arbitrary semantic faults that could be introduced by software upgrades. This protection is achieved through an analytic method that creates an envelope within which the trajectories of the high-performance avionics systems, such as the F-16, are required to lie. The determination of this envelope occurs during the design and implementation phases of the software development cycle, and it is deployed at run time. When combined with the temporal, spatial and system error protection afforded by the INSERT middleware, these switching rules provide a means to safely test new components early in the software development cycle.

The application-dependent switching rules play a role in the development, test, deployment, and maintenance phases of the software development cycle.

INSERT can be categorized as having a level three maturity (being used by integrators or other service/industrial organizations outside EDCS - Beta delivery). Specifically, INSERT middleware integration in the Lockheed Martin F-16 Ground Based test simulators at Ft. Worth has been successfully completed. The details of this integration are fully documented in this report. Note that the INSERT architecture is based in part on the Simplex architecture, which was developed and is being transitioned by the Software Engineering Institute (SEI). Research and Development into INSERT is also being transitioned through the SEI Simplex architecture. Specific transitions of Simplex include the following:

- U.S. Navy has identified the SEI Simplex architecture as a technology refreshment candidate on the National Standards Systems Network (NSSN) program, and is considering transitioning the technology into the DD-21 program.
- U.S. Army Aviation and Missile Command (AMCOM) has received the Beta version of the Simplex middleware and proposed a transition plan to support Simplex methods and concepts in the Real-Time Executive for Multiprocessor Systems (RTEMS) operating system environment.

In summary, INSERT is unique in providing a solution to the software fault tolerance problem for safety critical real-time systems. Moreover, INSERT offers a safe method to include new functionality into a system. Finally, INSERT offers coverage against a wide range of error types, supports the safe use of certain Commercial off the Shelf (COTS) products and permits early live testing of safety-critical systems. The INSERT project was designed to develop a comprehensive methodology for the safe upgrading of these systems accompanied by middleware, verification software, and dependency tracking software. Most importantly, the project was designed to demonstrate the viability of the approach on real DoD platforms and to estimate the cost benefits that might accrue from the presence of an INSERT infrastructure. The implementation demonstration and cost-effectiveness aspects were conducted primarily by a team at LMTAS. The INSERT infrastructure was implemented in the LMTAS Ft. Worth F-16 Ground-Based simulation facility. Using the INSERT, protected facility, the LMTAS team then implemented the AMAS and demonstrated protection against injected errors. The LMTAS team also conducted a cost-effectiveness study for real-time, safety-critical system software. They concluded that a manpower reduction of approximately 20 percent would accrue using an INSERT infrastructure.

The remainder of this report is organized in topical sections. Section 2 introduces the concept of analytical redundancy as the foundation for providing run-time protection from software faults. The run-time middleware services that support the INSERT system are described in Section 3.

Section 4 describes a new technology for performing formal verification of INSERT switching rules and its application to the F16 autopilot case study. Tools for validation of the INSERT component structure, including semantic features of the components, are described in Section 5. Sections 6 and 7 deal with the application of INSERT technology to avionics systems at Lockheed Martin. Section 6 describes the complete integration of the INSERT run-time system into the F16 ground-based simulation system and its support of software upgrades. Section 7 presents a full cost analysis demonstrating the potential impact of INSERT technology on the development of future avionics systems.

2. Analytical Redundancy

2.1 Introduction

Simplex [1] is an engineering approach to providing dependable upgrades to real-time systems. It combines several fault tolerance techniques including analytic redundancy [2] to protect against faults in upgraded software components, and it allows component variants to be replaced. From an application design perspective, this form of fault tolerance to support dependable upgrades is embodied in the concept of ARC [3]. Run-time support for fault tolerance and replacement is referred to as Simplex middleware, i.e., a *set* of services that implements the ARC concept in terms of services available through a real-time operating system or other form of run-time support.

The objective of the ARC concept is to provide fault tolerance to application components in order to make their upgrade more reliable. This is done by supporting multiple variants of a component, by monitoring component variants for misbehavior, and by utilizing alternate variants for recovery from faults.

An ARC is a multivariant component that supports run time variant selection. Fault tolerance is provided by switching variants as recovery action due to observed faults. The input is made available to all variants, and the output of each variant is passed to a Monitoring and Decision Unit (MDU) to determine which output is passed on as the output of the component. This is illustrated in Figure 1. The MDU supports the concept of a leader, i.e., an identified variant whose output is intended to be used as the component output. The MDU acts as a data flow router, passing on the output of the selected leader. From a configuration management perspective, the MDU acts as a logical reconfiguration agent, identifying at run time the component variant that is part of a logical system configuration.

The following sections discuss fault-tolerant components, then describe an approach to implementing ARCs. A discussion of Simplex-based application design also appears.

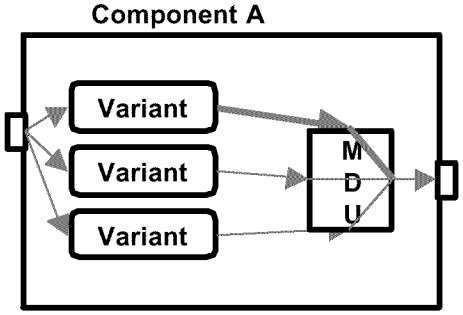


Figure 1. A Multivariant Component

2.2 A Fault-Tolerant Component

The ARC also acts as a fault manager. Some variants are considered *volatile*, i.e., they may misbehave, while others are "trusted" in that they are relied upon for recovery in case of misbehavior.

Address space protection is used to prevent volatile variants from damaging other parts of the application. The MDU of an ARC monitors volatile variants and takes recovery action as appropriate. The MDU monitors

- whether the leader produces output in a timely manner (observing the effect of execution time and deadline overrun as well as run-time errors that may stop or delay the execution of the variant)
- whether the leader output values are in an acceptable range
- whether the state of the controlled plant (either current state or projected state) is within an acceptable region.

This acceptable region, referred to a *safety region*, is determined by the performance characteristics of one of the variants identified as the safety variant. This is the variant relied upon to recover from volatile variant faults. This approach to fault tolerance differs from a majority voting approach. Here the output and its effects on external systems states are evaluated against a reference model, while in majority voting the output of all variants is considered in deciding the choice of output.

An ARC typically consists of three variants: a safety variant, a baseline variant, and an upgrade variant, although the use of these three variants is not an inherent restriction. The baseline variant is able to achieve the control objective of the component, while the focus of the safety variant is on providing a safety region that may be larger than the operational region of baseline. The safety variant's primary control objective is to recover the system to the operational region of baseline. In other words, the safety variant may not achieve the control objective by itself. The upgrade variant may achieve the control objective with improved performance. Figure 2 visualizes the use of the variants for fault tolerance.

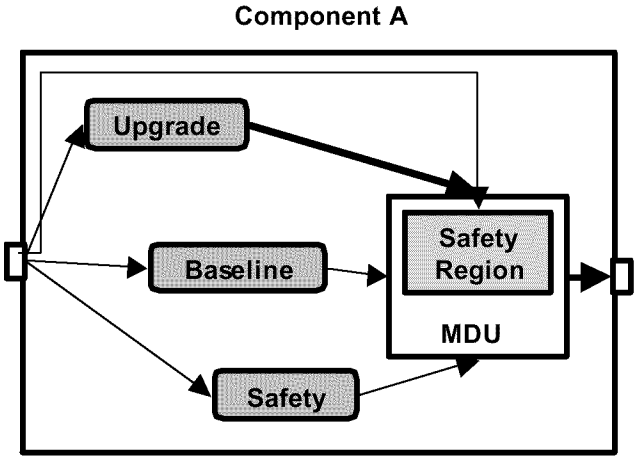


Figure 2. Analytically Redundant Component

The protocol for managing variant leadership works as follows. The baseline variant is the typical leader; changes are introduced through the upgrade variant, and leadership is given to it. If it fails, control is given to the safety variant to recover from a safety region violation and then handed back to the baseline variant, or handed directly to the baseline variant if a timeliness violation is observed though the safety region constraint not been violated. As a result of the protocol for variant management, the control objective of the component will be achieved with occasional transients of recovery through the safety variant, as shown in Figure 3.

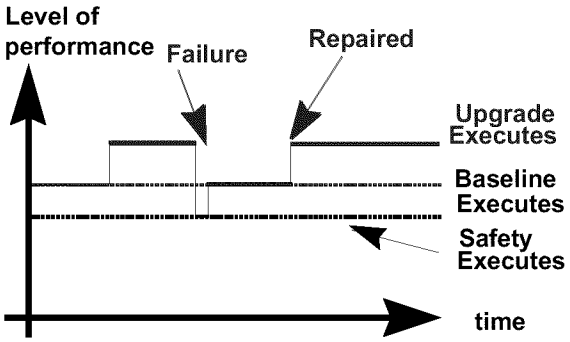


Figure 3. Achieving the Control Objective

2.3 Component Evolution

Components can be evolved by updating the upgrade variant or the baseline variant (see Figure 4). Those updates may be achieved by run time replacement of the upgrade and baseline variants, even while the system continues to operate. Modifications and improvements to the component are instantiated as an upgrade variant. If the upgrade misbehaves, the fault tolerance protocol will switch leadership to the baseline variant, possibly requiring recovery via the safety variant. The upgrade can then be corrected and replaced as the new upgrade variant. Once the upgrade variant has passed its test and is stable, it can become the baseline variant, and the next iteration of the evolution can begin.

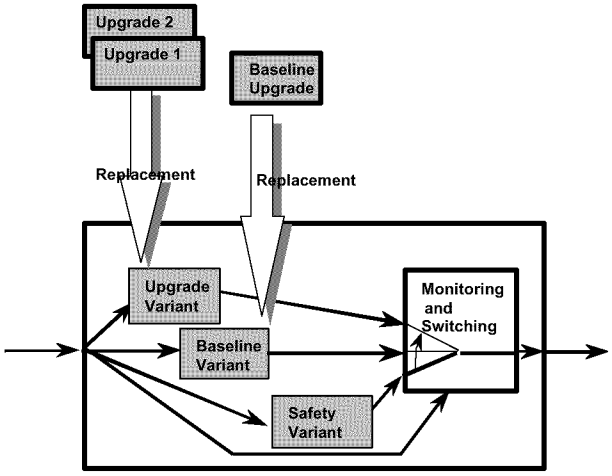


Figure 4. Component Evolution

2.4 Variability in ARCs

ARCs can vary along several dimensions:

- fault recovery protocol
- fault categories
- variant instantiation pattern.

The fault recovery protocol dimension focuses on variations in the protocol used to recover from faults. If the observed misbehavior is considered to be transient, a (limited) retry policy may be warranted. For example, if the cause of the fault can be attributed to another system component

rather than the volatile variant, then the variant can be retried. We may also consider temporarily delaying the retry until the faulty component has recovered.

A second dimension of variability focuses on the classes of faults to be monitored explicitly. At a minimum, Simplex assumes that mechanisms are in place to determine whether a variant is providing output and providing it within the allotted time. Simplex also assumes the operation of mechanism that determine whether a semantic condition representing the safety region is violated (typically focusing on prevention of damage to the controlled plant). Additional application conditions to be monitored include deviation from control objectives and certain *performance* characteristics in achieving them (see Sections 6 and 7), as well as *configuration constraints*, i.e., assumptions made about which variant of other components is the active one. The result is that additional monitoring constraints that complement the safety region constraint are provided by the application developer (see Figure 5). Some of these constraints are specific to the component variant and must be upgraded in conjunction with the component variant itself. The fault recovery protocol needs to be adjusted to accommodate these additional constraints. In some circumstances these constraints act as an enforcement mechanism, i.e., fault recovery is triggered by a constraint violation. In other circumstances, monitoring is used to validate the model embedded in the constraint, e.g., on-line comparison of a performance characterization of a control system component against the actual component.

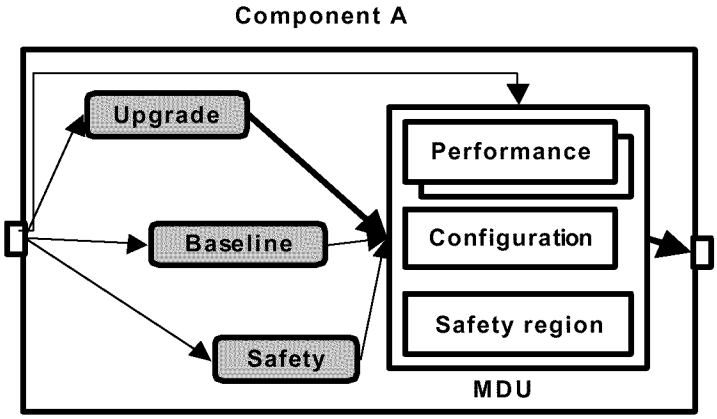


Figure 5. ARC with Multiple Constraints

Furthermore, system faults may be observed as unresponsive volatile variant, e.g., failure of the processor executing the volatile variant, or fault or intrusion in the communication mechanism used to pass the volatile variant output to the MDU. In some cases such faults are explicitly reported by the run-time system and can be taken into account in the fault recovery protocol.

A third dimension of variability is determined by the characteristics of the component variants and the fault tolerance objective for the component. For example, with a component that is a controller with a control objective, the fault tolerance objective may be to maintain this control objective, or to prevent damage to the controlled device. In the former case, ARCs get instantiated in one of two patterns: 1) a safety variant exists that can perform recovery as well as pursue the control objective, or 2) a safety variant performs recovery and a separate baseline variant achieves the control objective. In the latter case an ARC pattern with a safety variant that can perform recovery is sufficient. Thus, we have ARC instantiations with one variant (safety) or two variants (safety, baseline) to guarantee fault tolerance (fallback portion of an ARC). The

safety variant , and thus the fallback portion in the one variant instantiation, may exist as a pure recovery function or may be able to achieve the control objective as well.

2.5 ARC Implementation

This section discusses the implementation of ARCs in terms of real-time operating system services, in particular process mechanism and communication services.

The process mechanism is used to fulfill three different roles:

- address space protection
- execution time enforcement
- variant replacement

Variants are placed in different processes in order to prevent volatile variants from damaging the memory space of other parts of the application. The RMA-based scheduling analysis is used to determine schedulability of the application. This schedulability analysis assumes priority-based preemptive scheduling and enforcement of execution time budgets of processes and threads. Run time replacement of variants is accomplished by replacement of processes that represent the upgrade and baseline variants. Typically, a new process instance is created and takes place of the old process in a given process and communication topology.

An ARC is translated into the process and communication structure, as illustrated in Figure 6. Different variants, as well as the decision unit, are placed into different processes that communicate through a group communication mechanism. Message receive acts as a dispatch mechanism to the process. Another process (shown in Figure 6) acts as a physical I/O, the process that interfaces with a device being controlled, and provides input to the ARC. In the example shown in Figure 6, input is provided on a periodic basis. The input to the ARC is available to all variants and the MDU via subscription to a communication channel representing the component input. Upon receipt of the message, the baseline and upgrade variant processes execute and deliver their output to the safety/MDU process via ARC-internal channels.

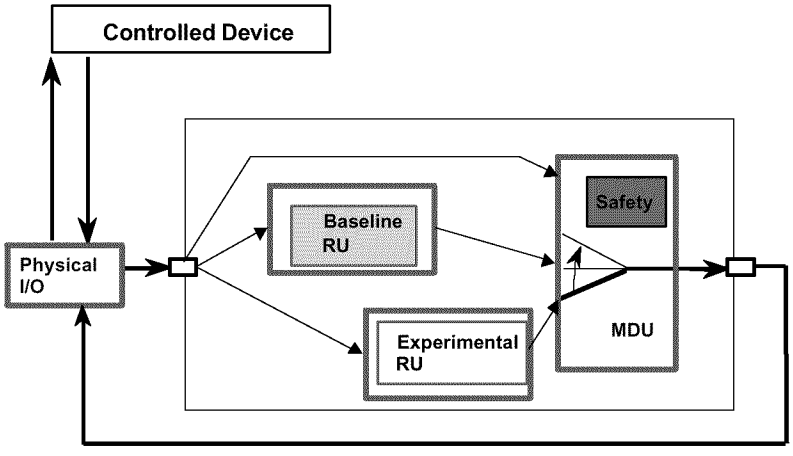


Figure 6. ARC Process Structure

The safety/MDU process can be viewed as two execution threads, one executing the safety variant immediately upon receipt of the component input, while the second delays execution time until all inputs are available. In the current implementation, this is achieved in a single process that executes the safety variant and then delays its execution until the time it expects output from

the variants. At that time, the process determines whether a variant (in particular the leader) has provided output, and checks for safety region violation and performs output range checking.

Delay and determination of whether variants have provided output can be accomplished in several ways. The safety/MDU process may explicitly execute a delay instruction and then perform an unblocked receive. If its priority is higher than that of processes executing the baseline and upgrade variant, it will preempt those processes (this is the current implementation). The safety/MDU process may also perform a blocked receive on the output from the variants and assume that a fault event will release the blocked receive with an appropriate error status in case of execution time limit violation or other execution failure.

In effect, the current implementation checks for deadline violation by the volatile variants rather than violation of execution time limits. This is doable only under the assumption that there is no lower priority ARC running on the same processor, which has been the case for most current prototypes. The coordinated pendulum prototype [4] consists of one ARC for each pendulum controller and of ARCs for the two instances of the coordination of the pendulums. These ARCs are mapped onto multiple processors in such a way that a processor either contains only the high assurance elements of two ARCs (a pendulum controller and a coordinator), or that the volatile components execute by themselves on a separate processor. Since the high assurance elements (safety variant and MDU) are assumed to adhere to execution time limits and no volatile components reside on the processor, their schedule is not impacted by volatile component misbehavior.

While the execution of component variants and the communication of data between them requires real-time performance, replacement of component variants can occur in the background. Replacement can be performed either incrementally (stopping the execution of the system, performing the replacement by process deletion and creation, and resuming execution) or on-line (carrying out the replacement concurrently with the application execution). By executing the replacement manager in the background, interference with the application execution can be minimized.

2.6 Simplex-Based Application Design

This section discusses issues in applying the concept of ARC to real-time applications.

2.6.1 Assumptions about the Application

Simplex makes some assumptions about the application. It assumes that the application is structured as a set of partitioned components that interact via message communication, at least those components that need to be upgraded dependably and are turned into ARCs. All information shared among components is explicitly communicated. Given the primary application domain of continuous control systems, application components operate on data streams. These data streams may be periodic or aperiodic (driven by external events). The primary pattern of execution for a component is *{ input, compute, output }**. At this time, application components cannot interact through shared data areas, although the ARC concept could be extended to support it. In short, Simplex assumes that applications follow a particular architectural style.

It is also assumed that an analytically redundant safety region can be defined for the component that is to be turned into an ARC. This safety region can either specify constraints purely in terms of output values, or in terms of the system state as it is affected by the output. Our primary application area has been continuous control, and we have been able to characterize the acceptable system state such that it could be encoded as a safety region.

2.6.2 Making a Component Analytically Redundant

In this section we focus on introducing analytic redundancy to an existing system by applying the ARC concept to an existing component. The purpose of turning this component into an ARC is to allow for dependable evolution of the component to improved versions.

In this scenario the capability of the existing component is considered to be the baseline, i.e., it becomes the baseline variant. The focus of the design activity is on identifying a safety region to be monitored. This can be the operating region of the current component. In this case, the current component also acts as the safety variant. Alternatively, a separate safety variant may be designed, for which the focus is to provide a larger operating region, but for which the control objective is limited to recovering the system into a state for the baseline variant to resume pursuing the component's control objective.

The safety region of continuous control systems can be determined empirically or through model-based analysis (Lyapunov and Linear Matrix Inequality (LMI); see Sections 6 and 7). In empirical determinations, there is uncertainty in the observed data samples. To compensate, conservative monitoring constraints are derived. In model-based analyses, the control system is modeled as a linear system, for which Lyapunov functions guarantee convergence, i.e., stability, and the LMI technique can derive the largest possible ellipsoid, characterizing a safety region.

2.7 Supporting an Upgrade

In this section we focus on introducing an upgrade to an existing capability, i.e., introducing an upgraded version of a component with improved performance characteristics. In this case we assume that the component has already been turned into an ARC (see previous section).

In this scenario, the design activity consists of three steps: the design of the improved component, the design of performance constraints, and the choice of recovery protocol policy. The improved component may be a new control algorithm or the result of changes to the control parameters in an existing algorithm. The design of performance constraints focuses on providing a monitor that checks whether the upgrade variant achieves the component control objective and whether it does so within performance expectations for either the baseline component or the upgrade variant.

Finally, the recovery protocol used by the ARC can be tuned to the robustness needs of the system at any time during the evolution. One policy focuses on validating the performance constraints. In this case, the modeled performance envelope is compared against the actual performance of the baseline (or upgrade) variant. A violation is this constraint is recorded, but leadership is not changed. A second policy focuses on validating the upgrade. In this case the safety as well as validated performance constraint violations will trigger change in leadership and the disabling of the upgrade variant. Observed violations are interpreted as being caused by misbehavior of the upgrade variant. A third policy gives the upgrade variant a second chance through limited retries in case of violation. This policy is useful when the monitored variant, either upgrade or baseline, are considered stable. The violation is assumed to be transient, and the cause of the violation may have been in another component or a support system function, e.g., transient fault or occasional excessive time delay in the communication system.

2.8 Introducing New Functionality

In this section we discuss a scenario in which new functionality, such as a new control objective, will be introduced. It is assumed that this functionality is different from the existing capabilities, e.g., a new mode may be introduced to an autopilot. In this case we have no existing baseline capability. There are two options for defining a safety variant and a corresponding safety region.

Option one focuses on existing components as a fallback for recovery. The safety region is characterized by the operating region of this component. The second option involves the design of a safety variant, for which the focus is recovery to one of the existing component capabilities. In this case, the safety variant has to be validated, its simple design contributing achieving a quality implementation.

In some cases, one of the existing capabilities may be used in such a way that it pursues the intended new control objective (for example, to provide a series of way points in the way point mode following an intended trajectory). This modified capability could then be viewed as a baseline version for satisfying the new control objective.

Since new functionality is being introduced, we may want to not only monitor the safety region, but also monitor performance against the new control objective. This performance monitor may be a very simple characterization of the control objective that may get refined over time. Initially, it may detect just divergence from the control objective and later be refined to observe convergence.

2.9 Periodic Execution Behavior

Many control applications are comprised of several components that process a data stream, i.e., sensor data is passed through several processing steps, resulting in output to be passed to actuators. Periodicity of a controller component that is implemented as an ARC and that processes a data stream is achieved as illustrated in the example shown in Figure 6. A unit referred to as Physical I/O is implemented as a periodic process. The unit interacts with the controlled device by reading sensor data and providing actuator data. It provides input to the controller as periodically emitted messages and it passes the output received from the controller component to the controlled plant in a periodic manner. By running the highest priority it guarantees the timeliness of the data stream elements both from the device and to the device (within the jitter of scheduling this process). The controller is implemented as an aperiodic ARC, i.e., the aperiodic processes for the variants and MDU are dispatched by message arrival.

2.10 Distributed ARCs

Two scenarios can be considered for distributed ARCs. The first scenario involves placing a whole ARC on a particular processor. In this case the scheduling issues are those of performing scheduling analysis of distributed components independent of whether they are ARCs.

The second scenario involves placement of the high assurance elements of an ARC on one processor and the volatile variants on other processors. Being a fault-tolerant component, the ARC does not depend on the completion of volatile variants. Thus, any uncertainty in interprocessor communication time is observed as missing output from the volatile component and can be treated as a transient fault. This approach has been chosen in the coordinated pendulum prototype.

2.11 Multi-ARC Systems

Applications may have several components that need to be upgraded dependably, and thus are turned into ARCs. In this case, we can identify two categories of upgrade: in-place upgrade of component variants, and changes to the system structure. In-place upgrade of component variants refers to component implementation changes that do not involve changing the communication topology to other components or addition and removal of components. In this case, the communication topology between components is preserved in that different variants of an ARC

may use subsets of ports and connections. The ARC implementation discussed so far supports this in-place replacement.

An application with multiple ARCs has a number of configurations and can change between them at run time. Each ARC autonomously monitors its leader's behavior and reconfigures itself as necessary. As a result, potentially all possible combinations of ARC-variant leadership may occur. Reconfigurations of some ARCs may have side effects on other components and may introduce configuration inconsistencies. If not identified and corrected at run time, these inconsistencies may trigger fault observations in other ARCs, causing further local reconfiguration. We refer to these configurations as *logical configurations* (see Figure 7) because their underlying task configuration may or may not change, i.e., the MDU may just ignore the output of a variant or actually disable its execution. Section 5 discusses an approach to manage such configurations.

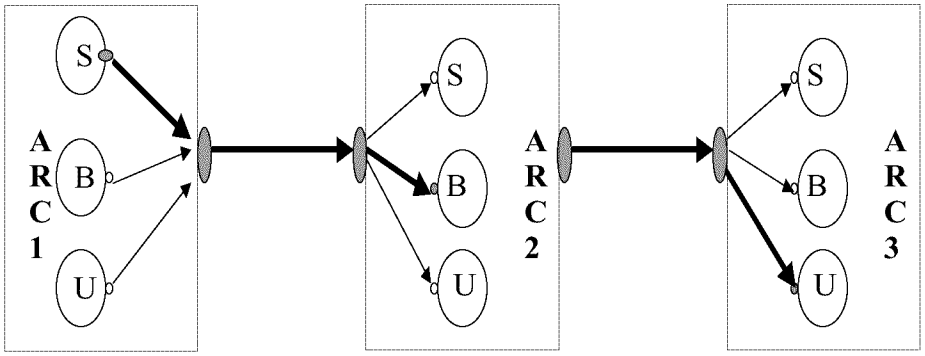


Figure 7. Logical Configuration

ARCs have scheduling characteristics that are like other components, i.e., they are periodic or aperiodic with specified periods, execution time limits, deadlines, etc. At this time we assume that all variants of an ARC are executing at the same rate. Notice that different ARCs, however, may execute at different rates.

Changes to the communication topology or in the set of components can be viewed as different variants of composite components, i.e., subsystems that consist of a set of communicating components. Switching between such variants involves switching between process configurations and communication topologies. This is an area that is still being explored for Simplex, and a promising solution approach can be found in the Honeywell MetaH real-time architectural language.

2.12 Multimode Components and ARCs

In many control applications, components have multiple operating modes. These operating modes may be externally controlled, i.e., mode switching is initiated by another component, or mode switching may be initiated through some mode logic that is part of the component. The output and effect of different modes may be observable by other components or may be transparent. Hybrid controllers are an example of a component that provides transparent mode switching.

The implementation of such a component may be monolithic or may be modularized into the different modes. In the latter case, it may be desirable to consider support for dependable upgrade of each of the modes. In this case, each mode is treated like a separate component to which the ARC concept is applied. If the multimode component has mode logic, the mode logic may evolve as well.

Alternatively, the component as a whole is treated as an ARC, and each of the variants may have multiple modes. In this case each of the variants may contain mode logic. Consistency must be established between the mode logic of each variant such that variant switching results in execution of the expected mode.

3. Middleware Services

3.1 The Distributed Publisher/Subscriber Communication Model

Distributed real-time systems are becoming more pervasive in many domains including process control, discrete manufacturing, defense systems, air traffic control, and on-line monitoring systems in medicine. The construction of such systems, however, is impeded by the lack of simple yet powerful programming models and the lack of efficient, scalable, dependable and analyzable interfaces and their implementations. We argue that these issues need to be resolved with powerful application level toolkits similar to that provided by ISIS [5]. In this section, we consider the IPC requirements that form a fundamental block in the construction of distributed real-time systems. We propose the real-time publisher/subscriber model, a variation of group-based programming and anonymous communication techniques, as a model for distributed real-time IPC which can address issues of programming ease, portability, scalability and analyzability. The model has been used successfully in building a software architecture for building upgradable real-time systems. We provide the programming interface, a detailed design, and implementation details of this model.

3.2 Background

With the advent of high-performance networks such as Asynchronous Transfer Mode (ATM) and upward-compatible 100 Mbps network technologies, the cost of network bandwidth continues to drop steeply. From a hardware perspective, it is also often significantly cheaper to network multiple, relatively cheap PCs and low-cost workstations to obtain an abundance of processing power. Unfortunately, these cost benefits have been slower in accruing to distributed real-time systems because of the still formidable challenges posed by integration issues, frequent lack of real-time support, lack of standardized interfaces, lack of good programming models, dependencies on specific communication protocols and networks, portability requirements and lack of trained personnel who perhaps need to be re-educated on the benefits and potentially major pitfalls of building working distributed real-time systems. We believe that the difficulty of programming distributed real-time systems with predictable timing behavior is in itself a significant bottleneck. Unfortunately, the problem is actually much harder because many other issues need to be addressed concurrently. These issues include the need to maintain portability, efficiency, scalability and dependability as the systems evolve or just become larger. Put together, these issues pose daunting challenges to the construction of predictable distributed real-time systems.

Three factors in particular seem to dominate the various phases of the life-cycle of developing and maintaining distributed real-time systems. First, the development, debugging and testing of distributed real-time systems is hindered by complexity; there are many degrees of freedom compared to uniprocessor systems which are conceptually easier to program. It would be extremely desirable to have a programming model which does not depend upon the underlying hardware architecture, whether it is a uniprocessor or a network of multiple processors. Second, systems change or evolve over time. Hardwiring any assumptions into processes, communication protocols and programming models can prove to be extremely costly as needs change. Finally, as distributed real-time systems grow larger, there is often a need to introduce new functionality by extending services and functions provided. Such extensions can become impossible if currently operational code needs to be rewritten to accommodate these changes, or sometimes even if the entire system just has to be taken down for re-installation of other systems. In other words, it would be desirable if a subsystem can be added online without having any downtime for the rest of the system.

3.2.1 Objectives for a Distributed Real-Time Systems Framework

Due to the many issues that need to be addressed simultaneously in building distributed real-time systems, we believe that a well understood framework for building distributed real-time systems is essential. Specifically, the following objectives should be addressed by a framework for building such systems:

- **Ease of programming**: The framework must provide simple programming models and interfaces to the programmer. The implementation of this programming model is hidden from the application programmer and completely bears the responsibility of hiding the complexity of the underlying hardware (processor and network) architecture, communication protocols and other configuration-dependent issues. Optimizations that are dictated by application performance requirements should still be done only at the implementation layer of the model. However, the programming model and interface exported to the programmer should be left untouched.
- **Portability**: The programming model should be portable across platforms and environments (at least at the source-code level). This objective basically implies that the model should not be bound to specific implementation-dependent features such as the availability of UDP/IP or a specific network medium.
- **Analyzability**: Since the system has real-time semantics, analyzability and predictability of the system's timing behavior cannot be sacrificed. It would be ideal if the programming model(s) and interface(s) come with schedulability models to facilitate scheduling analyses of applications using these interfaces.
- **Efficiency**: As long advocated by the real-time community, real-time systems are not "fast" systems but they are "predictable" from a timing point of view. Nevertheless, efficiency or performance is an important practical consideration in many applications. An elegant programming model whose implementation is unacceptably inefficient is unlikely to succeed in practice.
- **Scalability**: The natural appeal of distributed systems is that they can grow larger in order to provide added power and functionality as system needs grow over time. A programming model for these systems should be able to scale naturally. Similar to the requirement above for ease of programming, the responsibility for providing scalable performance and hiding programming complexity must lie in the implementation of the programming model.
- **Dependability**: As the system scales up in size, components or subsystems will need to be changed because of considerations such as failure, preventive maintenance, and upgrades. Since many real-time applications demand continuous and/or error-free operations, it is often critical in many cases that failure or maintenance problems do not cause damage to life and/or property.
- **Protection and enforcement**: Software errors in a new relatively untested module must ideally not bring the entire system down. Some form of protection and enforcement (such as denying a request for unauthorized access to a critical resource) can be very desirable.

Some of these requirements often seem to be (and can be) contradictory. For example, hard-wiring some assumptions may make analyzability easier. Scalability may sometimes conflict with both ease of programming and efficiency. Similarly, requiring portability or protection may sometimes affect efficiency adversely because of additional software layers and/or checks needed.

3.3 The Real-Time Publisher/Subscriber Communication Model

An IPC capability is a fundamental requirement of distributed real-time systems. All of the goals for a distributed real-time systems framework have direct relevance to this communication capability. For example, it would be very desirable if the IPC model is independent of uniprocessors or distributed systems, independent of specific communication protocols, independent of the network used, scales efficiently to a large number of nodes, works despite node or network failures, and fault(s) in some

application module(s) which do not bring down the entire communication layer. The IPC layer therefore constitutes a good test candidate for the proposed framework. In the rest of this paper, we present the real-time publisher/subscriber communication model for IPC, along with its design and implementation, which address many of the goals for a distributed framework[1].

3.3.1 Related Work

The real-time publisher/subscriber communication model we describe is based on the group-based programming techniques (isis18) and the anonymous communication model [6]. Such a model in a general non-real-time context is also sometimes referred to as "blindcast" (POSIX21). In the model that we propose, we are driven not only by the need to maintain ease of programming, but also by the need to maintain analyzability, scalability and efficiency for real-time purposes.

3.3.2 The Communication Model

The real-time publisher/subscriber model associates logical handles to message types. Messages can be generated and received based only on these logical handles without any reference to the source or destination of the messages, and independent of the underlying communication protocols and networks in use. Once a logical handle has been created, sources (publishers) can send or publish messages with that handle. Sinks (subscribers) can subscribe to and receive all messages published with that handle. At any given time, the publishers need not know the subscribers and vice versa. In fact, there may be no subscribers to a published message, nor any publishers on a logical handle subscribed to. Publishers and subscribers may also obtain or relinquish publication and subscription rights dynamically. It is also possible for an application thread to be both a publisher and a subscriber simultaneously. The logical handle itself can be as simple as a variable-length ASCII string.

In contrast to pure "blindcast" techniques, we also allow the publishers/subscribers to know, *if needed,* who the publishers/subscribers are on a certain message type, and the source of any received message. We allow this because additional publishers and/or subscribers have performance impact from a timing perspective, and it is impossible for a framework provider to judge all of the needs of application builders. However, by providing the information that applications may need, the framework can at least allow an application to make its own decisions. For example, the application could check the number of current publishers on a message type before allowing a new publication right to be requested. We believe that this concern of letting the application have access to performance-sensitive information and thereby have control over application performance is another significant issue that needs to be addressed by a framework for distributed real-time systems.

3.3.3 The Application Programming Interface

The C++ application programming interface for our real-time publisher/subscriber model is given in Figure 8. An illustration of how exactly the Application Programming Interface (API) would be used is presented later in Figures 9 and 10. The notion of a *distribution tag* is key to our communication model. It is the logical handle by which all messages can be published and received. As its name implies, the tag represents how a message will be distributed. In this case, the tag contains information regarding the publishers and subscribers on that tag (including their node addresses, and communication-protocol-dependent information such as port numbers, information which is not seen by the user). Hence, the API

---

[1] In this paper, we ignore the dependability requirement, but our current design and implementation have been carried out with this requirement in mind. Ongoing extensions to this design and implementation will allow the dependability requirement to be addressed as well.

provides calls to create and destroy distribution tags. Once a distribution tag has been created, it can be used to obtain publication and/or subscription rights, to publish messages with that tag, and to receive messages with that tag. Associated with each distribution tag can also be a set of real-time attributes such as the priority at which messages with this tag will be transmitted.

```
class DistTag_IPC {
private:
...
public:
// create or destroy distribution tag
Create_Distribution_Tag(tag_id_t tag, domain_t domain);
Destroy_Distribution_Tag( tag_id_t tag );

// get/release publication rights on distribution tag
Get_Send_Access( tag_id_t tag );
Release_Send_Access( tag_id_t tag );

// get/release subscriber rights to distribution tag
Subscribe( tag_id_t tag );
Unsubscribe( tag_id_t tag );

// publish on specified distribution tag
Send_Message( tag_id_t tag, void *msg, int msglen );

// receive a message on a subscribed tag:
// returns the process id of the sender: specify time
//  to wait and the node address of the sender is
// returned in optional "out" parameter.
Receive_Message( tag_id_t tag, void *msg, int *msglen,
timeval_t *tout, in_addr_t *from_addr = NULL );

// purge all input messages queued locally on tag
Purge_Messages( tag_id_t tag );

// is a message available on a specified tag?
int Message_Available( tag_id_t tag );

// query list of publishers/subscribers on a tag to
// allow applications make own decisions.
sender_list_t
Get_Senders_On_Distribution_Tag( tag_id_t tag );
receiver_list_t
Get_Receivers_On_Distribution_Tag( tag_id_t tag );

// Notes: In our implementation, tag_id_t is (char *)
// and tag id's are specified as ascii strings.
// "domain_t" used during tag creation can be
// GLOBAL or LOCAL, but only GLOBAL is supported.
};
```

Figure 8. The Application Programming Interface for the Real-Time Publisher/Subscriber Paradigm

3.3.4 Fault-Tolerant Clock Synchronization

Consider the problem of distributed clock synchronization [7]. Typically, clock slaves send a message to a clock master requesting the time, and the clock master responds with the time it has. The slaves use the master time and knowledge of the transmit time to update their local view of the globally synchronized

clock. If the clock master dies, some other node must become clock master and all slaves must now redirect their time requests to the new master. With the real-time publisher/subscriber communication model, two (or more) nodes can choose to act as cooperating clock masters. They obtain subscription rights to a distribution tag with logical name "Clock Master Request," and all slaves get publication rights to this tag. Slaves can also register the unique tags on which they expect to see their responses by sending registration messages to "Clock Master Request." A primary clock master response to a request from a slave would be to this tag (unique to that slave). In addition, the clock masters run an internal handshaking protocol (using a tag like "For Use Only by Masters") such that at most one publishes responses to slaves' requests, for example. If one master dies, another master can realize this because of the absence of heartbeat on the "For Use Only by Masters" tag, for example, and start publishing responses to the slaves. A slave's behavior does not change: it continues to publish its requests to "Clock Master Request" and to subscribe to its individual response tag. The slave literally does not care where the clock master responses come from. However, it must be noted that the masters themselves must take great care to keep *their* clocks more tightly synchronized and to respond to outstanding slave requests when a master fails.

3.3.5 Upgrading a Process

The fault-tolerant clock synchronization model can also be used to upgrade a process which is already running. All communications to the process's external environment must use distribution tags. When this process needs to be upgraded, its replacement process is created, and it obtains publication rights and subscription rights to all of the tags used by the process being replaced. Then, using a predefined handshaking protocol, the original process stops publishing its messages; simultaneously, the replacement process starts publishing the same messages and reacting to the messages it receives on its subscribed tags. As a result, as far as the rest of the environment is concerned, this process "looks" the same as the replaced process but can have enhanced functionality. This upgrade paradigm is at the heart of the Simplex architecture framework for building upgradable and dependable real-time systems [8] and is built upon our real-time publisher/subscriber communication model as just described.

3.4 The Design and Implementation of the RT Publisher/Subscriber Model

The key behind our design of the real-time publisher/subscriber communication model can be quickly summarized as follows. The generation and reception of messages happen in the "real-time loop" or the "steady-state path" and must therefore be as fast as possible. Conversely, the creation and destruction of tags, and getting publication or subscription rights are viewed as primarily non-real-time activities happening outside of the real-time loop and can be very slow compared to actual message communications using tags. We refer to this as the *efficient steady-state path requirement*.

The overall architecture of the design of our communication model follows the design presented by Rajkumar, Gagliardi and Sha [9]. Application-level clients (processes) use the communication model by making calls listed in Figure 8, which are implemented by a library interface hiding the complexity of the underlying implementation. The IPC daemons on various nodes communicate with one another, keeping them all apprised of changes in distribution tag status: tag creation, tag deletion, addition/deletion of publication/subscription rights. (Recall that the current implementation does not yet tolerate node failures.) A client that creates/deletes tags and gets publish/subscribe rights does so by communicating with its closest IPC daemon. Needed distribution tag information is then sent back to the client from the IPC daemon, and is stored in a local tag table in the client's address space. Any changes to this tag information because of requests on local or remote nodes are notified to the client (by the same daemon that processed the original tag request) and are processed asynchronously by an *update thread* created and

running in the library. Naturally, the local tag table is a shared resource among this update thread and the client thread(s) and must be protected *without* the danger of unbounded priority inversion.

3.4.1 The IPC Daemon

Each IPC daemon actually consists of three threads of control that run at different priorities:

- **Local Manager**: A local manager thread responds to requests for tag creation/deletion and publication/subscription rights from clients on or in close proximity to this node. *Not* being part of the efficient steady-state path, this thread can run at low priority.
- **Update Manager**: The update manager thread communicates with its counterparts in the other IPC daemons to receive tag updates. Any local manager thread receiving a request which changes the status of a distribution tag sends the status update notification to the update managers on all of the other nodes, and then updates the local daemon's copy of the tag table. The remote update manager threads update their respective copies of the tag table to reflect the change status. In addition, they may also notify any of their local clients that have a copy of that tag. These asynchronous notifications are handled by the update thread in the client library. A local manager thread can respond to its client only when all notifications to the remote update managers are complete. Hence, the update manager thread runs at a higher priority than the local manager thread.
- **Delivery Manager**: Strictly speaking, the delivery manager is not necessary. When a client needs to send a message with a valid publication right, the distribution tag will be in its local tag table in *its* address space and will contain the current publishers/subscribers to this tag. The client can therefore directly send a message to each of the subscribers on this list via the library. However, if there are multiple receivers on a remote node, it is often wasteful to send many duplicate network messages to a single node. The delivery manager is intended to address this (potential) performance problem. The client sends just one copy of its message to a delivery manager on the remote node, and the delivery manager, using locally efficient mechanisms, delivers the message to all receivers on its local node. In a larger system, only the delivery manager needs to be running on each node, and it acts more as a performance enhancement technique in that case. Since the delivery mechanism is directly in the steady-state path, this thread runs at the highest priority of the 3 threads. Actually, to avoid undesirable remote blocking effects [10], this thread may even run at priorities higher than many application threads.

3.4.2 The Client-Level Library

In the user-level library on the client side, three services acting as counterparts to the three threads in each IPC daemon are supported, as follows:

- **Local request service**: This library module in the client space translates client calls for tag status changes into a request that is sent to the local request manager of the closest IPC daemon. It then blocks until a successful or failed response is obtained. This service is executed as part of the calling client thread at its priority.
- **Delivery service**: This library module in the client gets activated when the client publishes a message. Distribution tag information from the local tag table is obtained, and if any receivers are on remote nodes, messages are sent to the delivery managers running on these remote nodes. At most one message is sent to any remote node, since the remote delivery manager will perform any needed duplication to multiple local receivers. This service is also executed as part of the calling client thread at its priority.
- **Update service and thread**: The update service is performed by a separate, user-transparent thread in the library. As mentioned earlier, this thread receives and processes notifications of tag status changes from its host IPC daemon. Any changes that it makes to the client tag table must be atomic. The atomicity can be provided either by semaphores supported by priority inheritance or by

emulating the ceiling priority protocol [10, 11]. This thread should run at higher priority than all client threads in this process using the IPC services.

3.4.3 Sequence of Steps on Various Calls

In this subsection, we summarize the list of actions taken for different categories of calls in the API of Figure 8.

When a non-steady-state path request (tag creation/deletion request, publish/subscribe right request) is issued, the following process is enacted:

1. The client's local request service sends a request to the local manager of the IPC daemon (on the closest node) and blocks after setting up a timeout.
2. The local request manager checks to see if the tag status change can be made. If so, it sends an update status to the update manager threads of remote IPC daemons[2]. Then, it updates its local tag table.
3. The local request manager then sends a response back to the client, which unblocks and checks the return value for success.
4. If there is no response from the IPC daemon (typically because it was not started), the client times out and detects the error status.

When the steady-state path request "publish message on a tag" is issued, the following process occurs:

1. The calling thread atomically checks the local tag information for valid information.
2. If the information is valid, it sends copies of the message to all receivers on the local node (if any) and at most one message to each remote delivery manager whose node has at least one receiver for that tag.

When the steady-state path request "receive message on a tag" is issued, the following steps occur:

1. The calling thread atomically checks the local tag information for valid information.
2. If valid, a specified timeout is set and the client thread blocks waiting for a message. If a message is already pending, the client returns with the message. The process id of the sender process and its node address are also available on return.
3. If no message is received within the specified time, the client returns with an error.

When a query call "obtain senders/receivers on a tag" is issued, the following process occurs:

1. The client's local tag table is atomically checked, and if the information is available locally, the information is returned.
2. If the information is not locally available, the request is forwarded to the nearest IPC daemon for obtaining the information.

3.4.4 Meeting the Goals of the Design

We discuss how the above design achieves the design goals outlined in the beginning of Section (model).

Ease of programming is achieved by providing location transparency, an identical programming model for unicast or multicast, and an identical programming model for uniprocessors or distributed systems. This programming ease is readily apparent in the benchmark code of Section (performance), in which the code does not change as the system grows from one processor to multiple processors as well as from one receiver to multiple receivers.

---

[2] An acknowledgement-based delivery mechanism is recommended here. However, the current implementation just uses datagrams.

Portability is achieved by maintaining protocol independence and keeping the interface constant. For example, our real-time communication model was first implemented on a single node using only UDP sockets. Then, for performance reasons, the underlying communication protocol was ported to POSIX message queues, but the user interface was unchanged. Finally, the interface remained constant as support for communication across nodes was added with only the addition of an *optional* new parameter to the Receive_Message() call to identify the source address of a message. Shown in the next section, POSIX message queues are still used locally on a node, and UDP sockets are used across nodes, but all of these details are hidden from the programmer at the interface.

Scalability is achieved by ensuring that there will not be significant bottlenecks when the size of the system grows. The implementation is supported by interacting IPC daemons running on various nodes, but as the number of nodes grows in the system, the communication between the daemons can become expensive. Hence, the current implementation is designed such that one daemon need not be running on every node. The daemons can be running only on an acceptable subset of the nodes in the system. Only the delivery manager needs to be replicated on every node.

Analyzability is achieved by assigning appropriate scheduling priorities to various threads and by providing a schedulability model of how the IPC interface operates. This is the key component from a timing predictability point of view. There are many threads of execution in client space *and* in daemon space, and it is critical that these threads run at appropriate priorities. The distribution tag information is also stored as data shared among these threads in each process, and unbounded priority inversion must be avoided on access to this data. More discussion on this topic is provided in Section 3.4.6.

Efficiency is achieved by ensuring that the steady-state path or the real-time path is as efficient as possible. Any message sending, reception or purging uses only information available locally within the client space. Remote deliveries involve network communication, and hence delivery to multiple receivers on the same node are accelerated by a delivery manager which does the local duplication.

Protection and enforcement is obtained by making sure that no process but the IPC daemons on various nodes actually can change the global publisher/subscriber information. Clients can at worst corrupt only the locally stored information, but this information is not accessible beyond the client's address space. Unfortunately, a misbehaving client can still try to flood the network or a remote delivery manager, and protection against such behavior can be provided at only a lower layer at the OS network interface.

3.4.5 Implementation Aspects

The current implementation of the above design uses the following:

- POSIX threads are used both in the IPC daemon and the client, which are scheduled using fixed priorities.
- POSIX semaphores are used for protecting the tag table shared information in the IPC daemon.
- POSIX message queues are used for local delivery of messages within a node, including direct delivery by a client to other local receivers and indirect delivery to local receivers by a delivery manager that receives messages from a remote client.
- All client-generated requests to the IPC daemon and their corresponding responses use UDP sockets such that an IPC daemon need not be present on every node.
- All communications which cross node boundaries also currently use UDP sockets.

- Each message that is sent from a client is prefixed with header information that includes the process id of the sender and its node address. This information can be used by higher level handshaking protocols at the receiver(s).

Current system initialization requires that IPC daemons be started on all of the networked nodes before any application-level IPC using the real-time publisher/subscriber model can commence. The list of nodes in the system is read from a file (identical copies of which have been installed on all the nodes) so that each IPC daemon and library knows what other daemons to contact for tag updates and message delivery.

3.4.6 Schedulability Analysis

From a schedulability analysis point of view for real-time systems, the implementation of the RT publisher/subscriber model has many implications, almost all of which are relatively well understood. The implementation presumes a fixed-priority-driven preemptive scheduling environment, but the model is only a library interface that is not bound to any particular application. Various real-time applications can use this model in desired ways. However, for these applications to be able to analyze the impact of the real-time publisher/subscriber model they use, sufficient information regarding the model's internal architecture, its cost, and priority inversion characteristics need to be known. The model's internal architectural design and implementation have been discussed in detail in the preceding sections. For example, the threads which provide various services or execute various modules are part of the design discussion, as are their priority assignments.

The techniques of generalized rate monotonic scheduling [12, 13] can be applied for uniprocessor analysis. The notions of remote blocking, deferred execution penalty, and global critical sections (if a remote client/server application architecture is used) as defined by Rajkumar [10] can be used for distributed system analysis. These techniques basically need two pieces of information: the amount of execution time for segments of code, and the worst case duration of priority inversion. Priority inversion can result due to the atomic nature of *local* tag table accesses in each client and in the IPC daemon. However, note that these shared resources are shared only within the confines of each client and the IPC daemon process. Hence, it is relatively easy to identify the maximum duration of critical section accesses, and the cost is bounded by measurements described in the next section.

The relatively tricky aspect of the schedulability analysis is that a tag status update can generate asynchronous notifications to many client threads, and hence the true value of preemption time from higher priority tasks and blocking time for a task must be based on the knowledge of the maximum number of clients (senders and receivers) on tags used by lower or higher priority tasks. However, results from real-time synchronization help determine these values. At any given priority level, at most one lower priority task from that node can issue a tag request (causing asynchronous notifications to be sent to other tasks) since this priority level can preempt subsequent lower priority requests and tasks. However, requests from higher priority tasks must be accounted for appropriately.

3.4.7 Object Hierarchy

The real-time publisher/subscriber IPC model has been implemented in C++ (the current implementation has about 7000 lines of code) and is based on an extensive and flexible object hierarchy. Basic classes encapsulate many needed OS functions such as timers, semaphores and threads to enable easy porting to other operating environments. Other library classes implement queues, hash-tables, protected buffers, and communication protocols such as sockets and POSIX message queues. Above these is built the tables to hold the tag information and messaging classes for constructing and decoding messages sent among the

IPC managers, and between the client and an IPC daemon. At the top level are the classes for instantiating the update manager, the delivery manager and the local manager on the daemon side, as well as the update thread, the local request service and the delivery service on the client side. The actual managers and services are provided by simple instantiations of these classes. Each of these classes can be ported to other platforms with different operating systems and/or communication protocols without affecting the other layered classes.

The key parameters used to control real-time performance in this context are scheduling priorities and access to shared resources. Hence, all classes which relate to schedulable entities (i.e. threads) and access to shared resources (e.g. semaphores) are parameterized such that priorities and semaphore accesses are under the control of instantiating entities.

3.5 Performance of the Publisher/Subscriber Model

The real-time publisher/subscriber communication model described in the preceding sections has been implemented and preliminary benchmarks obtained on a network of i486DX-66MHz PCs running on a POSIX-compliant real-time operating system LynOS Version 2.2 with 32MB of memory. The network used was a dedicated 10Mbps Ethernet but was very lightly loaded.

3.5.1 The Rationale Behind the Benchmarking

As discussed in Section 3.4.6, we attempted to measure the cost of each call that a client can make with the real-time publisher/subscriber model. We also tried to validate how far we met one of the design goals of efficiency compared to the traditional "build-from-scratch" communication facilities, despite the unoptimized nature of our prototype implementation and the optimized nature of the underlying UDP/IP and POSIX message queue communication primitives. In all measured cases, we published messages on a tag and measured the time it takes to get its response on another tag. This allows us to measure the elapsed time from a single point of reference, namely the sending end. For the sake of convenience, this is referred to as "sender-based IPC benchmarking." By sending multiple messages, sufficient elapsed time can be accumulated to eliminate any clock granularity problems.

"Sender-based IPC benchmarking" is conceptually simple in the case of either local or remote unicast: (a) send a message, (b) wait for a response, (c) repeat this several times, (d) measure the total time elapsed, (e) and compute the time elapsed per round-trip message. In the case of remote unicast, sender-based benchmarking also avoids the need for a tightly synchronized clock across nodes. With multicast in general and remote multicast in particular, the situation is trickier. If all multicast recipients respond with a unicast to the sender, the elapsed time contains a large mix of multicast and unicast messages. We measured this to use as a data point as well. However, if only one recipient responds with an *ack* message, then the total cost is dominated by the multicast and it is easy to identify the cost benefits of multicast. However, it is possible that the *ack* message returns concurrently before the multicast send has completed and the elapsed time measurements is distorted. We resolve this by multicasting a message to many receivers but only the lowest priority receiver responds with an *ack*. This ensures that the multicast send completes before the *ack* originates.

3.5.2 The Benchmark Code

The performance benchmark code on the sender and receiver sides for benchmarking is illustrative of the ease, power, and compactness of using the real-time publisher/subscriber communication model. All benchmarks (local and remote unicast as well local and remote multicast) are carried out with this code. The sender side is presented in Figure 9 and the receiver side is presented in Figure 10.

```
int benchmark_dist_tag_ipc( char *msg, int msglen,
                int num_messages, int num_acks )
{
  DistTag_IPC ipc; timeval_t start, end;
  int i, j;

  // get send rights to receivers' tag
  // (tag is currently an ascii string)
  ipc.Get_Send_Access( RECEIVER_TAG );

  // get receive rights for getting acks back
  ipc.Subscribe( ACK_TAG );

  // get time-of-day before sending messages
  getCurrentTime( &start );

  // send out messages
  for ( i = 0; i < num_messages; i++ ) {
    // send message and wait for a response
    ipc.Send_Message( RECEIVER_TAG, msg, msglen );

    for ( j = 0; j < num_acks; j++ ) {
      // block on required acks
      ipc.Receive_Message(ACK_TAG, msg, msglen, NULL));
    }
  }

  // get time-of-day we stopped
  getCurrentTime( &end );

  // compute delay per message sent
  // and print it out
  …
}
```

Figure 9. Benchmarking Code on the "Sender" Side

```
run_receiver_in_loop( char *msg, int msglen, int ack )
{
 DistTag_IPC ipc;

 // create tags: no problem if already created
 ipc.Create_Distribution_Tag( RECEIVER_TAG );
 ipc.Create_Distribution_Tag( ACK_TAG );

 // get receive rights on RECEIVER_TAG
 ipc.Get_Send_Access( ACK_TAG );

 // get send rights on ACK_TAG
 ipc.Subscribe( RECEIVER_TAG );

 while ( True ) {
  // block for messages waiting forever
  ipc.Receive_Message( RECEIVER_TAG,msg,msglen,NULL);

  if ( ack ) {
    // send ack to "ack tag"
    ipc.Send_Message( ACK_TAG, msg, msglen );
  }
 }
}
```

Figure. 10. Benchmarking Code on the Subscriber (Receiver) Side

3.5.3 Performance of Vanilla UDP/IP and POSIX Message Queues

It must be noted that the context-switching overhead from one process to another process is about 100 microseconds.

Table 1 shows the control case that does *not* use the real-time publisher/subscriber model. There is one sender and one receiver. The sender sends a message and the receiver returns a response using UDP/IP or POSIX message queues. Both sender and receiver are on the same node in columns 2 and 3 but are on different nodes in column 4. As can be seen, UDP protocol processing is costly relative to POSIX message queues but the latter work only within a node. This is the reason why we ported the local message delivery mechanism from UDP to POSIX message queues (without interface changes as intended).

3.5.4 Publication/Subscription on a Single Node

Table 2 presents the costs involved in multicast to multiple receivers on the same host node (when *all* receivers ack back to the sender) for a 256-byte message. A sender with lower priority than all the receivers (columns 2 and 3) performs better than having the sender with higher priority than all the receivers (columns 4 and 5) because the sender never blocks in the former case. Also, emulating the ceiling priority protocol (columns 2 and 5) to avoid the cost of semaphores to access the local tag table atomically (columns 3 and 4) helps to improve performance.

Table 1. Basic System IPC Costs

Message Size (bytes)	Round-trip delay on same node using POSIX message queues (ms)	Round-trip delay on same node using UDP messages (ms)	Round-trip delay from different nodes using UDP messages (ms)
32	0.3191	1.488	2.200
64	0.3368	1.481	2.400
128	0.3552	1.556	2.762
256	0.4050	1.633	3.569
512	0.4899	1.657	5.190
1024	0.6654	1.856	8.355
2048	0.8808	2.604	13.200
4096	1.4306	3.501	21.980
8192	2.6490	6.013	39.700

Table 2. Round-Trip IPC Costs for Multicast on a Single Node

# of Receivers	Low Priority Sender; *no* Semaphores (ms)	Low Priority Sender; Semaphores (ms)	High Priority Sender; Semaphores (ms)	High Priority Sender; *no* Semaphores (ms)
1	0.763	0.847	0.832	0.739
2	1.534	1.71	1.774	1.467
3	2.563	2.679	2.907	2.207
4	3.648	3.882	4.045	3.199
5	4.945	5.239	5.581	4.445
6	-	-	7.167	5.802

Comparing the control case (Table 1, column 2) with the case for 1 receiver, we see that our IPC model has roughly added at least 340 µs to a direct POSIX message queue send (0.4050 *vs* 0.739 ms for 256 byte messages) for a single receiver and this scales almost linearly as the number of receivers grows. We attribute this cost to the additional copying done by our implementation.

In the rest of this section, we use semaphores for critical section access so that the numbers are slightly higher than the performance obtained with emulating the ceiling priority protocol. Table 3 presents the multicast cost on a single node when only the lowest priority receiver acks. It is surprising to see that a sender with higher priority than the receivers performs better than a sender with lower priority than the receivers, the exact opposite of before! This is because of the tradeoff between increasing the number of context switches and letting one process block waiting for a message that has not yet arrived.

Table 3. Round-Trip IPC Cost for Multicast on a Single Node

# of Receivers	Low Priority Sender (ms)	High Priority Sender (ms)
1	0.83	0.829
2	1.37	1.173
3	1.847	1.522
4	2.259	1.831
5	2.762	2.254
6	3.178	2.597
7	3.553	2.877
8	4.096	3.181

3.5.5 Subscription from a Remote Node

Table 4 gives the costs incurred when multicast is performed to a remote node with 256-byte messages. Compare Table 1 column 4, 256-byte message cost (3.569 ms) with Column 3, 1 receiver cost of Table 4 (6.72ms). The added overhead of our library front-end and its sending a message to a delivery manager, which in turn delivers to local receivers has added an overhead of mare than 3 ms. However, with only two receivers on the remote node, two plain UDP messages sent across the network would have cost (3.569*2 = 7.138 ms) while the delivery manager is able to deliver two copies locally to accomplish the same function at 7.18 ms, i.e., the performance crossover occurs only with 2 receivers on a remote node.

Additional receivers on the remote node cost only 0.5ms each (with a single ack), while direct duplicate communication would not only load the network but also cost an additional 3.569 ms. Even out unoptimized prototype is better than a straightforward implementation which sends network messages to each remote receiver, while at the same time providing a much simpler and more powerful interface that deals with both unicast and multicast simultaneously, whether local or remote. This is a significant observation with respect to our goals of ease of programming, scalability and even efficiency, despite the unoptimized state of the current implementation. Note that this enhancement is obtained by using UDP sockets across the network (the most efficient at that level) and POSIX message queues locally on the receiving side (the most efficient within a node as seen in Table 1).

Table 4. Round-Trip IPC Costs for Multicast to Multiple Receivers on One *Remote* Node

# of Receivers	All Receivers Acknowledge (ms)	Only Low Priority Receiver acks (ms)
1	6.68	6.72
2	8.58	7.18
3	10.57	7.54
4	12.88	7.98
5	15.28	8.55
6	17.86	9.15
7	18.32	9.43
8	21.52	-

Table 5. Round-Trip IPC Costs for Multicast to Multiple Recipients on Many Remote Nodes

# of Receivers on Host "laurel"	# of Receivers on Host "hardy"	# of Receiver on Host "chaplin"	Round-trip Delay per Published Message on "shemp" (ms)
1	0	0	6.74
1	1	0	8.55
1	1	1	12.5
2	1	1	12.68
2	2	1	15.28
2	2	2	17.79
3	2	2	20.17
3	3	2	22.78

3.5.6 Subscription from Multiple Remote Nodes

Table 5 presents the costs involved for 256-byte messages when there can be up to 3 receivers on each of 3 remote nodes, "laurel," "hardy," and "chaplin." Messages are initiated from the node "shemp." *All* receivers must ack to the sender. As can be seen, if there is one receiver on each of the 3 remote nodes, the round-trip cost is around 12.5 ms. Also, the costs do not scale linearly because of some pipelining concurrency that happens naturally across nodes. This is clearly seen in rows 3 and 4 where there are 1 or 2 receivers on a remote host "laurel," but the total difference in cost is almost negligible. This is because the responses from the receivers on "laurel" happen in parallel with the messages sent to receivers on hosts "hardy" and "chaplin."

3.5.7 Lessons Learned

We have gained some valuable experience and insights into the communication model with our prototype implementation. As mentioned earlier, this communication model has been successfully used in implementing inter-processor communication in the Simplex architecture [8]. Due to the expense of UDP sockets in local node communications, as per Table 1, we re-implemented the interface using POSIX message queues. Since the interface could be reimplemented without any changes, it demonstrated that the goals of interface portability and performance optimization are not mutually exclusive.

The performance of the current implementation needs to be further optimized, however. We pay a performance penalty because of additional message copying necessitated by the need to insert and then delete message headers for use by the communication model. Currently, tag information is also stored in a hash-table and the hashing cost is relatively high. We therefore plan to eliminate strings for tags from being hashed once a tag has been created. However, despite these current inefficiencies, it is encouraging from Section 3.5.5 to see that the current implementation can actually do better for multicast to remote nodes and the improvement actually grows larger as the number of receivers increases.

Another alternative to consider is the use of shared memory segments to store the tag table information for sharing among clients and the IPC-daemon on a node. While this does not affect the steady state path significantly, it can make the implementation much simpler. However, two disadvantages would arise. First, with a single piece of shared memory having all the information, protection and failure issues need

to be addressed. Secondaly as the system grows, IPC daemons must run on all nodes to update the shared memory information.

3.6 Concluding Remarks

The construction of distributed real-time systems poses many challenges because of the concurrent need to satisfy many attributes, including ease of programming, portability, scalability, analyzability, efficiency, and protection. We have sought to identify how many of these attributes can be dealt with in the context of IPC, a fundamental component of any distributed system. We have identified the real-time publisher/subscriber model as an excellent candidate in this domain. We also presented a programming interface, which is quite simple and based on the notion of a distribution tag. The interface has been tested with the Simplex architecture and has remained stable with needed optimizations done without affecting the interface. We also provided a detailed design of the model, an implementation, and benchmarks to show that many attributes can indeed be satisfied.

Despite these positive results, much work remains to be done. The current prototype does not tolerate processor failures even though the current design and code is structured such that these changes can be incorporated relatively easily. In addition, redundant networks need to be supported to tolerate network failure. Finally, support for flexible, yet real-time, transfer of state between new and existing IPC daemons in different states is critical.

4. Verifying INSERT Switching Rules

4.1 Introduction

CheckMate is a Matlab toolbox used to perform verification of hybrid systems, systems with both continuous and discrete dynamics. This section describes the application of the CheckMate toolbox to the autolanding routine of the F-16 aircraft as implemented through the Simplex architecture. Both CheckMate and the Simplex architecture were developed at Carnegie Mellon University as part of the INSERT project.

In CheckMate, hybrid automata are used to model the behavior of a hybrid system [14, 15]. CheckMate uses quotient transition systems (QTS) to provide a discrete approximation of the hybrid automata [16], and then performs formal verification on the discrete approximation. System models are entered into CheckMate using the Matlab Simulink/Stateflow GUI, thus allowing the user to take advantage of the simulation capabilities available with the Simulink toolbox. Parameters and specifications are entered through both the GUI and the Matlab command window. The verification is carried out from the Matlab command prompt. There are several options available during verification such as saving data after each iteration, closing the Simulink window once the hybrid automaton is constructed, and a simulation-based validation routine useful for getting a quick first pass look at the system verification.

The Simplex architecture uses ARUs as a mechanism for fault-tolerant real-time software upgrades in control systems. Specifically, three controllers are present in a Simplex ARU: the *baseline*, *safety* and *upgrade* controllers. As the name suggests, the upgrade controller is a new or improved version of the control software. The baseline controller is a known reliable controller, usually the current or legacy version of the software to be upgraded. The safety controller is designed to control the system in a large operational region and to guide the system back into the operational range of the baseline controller. The system works as follows. The upgrade controller is given control as long as the system stays within the operational region of the safety controller. If it appears that the system is about to leave the operating region of the safety controller, the safety controller takes over and safely steers the system back to the operational region of the baseline controller. Once back within the operational region of the baseline controller, the baseline controller takes over and maintains control until the upgrade controller is reactivated. This architecture can be modeled as a hybrid system with discrete dynamics describing the Simplex switching logic and the continuous dynamics being the controlled system under each controller.

The Simplex architecture has been applied to the autolanding routine for the F-16 aircraft. The autolanding system works by directing a radio beam upward from the end of the runway at the ideal angle of descent and along the center of the runway. This beam then provides both horizontal and vertical guidance to the aircraft's autopilot system during the landing procedure. The autolanding system senses deviations from this predetermined glideslope and uses the pitch and roll input commands to make the appropriate corrections and bring the aircraft back in line with the glideslope. As the aircraft descends, it must remain within certain safety constraints about the glideslope. There are constraints on the vertical and horizontal deviations from the glideslope as well as on the aircraft dynamics such as the roll, pitch, and yaw angles and rates. As a safety measure before landing, there is a point designated as the decision height. If the aircraft is not within the constraints at the decision height, the landing must be aborted.

Through linearization and order reduction, the model of the F-16 dynamics can be decomposed into decoupled models of the longitudinal dynamics and the lateral dynamics, respectively. This project focused on the longitudinal dynamics. Specifically, the goal of this project was to verify that if the

Simplex system switched to the baseline controller at a certain height and within a certain set of constraints, the aircraft would reach decision height without violating the safety limits. Thus, this project contributes to the INSERT project a formal verification of a portion of the F-16 autolanding routine. In doing so, a general procedure for verification of hybrid systems using CheckMate was developed.

The report is structured as follows. Section 4.2 provides an overview of CheckMate. Section 4.3 details the F-16 autolanding system. Section 4.4 follows with a discussion of modeling the F-16 problem in CheckMate. Section 4.5 presents the verification procedure and results, and Section 4.6 concludes the report with a review of the project and its results.

4.2 CheckMate

4.2.1 Model Checking for Hybrid Systems

Systems are modeled in CheckMate using *polyhedral invariant hybrid automata* (PIHA). The PIHA are a restricted class of *hybrid automata* (HA)--a generalization of finite state automata [14, 15]. States in the finite automaton are called *locations* in the HA, and each location has associated with it a set of continuous dynamics and an *invariant*. The continuous dynamics are described by a set of flow equations in the continuous state space, and the invariant describes a region in which the continuous state must remain in order for the HA to remain in that location. Locations in the HA are connected via edges. Each edge describes a transition between locations and is labeled with guard and reset conditions. The transition becomes enabled when the continuous state satisfies the guard conditions. Similarly, when the transition is taken, the continuous state before and after the transition must satisfy the reset conditions. This structure is shown in Figure 11.

Analysis of hybrid automata can be a very difficult problem in general. For this reason CheckMate models systems with a restricted subclass of HA known as the polyhedral invariant hybrid automaton. A PIHA, as illustrated in Figure 12, is a hybrid automaton that conforms to the following restrictions:

- Each guard condition is a linear inequality (a hyperplane guard).
- Each reset condition is an identity.
- The continuous dynamics for each location are governed by an ordinary differential equation (ODE).
- For the hybrid automaton to remain in any location, all guard conditions must be false. This restriction implies that the invariant condition for any location is the convex polyhedron defined by the conjunction of the complements of the guards. This gives rise to the name polyhedral-invariant hybrid automaton

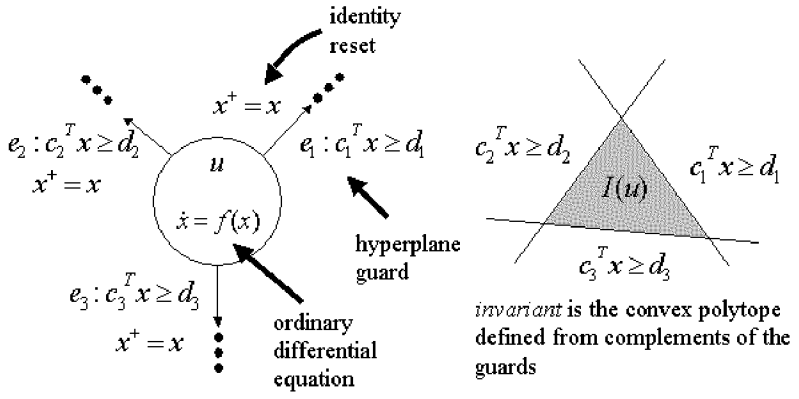


Figure 11. Hybrid Automaton

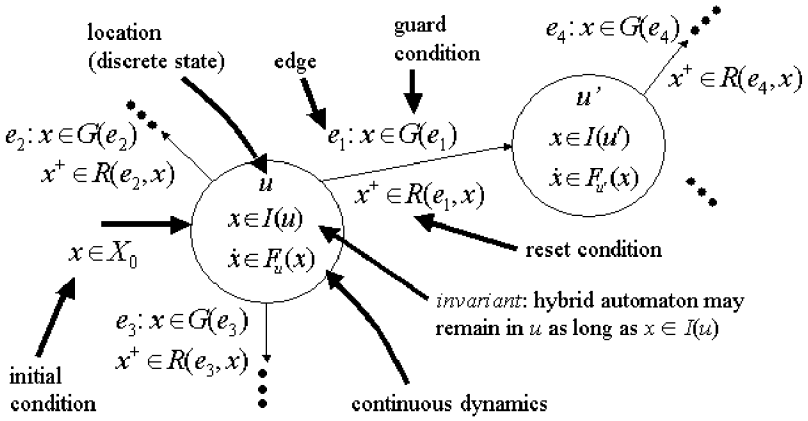
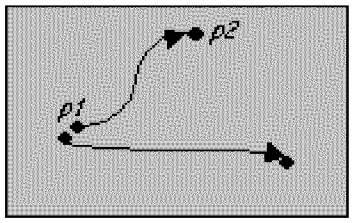


Figure 12: Polyhedral Invariant Hybrid Automaton

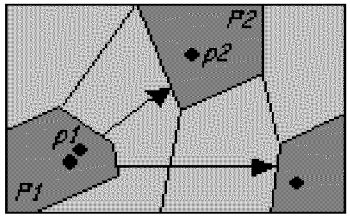
The QTSs are finite approximations to hybrid automata [16]. In a QTS, the state space of the hybrid system is partitioned, and each state in the QTS corresponds to a region of the partitioned state space. Transitions are defined between states in the QTS under the following conditions (see Figure 13). A transition is defined from a partition *P1* to another partition *P2* if and only if there is a state *p2* in *P2* that is reachable from a state *p1* in *P1*. CheckMate approximates a QTS by examining the original system at the switching instants. Thus, an *approximate quotient transition system* (AQTS) is formed using the switching surfaces to partition the state space and reachability analysis from the set of initial continuous states to determine connectivity [17].

The reachability analysis is performed using a method known as flowpipe approximations [18]. Flowpipe approximations provide polygonal approximations to the continuous flow of the system from a given set of states under the continuous dynamics associated with that location. These approximations are then used to define transitions in the AQTS. This results in the QTS on which CheckMate performs formal model checking using fixed-point iteration methods [24].

The AQTSs are conservative in that if a trajectory exists in the original system, there is a corresponding trajectory in the AQTS. Therefore, if we can prove that all trajectories in the AQTS satisfy some



(a)



(b)

Figure 13. (a) State Space of Original System; (b) State Space Partitioned for QTS

property, we can affirm that all trajectories in the original hybrid system satisfy the same property. This allows CheckMate to verify a class of specifications known as universal specifications, which are represented using the *ACTL logic*.

ACTL is a subset of the logic known as *computation tree logic* (CTL) [19]. In CTL the state transition graph is unfolded into an infinite computation tree. The root of the computation tree is a designated initial state. The CTL specifications are then used to describe properties of the system along branches of the tree. Temporal *path operators* such as X ("the next time"), F ("sometime in the future"), and G ("it is globally true") are used to describe properties along a single path through the tree. *Branching operators*, E ("there exists a path") and A ("along all paths"), are added to temporal specifications to arrive at a system specification in the CTL logic. In CheckMate, these concepts are applied to the AQTS. However, for the specification to hold true for the original hybrid system, we can describe only the universal properties of the AQTS-derived computation tree. That is to say, due to the conservative nature of the AQTS, we can ask only, "Do all paths in the AQTS have a certain property?" The CTL operator for "all paths" is A. Therefore, this extension of CTL is named ACTL.

Table 6. SCSB Parameters

Parameter	Description
Number of Continuous Variables	Number of continuous state variables in the continuous dynamics
Number of Discrete Inputs	Number of elements in the discrete input vector
Initial Continuous States	Initial values of continuous state variables
Switching Function m-File	Name of the m-file function which takes as arguments the value of the SCSB input u and the continuous state vector x, and returns the system type and the continuous state derivatives
Initial Continuous Set	Region in the continuous state space where the AQTS reachability analysis begins
Analysis Region	Bounded region of the state space in which the verification takes place

4.2.2 Entering System Models in Checkmate

Hybrid systems are entered into CheckMate through the Matlab Simulink/Stateflow GUI. CheckMate models are built using two customized (masked) Simulink blocks, along with several of the standard Simulink blocks.

One of the custom blocks in CheckMate is the *switched continuous system block* (SCSB). The SCSB represents a continuous dynamic system with the equation $\dot{x} = f_u(x)$, where u is a discrete-valued input vector, and the output of the SCSB is the continuous state vector x. The parameters associated with the SCSB are described in Table 6. Currently, three types of dynamics are supported in the SCSB: *clock dynamics* ($\dot{x} = c$, where c is a constant vector), *linear dynamics* ($\dot{x} = Ax + b$, where A is a constant matrix and b is a constant vector), and *nonlinear dynamics* ($\dot{x} = f(x)$).

The other custom block used by CheckMate is the *polyhedral threshold block* (PTHB). Each PTHB represents a polyhedral region, $Cx \le d$, in the continuous state space. The output of the PTHB is a

binary value indicating whether or not the continuous input vector x is inside the polyhedral region, i.e. whether or not the continuous state vector x satisfies the condition $Cx \leq d$.

Discrete dynamics are modeled in CheckMate using *finite state machine blocks* (FSMBs). An FSMB is a regular Matlab Stateflow block with the following restrictions:

- No hierarchy is allowed in the Stateflow diagram.
- Data inputs must be Boolean functions of PTHB and FSMB outputs only.
- Event inputs must be Boolean functions of PTHB outputs only, i.e. events can be generated only by the continuous trajectory leaving or entering the polyhedral regions.
- Only one data output is allowed.
- Every state in the Stateflow diagram is required to have an entry action that sets the data output to a unique value for that state.
- No action other than the entry action is allowed in the Stateflow diagram.

Other Simulink blocks supported by CheckMate include multiplexers for vectorizing signals and logic blocks (AND, OR, NOT, etc.) for creating logical combinations of PTHB and FSMB outputs. Additionally scopes, x-y plots, and other sink blocks can be used when running simulations. Figure 14 shows a typical CheckMate block diagram.

Several parameters are used throughout the verification process. These parameters as well as any variables used in the Simulink/Stateflow front-end model are defined and stored in the Matlab workspace. Parameters and variables can be defined manually or through the use of Matlab m-files. CheckMate uses

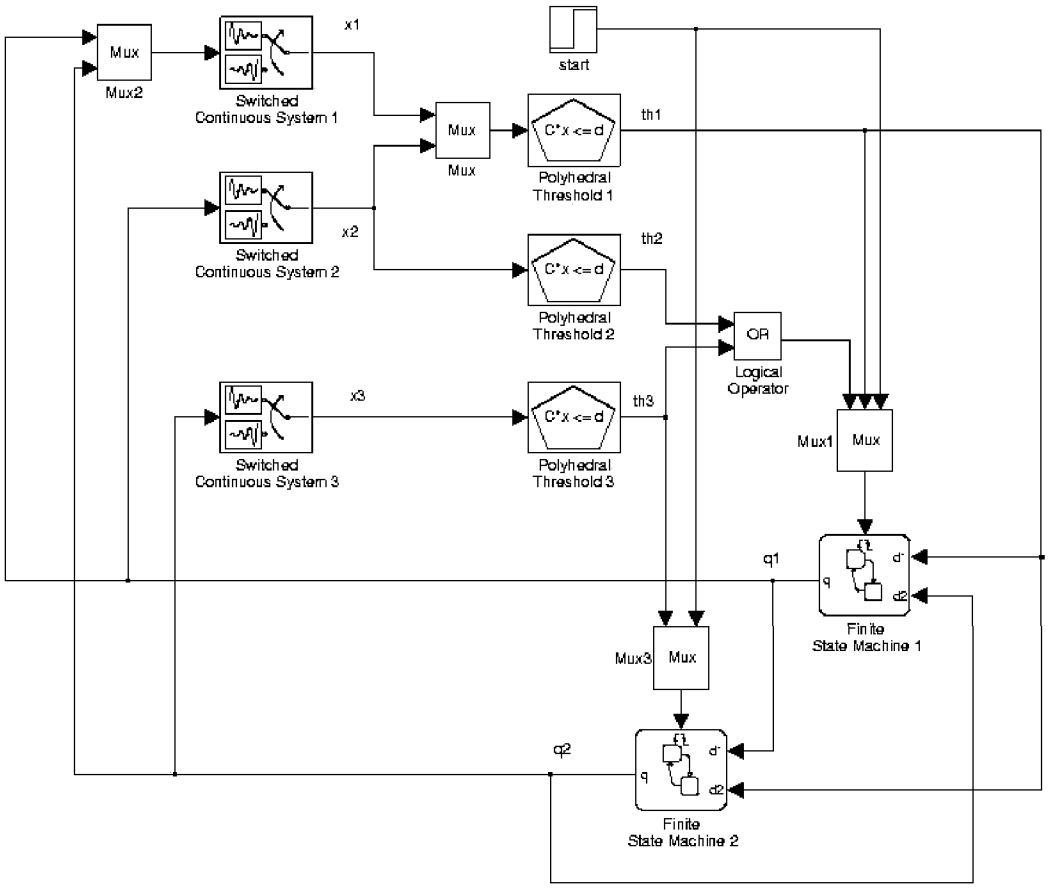


Figure 14. A CheckMate Block Diagram.

a set of global variables throughout the verification process. These variables must be declared in the base workspace before the verification is started. This can be accomplished in CheckMate using the *global_var* m-file script. The following three global variables must be defined before verification is possible. The GLOBAL_SYSTEM must be assigned the name of the Simulink model to be examined. The ACTL specification for verification must be contained as a string in GLOBAL_SPEC, and GLOBAL_APARAM must contain the name of an m-file function that takes as input the discrete state vector and returns a structure named `approx_param`. This `approx_param` structure contains several tolerance parameters that CheckMate uses to resolve numerical precision issues. CheckMate uses some of these parameters to partition the switching surfaces and initial continuous set (ICS), and others to eliminate extraneous faces in polyhedra.

4.2.3 Running the Verification

One of the tools available in CheckMate is the validation tool. The `validate` command simulates the model from the initial conditions specified in the initial continuous states parameter of the SCSB, uses that trajectory to build a QTS approximation, and performs verification on this approximation. If the `-vertices` flag is added to the `validate` command, the validation routine is performed on the vertices of the region given as the ICS in the SCSB parameters. This tool provides the user with an opportunity to get a quick first pass look at the overall verification process. Many times, a problem that would not be as obvious in the full verification routine can be spotted quickly with the validation tool.

The verification tool in CheckMate is invoked using the `verify` command. Several options are available with the `verify` command. The `-close` option tells CheckMate to close the Simulink model once the PIHA has been constructed. This is useful for verifications of larger models in which system resources become a concern. Also useful when system resources are a concern is the `-discard` flag. In each iteration, CheckMate determines which states are unreachable from the ICS. Nonetheless, CheckMate retains information about all of the states in the system. The `-discard` flag causes CheckMate to throw away all information about the unreachable states in each iteration. Two options useful for verifications requiring several iterations are `-save` <*filename*> and `-nopause`. The optional flag `-save` <*filename*> saves the Matlab workspace at the end of each iteration in a file named <*filename*><*iteration number*>`.mat`. This option is very useful for off-line data analysis. At the end of each iteration, if the verification does not return a positive result, CheckMate pauses and prompts the user with the choice to refine the approximation and continue with another iteration, or quit the program. The `-nopause` option causes CheckMate to skip this pause and automatically begin the next iteration, hence the name nopause. The actual verification process in CheckMate is broken down into several steps. The last two options allow the user to work with these steps individually. The `-step` <*step*> option instructs CheckMate to perform the <*step*> step of the process. CheckMate looks for the appropriate information in the Matlab workspace, and performs the specified step. Similarly, if the user has stopped the process after or in the middle of an iteration, the `continue` flag tells CheckMate to look at the workspace information and continue the verification from the last completed step in the process.

During the verification process (i.e., after issuing the `verify` command), CheckMate provides the user with feedback indicating the current step and the level of progress within that step. As each iteration is completed, the user will be told either that the system satisfies the ACTL specification, or that the verification returned a negative result. If a negative result is returned, the user can choose to begin another iteration or quit the program.

4.3 The F-16 Problem

4.3.1 F-16 Dynamics

The dynamic model of the F-16 used in our example was adapted from the eleventh order linearized model presented in [20]. This model takes into account the dynamics of the inner loop control between the command inputs (pitch and roll rates) and the flight control surfaces on the aircraft (i.e. rudder, aileron, and elevator). These inner loop dynamics are fast relative to the outer loop controllers. Therefore, when examining the first order effects of switching in the outer control loop, it is reasonable to use a reduced order model that does not involve these fast inner loop dynamics. With this in mind, order reduction was performed on the eleventh order model from [20] resulting in an eighth order model with the state variables and inputs described in Table 7. As a consequence of this order reduction, it was also possible to decouple the model into the longitudinal and lateral dynamics of the aircraft. Figure 15 shows how the variables are separated into lateral and longitudinal dynamics. Equation 1 gives the state equations governing each situation ((a) longitudinal sate equations (b) lateral state equations). This project focused on the longitudinal dynamics of the F-16.

$$\begin{bmatrix} \dot{\theta} \\ \dot{q} \\ \dot{dv} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -0.0183 & -0.42 & 0 \\ 4.5379 & 0 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ q \\ dv \end{bmatrix} + \begin{bmatrix} 0 \\ 27.7254 \\ 0 \end{bmatrix} Q \tag{1a}$$

$$\begin{bmatrix} \dot{\phi} \\ \dot{\psi} \\ \dot{p} \\ \dot{r} \\ \dot{dh} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0.1698 & 0 \\ 0 & 0 & 0 & 1.0143 & 0 \\ -0.1149 & -0.0508 & -1.5297 & -0.4294 & 0 \\ 0.0094 & -0.0071 & 0.0821 & -0.2788 & 0 \\ 0 & 4.5379 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \phi \\ \psi \\ p \\ r \\ dh \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 31.5814 \\ 1.4954 \\ 0 \end{bmatrix} P \tag{1b}$$

During the autolanding procedure, the aircraft is guided along a predefined glideslope. Deviations from the glideslope are measured in both the horizontal (*dh*) and vertical (*dv*) directions. These deviations are then used to calculate the appropriate pitch and roll rate commands to bring the aircraft back on line with the glideslope. Since the command inputs are functions of the state variables, the control function can be embedded into the model. A simple PD controller was developed for use with our reduced order model. The development of this controller, and the resulting affine system model are described in the next section.

Table 7: State Variables and Inputs

Variable	Description
ϕ	roll angle
θ	pitch angle
ψ	yaw angle
p	roll rate
q	pitch rate
r	yaw rate
dv	vertical deviation from glideslope
dh	horizontal deviation from glideslope

Input	Description
Q	pitch rate command
P	roll rate command

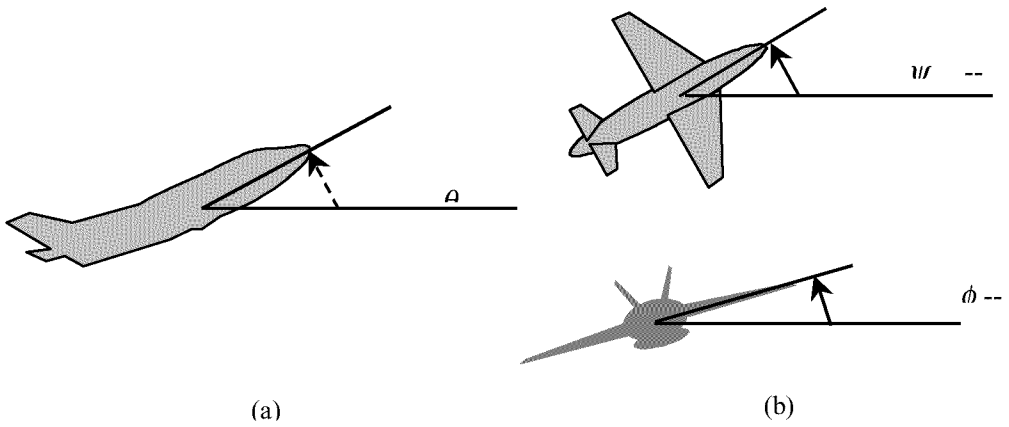


Figure 15. (a) Longitudinal and (b) Lateral F-16 Dynamics

4.3.2 Application of Simplex to the F-16

The Simplex architecture uses ARUs to provide fault-tolerant software upgrades in control systems [21]. Simplex switches among an upgrade, safety, and baseline controller based on the performance of the system. In the F-16 application, we are interested in the behavior of the system after control has been switched to the baseline controller. For this reason, it is only necessary to develop a single controller for the dynamic model. In addition to developing a controller for our model, it is also necessary to define constraints for that controller, as well as safety constraints for the aircraft in general. Once these details are in place, we will have all of the elements necessary to create our CheckMate model.

A simple PD controller was developed for use as a baseline controller in our example. The variable to be controlled is *dv*, so from the open loop state equations (see equation 2), we see that $\dot{dv} = 4.5379\theta$. Selecting gains of $kp = 0.0472e-3$ and $kd = 0.4384e-3$, we arrive at the closed-loop system.

$$\begin{bmatrix} \dot{\theta} \\ \dot{q} \\ \dot{dv} \end{bmatrix} = \begin{bmatrix} 0 & 1.0 & 0 \\ -0.0735 & -0.42 & -0.0013 \\ 4.5379 & 0 & 0 \end{bmatrix} \tag{2}$$

The safety constraints are derived from the operational region of the safety controller. We are not concerned with the performance of the safety controller, however, but with whether the baseline controller lands the aircraft without switching to the safety controller. With this in mind, the safety constraints were chosen so as to be reasonably close to constraints resulting from an actual safety controller. Specifically, the constraints on θ and q were based on physical limitations of the aircraft and

Table 8. (a) Safety and (b) Baseline Constraints

height	22.7	113	340	680
$\lvert dv_{max}\rvert$	5.0	10	50	200
$\theta_{min}, \theta_{max}$	-29.64,10.36			
$\lvert q_{max}\rvert$	5.0			

(a)

height	22.7	113	340	680
$\lvert dv_{max}\rvert$	5.0	1.0	10	50
$\lvert \theta_{max}\rvert$	0.05	0.1	1.0	2.0
$\lvert q_{max}\rvert$	0.01	0.05	0.5	1.0

(b)

pilot, and the *dv* limits were derived from reasoning about vertical deviations from which it would be possible to safely recover. As given in Table 8(a), the constraints on θ and q are constant, and the *dv* limits are piecewise linear functions of height given by the data points in the table. The baseline constraints for our model were chosen to assure a quick return to the glideslope under baseline control. The baseline constraints for all three variables are piecewise linear functions of height, as defined by the data points give in Table 8(b).

It is easy to see that there are many properties to be investigated with a system using the Simplex architecture. For example, does the safety controller drive the system back to the baseline operating region? Are the safety limits set such that the safety controller can gain control before the system fails? What constraints are sufficient to allow the system to switch back to the baseline controller and terminate safely? In this project we examined this final question. Specifically, we investigated the following question: if the system switches to the baseline controller within the baseline constraints at 1500 feet, will it land safely?

4.4 CheckMate Model of the F-16 Problem

The Checkmate model of the F-16 autopilot problem is shown in Figure 16. In the sections that follow we describe the details of this model.

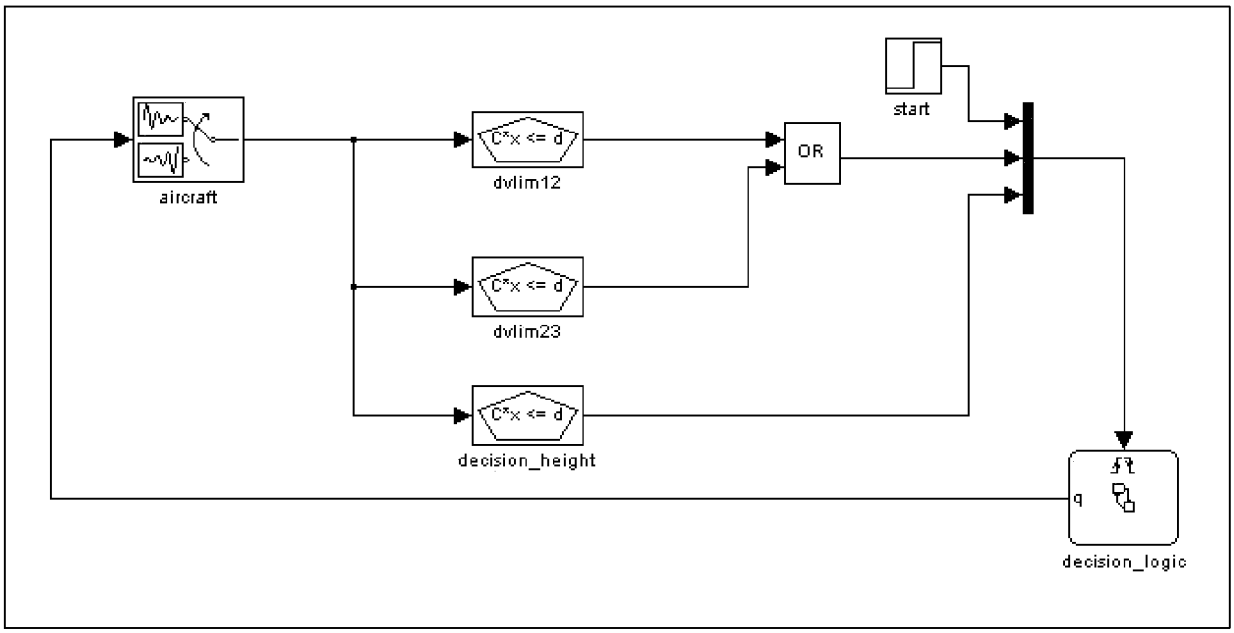


Figure 16. CheckMate Model of the F-16 Autolanding Problem

4.4.1 Switched Continuous System

A CheckMate SCSB is used to model the continuous dynamics of the system. The number of continuous variables, and the number of discrete inputs are four and one, respectively. There are three continuous variables associated with the dynamic model of the F-16 aircraft, and a fourth continuous state variable is introduced to track height. The single discrete input indicates the state of the FSMB, which is used to model the discrete dynamics of the system, discussed in detail below. The initial continuous state is used

only for simulation and can be set arbitrarily as needed for simulation analysis. This parameter can also be left blank if no simulations are going to be run.

The switching function m-file is named `F16_long_sw.m`. The body of the `F16_long_sw.m.` function consists of a simple switch statement.

```
function [sys,type] = f16_long_sw_bu(x,u)

% state variables
%   theta : pitch angle (deg)
%       Q : pitch rate (deg/s)
%      dv : vertical deviation (ft)
%       h : altitude (ft)

type = 'linear';
switch u
case 1,
  % normal control
  Aopen = [
         0    1.0000         0
   -0.0183   -0.4200         0
    4.5379         0         0
  ];
  Bopen = [0 27.7254 0]';
  kp = 0.0472e-3;
  kd = 0.4384e-3;
  A = Aopen - Bopen*[4.5379*kd 0 kp];
  A = [A zeros(3,1); zeros(1,4)];

  % true velocity = 260 ft/sec and glide slope angle is 2.5
deg
  b = [0 0 0 -260*sin(2.5*pi/180)]';

otherwise,
  A = zeros(4,4);
  b = zeros(4,1);
```

The discrete input is evaluated, and if it takes on the value of one, indicating normal safe operation, the A and b matrices under the baseline controller are returned. Otherwise, the system has either violated a safety constraint or reached the decision height, and the A and b matrices are returned as zero matrices, indicating that the FSMB has reached a terminal state (commit to the landing or abort). In either case, the system type is returned as linear.

The analysis region (AR) and ICS parameters take the name of workspace variables representing a linearcon object and a cell array of linearcon objects, respectively. A linearcon object is a CheckMate data structure used to represent polyhedra. In the case of the F-16 model, the ICS is represented by the variable F16_ICS, in a similar manner; the AR is represented by F16_AR. The actual AR is defined in the range of height from 1600' to 0' by the safety constraints on θ and q as well as arbitrary extrema of dv guaranteed to be outside the height-dependent dv safety constraints at all times. This guarantees that any state outside of the AR is an unsafe state. Specifically, the AR is characterized by the following inequalities: $-29.64 \le \theta \le 10.36$, $-5 \le q \le 5$, $-600 \le dv \le 600$, and $0 \le h \le 1600$.

We wish to verify that if the F-16 switches to the baseline controller at 1500 feet, it will reach decision height without violating any safety constraints. Given this goal, the ICS is defined as the entire region at 1500 feet in the state space in which the baseline controller could be switched into operation. Using the limits discussed in the previous section, we arrive at: $-4.41 \le \theta \le 4.41$, $-2.21 \le \theta \le 2.21$, $-146.47 \le dv \le 146.47$, and $h = 1500$.

4.4.2 Polyhedral Threshold Blocks

Three PTHBs are used to define the safety constraints in the CheckMate model. One is used to define the decision height, and the other two are used in a logical combination to define the height-dependent *dv* safety constraints. All PTHBs take as a parameter the name of a linearcon variable in the workspace.

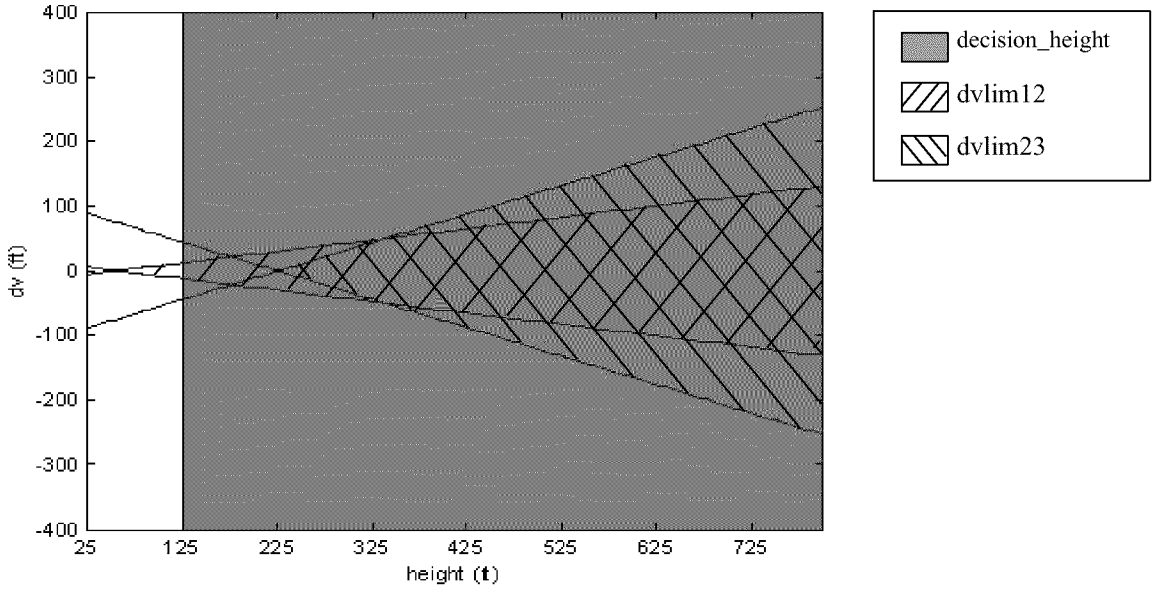


Figure 17. PTHB Regions (in the dv versus Height Space)

Figure 17 illustrates the regions defined by the PTHBs. The decision_height PTHB defines the region $h \ge 125$. This means that a falling edge event from the decision_height PTHB to the FSMB indicates that the system has reached the decision height. The other two PTHBs are named dvlim12 and dvlim23. They are used to define the safety constraints in the two segments of the glideslope discussed in Section 3. The constraints in the first segment ($113 \le h \le 340$) are represented by dvlim12, and dvlim23 represents the constraints in the second segment ($h \ge 340$). Using the limits discussed in Section 4.3, the dvlim12 constraints are defined as $-0.18h + 10 \le dv \le 0.18h - 10$, and dvlim23 represents the constraints $-0.44h + 100 \le dv \le 0.44h - 100$. Due to the relationship of these lines and the segmentation of the glideslope, in order for the system to reach an unsafe state it must leave both dvlim regions. We represent this relationship with a simple OR block from the standard Simulink library. The output of this OR block is in turn fed as an event input to the FSMB, which uses a falling edge event from that input to indicate that the continuous system has violated a safety constraint.

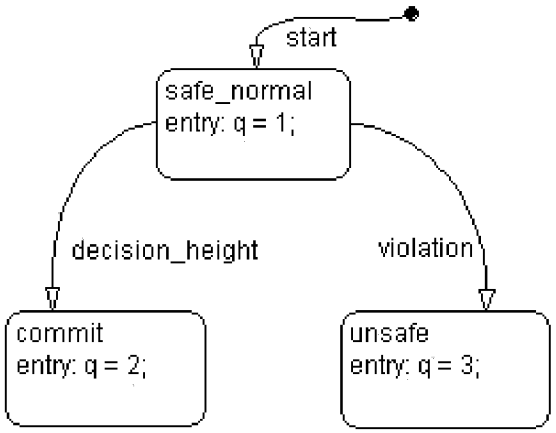


Figure 18: F-16 Finite state machine – decision_logic

4.4.3 Finite State Machine Block

One FSMB is used to model the discrete dynamics of the F-16 autolanding system. The FSMB, shown in Figure 18, consists of three states: normal, commit, and unsafe; three event inputs: start, decision_height, and violation; and one data output, q. As the name suggests, normal is the state used to represent proper operation within the safety constraints and above the decision height. The start event is used to initialize the FSMB to the normal state. If at any point during normal operation the FSMB receives a violation event (a falling edge from the OR of dvlim12 and dvlim23), indicating that the system has violated the safety constraints, a transition is made to the unsafe state indicating an aborted landing. Similarly, if the system remains in the normal state until it reaches decision height (indicated by a falling edge from the decision_height PTHB), the FSMB transitions to the commit state indicating the decision to commit to a completed landing.

4.4.4 Specifications

The final portion of modeling the F-16 problem in CheckMate is generating the specification to be verified. For our example, we want to verify that if the aircraft intercepts the glideslope within the baseline constraints at 1500 feet, the aircraft will reach decision height without violating the safety constraints. Using the CheckMate model described above, this property can be expressed with the following ACTL specification: (AF (decision_logic == commit)) & (AG ~out_of_bound). In English, this specification reads, "along all paths some time in the future, decision_logic == commit is true, and along all paths, it is globally true that out_of_bound is false." For our model, this means that the aircraft reaches decision height without entering the unsafe state (there is no way to get to commit from unsafe) and that the aircraft never leaves the AR. This last part of the specification is necessary for two reasons. First, in our example, the AR is constructed such that anything outside of the AR is a violation of the safety constraints, and secondly, all of the methods used in CheckMate are valid only in a bounded region of the state space. Therefore, if the system leaves the analysis region, the CheckMate approximations and verification are no longer valid. The following section describes how this specification was verified using CheckMate.

4.5 Verification Procedure and Results

These sections describe how the general procedure of parameter selection and entry, validation, validation-based refinement, and full verification is applied to the F-16 problem to achieve a positive verification result. In this case it is necessary to use a technique known as *funneling* [22] to achieve a full

verification result. A discussion of this technique and the positive verification results conclude this section.

Table 9. Fields of the approx_param Structure

Field Name	Value for F-16 Problem	Description
dir_tol	--	Used to determine if vector field is pointing in the same direction on all parts of the partition
var_tol	2	Used to determine if variations in the vector field are within a certain range of the normal vector to that face
size_tol	2	Sets the maximum size of any partition
W	diag ([1/50, 1/20, 1/1600, 1/1600])	Weighting matrix to scale and square the axes of the state space
T	2	Length (in time) of the flowpipe segments
Tsim	2	Time step for simulation between switching surfaces
max_bisection	6	Maximum number of steps in "back and forth" routine to approximate exact crossing of a switching surface
quantization_resolution	0.05	Resolution used to approximate exact crossing of a switching surface
max_time	Inf	Maximum amount of time examined in the reachability analysis
reachability_depth	Inf	Maximum number of switching instances examined in the reachability analysis
min_angle	5	Angle values used in eliminating extraneous faces of polyhedra
med-angle	10	
Extra_angle	30	
max_angle	110	
unbound_angle	160	
edge_factor	2	Edge values used in eliminating extraneous polyhedra
edge_med_length	1000	

4.5.1 Parameter Selection and Entry

The first step in the verification procedure is to choose appropriate tolerances in the `approx_param` structure. This structure contains several tolerances used in partitioning the state space, performing the reachability analysis, and removing unnecessary or redundant polyhedra faces. The fields in this structure and the values assigned to the fields for the F-16 verification problem are shown in Table 9. The first field, `dir_tol,` is used to assure that the vector field is pointing in the same direction on all parts of the partition, and is not necessary in the linear case. The weighting matrix `W` was set to a diagonal matrix with the values 1/50, 1/20, 1/1600, and 1/1600 on the major diagonal. This guarantees that all values will fall in the range of -1 to 1. With this in mind, `size_tol` and `var_tol` were both set to 2. This allows for the maximum size of a partition to include the whole range of values in any one direction, and similarly allows the vector field to vary across the same range of values.

The last seven parameters of Table 9 (mid-angle through edge_med_length) are used in eliminating redundant faces from polyhedra. Due to numerical issues in the flowpipe approximation method, it is possible to get approximations with faces that are nearly parallel, but maintained as separate faces.

CheckMate uses seven parameters involving angle limits and edge length factors to eliminate small redundant faces. For example, in Figure 19, F2 would be eliminated. Parameters F1 and F3 would be extended to form a new polyhedron with one less face. This aids in reducing the computational complexity of the next step in the reachability analysis. For the F-16 example, these seven parameters were left at their default values given in Table 9.

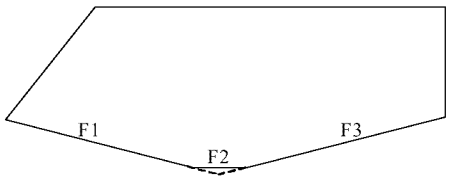


Figure 19. Polyhedral Face Elimination

The six parameters in light gray are used in the reachability analysis during the construction of the AQTS. The `T` field determines the length (in time) of the flowpipe segments. In the F-16 example, this was set to 2 seconds. This was based on some simulation analysis. The fastest modes of the system have time constants on the order of 10 seconds, so a time step of 2 seconds allows for a reasonable approximation of the system. Similar reasoning holds for `Tsim`, the time step used for simulation between switching surfaces, so it was also set to 2 seconds. Default values were used for `max_bissection` and `quantization_resolution`. These parameters are used once the simulation finds that the flow crosses a switching surface. At that point, CheckMate uses a "back and forth" routine governed by `max_bissection` and `quantization_resolution` to approximate the exact crossing of the switching surface. There is a clear termination criterion in the F-16 example (the plane reaches decision height), so `max_time` and `reachability_depth` were set to infinity.

```
function setup()

% SETUP.M Sets up workspace variables and verification parameters for
% F-16 auto landing verification.

% Keith Richeson
% 4-Nov-99

% declare global variables in base work space
evalin('base','global_var');

% declare global variables locally
global_var

% setup block diagram parameters
ICS_CE = [0 0 0 1];
ICS_dE = 1500;
ICS_CI = [-1 0 0 0;1 0 0 0;0 -1 0 0;0 1 0 0;0 0 -1 0;0 0 1 0];
ICS_dI = [4.411765;4.411765;2.205882;2.205882;146.470588;146.470588];

AR_C = [1 0 0 0;-1 0 0 0;
        0 1 0 0;0 -1 0 0;
        0 0 -1 0;0 0 1 0;
        0 0 0 -1;0 0 0 1];
AR_d = [10.36;29.64;5;5;600;600;0;1600];

dvlim12_C = [0 0 1 -0.176350658661949;0 0 -1 -0.176350658661949];
dvlim12_d = [-10;-10];
```

```
dvlim23_C = [0 0 1 -0.440876646654872;0 0 -1 -0.440876646654872];
dvlim23_d = [-100;-100];

decision_height_C = [0 0 0 -1];
decision_height_d = [-125];

assignin('base', 'F16_ICS', {linearcon(ICS_CE, ICS_dE, ICS_CI, ICS_dI)});
assignin('base', 'F16_AR', linearcon([], [], AR_C, AR_d));
assignin('base', 'dvlim12', linearcon([], [], dvlim12_C, dvlim12_d));
assignin('base', 'dvlim23', linearcon([], [], dvlim23_C, dvlim23_d));
assignin('base', 'decision_height', ...
    linearcon([], [], decision_height_C, decision_height_d));

% Set up verification parameters
GLOBAL_SYSTEM = 'landing5';
GLOBAL_APARAM = 'F16_long_param';
GLOBAL_SPEC = ['(AF (decision_logic == commit)) & (AG ~out_of_bound)'];
```

The `approx_param` structure is returned from a MATLAB m-file function. The name of this file, as well as the ACTL specification and the name of the Simulink system used to model the system are assigned to GLOBAL CheckMate variables. In the case of the F-16 example, this was accomplished through a MATLAB m-file function named setup.m, listed in Fig. 11. In addition to setting up these GLOBAL variables, setup.m was also used to assign values to variables used in the Simulink model.

4.5.2 Funneling Approach to Avoid Numerical Problems

Before the full verification routine was run, the F-16 example was subjected to the simulation-based validation routine. This routine was run for the eight vertices of the ICS defined by the baseline constraints at an altitude of 1500 feet. Validation results showed that all eight vertices returned a positive verification result. However, by using x-y plot blocks during the validation, it was noticed that two of the four dimensions (θ and q) converge to zero in a relatively short amount of time. This result indicated a potential problem in the flowpipe approximation. In a situation such as the F-16 example, as the flow collapses, it becomes a challenge to distinguish points that are close together from points that are the same. This causes problems in the flowpipe approximation routine as it becomes unclear which points are distinct and which are coincident. This problem manifests in the form of warnings from the Matlab optimization routines during the reachability analysis, or by causing indexing errors when array elements are left empty or undefined by a failed optimization routine. With this in mind, the full verification was attempted, and this convergence problem was quickly confirmed.

Funneling is a technique being used in designing hybrid control systems [22]. The basic concept as applied to control synthesis is to use multiple controllers each with an individual goal state to drive the system to an overall goal. This can be accomplished if each individual controller's goal (the small end of the funnel) lies within a region in which another controller can drive the system to its individual goal (the large end of another funnel). In this situation, the system switches between controllers, each driving the system to its own goal until the overall system goal is reached. A similar methodology can be applied to the verification problem. That is, if we can verify that all trajectories from one set of states eventually reach another specific region in the state space, and in turn we can verify that all trajectories from that region return a positive verification result, then the overall verification holds. This can easily be expanded to include several steps (i.e., one region leads to a second, the second leads to a third, and so on until the last region returns a positive result).

The funneling technique is applied to the F-16 example in the following manner. During the verification routine, if it is discovered that the flow is converging to a point that is likely to cause numerical problems, the routine is stopped and the flow is "blown up" to a larger region. The verification is continued starting from this new larger region, and this procedure is repeated several times throughout the verification. This results in a chain of reasoning similar to the funneling approach. The procedure is illustrated in Figure 20. The ICS flows to the inflated region, which in turn leads to a positive verification result, thus providing a positive overall verification result.

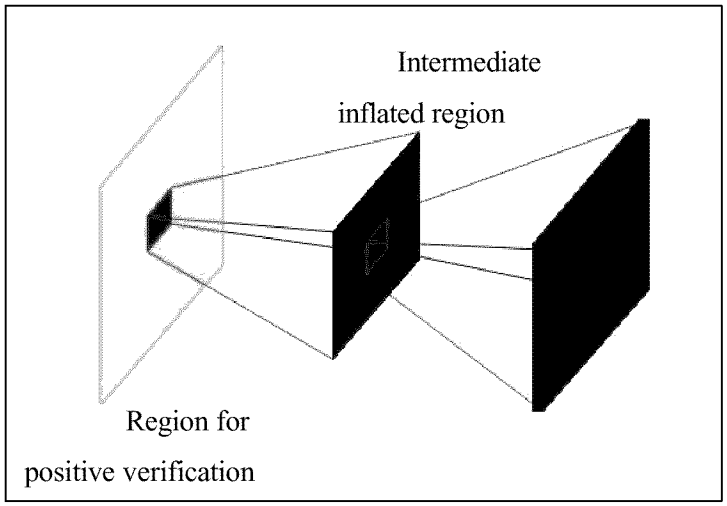


Figure 20. Illustration of funneling

This method was manually applied to the F-16 example using CheckMate. The following heuristics were used as a starting point. The glideslope was divided into several segments defined by the altitude of the aircraft. The limits on each segment were decided using the validation routine and the simulation capabilities of CheckMate. Five segments were defined: 1500-1000 feet, 1000-500 feet, 500-340 feet, 340 feet-250 feet, and 250 feet-125 feet. These segments are illustrated in Figure 21. At the end of each segment, the flow from that segment was inflated to the baseline limits at that height for θ and q, and the dv limits were extracted from the flow approximation and left unchanged. This process led to a successful verification in the first segment of the glideslope, but a failure was discovered for the second segment. Given this result, the heuristics were modified, and less conservative initial sets were chosen for each segment. With this modification, the process was repeated and a successful verification was obtained. The verification results are discussed in the next section.

4.5.3 Positive Verification Result

For each segment of the glideslope, a positive verification was returned after a single iteration. This indicates that the system came through each section without violating the safety constraints. Further,

Table 10. Results of Funneling Verification Procedure

Segment	Time to verify (min)	Volume of initial set	Volume of final set
1500-1000	1.92	1.1403×10^{4}	1.0359×10^{-4}
1000-500	3.83	5.12×10^{-1}	4.6517×10^{-9}
500-340	2.03	7.2×10^{-5}	1.9247×10^{-7}
340-250	4.57	6.0×10^{-5}	2.1439×10^{-6}
250-125	4.44	1.92×10^{-4}	1.8757×10^{-6}

since the initial set for each segment was developed from the approximation of the flow from the previous segment, it is trivial to prove that the flow from each segment ends up within the initial set for the next segment. Finally, since the initial set of the first segment is the ICS for the overall verification and the lower limit of the final segment is the decision height, it is reasonable to conclude that the system proceeds from the ICS (the baseline limits at 1500 feet) to the decision height without violating any of the safety constraints. That being the case, this is a positive verification of the specification defined in Section 4.4.4.

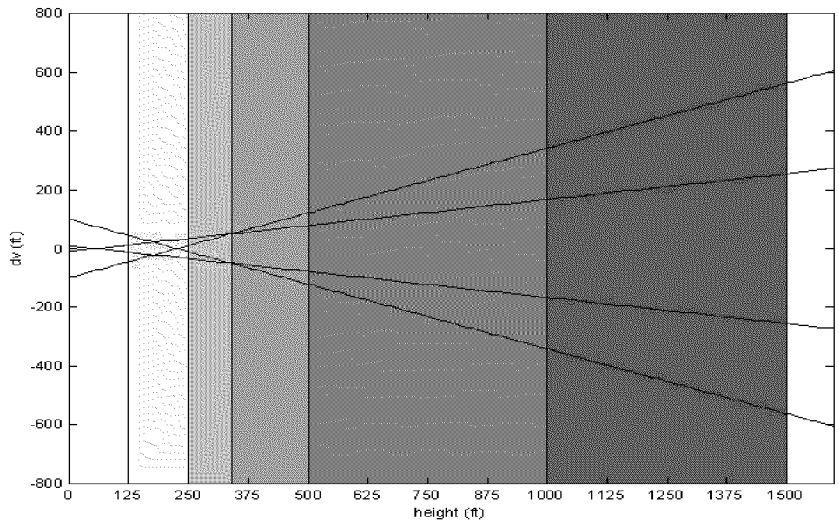


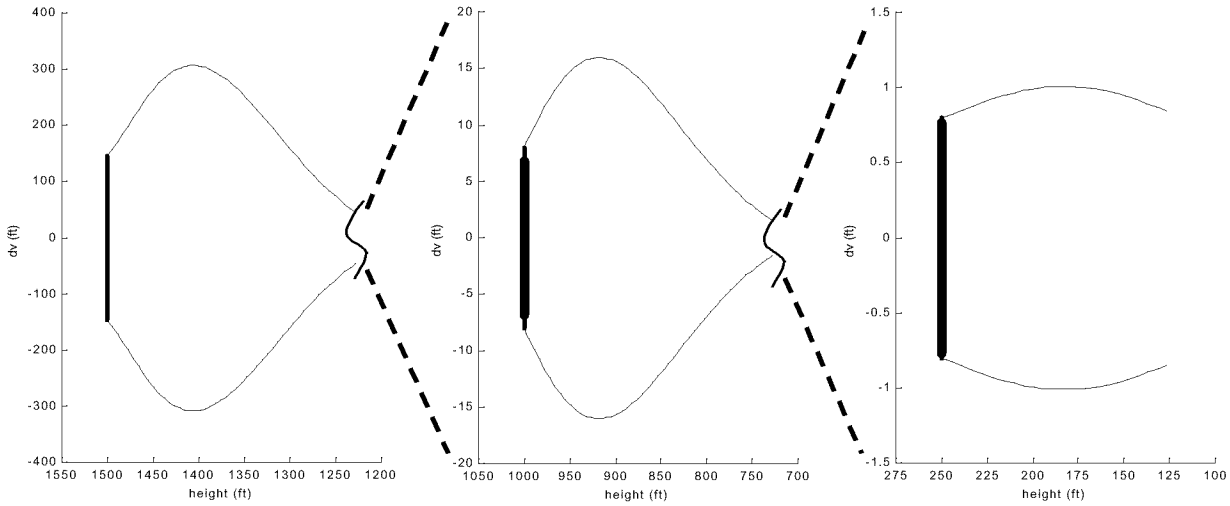
Figure 21. Height Segments for Full Verification

The results of the funneling verification procedure are shown in Table 10 and Figure 22. Table 10 shows the time required to verify each segment and the volume of the initial ("blown up") and final (converging) sets. Figure 22 illustrates the results in the dv versus height plane. Figure 22(a) shows the ICS on the left, and its rapid convergence toward zero. In Figure 22(b), the verification is stopped at 1000 feet, and the converging flow (the wide line) is enlarged slightly. The verification restarted, and this procedure is repeated at 500 feet, 340 feet, and 250 feet. In the final step (Figure 22(c)), we see that the inflated region at 250 feet, reaches the decision height (125 feet) before the flow converges significantly. If the procedure had not been successful at some stage, less conservative initial sets could be used to see if that would lead to a positive verification result. In general, since verification of hybrid systems is only semidecidable, continued failure of the verification would not be conclusive, i.e., one could not necessarily conclude that the system is unsafe. It would, however, indicate that the system should probably be redesigned to make it verifiable.

4.6 Conclusion

Several tools have been developed for verification of hybrid systems [23-28]. This project used one of those tools to verify a complex hybrid dynamic system. The CheckMate tool uses hybrid automata to approximate the behavior of the real system and then uses model checking routines to perform verification on these conservative approximations. This tool was applied to the verification of the F-16 autolanding routine as implemented through the Simplex architecture. A funneling approach was used to avoid numerical convergence problems in the approximation routines. It was also verified that the aircraft

reaches decision height safely if the baseline controller takes control within the baseline constraints at 1500 feet.



(a) (b) (c)

Figure 22: Verification Results in the dv vs. Height Plane for Selected Glideslope Segments

There are several directions for future work. This project dealt with a system that converges significantly and quickly from a large initial set. The funneling procedure, when applied to accomplish the verification, is an exercise in dynamic scaling. The tools are available in CheckMate to automate this process. The automation resulting from the use of such tools would be a great improvement when dealing with systems such as the F-16 example. Another direction for future investigation is numerical precision and sensitivity. This project showed that increasing the flow approximation by a relatively small amount caused a large difference in the outcome of the approximation routines. Further, there are a great number of parameters used throughout the verification routine. A numerical analysis of these parameters and their effect on the verification routines and numerical sensitivity would certainly be appropriate. The final issue to be discussed here is the computational complexity of the routines. Everything in CheckMate is represented using polyhedra. This means that any operation performed in CheckMate must be extensible to higher dimensions. It is currently the situation that the routines will work in higher dimensions, but it is also the case that for dimension greater than three, the computational complexity causes lengthy verification runs and presents challenges to system resources such as memory and processor loading. Improvement in any of the areas discussed here would certainly serve to improve the CheckMate tool and contribute to the hybrid systems community.

5. ARC-Based Application Validation

5.1 Introduction

This chapter discusses tool support and an approach for identifying application inconsistencies at design time. These inconsistencies may be introduced during an upgrade to the application and may manifest themselves as residual errors. While INSERT offers technology to tolerate such errors, the focus of this technology is to identify inconsistencies before an application is deployed. These inconsistencies may be due to a change introduced to the application or an ARC-based component reconfiguring itself, i.e., switching variants at run time. In the former case it is desirable to understand the impact of a change and to propagate the change such that hidden side effects are minimized. In the latter case it is desirable to determine at design time whether certain combinations of component variants result in inconsistent configurations, in which they should be avoided if possible. Notice that, even if not handled proactively, the effects of this inconsistency may still be observed by the ARC mechanism of other components, resulting in further reconfiguration. Proactive treatment of such configuration inconsistencies, however, results in better system performance.

We proceed as follows. In section 5.1 we examine characteristics of example avionics and other control system software that are possible culprits in causing residual errors as part of an upgrade. In section 5.2 we examine how these characteristics can be captured in a model, and analysis can identify potential inconsistencies. In section 5.3 we discuss how this consistency analysis is used to identify potentially inconsistent logical configurations of applications that consist of multiple ARC-based components and how these configuration constraints can be enforced at run time. Section 5.4 elaborates on how the same model of the system can be used to determine the impact of a change to an application, i.e., how the model aids in identifying all components that are potentially affected and ensuring that the change is propagated consistently. Section 5.5 describes the prototype analysis tool implementation

5.2 Issues in Avionics System Upgrades

This section focuses on characterizing avionics systems. The purpose is to identify key characteristics of avionics system components that other components make assumptions about. Those assumptions are often not documented in the code, which is the primary artifact available for maintenance. At the same time these assumptions are reflected in parameters and constants embedded in the code. Once identified, key characteristics can be reflected in a model and made available for system-level dependency analysis during maintenance. As a result of such an analysis the maintenance engineer can then revisit detailed models, such as a predictive control model or models used for timeline analysis, as appropriate.

5.2.1 General Observations

Legacy systems, i.e., many systems currently in operation, can be described as federated single-processor systems that make up a distributed control application. Application scheduling for each component is performed locally by cyclic executives. The application functionality is tightly coupled with and contains extensive knowledge of the physical architecture and capacities of the execution platform. Limited throughput of the platform results in limitations of some components that are hard-coded into other components. For example, the number of targets being tracked is limited. Prediction algorithms are based on low-order estimation techniques. Control loops exist between these distributed components. Time constants that are hard-coded in the components are based on processor and communication speed. System control equations are time sensitive, i.e., changes in temporal system characteristics affect control parameters, which are embedded in the application code as control constants.

Avionics systems are evolving to make use of ever increasing processing speeds and capacity of commercial hardware. They are migrating to tightly coupled multiprocessor architectures with real-time operating systems supporting more flexible task execution through preemptive scheduling techniques. Multiple avionics system components are sharing computing resources. Changes in processor speed, communication services, and scheduling models impact many parameters of an avionics implementation. In many cases such dependencies between temporal characteristics in terms of task execution and in terms of the application semantics are based on assumptions that are not explicitly recorded. As a result, side effects of changes to one such time-sensitive parameter are not detected until integration testing. At the same time, additional computing power allows functionality to be increased, whether in the form of more precision in predictive algorithms, an increase in the number of targets to be tracked, or new services such as handling of new tactical situations or mission planning. In other words, there is increasing demand for incremental upgrade and continuous evolution of avionics systems.

We will examine two scenarios of avionics system upgrades to further identify component characteristics that are key to identifying assumptions and dependencies that previously were hidden, i.e., not considered in maintenance and upgrade activities. Then we will examine the implications of using INSERT, a technology in support of incremental upgrade.

5.2.2 Visual Bombing Sights

The system depicted in Figure 23 represents the generic problem of visual bombsights in existing tactical aircraft. In this system, ranging sensors and weapon impact prediction algorithms are used to generate real-time aiming cues on avionics displays for an operator (pilot). The operator steers the aircraft until the aiming cues align with the target, whereupon weapons are released. This system is therefore categorized

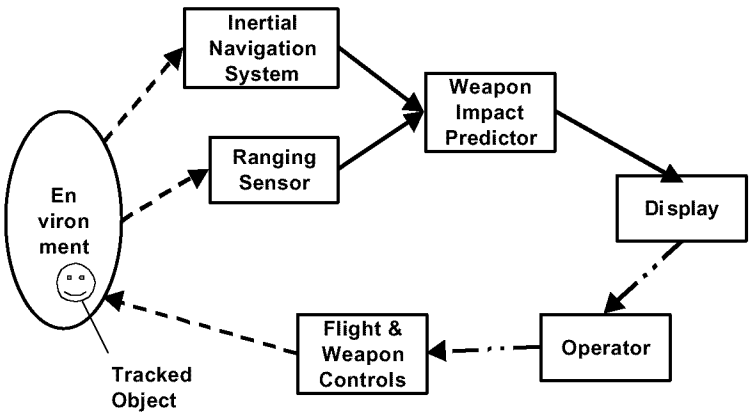


Figure 23. Visual Bomb Sight

as a distributed, closed loop control system. Some of the components in the system have internal control loops as well (i.e., flight controls and ranging sensors).

Weapon delivery solution computation involves processing information from various sources, such as the following:

- aircraft position and velocity
- aircraft mean sea level attitude
- slant range to weapon impact point
- aircraft attitude

- weapon ballistic characteristics (weight, coefficient of drag, etc.)
- atmospheric data (pressure, temperature, air density, winds, etc.)
- other factors such as weapon ejection velocity and orientation

To construct a correctly performing design solution for the visual bombsight problem, much information, such as the following, must be known about the quality of the input data used:

- latency of sensor information describing aircraft kinematics and relationship to earth
- characterization of sources of error for all input data (often modeled and important in prediction computations).

Discussion of the computational steps involved in the solution can be divided into the following two categories:

1. Compensation for latencies in input data (temporal alignment): Tag the measurement with the time of measurement. Use rate information (aircraft velocities, attitude rates, etc.) to compensate (extrapolate) for the time from measurement. The longer the extrapolation period, the more error is introduced. Errors in the ability to tag the measurement with the time and the ability to resolve the current latency in the measurement time can also be a major error contributor. These timing and resulting extrapolation errors generally manifest themselves as additional noise (uncertainty) in the parameters being estimated. Note that sometimes rate information is not available and often must be estimated or possibly ignored. This creates additional errors.
2. Minimization of error in geometric predictions: Predictions are handled similarly to measurements in the past, except that the future events are generally less certain than future events (given good time tagging mechanisms). Predictability of timing becomes even more important with future events. Additional errors occur in the uncertainty of rate information. At best, current rates are known, and future rates that affect the parameter being estimated in the future are generally not known and must be estimated (or the effects ignored). These all point to minimizing the absolute time between prediction and usage in addition to minimizing the uncertainty.

The design objective of this control system can be expressed in terms of symbology positioning errors. Timeline analysis in terms of absolute latency, variability, and predictability from measurement to display or weapon release and rate of symbol positioning update drive the stability of the symbology (jitter, drift, etc.).

This example shows that the resulting model for calculating the predictions is quite elaborate. Such models are often developed and validated through extensive modeling and simulation. Our goal is to identify key characteristics of components and may be reflected in the output data stream of the component. Other component designs may be based on assumptions about those characteristics, which results in hidden dependencies.

Some of the characteristics of a component, i.e., the input it processes, are expressed in terms of assumptions about data type, measurement unit, coordinate system, and legal values of elements in the data stream. Data stream characteristics may include rate, guaranteed availability of elements, and first order constraints on data values in the stream. The data may have error characteristics in terms of accuracy from sensing, prediction, and from time-related variability. An upgrade scenario of the visual bombsight system illustrates the intricacies of time-related properties.

The visual bombsight system was originally implemented on a federated hardware configuration with cyclic executives. The introduction of the Modular Mission Computer (MMC) resulted in a port to a new hardware platform with faster processors and preemptive scheduling, a port to Ada, as well as

enhancements to the functionality. Timing-related hidden dependencies manifested themselves in erratic behavior (oscillation) of the displayed symbology.

Root cause analysis focuses on providing an understanding of the effects of changing the scheduler and the deadline on the jitter. As pointed out by domain experts, there is a close interplay between the timing characteristics in terms of scheduled tasks (processes), and the temporal and accuracy characteristics of the processed data in terms of application semantics.

Changes in the application can have unforeseen impact on component characteristics. A change in the task timing characteristics (an explicit change in deadline, or an implicit change in jitter as a result of changing the process scheduling model from cyclic to preemptive or from periodic processing to message-driven aperiodic processing) can affect the characteristics of the data stream being processed. Therefore, it is beneficial for maintainers to be aware of such dependencies.

In summary, we can have time-sensitive component characteristics that are affected by timing properties of the implementation in terms of real-time processes.

5.2.3 INS Upgrade

This section summarizes a scenario of avionics system upgrade that involved navigation fusion and "standard" Inertial Navigation System (INS) replacement. This section could be called "the myth of Form, Fit, Function, Interface (FFFI) compatibility when upgrading sensors."

The original F-16 used an INS developed by Singer-Kearfott. It was desired to position symbology on the Head Up Display (HUD) using the attitude information provided by this INS. There is latency from the time of attitude measurement to the time of display on the HUD. Proper positioning would require the prediction of the attitude of the aircraft at the time the symbology is displayed on the HUD. The attitude measured by the INS, therefore, needed to be extrapolated using attitude rates to the time of display. Attitude rates must be estimated based on attitude outputs from the INS. This was accomplished and the system worked. Later, a new INS was added to the F-16. The INS was billed as "form, fit, and functionally" and was compatible with the previous system. When it was installed, the symbology position was extremely noisy. After much investigation, it was determined that the new system filtered its output attitude, whereas the earlier system did not. This had the effect of providing smoother attitude, but it destroyed the assumptions made by the attitude prediction algorithm. The algorithm had embedded knowledge about the behavior of the INS system. This knowledge can be expressed as an assumed characteristic of the input data stream: in terms of processing unfiltered, raw INS sensor data; or in terms of delay of the sampled data due to filtering.

5.2.4 Multiple Modes and INSERT

Avionics components and other control system components often support multiple operating modes. Different modes exhibit externally observable differences in behavior. Selection of modes may be externally controlled by operator or another component. Modes may be usable under certain conditions. Some components may depend on other components operating in certain modes. Mode condition and mode transition tables are a common way to model such system behavior.

The INSERT package supports dependable upgrade of avionics components. It does so by supporting the presence of multiple component variants and by a fault tolerance mechanism that monitors an upgrade variant and at run time reverts to a well established baseline variant.

In an INSERT-based application, different variants of an ARC may show different characteristics that are externally visible. Introduction of an upgrade variant may impact other components and require changes to them. Other components may want to take advantage of the enhancements in the upgrade variant, resulting in new variants of their own. The result is a set of dependencies on particular component variants similar to those between modes of different components.

When INSERT is applied to systems with multiple multimode components, the number of potential combinations of component modes and variants increases quickly, and tool support is essential for examining acceptable combinations. A multimode component may evolve as a whole in multiple variants, i.e., the whole component is upgraded even though only one mode may have changed. Modes within a multimode component may also evolve independently, i.e., for each mode, new variants are introduced separately. The first approach lends itself to designs with highly interdependent modes, while the second approach better supports components with independent modes.

5.3 Identifcation of Inconsistency through System Models

Recently architecture description languages (ADL) have emerged from the research community [29]. These notations allow you to model a system in terms of a collection of interconnected components, in which the components may themselves be composites of other components. The notations provide facilities for specifying additional constraints of various forms, typically in terms of the types of data and event communicated and in terms of behavior (e.g., Wright [30] and Rapide [31]). Systems modeled in ADLs can be analyzed and simulated. MetaH [32] is an ADL that is tailored to support embedded real-time applications, i.e., it allows modeling of applications in terms of periodic and aperiodic processes. System models expressed in MetaH can be analyzed for Generalized Rate Monotonic Analysis (GRMA) [33] schedulability. The system model is also the basis for automatically generating an executive and performing a system build. As result, the MetaH toolset can make certain guarantees about the consistency between the system model and the actual implementation. These guarantees include consistency in the data types communicated between components, and satisfaction of timing requirements. Together with address space protection between processes, these guarantees provide a high degree of fault avoidance and tolerance.

We are building on the capabilities of a notation such as MetaH and expanding its expressive power to capture application characteristics common in avionics applications that reflect application semantics, including continuous processing of data streams. We proceed by first summarizing the capability of describing the component interconnection structure of applications, and then elaborating on the port interface specification to model data stream properties, followed by predicate-based constraints on the flow of data, and closing with constraints on component properties.

5.3.1 Modeling Component Interconnection Structures

The capability to model component interconnection structures is modeled after MetaH. A component consists of a component interface specification and a component implementation description. The interface specification describes how the component interfaces with other components—expressed in terms of ports and shared objects. An implementation description indicates what source code files make up the implementation, and how a component is composed of other components (component instances). We have several component types, as follows:

- *Process:* a schedulable unit of execution (task)
- *Macro:* a collection of interconnected processes, i.e., a composite component
- *Mode:* a collection of interconnected processes; only one of multiple modes can be active at a time, i.e., modes represent process (task) configurations

- *Subprogram:* a subunit of a process

In addition to these software component types, we have hardware component types such as *processor*.

Processes can be *periodic* or *aperiodic*. Periodic processes execute at a specified period, with a given deadline, maximum execution time, and possibly an offset start time. They will operate on input that is available at the time of dispatch. Execution of aperiodic processes is triggered by an external event. Processes have input and output ports, i.e., interaction is directional. They can be data ports or event ports. Data ports specify the type of data being communicated through a port by referring to an abstract data type in the source code of the component implementation. Event ports specify control events. Control events can be raised by a component and they can be accepted by an aperiodic component. An example is illustrated as follows:

```
port type sensor_data   : source_file = "Port_types.a";
port type actuator_data : source_file = "Port_types.a";

periodic process controller
   in port position: sensor_data;
   out port feedback: actuator_data;
end controller;

periodic process implementation controller.impl is
   attributes
      source_file = "controller.a";
      execution_time = 4 ms;
      period = 20 ms;
         end controller.impl;
```

An application is a collection of interconnected component instances. The interconnections are port-to-port connections from the output port of one component instance to the input port of another component instance. The data port connections can be delayed or undelayed. In delayed connections, the receiving component receives the data available at the time of dispatch, i.e., the data currently produced by the sender is not available until the next dispatch. For periodic processes this means that data is passed at period (frame) boundaries. In undelayed connections, the receiving component waits for the sender to complete, if the sender is active, to receive the latest data, i.e., data is processed as soon as it is available. This represents midframe communication and precedence ordering between processes. Figure 10 illustrates a simple two-process application. It utilizes the process specification for a controller identified earlier, as well as an assumed specification for a device.

The analysis of such a system model will determine the following:

- *Syntactic* correctness of the model: only ports connected to ports (a port is defined in the interface specification rather than the implementation) etc.
- Satisfaction of *connection constraints*: connection constraints include direction of a connection from an output port to an input port, matching of the data type communicated through the port, and limit on

the number of connections on a port (multiple outgoing connections and a single incoming connection).
- S*chedulability*: given a binding of the application onto hardware components, scheduling analysis can be performed utilizing the specified timing properties.

Macros and modes are composite components, i.e., their implementations consist of component instances and connections as illustrated for an application. In addition, a composite implementation specifies which input (output) ports of component instances are bound to the input (output) ports in the interface specification. The *binding constraints* are similar to the connection constraints, differing only in that input ports are bound to input ports and output ports to output ports.

5.3.2 Refined Input/Output Port Specifications

In this section we further define the port specification. Notice that an input port specification represents expected or *require*d constraints on data, and an output port specification represents assertions on the *provide*d data. We enhance the port specification in the following four ways:
- Refinement of the data type with measurement unit and coordinate system (measurement reference point)
- Range restrictions on the data values
- Rate of the elements in a data stream
- First order range restriction on the data values, i.e., limit on the delta between two successive data values

In terms of connection and binding constraints this means that measurement unit, coordinate system, and rate must match between the source and destination of a connection/binding. Notice that the default constraint for port rates is that data is processed at the rate it is supplied. Designers may choose to over- or under-sample a data stream. This intention may be documented as a constraint on the port rates of a particular connection, as shown in the following code:

```
connections c1.feedback -> d1.input sampled 2:1;
```

For ranges and deltas it must be the case that the source range and delta must be contained in the destination range and delta, as shown in the following:

$$range(source) \subseteq range(destination)$$

and

$$delta(source) \subseteq delta(destination)$$

These conditions illustrate a refined specification. The controller expects position data about a cart on a track in centimeters, relative to one track end, and at a rate of 20 milliseconds. The output to the device in units of volt at the specified rate. The controller also expects setpoints as input at one tenth of the position/feedback rate. Furthermore, it consecutive setpoints to differ no more than 10 cm, which is the maximum step size the controller can handle at the given rate. Notice that the data stream rates are expected to be a multiple of the component execution period specified in the component implementation.

```
periodic process controller
   in port position: sensor_data <cm, trackend>
      range [0 : track.length ] every 20 ms;
   in port target: set_point <cm, trackcenter>
      range [ 0 : track.length ] every 200 ms delta [ +- 
max_step_size ];
   out port feedback: actuator_data <volt>
      range [ +- 4.5 ]every 20 ms;
   attributes max_step_size = 10;
end controller;
```

The measurement unit and coordinate system can possibly be encoded in the user-defined data types. For example, a class representing the actuator data (actuator_data) may have several subclasses one of each unit of measure, e.g., actuator_data_volts and actuator_data_millivolts. In this case the type that matches the constraint enforces that the same measurement unit is assumed. This, however, may result in a cluttered class (type) hierarchy. As an alternative, we permit the user to specify measurement unit, coordinate system and data type. This can be done for every element (field) in a composite type (record). This approach allows for independent specification of components and delegates checking to the model analysis. A third alternative is to specify this information in the port type declaration. In this case, all instances of this port type satisfy these specification constraints. However, if components are developed independently, their port type specification sets must be merged to assure consistency between assumed data characteristics, an activity that corresponds to merging data dictionaries.

5.3.3 Assumptions and Assertions as Propositional Predicates

We introduce the propositional predicate notation in a manner similar to Perry [34] to capture additional assumptions and assertions about the data communicated via ports. Predicates represent preconditions, postconditions and obligations. Preconditions reflect assumptions about the predecessors along data streams, while obligations are assumptions about the successors. Postconditions are assertions made by components. An example propositional precondition is that the data has been filtered, while an example obligation is that a setpoint will be reached – as illustrated in the following:

```
periodic process controller
   in port position: sensor_data ...;
   in port target: set_point ...;
   out port feedback: actuator_data ...;
   pre position.sensor_data_filtered;
   post setpoint_reached;
end controller;
periodic process supervisor
   out port target: set_point ...;
   obl target.setpoint_reached;
```

Notice that propositional predicates are in essence labels that reflect a certain data characteristic, i.e., a certain type of processing performed on the data.

To determine whether a precondition (obligation) is satisfied, we check whether the direct predecessor (successor) can provide the matching postcondition. If the names match, the predicate is satisfied. If the predicate is in negated form and the names match, the predicate is not satisfied. Otherwise, we continue the search by checking their predecessors (successors). If we exhaust all of the predecessors (successors) in terms of connections and we processed a composite component, we check whether the predicate is declared in the component specification. This means that the precondition or obligation is to be satisfied externally, a fact that is checked for every instance of this component. Notice that for each postcondition in a component specification there must exist a component in the composite implementation in the chain of predecessors, starting with the output port which is bound to the specification without an intervening negated version of the predicate.

For any given predecessor (successor), its predecessors (successors) are determined as follows:

(1) by considering all input (output) ports as predecessors (successors)
(2) by taking into account data flow between input ports and output ports, which may result in a subset of (1). This data flow may be

- provided through explicit user specification of flow from input ports to output ports, e.g., ***paths*** *position -> feedback* for the controller in
- derived for simple components by program slicing [31] analysis of the source code
- derived for composite components by deriving the flow from their component dependency graph.

(3) by considering an explicitly declared path that represents a data stream through multiple components. Users can declare the flow of a data stream by explicitly listing a sequence of component instances even though the data representation, as declared in the port specifications, may change. For example, ***paths*** *control_flow: device1.sensor -> control1.position -> control1.feedback -> device1.actuator.*

In cases (1) and (2) satisfaction of a precondition or obligation by any one candidate is considered acceptable. In these cases a predicate is being considered as satisfied although it may not be by the intended path. A refinement by use of a path declaration (case (3)) remedies unintended and incorrect predicate satisfaction. In other words, the more precise a specification the user provides, the better the analysis in detecting inconsistencies.

Propositional predicates provide a quick and easy way to document assumptions and assertions. However, users must be aware that the analysis capability does not understand relationships between different labels. For example, sensor data may be raw, high band filtered, or low band filtered. If a component specifies as an assumption that it expects input not to be raw, this precondition (***not*** *raw*) is not recognized as satisfied by a postcondition stating that the data has been high band filtered (***post*** *HB_filtered*). The next section introduces property constraints that overcome this shortcoming.

5.3.4 Component Properties and Property Constraints

We can associate properties with components. Properties have constant values, including values of user-defined enumerated types. Specific property values may apply to all component instances, i.e., values are specified in the component interface specification, or each instance may have a separate property value, specified separately with each instance. The example of the shortcoming of propositional predicates can be modeled similarly to coding shown as follows:

```
Property type data_processing = enum {raw, HB_filtered, LB_filtered};
periodic process device is
   out port sensor: sensor_data ...;
   attributes processed : data_processing = raw;
end device;
periodic process controller is
   in port position: sensor_data ...;
   constraint position.processed != raw;
end controller;
```

The device has a declared property called *processed*, which is of the enumerated type data_processing and has the value raw. Enumerations are considered ordered, meaning that values can be compared for

equality as well as ordering. Property constraints specify limitations on properties. In our example, a constraint is specified on the property of a connected component, i.e., a component connected via the indicated port. For input ports, this corresponds to preconditions, and for output ports, to obligations.

Property constraints can be used in other ways to document assumptions made about the properties of components.

1. A property value may be used in a range or delta constraint specification. Two examples are located in *Track.length,* referring to the property of another component and in max_step_size, referring to a property within the same component.

   A composite component may make assumptions about the relationship of properties of several of its component instances. For example, the application in

2. may make the assumption that both component instances have the same period, i.e.,

   ```
   constraint c1.period == d1.period.
   ```

3. Component instance properties in the implementation may be constrained relative to a property of the component itself. For example, the sum of execution time properties of the implementation parts are to be less than the execution time property specified for the component. In order to support the cumulation of property values along a path, we have introduced a cumulation construct, which can be used as part of a constraint. For example, phase delay resulting from a flow path through a component implementation is expected to be less than the phase delay specified for the component, i.e.,

   ```
   constraint sum(c.phase_delay: c in controlflow.components) <
   self.phase_delay;
   ```

Properties can be used to describe key semantic characteristics of components, such as control components, without requiring a full set of control equations to be part of the model. Relevant property values are derived by control engineers and reevaluated as components change. By documenting relevant properties and assumptions that other components make about them, we can determine potential inconsistencies, i.e., we can determine otherwise hidden side effects through model analysis at design time [35].

5.3.5 Incremental Inconsistency Analysis

In summary, design time analysis of a system model can identify the following types of inconsistencies:

- *Syntactic inconsistency:* syntax and topological constraints are violated.
- *Connection and binding inconsistency:* connection and binding constraints are violated.
- *Resource inconsistency:* if the demand for resources exceeds available resources. In particular for real-time applications, schedulability analysis based on particular scheduling approaches, e.g., GRMA [33], can determine whether a particular task configuration is schedulable.
- S*emantic inconsistency:* violation of assumptions expressed as propositional predicates or property constraints.
- *Timing-related properties*: period of components and periods of ports; all rates on ports; predefined timing-related property constraints such as execution time; intentional over- and under-sampling.

A system may have a hierarchical structure, which is modeled with composite and simple components. As discussed earlier, each composite component has an interface specification that includes properties, assumptions, and constraints. Composite component implementations are described in terms of

interconnected instances of other components, while simple component implementations refer to the source code. For each simple component, the analysis tool checks for consistency between the port type definition in the model and in the source code. Other constraints and assumptions must be validated by the developer, since much of that information is not directly recorded in the source code. For each composite component, the analysis tool checks for syntactic, connection and binding, and semantic inconsistency of the interconnected component instances making up the implementation according to their interface specification. Separation of interface specification and implementation allows us to check for inconsistency incrementally, i.e., one component at a time. Only resource inconsistency analysis transcends this interface/implementation hierarchy. All components sharing a resource must be taken into account. However, many resource analyses can be performed considering one resource at a time. For example, GRMA-based schedulability analysis of tasks on a simple processor without any precedence constraints can be analyzed one processor and channel at a time.

5.4 Managing Configuration Inconsistency

In Simplex-based systems analytically redundant components may reconfigure themselves, if constraints that they monitor are violated. The result of this variant change may be a new logical configuration, which may be known to be inconsistent according to the inconsistency analysis based on the system model. For example, two controller variants may have different control characteristics, e.g., they may have different maximum step size or speed. A supervisory component may have a variant that assumes the higher speed controller. A change from the higher speed controller variant to a slower speed variant will cause further ARC constraint violations to be observed and variants to be switched, if the supervisory component variant with the higher speed assumption remains active.

Through design time analysis we can determine which of all possible logical configurations are inconsistent [36]. From this analysis we can derive configuration constraints, i.e., determine what are acceptable and unacceptable combinations of ARC component variants. By monitoring violations of these configuration constraints at run time, we can proactively reconfigure the system to reestablish consistent logical configurations. Proactive reconfiguration avoids snowball effects and maintains the overall system performance at a higher level than would otherwise occur.

The results of this analysis can also be used to support changes in system configurations that operators request. Operators may be presented with choices that include only consistent configurations. Furthermore, when given a desired configuration, we can determine the most appropriate way to reconfigure the system.

In this section, we illustrate how ARC-based components are described, discuss how these configuration constraints are determined through analysis of the system model, and describe what it takes to monitor these configuration constraints at run time and to support operator reconfiguration requests.

5.4.1 ARC-Based Component Modeling

ARC-based components appear to other components as regular components. The interface specification indicates input and output ports. An ARC-based component implementation consists of component instances that represent the safety, baseline, and upgrade variants. The input and output ports of these variants are bound to the ports in the specification. The constraints associated with the ports in each variant may differ from each other. For example, the acceptable range of values for setpoints may be larger for the upgrade variant than the baseline variant resulting in a higher speed characteristic. Notice in the example code that follows that a variant may service only a subset of the ports.

```
periodic process recovery_control is
   in port position: ...;
   out port actuation: ...;
end recovery;
periodic process normal_control is
   in port position: ...;
   in port setpoint:... delta [ +- 15 ];
   out port actuation: ...;
end controller;
periodic process high_speed_control is
   in port position: ...;
   in port setpoint:... delta [ +- 25 ];
   out port actuation: ...;
end controller;
periodic process controller is
   in port position: ...;
   in port setpoint: ...;
   out port actuation: ...;
end controller;
periodic process implementation controller.impl is ARC
   safety recovery_control;
   baseline normal_control;
   upgrade high_speed_control;
end controller.impl;
```

If the safety variant is the leader, the setpoint port is not serviced. As a result, we may have an inconsistent configuration if the connected component or component variant expects setpoints to be processed, i.e., its output port is connected. Similarly, a switch from the upgrade to the baseline variant may result in an inconsistent configuration if the supplied setpoints fall outside the smaller delta range of the baseline variant.

An ARC-based component is instantiated and connected to other components in the usual way, as illustrated in Figure 24.

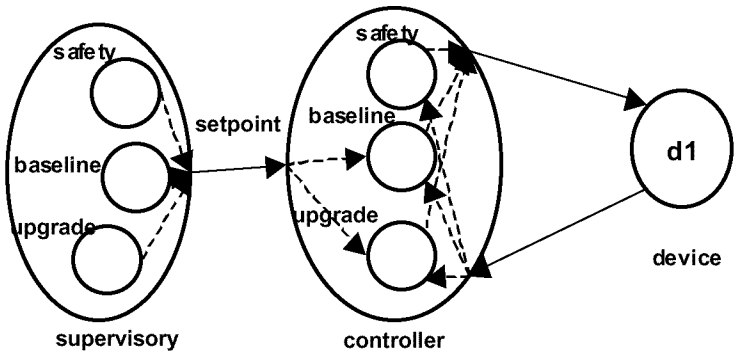


Figure 24. Interconnected ARC-Based Components

Only the solid connections have to be explicitly declared. The dashed lines represent the binding between the ARC-based component ports and the respective variant ports. They are inferred from the port names. The interconnection topology of these instances determines how component variants communicate with other components. From this topology and the port bindings, we derive logical configurations and their topologies. These system models are subjected to the consistency analysis described in the previous section to identify inconsistent configurations.

5.4.2 Configuration Constraints

Beladi [37] and Hiltunen [38] have shown that configuration constraints can be used to identify inconsistent configurations. A configuration constraint indicates whether a certain combination of component variants is acceptable or unacceptable. For example, in a system of three ARC-based components (A, B, C), the configuration `{A.baseline, B.baseline, C.baseline}` is acceptable, while the configuration `{A.baseline, B.safety, C.baseline}` is unacceptable. We may also know that the combination `{A.baseline, B.safety}` is unacceptable for all variants of C. This knowledge of acceptable or unacceptable configurations may be supplied by the users based on their experience with and observations of the system. In this case, users can explicitly record these configuration constraints in the system model. They can do so for an application and for each of its subsystems that are represented by composite component implementations. One of the constraints in the acceptable constraint list must be satisfied, while none of the constraints in the unacceptable list can be satisfied. In other words, **`Acceptable`** `C11, C12, C13; Unacceptable C21, C22;` with C_{ij} representing constraints as shown, implying $(C_{11}$ **`or`** C_{12} **`or`** $C_{13})$ **and not** $(C_{21}$ **`or`** $C_{22})$.

Configuration constraints can also be derived from inconsistency analysis. A naïve approach is to simply record all configurations for which inconsistencies are determined as unacceptable. The cause of the inconsistency can be taken into account when recording the constraint. For example, the connection between components A and B is inconsistent for variant A.baseline and B.upgrade. This inconsistency is due to a violation of a range constraint and applies to all configurations that contain `{A.baseline, B.upgrade}`. Thus, the analysis tool can capture the root cause of this inconsistency as a derived **unacceptable** `{A.baseline, B.upgrade}` rather than recording all possible configurations containing this combination. Similarly, property constraints on composite component implementations may involve a subset of the component instances. In this case, only this subset can be considered in identifying a configuration constraint. Furthermore, the set of already declared and derived configuration constraints can be applied to determine whether a full logical configuration can be skipped in deriving a configuration constraint from consistency analysis, since they are already known to violate a constraint.

This reduces the number of logical configuration permutations that must be considered for a full consistency analysis.

5.4.3 Run Time Recovery from Configuration Inconsistency

When a component implemented as an ARC reconfigures itself, we know that we potentially have an inconsistent configuration if the new variant is part of at least one unacceptable variant constraint. The variant constraints, which contain this variant, list the other components that may be affected. If the active variant of any of those other components corresponds to the variant in the constraint, then it will have to be reconfigured. Their reconfiguration may trigger other configuration constraint violations, and this process is repeated until the transitive closure is reached, i.e., when all inconsistencies have been eliminated. Notice that the number of iterations through this process is bounded and can be determined at design time.

Proactive recovery from inconsistent configurations can be implemented through central agent, or it can be distributed across the ARC run time support mechanism of the affected components. Let us examine each in turn.

In the case of a central agent, all ARCs that are involved in any configuration constraint must report a change in their variant state, i.e., a change in active variant, to the agent. The agent then consults the variant constraints to determine whether and which components have to be asked to reconfigure themselves to reestablish consistency. The variant constraints can be organized into a lookup table for the agent to quickly determine the transitive closure of affected components, i.e., the *change set*. Based on this change set, the agent requests the respective components to reconfigure themselves. This central agent solution requires variant status connections between all components and the agent, as well as the ability of the agent to cause variant reconfiguration in any component. Furthermore, the agent must maintain a consistent view of the current system configuration despite the fact that individual components may reconfigure themselves.

The distributed solution takes the following approach. From the set of variant constraints we can determine at design time which components need to be aware of changes in the variant state of another component. In other words, we can determine the necessary connections to communicate changes in the variant state. Potentially affected components will check the incoming variant state against their own according to the variant constraints relevant to them and reconfigure themselves as necessary. This component reconfiguration may itself cause other components to be affected, i.e., propagate through the transitive closure. In essence we have extended the responsibility of the MDU in an ARC to monitor not only safety region constraints and performance constraints, but also configuration constraints. This is the solution that has been chosen in multicomponent Simplex prototypes (the coordinated pendulum).

If the variant state is communicated periodically at the rate that the components operate and communicate application data, the distributed solution will have the effect of reaching a consistent configuration that may take more than one period. Fortunately, in the domain of control systems there is enough lag [39] in the system to tolerate temporarily inconsistent configurations, which permits the incremental execution of reconfiguration steps within time bounds.

If this iterative solution is not acceptable in a particular system, the transitive closure can be translated into additional variant state communication connections and additional conditions to be checked by all components affected by a reconfiguration.

Notice that many control applications have continuous feedback controllers and supervisory controllers. Supervisory controllers are responsible for operating the feedback controllers appropriately, and can thus take the responsibility for managing the inconsistency that results from a component reconfiguration due to a fault. Thus, we can limit explicit variant constraint checking status of peers and supervised components. In the coordinated pendulum prototype [40], the coordinator will adapt to reconfigurations in the feedback controllers, while the feedback controllers are assumed to be ignorant of reconfigurations in the supervisory controller. The coordinator monitors the variant state of both pendulum controllers. If either of the pendulum controllers is executing the recovery variant, the coordinator switches itself to a control mode (variant) where the second pendulum controller is following the other in order to avoid misalignment. Similarly, the coordinator monitors whether either of the pendulum controllers switches from an upgrade variant to a baseline variant. In this case, it switches to a control mode (variant) that operates the pendulum controllers at the slower baseline speed.

5.4.4 Managing Reconfiguration Requests

Consistency of reconfigurations requested by operators can be assured by limiting the operator's choices to configurations identified as consistent by design time analysis. Reconfiguration from the current configuration to the desired configuration may involve reconfiguring a number of components. Individual components with multiple variants may not be able to immediately switch to a requested variant. For example, the controlled plant must be brought within the operating region before control can be given to the desired variant. Thus, an operator requested reconfiguration may not be immediately achievable.

It is desirable to reduce the risk of staying in inconsistent configurations for longer periods of time during an operator-requested reconfiguration. We can do so by determining whether the desired configuration can be reached in multiple steps by reconfiguring to intermediate consistent configurations that have steps that require a smaller number of components to be reconfigured. We can view the set of consistent configurations as nodes in a graph. The edges between the nodes represent the change set, i.e., the components that need to be reconfigured to change from one configuration to another. These edges are weighted by the size of the change set.

Given two nodes in the graph, i.e., the current configuration and the target configuration, the goal is to identify a path whose edges have minimal weight. In addition, the path should be as short as possible, i.e., only those edges and change sets that reduce the number of components to be reconfigured should be considered.

To perform this analysis effectively, we start with a fully connected graph. Through design time analysis, we reduce this graph to a minimal graph by eliminating edges if a path exists with lower weighted edges. This minimal graph can then be used at run time to identify a path between the current and the target configuration with the minimum number of component reconfigurations per reconfiguration step. More than one path with minimally weighted edges may exist. For example, reconfiguration from the current configuration to the target configuration may involve five components. Two minimum weighted paths may exist, one involving two reconfiguration steps that involve two components, and one step involving one component. The other path consists of one step that involves two components and three steps involving one component. In this case we have the choice of minimizing risk by choosing the path with the smaller number of higher weighted edges, or minimizing the number of intermediate steps.

5.5 Analyzing the Impact of Change

The model also supports impact analysis, i.e., "identifying the potential consequence of a change, or estimating what needs to be modified to accomplish a change" [41]. Impact analysis can occur in two

forms: identification of all potentially affected components given an intention to change a component; and determination of actually affected components given a completed change and the incremental propagation of this change as affected components get corrected. The effectiveness of impact analysis in identifying affected components is strongly influenced by the granularity of a change. Let us illustrate with some examples.

A change to a component interface specification impacts all composite components that contain instances of this component. We can further identify which portions of those composite component implementations are affected by utilizing the component connection graph. Potentially, all direct and indirect predecessors and successors of the component instance whose interface changed are affected. Similarly, the interface specification of the component may be potentially affected. Furthermore, component instances that are part of a property constraint that refers to an affected instance may be potentially affected. Any of the potentially affected components may require modification, resulting in propagation of the change.

A change to a component implementation potentially affects the component interface specification. The change may result in different resource requirements, i.e., for all instances of this implementation, component instances sharing resources are potentially affected. Again, any of the potentially affected components may require modification, resulting in propagation of the change.

The actual impact of a change to an interface specification is determined by considering the potentially affected components as candidates and reevaluating the constraints associated with them against the changed component. Given the changed interface specification, all composite component implementations containing instances of this specification are reanalyzed for inconsistency. This involves reevaluating the following:

- connection and binding constraints associated with the instance to determine whether the connected component or the composite specification is actually affected
- satisfaction of preconditions and obligations associated with the instance to determine whether the composite implementation becomes inconsistent
- preconditions and obligations of all direct and indirect predecessors and successors that refer to the instance's postconditions
- property constraints that include an instance of the changed interface.

The results of the reevaluations determine whether the potential candidates are actually affected.

The actual impact of a change to a component implementation is determined by reevaluating the component implementation itself and by validating it against its interface. An inconsistency between the interface and the implementation may be resolved by a change to the implementation or to the interface. The latter causes incremental propagation of the change. In addition, schedulability analysis (and other resource constraint analysis) of resources that include an instance of the changed interface must be performed again.

We may have more knowledge of the change, e.g., that it is a change to a specific port or some specific characteristic of a port, a change in property values or constraints, or a change to a component instance or a connection. Let us examine in more detail the changes to ports and to properties.

A change to a port can occur in a component interface specification. Potentially affected components are those connected to the modified port for each instance of the component, if the change involves characteristics relative to connection constraints. Their connected port is considered potentially affected,

which in turn may affect that component's implementation as well as other ports related through data flow. If the change is to predicates then earlier stated rules for predicate, reevaluation identify potential candidates.

A change to a property value may be specific to one instance or may affect all instances of a component specification. A property value is potentially used in range and delta constraints or in property constraints. In the former case, the respective port characteristic is affected by the change and propagated accordingly. In the latter case, the properties of all component instances involved in a property constraint are potentially affected and handled accordingly. Changes to resource properties, such as timing, potentially affect timing properties of other components sharing a resource. The relationship between port rates and component periods represents an implicit property dependency that must be taken into account. We also support explicit declaration of property dependencies. Such declarations allow developers to capture the essence of relationships between properties that may exist and are embedded in detailed design and analysis models, and not lose sight of those facts. For example, the maximum step size acceptable to a controller component is dependent on the period at which it is processed. The details of the relationship are embedded in control equations, but the fact that one property affects the other is captured in the model and can be utilized during impact analysis. The dependency can be denoted as "→," which can be interpreted as "depends on" or "is a function of." Changing the right side property potentially affects the left side property. One property can be dependent on multiple properties. Properties may affect each other, expressed as "↔." An annotation to this dependency can provide a reference to the detailed design model to be reevaluated to determine the actual impact.

In summary, the following relationships must be taken into account when determining the impact of a change (see also [42]):

- interface-instance relationship: all instances of a component with interface modifications are affected
- interface-implementation relationship: all implementations of a component are affected by changes in the interface specification
- part-of relationship: ports and properties are part of components, and component instances are part of composite component implementations
- connection relationship: component instances may be affected through interaction
- binding relationship: an instance port in an implementation may affect the component interface and vice versa
- resource relationship: components sharing resources affect each other with respect to relevant properties
- property definition-use relationship: property values are referred to by range/delta specifications, property constraints, and property dependencies
- property constraint and dependency: properties affect each other and the components with which they are associated.

5.5.1 Reducing the Impact of Change

The number of impacted components can be reduced in two ways: reduction of dependencies, and reduction of assumptions, i.e., relaxation of constraints. Reduction of dependencies requires restructuring of the system by introducing new abstractions and localizing information. The result is a simplified system interconnection structure. Reduction of assumptions results in fewer constraints that can be violated as a result of a change. Relaxation of a constraint means that it is easier to satisfy the constraint, i.e., more variation is acceptable. Reduction and relaxation of constraints has the effect of an increased propagation barrier in that there is less chance that a change will affect other components. Let us illustrate how reduction and relaxation of constraints can be achieved in practice.

A component may be redesigned to be more flexible, i.e., fewer assumptions are made about the interaction with other components. For example, the precondition *HB_filtered* may be removed if a controller implementation has been modified to handle unfiltered data. Similarly, the obligation of a setpoint being reached may be removed, if the supervisory component has been redesigned to be closed loop, i.e., the component takes into account feedback and adjusts its output accordingly. A component may have been modified to accept data that has been augmented with metadata such as the measurement unit used. As a result, it does not make assumptions about the measurement unit. A component design may also be examined or redesigned to accept more variability in data or property values. For example, a controller may be able to maintain stability within a range of sampling rates [43]. This can be reflected in the model by specifying a range for the input arrival rate. Similarly, sensitivity analysis of a set of tasks for schedulability [44] may determine that the schedulability constraint remains satisfied within a range of execution time values for components sharing the resource.

The concept of compatibility can also contribute to reducing the impact of change. Variants of a component can be compatible, i.e., they are interchangeable with respect to their interaction with other components. The concept of compatibility has been investigated and formalized previously [45, 46, 47]. In our context, two types of compatibility are of interest: strict compatibility, and upward compatibility. Two variants are considered strictly compatible if other components are dependent only on an interface specification that is satisfied by both variants. This means that replacement (either upgrade to reflect a change or run time switching within an ARC) of one variant or the other in either direction has no detectable side effects. A variant is considered upward compatible with a second variant if the first variant can replace the second and all specified dependencies by connected components continue to be satisfied, i.e., a reconfiguration from the first to the second variant has no side effects. This does not hold for the inverse.

In our context these conditions can be expressed as follows. Variant B is upward compatible with variant A if

$$elem(A.in) \supseteq elem(B.in) \wedge range(A.in) \subseteq range(B.in) \wedge elem(A.out) \subseteq elem(B.out) \wedge$$
$$range(A.out) \supseteq range(B.out) \wedge pre(A) \supseteq pre(B) \wedge post(A) \subseteq post(B) \wedge obl(A) \subseteq obl(B).$$

Variants are strictly compatible if they satisfy a common interface C, i.e., an interface that other components are matched up with, ($x(A/B) \subseteq x(C)$ being a shorthand for $x(A) \subseteq x(C) \wedge x(B) \subseteq x(C)$:

$$elem(A/B.in) \subseteq elem(C.in) \wedge range(A/B.in) \supseteq range(C.in) \wedge elem(A/B.out) \supseteq elem(C.out) \wedge$$
$$range(A/B.out) \subseteq range(C.out) \wedge pre(A/B) \subseteq pre(C) \wedge post(A/B) \supseteq post(C) \wedge obl(A/B) \subseteq obl(C)$$

In practice, compatibility between two component variants or implementations means that other components make fewer assumptions about a component than is reflected in the interface specification. This means that declaring fewer component properties and assumptions as externally visible can reduce the interface specification.

5.6 The Analysis Tool

Our prototype implementation of an analysis tool demonstrating the above capabilities is based on the Java-based toolkit for Acme [48]/Armani [49], an ADL interchange notation and constraint engine. Our notation MetaS (MetaH with extensions) is mapped into an Acme style and into Armani invariants, heuristics, design analyses, and external Java functions. AcmeStudio provides a graphical front end for the Acme-based representation of our notation.

Application models can be developed and maintained in three ways: in MetaS, in Armani/MetaS, and in AcmeStudio. In the first two cases, application models are edited in textual form. MetaS descriptions are then translated into Armani/MetaS. Armani/MetaS descriptions are processed by the Armani toolkit, based implementation of our analysis tool. AcmeStudio provides a graphical front end for developing Armani/MetaS descriptions. AcmeStudio stores these descriptions in textual Armani/MetaS form. The analysis capability can be directly invoked from AcmeStudio and the results of the analysis reported back within AcmeStudio.

5.7 Summary

In this section, we have described a design time analysis approach for avoiding faults to complement the fault tolerance capability of INSERT. Many residual faults are attributable to hidden side effects of changes that are due to assumptions made about component characteristics that are specific to the application semantics that are often time sensitive in nature. We have introduced a notation based on an ADL for real-time embedded systems to support modeling of avionics applications. This ADL has been extended with constructs to capture these additional properties and constraints. Such application models can be analyzed for several types of inconsistency. In the context of INSERT-based applications, the results of such inconsistency analysis can be used to derive configuration constraints. By monitoring such configuration constraints at run time and proactively recovering to consistent configurations, we can improve the overall system performance. Finally, the application model can be used to assist in identifying the impact of a change.

We have demonstrated the feasibility of such an analysis capability by the prototype implementation of an analysis tool. This prototype implementation is based on Acme/Armani EDCS technology.

6. An Application of INSERT Technology

6.1 Introduction

Developers of high assurance software systems face increasing challenges as their products evolve and increase in complexity. Costs must be controlled, but safe, predictable operation must be guaranteed throughout the product life cycle. This places great importance on the ability to confidently perform repeated incremental upgrades. Solutions to the incremental upgrade challenge must preserve and isolate "untouched" existing functionality, allow new capability additions, and provide safety backup coverage in the event of latent errors within newly added components.

Use of commercial components is seen as a significant avenue for reducing costs in DoD systems. As a result, the INSERT architecture (Figure 25) starts with a hardware platform based on commercially available CPU, memory, and I/O devices. On top of that, a commercial Real-Time

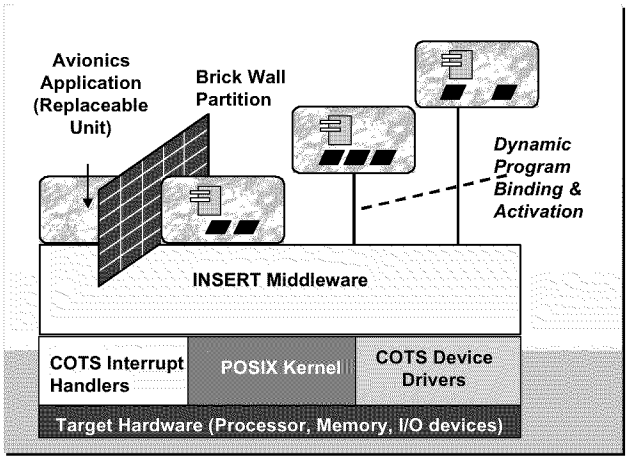


Figure 25. INSERT Architecture

Operating System (RTOS) runs along with COTS interrupt handlers and device drivers. The RTOS is based on the IEEE POSIX standard, which provides maximum portability between Unix-based operating systems as well as real-time extensions to provide deterministic computing.

The INSERT middleware is a set of architectural constructs that makes maximum use of the hardware and RTOS features to provide high assurance capabilities. The various avionics computer programs, known as replacement units in the INSERT system, operate on top of the INSERT middleware. The INSERT middleware insulates the applications from the underlying RTOS and hardware. The middleware also provides dynamic binding and activation of each replacement unit, provides run-time safety monitoring, prevents corruption of code and data in one program by another, and allows dynamic reconfiguration.

The avionics application software is partitioned into separable units, and with each Computer Software Configuration Item (CSCI) can be hosted in its own virtual memory partition. The CSCIs interact and communicate necessary information using asynchronous messages. To ensure that each of the independently scheduled, dynamic programs are able to meet necessary deadlines, the well established techniques of Rate Monotonic Scheduling [50, 51] and Rate Monotonic Analysis [52] are used. If a fault is detected, backup replacement units can be

activated to guarantee a specific level of performance. New capabilities can be safely integrated into the system while it is on-line.

6.2 Methods, Assumptions, and Procedures

The INSERT project required integration of the technology into an actual weapon system–in this experiment, the F-16 AFTIs avionics software. We used an F-16 AFTI simulator, complete with a functional cockpit that includes controls, instrumentation, and displays. The AFTI simulator employs the same General Avionics Computer (GAC) found in the aircraft. The original Operational Flight Program, or OFP, is hosted on a MIL-STD-1750A 16-bit processor, communicating via a MIL-STD-1553B data bus. The original OFP, coded in the JOVIAL programming language, was modified to remove pieces of the avionics algorithm and replace it with a series of bus controller commands to communicate with another remote terminal added to the 1553 bus. For this case the remote terminal was an Intel-based PC running the INSERT platform.

6.2.1 Hardware and Software Specifics

The re-hosted code implements the F-16/AFTI AMAS. The AMAS is a fully automated, hands-off weapon delivery algorithm that provides a stand-off capability using a curvilinear flight path approach to the target. The GAC determines the correct flight path and attack profile and can either provide steering cues to the pilot via the head up display or can be coupled directly to the aircraft flight control computer, allowing the aircraft to automatically fly three phases: Ingress, Non-wings Level Attack, and Egress.

The desktop PC has an Intel Pentium II 233 MHz processor and is configured with the Lynx real time operating system, Version 2.5.0. The PC was augmented with the System Timing Analysis Tool (STAT) hardware timer card by Alpha Logic Technologies. This allowed us to gain increased timer granularity (as small as 250 nanoseconds) as compared with the standard Intel and POSIX clock mechanisms. Communication with the AFTI Simulator was via a 1553B data bus device that is packaged in a Personal Computer Memory Card International Association (PCMCIA) card format from Excalibur Systems, Inc. The ported JOVIAL AMAS application code was translated to C/C++ and compiled using the GNU compiler suite.

The PC is connected using standard coaxial cable to the F-16 simulator's bus system. After translation into C/C++, the ported AMAS code was "wrapped" in the INSERT architecture to make use of the advanced capabilities previously discussed. In fact, the INSERT technology provided an easy skeleton of C/C++ source code into which to plug the application.

6.2.2 Risk Reduction Steps

The re-engineering of OFP and the integration of the PC with the 1750 proceeded in four stages:

1. The first was to remove the Egress part of the algorithm (the phase of flight following weapon release) from the OFP, translate it to C, and test it. This risk reduction step was performed because the Egress part of the algorithm has minimal complexity and requires minimal data transfer across the1553 bus. This phase also required development of a driver for the PC 1553 card. By limiting this first experiment's size we could ensure the timeliness of the PC and 1553 communication protocols with the simulator.
2. After successful integration, we wrapped the Egress algorithm in the Simplex code (written in C) provided by Carnegie Mellon University (CMU). The goal was to take advantage of the

small size of the algorithm to effectively evaluate the impact of the middleware's messaging and analytic redundant concepts on overall system timing.

3. The third step involved migrating the entire AMAS algorithm to the PC as a stand-alone application without the Simplex middleware. The goal was to simplify the evaluation and confirm that the timing constraints were being met and that the interface between the two computers was complete and correct.
4. The final integration experiment used the full AMAS with the Simplex architecture engaged. This phase actually required only a week of effort to complete, mostly due to the difficulties in timing and messaging being resolved by the first three risk reduction steps.

6.2.3 Basic INSERT Operation

INSERT's basic structure is the replacement unit. By using a name-tagged publisher/subscriber communication technique, INSERT can dynamically add or delete tasks from run time execution. These tagged communications are built upon IEEE POSIX message queues supported by LynxOS. In addition, LynxOS supports 256 different priority levels for process and threads.

This approach provides flexibility in dynamically activating or deactivating the processes as well as their communication links. The middleware is comprised of these data-tagged communication processes, a replacement manager task, a decision logic process, and a timer process to enable the STAT timer.

The INSERT PC runs three different algorithms that can control weapons delivery: the baseline, upgrade, and backup controllers. The baseline controller uses a reliable algorithm that can fly the

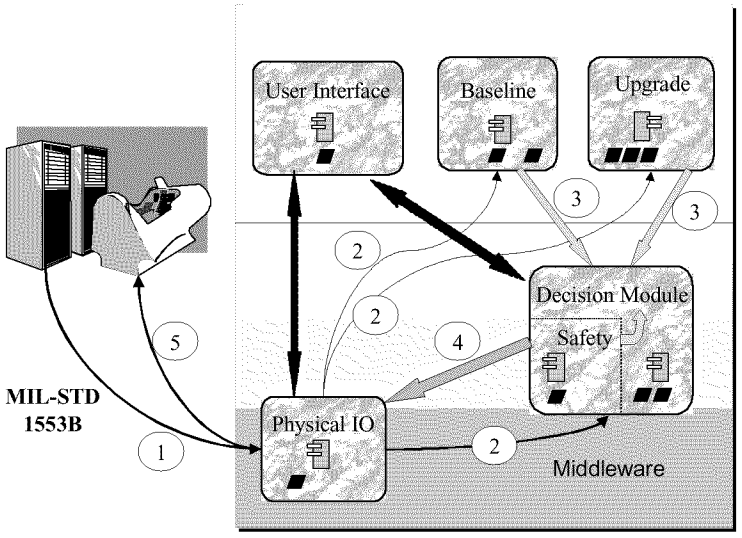


Figure 26. INSERT Messaging

aircraft through a weapon delivery that has previously undergone extensive testing to ensure a high degree of reliability. The upgrade controller implements an "improvement" to the baseline that the developers need to evaluate. Examples of improvements include increasing the accuracy of the algorithm, modifying it for a moving target, or changing the aircraft maneuvering

parameters. The backup, or safety controller, which is used only when severe errors are detected in both the baseline and upgrade controllers, discontinues the attack and transitions the aircraft into a terrain following ground collision avoidance mode. As a result of multiple versions coupled with monitoring and switching infrastructure, INSERT provides a guaranteed level of system performance at all times. Residual errors existing within experimental components are prevented from propagating and the system recovers in real-time by invoking a trusted backup component.

The data flow is shown in Figure 26 with the sequence numbers indicating the order of messages. The physical I/O task runs a timer thread that will generate a message to a queue that is read by the physical I/O task. Upon receipt of 1553 data, the timer thread sends a message to the physical I/O task which then retrieves data from the 1553 card.

Table 11. Process Priorities

Process/Thread	**Priority**
RTOS terminal and XWindows	<= 17
Minimum priority for INSERT application (PRI_MIN)	20
1553 data arrival thread	PRI_MIN
User interface initialization	PRI_MIN+30
User interface real-time	PRI_MIN
Console handler	PRI_MIN
Command and control handler	PRI_MIN
Physical I/O	PRI_MIN+25
Decision module	PRI_MIN+20
Baseline controller	PRI_MIN+15
Upgrade controller	PRI_MIN+10

Process and thread control is established using the RTOS prioritization and scheduling services. The various priorities are shown in Table 11. The use of the RTOS scheduler and prioritization makes adding extra processes, more variants, and modified code far simpler than in the legacy F-16 cyclic executive.

The selection of the run time priorities is critical to the correct operation of the system.[53] The user interface initialization task is the highest but stays at this level only during the starting of the PC. Since the system is data-driven by 1553 messages received from the GAC, the physical I/O process has the highest priority but blocks until it receives the solution from the decision module. The decision module has the next highest priority because of the need for it to be able to awake after its midframe suspension. To ensure that a solution will be available should the upgrades experience hard failures, the baseline controller, which represents a more trusted process, has higher priority than any of the upgrade variants.

Figure 27 depicts the scheduling time line. The processes are scheduled in order of their priorities but are blocked or suspended depending on the availability of data in their message queues.

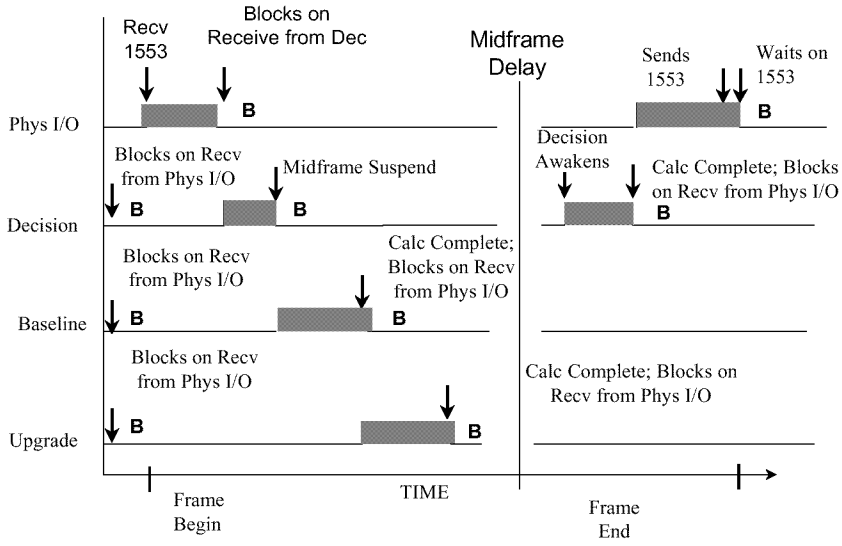


Figure 27. RTOS Scheduling

6.2.4 Modeling Practical Design Constraints

Object-oriented (OO) application design methods focus on "packaging or grouping" application data and computational procedures with the goal of achieving a maintainable lexical structure. There are many sets of notations and language semantics for OO representations in use currently. In general, OO techniques build on the concepts of information hiding and interface standardization. Often, OO techniques focus on developing class hierarchies of reusable units of behavior and data, referred to as Abstract Data Types (ADTs).

The real-time software industry has been quick to adopt OO methods, as they promise reduced life cycle costs and enhanced ability to upgrade fielded applications. While OO implementations often increase function call overheads, this problem is generally well understood and can be solved through better measuring, or rendered irrelevant by a faster processor.

6.2.4.1 Reliable and Cost-Effective Migration of Code (JOVIAL to C/C++)

In this phase of the INSERT research project, the F-16 AMAS algorithm was re-engineered from JOVIAL into a Unified Modeling Language (UML) design using the Rhapsody Tool from Ilogix, Inc. The C++ code was automatically generated and the resulting executable was hosted on the Intel based PC running the LynxOS RTOS. This variant is similar to the experiment described in para. 3.2.C.

We were originally tasked in year three to re-engineer the AMAS code into an OO format. We would then wrapper the MMC Software with Simplex and mesh the AMAS into this MMC architecture. Because of the extreme success of the year two demo, with the customer's concurrence, the year three objectives were modified: change the avionics target and look at reliable software modeling approaches for re-engineering the AMAS. The MMC conversion was dropped in favor of evaluating fault-tolerant approaches on the newer LM-Aero middleware: the Advanced Software Execution Platform (ASEP). The ASEP is built in C++ using the Rhapsody

Toolkit from Ilogix. The AMAS re-engineering task was performed using the UML and Rhapsody.

6.2.4.2 Manual Effort

As previously mentioned, JOVIAL to C conversion of the modules was manually performed in the first year with integration, test and execution on the F-16 AFTI Simulator. A restructuring of the C code into a format that would fit with Simplex was performed in the second year. The LM Aero - FTW JOVIAL programming standards called for developers to place one function within one individual file or module. The converted files were originally converted to C using a one-to-one file conversion process: for example "AGSUST.J73" became "agsust.c".

Table 12. AMAS Rehost to Final INSERT C Format

Task	Hours
Code translation (JOVIAL to C)	200
Redesign and test of C	500
Redesign, integration, debug on PC with INSERT	465
CMU/SEI INSERT integration support	60
TOTAL	1,225

This phase yielded a total of 45 C-code files and 13 header files. Initially, the COMPOOLS were treated as global variables and the code was compiled, integrated to the simulator and tested. These were then restructured into a more conventional C-like style. Subroutines (modules) were gathered into clusters of fewer C files according to logical grouping of functionality. Since the new code was now running on a distributed Intel PC, many of the COMPOOLs were formed into structures that would either facilitate the 1553 communication or be logically grouped depending on coupling and cohesion factors. In addition, with the reduction of the COMPOOLs the use of parameters to pass information between function calls necessitated a further structuring of data variables. Not all "global" variables were converted. Some were left as is due to the nature of the problem.

6.2.4.3 Auto Code from UML Tools

This part of the experiment was accomplished by tasking an individual knowledgeable in Rhapsody but not familiar with OFP development. He was given AFTI documents containing an overview of the AMAS algorithm and was tasked to produce an initial UML Rhapsody Model including an object diagram, sequence diagram, and state charts. After the initial model was designed, he was given access to the source code of the final "AMAS only" C version (the one that was wrapped into Simplex) and then added the methods and attributes to the classes. These steps were taken to "emulate" the idea of top level design without preknowledge of a solution. (While this author could have done this, my intimacy with the current C code version could potentially "bias" the OO design.)

After the engineer finished the modeling step, he and I reviewed the design, made some tailoring decisions, and revised the model. This step was necessary because the original UML model (and documents) assumed an AMAS running on a single processor. In actuality, with the distribution of the algorithm on the Intel PC, the break was not as clean as the OO analysis would predict. The AMAS in reality ran mostly on the PC, but parts of it still ran on the GAC. Even more of a nightmare was that the state machine logic resided mostly on the PC but some on the GAC. In reality either machine could trigger transitions and both had to stay in sync! Thus we had to

modify the original design to accommodate the existing legacy C code. The Object Model Diagram (OMD) is shown in Figure 28. (NOTE: It should be pointed out that a couple of the class blocks are placeholders at this time for future work on INSERT and are not shown with any associations.)

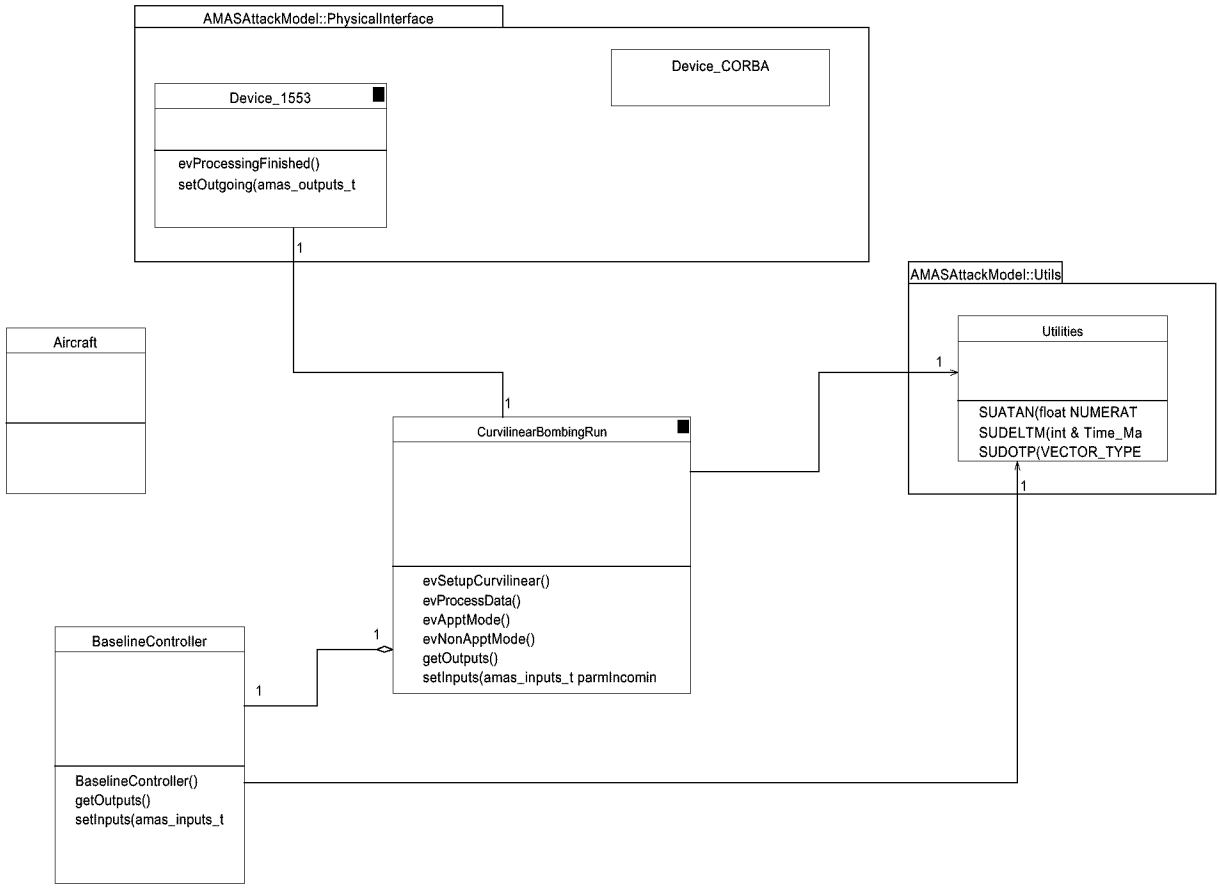


Figure 28. Object Model Diagram

Once the UML model was finalized, the process of moving the implementation code began. Most of this part of the re-engineering was handled by me because of my familiarity with the code and the lack of OFP experience on the part of the young engineer. The structures, constant declarations, attributes, and methods were created in the Rhapsody model using a text editor to reference the "legacy" C files. About 80 percent of the original C code could be cut and pasted directly into the Rhapsody model with the remainder, structures, and type definitions, for example, being created by hand within Rhapsody. In the conversion, functions became methods, variables became attributes, and so forth, along with defining the public/private access to all methods and attributes. Accessor and mutator functions were created within Rhapsody . Constructors were added and state machines using the Harel notation within Rhapsody were added to two of the classes.

Rhapsody does not support code generation to all RTOSs. However, they do provide a mechanism to extend the tool so that users can build the correct libraries for the target environment as well as generate the correct application files for the target. This setup task required 68 hours and was well documented and very straightforward.

The end result of this effort was a "code complete" UML model from which we could generate compilable code. In fact the code was moved to the LynxOS machine and compiled without error on first try. An additional 10 hours of integration and test were required to get the C++ version working.

6.2.4.4 The Experiment Using Automated Re-Engineering Capability

Within the same time frame as this effort, the AFRL/IFTA, together with LM-Aero was demonstrating a re-engineering capability under the Embedded Information Systems Re-engineering (EISR) project. The goal of EISR was to develop and demonstrate an automation-assisted re-engineering capability for transforming JOVIAL source code into C (Contract # F33615-97-D-1154-002). Xinotech Research, Inc., a vendor specializing in source code transformation tools, was subcontracted to LM-Aero to extend their existing product line by adding a JOVIAL-to-C conversion capability. A collaborative effort to repeat the experiment using the EISR capability was initiated.

The engineer responsible for evaluating the EISR capability generated the C code from the JOVIAL code, compiling it, and ensuring its syntactic accuracy. The EISR has between 92-98 percent coverage in correctly converting all code. The last few percent has to be hand massaged. This effort required 24 hours versus the several hundred hours required in the original manual effort.

A senior software developer with F-16 OFP and Rhapsody experience was tasked to repeat the experiment. The EISR-converted C code was provided to this engineer, along with the AMAS documents (the engineer had not worked AMAS previously).

As expected, the UML model was different in organizational pattern but still behaviorally equivalent. In this case, the engineer handled the entire conversion effort alone. Differences between the two experiments regarding individual hourly efforts are as follows. In the original, the time to create the Rhapsody model was for the first skeleton that this author reviewed. In addition, I was responsible for completing the move of the code and development of the model so that the task of "moving" the C code into the Rhapsody model included some "massaging" of the model. In the first experiment, my knowledge of LynxOS resulted in a smaller effort to compile the program. In the second, the engineer found some problems with the code in his model that required more time to correct during the compilation phase. Corrections were made within the model and Rhapsody re-generation of the code was accomplished so that the model always stayed in sync with the target.

6.3 Results and Discussions

6.3.1 Processor Utilization Performance of INSERT Middleware

6.3.1.1 Data Gathering Procedures

To analyze the performance of the middleware, the STAT card, accessed via a Unix "ioctl" device driver call, was configured to provide a timer granularity of 1 μs. Each call to the card provides the elapsed time since card activation. Logtime calls were made at points of interest in the various process modules.

Table 13. Event Message Logging

Event Time (seconds)	Source	Event Message	Calc	Phase of Flight
158.078499	PIO	Wait1553	Ingress	Nwl_atk
158.078529	PIO	Endlog	Ingress	Nwl_atk
158.092060	PIO	Pre_recv1553	Ingress	Nwl_atk
158.092284	PIO	Recv1553	Ingress	Nwl_atk
158.092322	PIO	Pre_msgsent	Ingress	Nwl_atk
158.092431	PIO	Msgsent	Ingress	Nwl_atk
158.092505	DEC	Physiorecv	Ingress	Nwl_atk
158.092538	DEC	Endlog	Ingress	Nwl_atk
158.092567	DEC	Beforemidframe	Ingress	Nwl_atk
158.092653	SIM	Physiorecv	Ingress	Nwl_atk
158.092728	SIM	Msgsent	Ingress	Nwl_atk
158.092760	SIM	Waitphysio	Ingress	Nwl_atk
158.092790	SIM	Endlog	Ingress	Nwl_atk
158.092875	UPG	Physiorecv	Ingress	Nwl_atk
158.092952	UPG	Msgsent	Ingress	Nwl_atk
158.092984	UPG	Waitphysio	Ingress	Nwl_atk
158.097979	DEC	Aftermidframe	Ingress	Nwl_atk
158.098112	DEC	Controllerrecv	Ingress	Nwl_atk
158.098193	DEC	Msgsent	Ingress	Nwl_atk
158.098232	DEC	Waitphysio	Ingress	Nwl_atk
158.098298	PIO	Msgrecv	Ingress	Nwl_atk
158.098331	PIO	Endlog	Ingress	Nwl_atk
158.098368	PIO	Pre_sent1553	Ingress	Nwl_atk
158.098467	PIO	Sent1553	Ingress	Nwl_atk
158.098499	PIO	Wait1553	Ingress	Nwl_atk

To minimize the impact of collecting this performance data, each logging call simply places a timestamp in a shared memory buffer that persists after the test run is complete. Each timestamp is a structure of binary data, 12 bytes long, containing 2 longs (time) and 4 chars: (event message source, event message, calculation state, and attack phase state).

The F-16 OFP uses a cyclic executive designed to run frames at 50 Hz with a frame time of 20 ms. It triggers the PC to run by sending data at the 50 Hz rate, thus running the PC in lockstep with the OFP. The information in Table 13 shows a one-frame event log.

The *Endlog* messages occur four times in each 20-ms frame. They are executed immediately after the previous log message. The time difference between the *Endlog* and previous timestamp represents the total time for logging a single event message, including hardware interrupt to the STAT card, context switches between OS tasks, etc.

The *Wait1553* message is printed at the top of the process loop in physical I/O, awaiting the data from the 1750. The *Pre_recv1553* is logged after the 1553 card signals arrival of the data but before the actual retrieval of the blocks of data, with the *Recv1553* being logged immediately after retrieval of the data. The delta provides time data on the speed of the Excalibur 1553 card.

The *Pre_msgsent* and *Msgsent* pair in physical I/O bracket the call to the publish routine in the publisher/subscriber message system. The delta between the *Msgsent* in physical I/O and the *Physiorecv* in decision module (DEC) represents the time for a context switch between the two processes and the time to pull the data out of the subscriber's message queue.

Beforemidframe signals the time of suspension of the DEC. The delta-time in SIM and UPG between *Physiorecv* and *Msgsent* reflects the time to compute the tracking and targeting solution for this frame. The time between UPG *Waitphysio* and *Aftermidframe* reflects the time in which DEC is still suspended and represents the "slack" or free time.

The interval from *Aftermidframe* to *Controllerrecv* in DEC is the time to retrieve all the messages out of the subscriber queues from baseline and upgrade. The interval between *Controllerrecv* to *Msgsent* represents the time to evaluate the safety, baseline, and upgrade controller data, make a selection of which controller to use, and publish the data to the physical I/O process.

The physical I/O receives the message and the difference between *Pre_sent1553* and *Sent1553* is the time for the 1553 card to output the data to its buffers.

In terms of statistical sampling rates, there were 750 frames of data captured in each of four test runs: baseline controller only, baseline controller with upgrade experiencing a fault (two runs), and baseline and upgrade controllers without fault. The data from the four samples were consistent. Tabular data for this report is from the last test run, with a few values from the baseline provided only for comparisons.

6.3.1.2 Data Analysis: Logging and Middleware Components

Several individual items were of interest, including the performance of the publisher/subscriber message queues and the decision controller. These time values are shown in Table 14. Before analyzing these it was necessary to determine just how much CPU time the call to the STAT card and data logging activity was "interfering" with the normal processes.

The time to log event messages was 0.031 ms. The fifth row shows the times adjusted for the log activity.

Table 14. Middleware Data

	Log Time	Publish	Subscribe	Drain Msgs	Check Data
Average	0.031	0.109	0.074	0.134	0.082
Std Dev	0.002	0.002	0.003	0.003	0.002
Min	0.028	0.107	0.072	0.131	0.080
Max	0.046	0.122	0.129	0.151	0.096
Adjust for log		0.078	0.043	0.103	0.050
Utilzn (%)	0.16	0.39	0.21	0.51	0.25

* Times shown in milliseconds
** Utilization based on 20 ms frame

Of interest in determining middleware overhead is the time of 0.078 ms to publish versus the time of 0.043 to subscribe. Unfortunately, it is not possible to gather the exact time for the access of the queues. The publish activity is from the Physical I/O which has three subscribers (DEC, SIM, UPG). The subscribe by DEC is the first action after the publish but includes a process context switch as well as the queue access.

The Drain Msgs activity is within DEC and has no context switch. This process gathers the information from the SIM and UPG controllers via the message queues, but there is extra processing that occurs in this loop, and the total time is 0.103 ms. The Drain Msgs loop in the "baseline only" test run has a time factor of 0.073, which suggests that the actual queue access time is on the order of 0.030 ms. This is speculative but should be reasonably close.

Based on these data points it could be assumed, since the publish takes 0.078 ms and has three receivers, that the time to send individual messages could be on the order of 0.025 or less. While more test runs with other log points could firm up the publisher/subscriber time factors, the overall numbers are impressive enough in terms of their utilization rates that we can conclude that there is little penalty in the use of this messaging technique between processes.

The Check Data time of 0.050 ms represents the heart of the fault-tolerant feature of INSERT. It is here that the boundary conditions are run against the proposed output data of safety, baseline, and upgrade. A 0.25 percent utilization factor demonstrates minimal impact to the overall system resources. Our version was not very robust for this experiment, but even a tenfold growth in CPU time would be acceptable. This code is basically a series of comparisons between minimum and maximum values. Generally, CPU memory accesses and comparisons on most COTS processors require between 2 and 8 cycles each. (Of course, for a processor running 233,000,000 cycles per second, the time for each additional boundary check would be negligible.)

6.3.1.3 Data Analysis: Avionics Algorithm Processing

The processing time (shown in Table 15) is measured between the Pre_recv1553 and the Sent1553 events. This represents the entire PC processing cycle out of the 20 ms frame. The 21 message logs provide the data points that allow computation of the actual processing time as 5.756 ms. However, as stated previously, the decision module sets a midframe delay of 6 ms to provide time for the controller variants to execute. One can see that most of the end-to-end execution is spent in sleep mode. By factoring out the delay, the actual processing time is 0.806 ms for a utilization of 4.03 percent. This is an extremely small CPU load for the algorithm plus associated overhead.

The actual avionics calculations are included in the baseline and upgrade controllers. These accept the data from the 1750 via the previously described processes and calculate the new information to be forwarded to the flight controls. Typical data includes inertial navigation accelerations, target bearing and distance, A/C heading, directional axes, airspeed, G-loading, and predicted bomb impact. The calculations involve a third order Runge-Kutta integration to predict the trajectory and impact point of a given ballistic based on aircraft parameters.

As can be seen, the core calculations require only about 0.05 ms to complete. The 1553 task has 0.194 ms and 0.07 ms to receive and send. The difference here is attributed to the data sizes. The PC receives from the GAC about 2.5 times as much data as it sends back.

It should also be mentioned that the times do not totally reflect the actual performance of the 1553 card. The data had to be unpacked and packed as part of the PC's 1553 task. This was precipitated by two differences between the 1750 and Intel CPU's: floating point representation and endian storage. Floating-point values require two 1553 words, and the words were converted on the PC side of the 1553 data bus to accommodate the two differences.

The floating point representations for the 1750 and Intel can be seen in Figure 29 and Figure 30 with the Intel adhering to ANSI/IEEE 754-1985. In addition to different byte ordering, the exponent is represented in sign-magnitude form on the 1750, while ANSI uses an excess-N notation, sometimes called biased form.

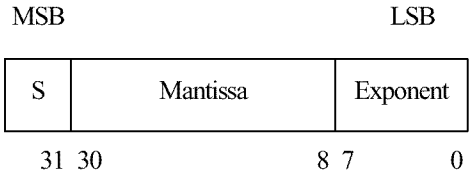


Figure 29. 1750 FP Format

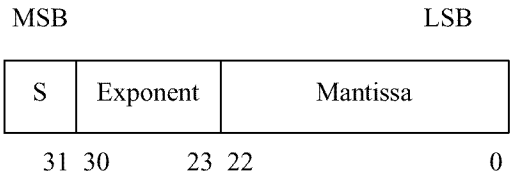


Figure 30. ANSI/IEEE 754 FP Format

Finally, the data storage across address spaces had to be corrected. The 1750 byte ordering of stored data is big-endian, which has the most significant byte of a number stored at the lower address when those data span multiple addresses. The Intel architecture uses little-endian addressing.

By deducting the AMAS and 1553 execution times from the processing time, it is possible to determine the processing for all of the associated message queues, process context switches, and decision switch logic. The "cost" of the fault-tolerant INSERT middleware is about 2.2 percent. The middleware utilization for the first test run that has only the baseline controller active is 1.73 percent. This difference would suggest that the cost of adding extra controller variants would increase the middleware expense by about 0.50 percent for each. As a point of note, the execution of the decision logic rules shown in Table 15 requires 0.05 ms, which accounts for about 10 percent of the middleware time, and this would grow as more boundary checking and the number of controller variants are added.

6.3.1.4 Excess Processing Availability

Certainly the use of COTS CPUs and RTOSs provide a major increase in capability compared with the MIL-STD-1750A. It should be pointed out that our choice of 6 ms for a midframe delay was arbitrary. In reality it could be shortened even more, for it needs to occur only after the controller variants have had sufficient time to execute. Its delay is timed to check controllers that may have experienced hard failures and will not be publishing any data.

Table 15. Processing Times and Middleware Utilization

	Process Time	Midframe Delay	Proc Time Less Delay	Baseline Controller	Upgrade Controller	Receive 1553	Send 1553	AMAS+1 553	Middleware
Average	6.412	4.981	1.432	0.082	0.083	0.226	0.101	0.491	0.940
Std Dev	0.030	0.034		0.009	0.009	0.004	0.003		
Min	6.138	4.685		0.074	0.075	0.193	0.099		
Max	7.041	5.546		0.118	0.125	0.241	0.128		
Adjust for log	5.756	4.950	0.806	0.050	0.052	0.194	0.070	0.366	0.440
Utizn (%)	28.78%		4.03%	0.25%	0.26%	0.97%	0.35%	1.83%	2.20%

* Times shown in milliseconds
** Utilization based on 20 ms frame

Likewise, the midframe delay could be pushed back to the end of the frame, allowing for significant growth in the number and size of the controller variants. As long as the decision controller can execute, respond to physical I/O, and send the 1553 data in time for the GAC to pick it up, the necessary deadlines can be met.

The collection of timing data shows that the INSERT approach with accompanying middleware requires a relatively small amount of additional computing resources. This cost is easily offset by the additional reliability and maintainability benefits that result from the INSERT architectural approach.

6.3.2 Efforts to Migrate JOVIAL to C++ via UML

6.3.2.1 Manual Conversion Effort

This study yielded some very interesting data on the conversion from legacy code to UML format. The effort to extend the Rhapsody environment and libraries to target LynxOS, create the Rhapsody OO models, transfer the previous C declarations and constructs to the Rhapsody model can be seen in Table 16.

Table 16. Engineering Effort for UML

TASK	Hours
1. Review AMAS document	6.8
2. Review AMAS C code	3.2
3. Create Rhapsody model	8.0
4. Move AMAS C to new model	49.8
5. Setup LynxOS for Rhapsody models	68.0
6. Move AMAS C++ to LynxOS and compile	5.0
7. Integrate and test	20.0
TOTAL	155.8

It should be reiterated that tasks 1, 2, and 3 were accomplished by a young engineer not associated with the project but very knowledgeable of UML and the Rhapsody tool. This use of personnel was selected to provide more realistic data points on what a "from scratch" re-engineering effort would entail.

As noted earlier, task 5 represents a one-time-only set up cost associated with each new target. As such the setup costs would not factor into scale up extrapolations to estimate the conversion costs of the entire OFP. Therefore the effort to produce the C++ code only is 87.8 hours!

6.3.2.2 EISR Tool Conversion Effort

For the second experiment the data can be seen in Table 17. At the time of this writing, we are awaiting simulator availability to complete the integration and test. We have no reason to expect more than a 20-hour additional effort. Given that assumption, the difference between the experiments, 88 vs. 136 hours, is due to the differences between the C source code provided to the engineer.

In the first experiment, we used the final design structured C code that had been hand-massaged to get it organized for INSERTion into Simplex. In this second, the C code was a mirror image of the JOVIAL code with all of its "spaghetti like" quality and COMPOOLS. Thus the engineer had to spend more time reworking the C code. The second experiment has an effort that is more representative of what is involved to start with JOVIAL and end up with a C++ equivalent using two COTS modeling/programming tools.

6.3.2.3 Code Growth: Procedural to OO Paradigm

In addition to the labor effort, something can be learned about the "code overhead" from C versus C++ as well as the Rhapsody run time support. Table 18 provides the data on those changes.

Table 17. Second Engineering Effort for UML

TASK	Hours
1. Convert JOVIAL to compilable C	24.0
2. Task explanation	0.4
3. Review AMAS document	5.9
4. Review material on EDCS web	3.2
5. Review AMAS C code	3.7
6. Create Rhapsody model	22.4
7. Move C code to new model	41.4
8. Move AMAS C++ to LynxOS, compiling and correcting errors	14.8
9. Final integration and test	TBD
TOTAL	115.8

Table 18. Code Growth Metrics

Entity	C Code	Rhapsody-Generated C++	Increase (%)
LOC - Code Body	2,047	2,382	16
LOC - Headers	562	741	31
LOC Total	**2609**	**3123**	**19**
Bytes Code	145,500	158,882	09
Bytes Headers	28,321	45,558	60
Bytes Total	**173821**	**204440**	**18**
Bytes Executable	191,998	326,540	70

*(LOC - Lines of Code)

As one would expect, there is some additional overhead associated with the OO paradigm versus the procedural paradigm. This modest increase in code size is offset by the advantage of Rhapsody generating all the logic for associations between classes, header specifications, and state model implementation. There is a labor savings since the detailed coding at this level requires no hours to create.

In the C++ language, much more information is contained in headers than the information in the headers of traditional C language and this fact is reflected in the larger byte size of the C++ files. Estimates of lines of code were determined by eliminating blank lines and comment lines. Of biggest note is the growth in the size of the executable file. The Rhapsody libraries provided more run-time support than would be seen in more conventional C++ libraries, such as messaging for events between state machines, state machine transition actions, association connections, and Accessor/Mutator functions. These auto-generated items result in a larger overhead.

However, it is this author's opinion that the size is more than acceptable in today's hardware environments of large size DRAMs and high-speed processors. This perceived disadvantage of a

slightly larger executable is more than offset by the advantage of future maintainability of projects using the UML models rather than direct C++ code modules.

It was obvious that Rhapsody provided a very good visual framework in which to plug the legacy code. The system's autocode generation is very fast, complete, and precise. In fact, an accomplished C programmer having no C++ experience but rudimentary training of the UML paradigm and Rhapsody tool could easily convert any C program into a C++ system. The tool handles all of the syntactic details of the C++ language and accurately generates the code from the graphical models. This allows the programmer to concentrate at a higher level of abstraction, "making certain the actions and behaviors are right"[54, 55].

6.3.2.4 Level of Effort Metrics

The 787.8-hour effort of manually converting JOVIAL to final C++ when compared to the re-engineering effort of 1,088 hours from Block 25 JOVIAL to Block 40 JOVIAL is only a fraction at 72.4 percent. That represents about a 30 percent reduction in coding costs even using manual translation techniques for the JOVIAL to C phase.

Using the Xinotech JOVIAL to C conversion system, instead of a 787.8-hour effort, the total time to convert this algorithm from JOVIAL to UML to C++ running on a COTS CPU requires 136 hours. That represents a cost savings of 87.5 percent front to end.

Using another metric, the cost per line of code would be 4 lines of code per hour using the manual conversion of JOVIAL, and 23 lines of code per hour using the Xinotech tool. Extrapolating that process to the conversion of the entire OFP, we would expect that the entire OFP could be converted to C++ in approximately 5500 man-hours. It could be argued that scale up of this process might not be linear, but even using a conservative estimate, it would be realistic to expect an 80 percent cost savings for the entire OFP using this methodology and tool set.

6.3.3 Evaluation of ASEP Platform for INSERT FT Capabilities

6.3.3.1 Introduction

This is a discussion of how to use the ASEP to execute INSERT analytic redundant units (ARU, or baseline/upgrade variants) and a decision module (DM) to allow for future, reliable upgrade of the mission systems applications.

The INSERT approach is for the baseline variant to run at a higher priority than the upgrade variant. The DM runs at a higher priority than both variants and is activated at the beginning of a frame in which the two variants run, but then suspends for a predetermined time. It is awakened prior to the end of the frame to evaluate the data provided by the two variants and make a selection of the correct leader.

6.3.3.2 ASEP Concepts

The context in which mission systems applications run is ASEP. The ASEP is a middleware layer that controls interfaces to the COTS RTOS and hardware platform, precluding the use of platform-specific APIs. The ASEP promotes standardized patterns for accomplishing a given task, avoiding the problem of different developers using different structures. Application programs are isolated from each other through memory protection and privilege levels. Each application can be thought of as running its own copy of ASEP.

The ASEP provides frame-based scheduling and anonymous, labeled messaging. The ASEP frames consist of rate groups, each of which operates as a separate thread of execution. Rate group threads are prioritized such that higher rate groups have higher priority than lower rate groups; lower rate groups are, therefore, preemptable. The ASEP scheduling services are anonymous, which means that applications identify themselves to ASEP by address during initialization, and are called back on the rate group thread that they request.

The ASEP messaging operates in a similar fashion in that applications sign up for messages during initialization and are called back at the top of the frame following the one in which the message has arrived. The ASEP (and, by extension, the application) has no knowledge of the source of a received message. There is also the concept of a blocking message, which has associated with it a dedicated, high-priority thread on which an application will be notified immediately of the arrival of the message at a priority higher than that of any of the rate groups. Messages are sent in chains; when an application makes use of the ASEP "send message" service, the message is placed into a chain of messages which is transmitted at the end of the current minor frame. The ASEP also provides a service to enable an application to send a message and have it transmitted immediately.

Specifics of the ASEP service calls, as well as examples of how ASEP services are used, can be found in the ASEP Application Programming Interface, and the ASEP Users Guide.

6.3.3.3 INSERT Implementation under ASEP

Each of the variants and the DM should run as separate applications; that is, they should run in separate address spaces. If one of the variants fails, the other would be able to continue running. In an RTOS like Integrity, this would be accomplished by running each variant and the DM as separate programs, each in its own virtual memory space. In a Unix-like environment, each of the variants and the DM would simply be separate processes. All communication between the three programs would be through labeled messages.

Each of the variants would need to define a message destined for the DM and the message would contain enough information for the DM to make its decision. The decision rules would, therefore, necessarily reside in the DM. The messages wouldn't have to be identical in structure, but the DM would need to be smart enough to distinguish between them in order to correctly evaluate the results reported by each variant. The simplest way to accomplish this would be to use a different label for each variant's message.

Since the variants should be able to run stand-alone (i.e., without the existence of the DM or another variant), each variant needs to be able to communicate its results via labeled messages to the rest of mission systems in the absence of a DM. When a DM is present, mission systems should get these results from the DM itself. The actual source of the information is irrelevant to mission systems because all ASEP messaging is anonymous.

So there needs to be some way of interposing the DM between the variants and the rest of mission systems with regard to flow of information. This could be accomplished through conditional compilation or through some kind of configuration file that the variants would read at run time to instruct them as to how to report their results. Either method requires that this capability be designed into each and every algorithm that has the potential to be upgraded.

Since most operating systems provide no way to assign priorities at the address space level, achieving the desired prioritization of the two variants and the DM could be done in one of two ways:

1. All three of the applications could be run at the same priority. Communication from the variants to the DM would be through blocking messages, which are handled at a priority higher than that of any rate group thread within a program. This approach would be the simplest, but does not guarantee the order of execution of the two variants. There is also no consideration for the DM "waking up" to evaluate whether both variants have finished their computations on time.
2. Since priorities are assigned at the thread level across all programs running on the same CPU, the priorities of the DM's and each variant's threads could be adjusted to make the DM's threads higher priority than the baseline variant's, and the baseline's threads higher priority than the upgrade variant's. Within ASEP, thread priorities are assigned by the frame manager and are derived by taking a base priority (at which the frame manager itself runs), and adding a fixed offset iteratively for each rate group thread. The base priority and the offset are currently hard coded but could be made configurable through the program configuration file that ASEP uses during initialization. Each program has a unique program configuration file. By making the DM's base priority higher than that of the baseline variant, and the baseline's priority higher than that of the upgrade variant, the three applications would run with the desired prioritization. Of course, this prioritization would have the desired effect only if all three programs were running on the same CPU.

The DM presents a unique problem in that it needs to "sleep" while the two variants perform their computations, and then wake up before the frame completes in order to evaluate the results (or the lack thereof). One of the basic premises of ASEP is that applications are not permitted to block rate group threads. This ground rule is intended to ensure that RMA can be used to ensure system schedulability. There are two potential solutions to this problem:

1. If the algorithm under evaluation runs at a lower rate, it may be permissible to run the DM at a rate at least twice that of the variants. The variants would use normal messages to report their results to the DM. The DM could either evaluate results every other frame, or could use a ASEP timer to determine when it is time to evaluate results (note that there are resolution considerations with ASEP timers). With this method, the results of the "winning" variant would be reported to mission systems in the minor frame *following* the completion of the variants, not in the same frame. Thus, there would be some time delay between the completion of the computations and the availability of the results. This may or may not be acceptable, depending on the nature of the algorithm.
2. If the algorithm under evaluation runs at the highest rate, or if it is necessary to make the results of the algorithm available within the same frame in which they are computed, the DM could be run at the same rate as the variants and allowed to block and "sleep" for a fixed period of time during the frame. This time period would have to be less than the frame duration minus some small amount to allow for ASEP overhead and DM computation time. This method assumes that the DM runs as a separate program in its own address space, and that there is nothing in the DM that runs at a higher rate than that portion which blocks its rate group thread; that is, no higher priority rate group threads are active. The variants would need to send their results to the DM using immediate, rather than chained, messages; the DM would need to use blocking messages to receive these results, since blocking messages are processed on dedicated, high-priority threads. This would allow for same-frame communication between the variants and the DM. The RMA schedulability analysis would

not be possible with a DM that uses this method, but this is probably not a major concern during an evaluation phase.

6.3.3.4 Rehost of ASEP to LynxOS and LINUX/RT

In order to evaluate the success of the ASEP fault-tolerant techniques, it was necessary to take the existing baseline of C++ code that represents the middleware and port it to two different operating systems: LynxOS and the open source product, LINUX, with the real-time extensions provided by Timesys Corporation. The original ASEP test bed is based upon PowerPC Single Board Computers (SBC) with the Integrity RTOS by Green Hills Software.

The ASEP middleware was modeled in UML with the Rhapsody toolkit. This allowed us to leverage the prior re-engineering work on AMAS and target the two operating systems. By porting ASEP to these, we could eventually tie this ASEP test bed into the AFTI simulator and running the re-engineered (UML version) of AMAS under a Fault Tolerant (FT) ASEP.

Retargeting the two new RTOSs involved about 300 hours of effort, adding conditional defines for Lynx and LINUX to the existing model that supported Solaris, VxWorks, and Integrity to ensure that the OS-specific system calls would be compiled. Preliminary experiments with the LINUX/RT indicate that it was a highly capable product with full preemptive priority-based scheduling support. Informal test suits run against the Lynx RTOS on the LINUX/RT had similar results.

Notionally this test bed would appear as shown in Figure 31.

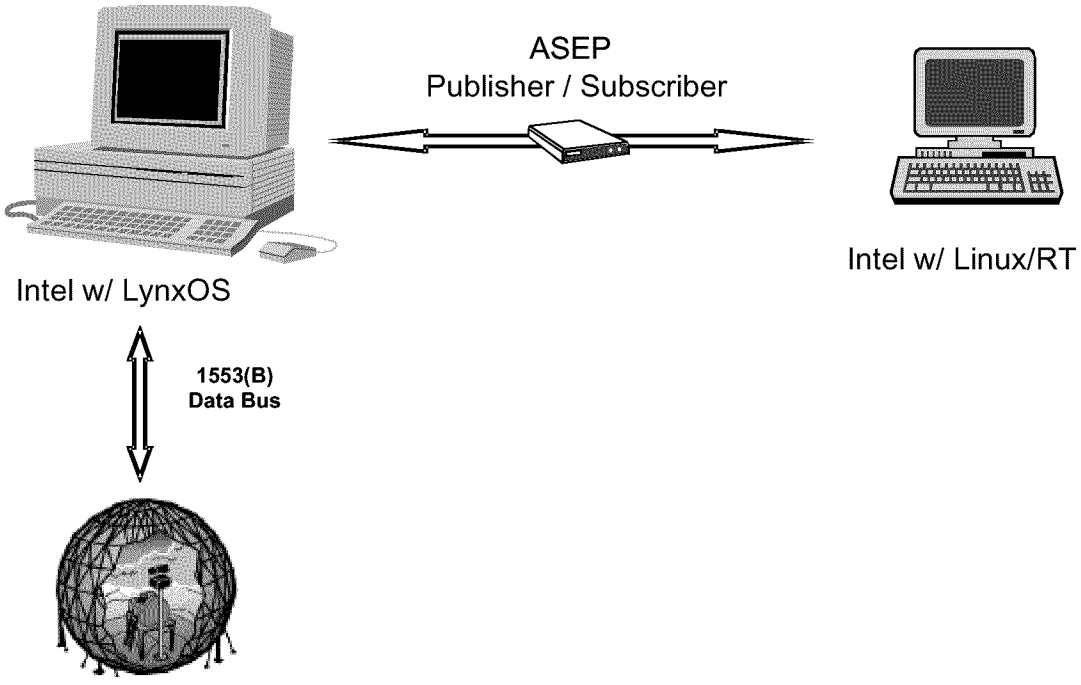


Figure 31. ASEP/AMAS Test Bed

At the time of this report, that effort is not complete due to two factors. The first is that while we have built the ASEP on the LINUX/RT machine and have run test suites, ASEP is not built on Lynx. Although both use the GNU compilers, the compiler supplied with Lynx, even for 3.0.0,

will not support C++ namespaces. Two approaches would be to get a later version of Lynx and its compiler or modify the UML model to not use namespaces.

The second difficulty is the unavailability of the AFTI Simulator. Given the time remaining under contract and the AFTI schedule, the simulator will not become available until after this contract expires.

6.4 Conclusions

6.4.1 INSERT Fault-Tolerant Technologies

The INSERT technology has demonstrated itself to be scalable to major weapon system avionics software. Initially developed at the SEI for providing software fault tolerance, its clean architecture and COTS approach enables its application to many software application domains. Under DARPA's EDCS, this research has demonstrated the feasibility of using this technology to start with a reliable baseline algorithm, wrap it in the INSERT architecture, and then safely and reliably implement new functionality with minimal testing.

Along the same lines, one could use this architecture to "design a little, build a little, test a little" software development approach in real-time systems. A design team could use the INSERT Architecture to "bootstrap" an OFP from the ground up with highly reliable interim versions.

6.4.2 Legacy Conversion to C++

This effort to port a legacy C algorithm to a UML model and to executable code proved to be one of the most straightforward and low cost conversions this author has ever personally encountered, both with the LM organization and in other commercially based companies. Conservatively, large scale migration from legacy JOVIAL code to modern, COTS-supported C++ languages, including UML model representations, could easily be achieved at 80 percent reduction in costs, compared to other methodologies.

In this phase of the INSERT research project, we re-engineered the AMAS algorithm from JOVIAL to C code to a UML model in the Rhapsody tool to a C++ code that compiled and executed on the LynxOS PC. The growth rates of the size of the source code in that transition ranged from about 109 percent to 160 percent, depending on which metric is applied. The conversion rate of the conversion from the original AMAS algorithm in JOVIAL to the final C++ version was four lines of code per hour. The effort to go from a JOVIAL version to the UML / C++ executable was achieved at 23 lines of code per hour. Note that all of the code came out of the Rhapsody model syntactically correct and ready to compile. Very minimal modifications/additions were required after generation and the code compiled and executed correctly on the target system.

6.4.3 ASEP Fault Tolerance

The investigations into the ASEP middleware could be completed only through study and analysis. Reductions in funding on the contract resulted in insufficient funding to implement the proposed changes and test them on the testbed. However, due to the early influence of INSERT concepts on the ASEP design, the merger of these fault-tolerant techniques should be straightforward.

7. Cost Reduction Benefits of INSERT Technology

7.1 Introduction

The cost benefits associated with applying INSERT technologies to DoD embedded applications were determined using design data obtained from project demonstrations. The SEER-SEM is the official software estimation model used by Lockheed Martin Aeronautics Company. This model was employed to substantiate both labor and schedule savings associated with the use of INSERT.

7.2 Background

A key accomplishment of the INSERT project was a successful rehost of a high assurance DoD application within an innovative fault-tolerant software framework (i.e., the INSERT middleware). An advanced attack guidance algorithm was rehosted on INSERT middleware and integrated into a full resolution, ground-based F-16 avionic system simulator. Subsequent performance evaluation verified correct algorithm operation and failsafe operation in response to a variety of injected faults.

Then INSERT framework reduces development cost through reuse (i.e., the fault tolerance framework) and reduction in the amount of required application testing. Such reductions can be directly accounted for in the SEER-SEM estimation model. The following application characterization data was input into the SEER-SEM estimation model to construct three runs:

The guidance application size was 4,000 LOC. Industry standard productivity parameters were applied throughout all runs.

1. *Control Case:* This baseline case assumes no reuse and full (100 percent) testing effort.
2. *Case 1:* This case assumes 20 percent design reuse and 15 percent implementation reuse based on actual code counts from the INSERT project experiments.
3. *Case 2:* This case assumes 20 percent design reuse, 15 percent implementation reuse, and a 15 percent reduction in testing based on actual code counts and analyses from the INSERT project experiments.

7.3 Results

A comparison of the Control Case and Case 2 provides the best indication of INSERT benefits. The results show that the INSERT technology application can reduce the software development effort by 20 percent and developmental span time by 7.2 percent under the stated conditions.

Table 19. Run Results in the SEER-SEM Estimation Model

Run	Total Size	Hours	Months
Control	4,000	4,204	16.36
Case 1	4,000	3,618	15.56
Case 2	4,000	3,362	15.18

7.4 Additional Details

The SEER-SEM parameter settings used for the three runs are shown in the following table:

Table 20. SEER-SEM Run Parameters

WBS Element Description	**Control**			**Case 1 (Full Retest)**			**Case 2 (Limited Retest)**		
LINES (Classic)									
New Lines of Code	4000	4000	4000	**0**	**0**	**0**	0	0	0
Pre-exists, not designed for reuse									
Pre-existing lines of code NDR	0	0	0	**4000**	**4000**	**4000**	4000	4000	4000
Lines to delete in pre-exstg NDR	0.0%	0.0%	0.0%	0	0	0	0	0	0
Redesign required Lines NDR	5.0%	10.0%	40.0%	**80.0%**	**80.0%**	**80.0%**	0.8	0.8	0.8
Reimplementation req. Lines NDR	1.0%	5.0%	10.0%	**85.0%**	**85.0%**	**85.0%**	85.0%	85.0%	85.0%
Retest required Lines NDR	10.0%	40.0%	100%	**100%**	**100%**	100%	**85.0%**	**85.0%**	**85.0%**
Pre-exists, designed for reuse									
Pre-existing lines of code DFR	0	0	0	0	0	0	0	0	0
Lines to delete in pre-exstg DFR	0.0%	0.0%	0.0%	0	0	0	0	0	0
Redesign required Lines DFR	1.0%	5.0%	10.0%	1.0%	5.0%	10.0%	0.01	0.05	0.1
Reimplementation req. Lines DFR	1.0%	1.0%	5.0%	1.0%	1.0%	5.0%	1.0%	1.0%	5.0%
Retest required Lines DFR	5.0%	10.0%	100%	5.0%	10.0%	100%	5.0%	10.0%	100%
Function Implementation Mech.	3rd Generation Languages			3rd Generation Languages			3rd Generation Languages		
Programs Included in Size	1	1	1	1	1	1	1	1	1
PERSONNEL CAPABILITIES & EXPERIENCE									
Analyst Capabilities	Low	Nom	Hi	Low	Nom	Hi	Low	Nom	Hi
Analyst's Application Experience	Nom-	Nom	Nom+	Nom-	Nom	Nom+	Nom-	Nom	Nom+
Programmer Capabilities	Low	Nom	Hi	Low	Nom	Hi	Low	Nom	Hi
Programmer's Lang. Experience	Nom	Hi	VHi	Nom	Hi	VHi	Nom	Hi	VHi
Host System Experience	Nom	Hi	VHi	Nom	Hi	VHi	Nom	Hi	VHi
Target System Experience	Nom	Hi	Hi	Nom	Hi	Hi	Nom	Hi	Hi
Practices & Methods Experience	Nom	Hi	VHi	Nom	Hi	VHi	Nom	Hi	VHi
DEVELOPMENT SUPPORT ENVIRONMENT									
Modern Dev. Practices Use	Nom	Hi	Hi+	Nom	Hi	Hi+	Nom	Hi	Hi+
Automated Tools Use	Hi-	Hi+	Hi+	Hi-	Hi+	Hi+	Hi-	Hi+	Hi+
Logon thru Hardcopy Turnaround	VLo	Low-	Nom	VLo	Low-	Nom	VLo	Low-	Nom
Terminal Response Time	Low-	Hi-	Hi	Low-	Hi-	Hi	Low-	Hi-	Hi
Multiple Site Development	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom
Resource Dedication	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom
Resource and Suppoort Location	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom
Host System Volatility	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom
Process Volatility	Low	Low+	Nom	Low	Low+	Nom	Low	Low+	Nom
PRODUCTION DEVELOPMENT REQUIREMENTS									
Requirements Volatility (Change)	Nom	Nom	Hi	Nom	Nom	Hi	Nom	Nom	Hi
Specification Level – Reliability	Hi+	VHi-	VHi	Hi+	VHi-	VHi	Hi+	VHi-	VHi

WBS Element Description	Control			Case 1 (Full Retest)			Case 2 (Limited Retest)		
Test Level	Hi+	Hi+	VHi	Hi+	Hi+	VHi	Hi+	Hi+	VHi
Quality Assurance Level	Hi-	Hi+	VHi	Hi-	Hi+	VHi	Hi-	Hi+	VHi
Rehost from Dev. to Target	Nom	Hi	VH+	Nom	Hi	VHi+	Nom	Hi	VHi+
PRODUCT REUSABILITY REQUIREMENTS									
Reusability Level Required	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom
Software Impacted by Reuse	100%	100%	100%	100%	100%	100%	1	1	1
DEVELOPMENT ENVIRONMENT COMPLEXITY									
Language Type (complexity)	Nom	Nom	VHi	Nom	Nom	VHi	Nom	Nom	VHi
Host Dev. System Complexity	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom
Application Class Complexity	Nom-	Nom	Nom+	Nom-	Nom	Nom+	Nom-	Nom	Nom+
Process Improvement	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom
TARGET ENVIRONMENT									
Special Display Requirements	Hi	Hi+	VHi	Hi	Hi+	VHi	Hi	Hi+	VHi
Memory Constraints	Hi-	Hi	Hi+	Hi-	Hi	Hi+	Hi-	Hi	Hi+
Time Constraints	Nom	Nom	Nom+	Nom	Nom	Nom+	Nom	Nom	Nom+
Real Time Code	Nom	Hi-	VHi	Nom	Hi-	VHi	Nom	Hi-	VHi
Target System Complexity	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom
Target System Volatility	Hi-	Hi	Hi+	Hi-	Hi	Hi+	Hi-	Hi	Hi+
Security Requirements	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom	Nom
SCHEDULE & STAFFING CONSIDERATIONS									
Required Schedule (Cal. Mos)	1/0/00			0			0		
Start Date	9/27/00			9/27/00			36796		
Complexity (Staffing)	Nom+	Hi-	Hi+	Nom+	Hi-	Hi+	Nom+	Hi-	Hi+
Staff Loading	VHi			VHi			VHi		
Min Times Vs. Opt. Effort	Optimal Effort			Optimal Effort			Optimal Effort		
RISK ANALYSIS									
Effort Probability	50.0%			50.0%			0.5		
Schedule Probability	50.0%			50.0%			50.0%		
REQUIREMENTS									
Requirements Complete at Start	Low			Low			Low		
Requirements Definition Formality	Nom	Nom+	Hi+	Nom	Nom+	Hi+	Nom	Nom+	Hi+
Reqt's Effort After Baseline	YES			YES			YES		
SYSTEM INTEGRATION									
Programs Concurrently Integrating	1			1			1		
Concurrency of I&T Schedule	Hi			Hi			Hi		
Hardware Integration Level	Hi	VHi	VHi	Hi	VHi	VHi	Hi	VHi	VHi

References

1. L. Sha, R. Rajkumar, and M. Gagliardi, "Evolving Dependable Real-Time Systems," *IEEE Aerospace Applications Conference*, Aspen, CO, Feb., 1996, New York, NY: IEEE Computer Society Press, 1996. Also published in "Component-Based Software Engineering," Selected Papers from the Software Engineering Institute, Alan Brown, ed., IEEE Computer Society Press 1996 (ISBN 0-8186-7718-X).
2. M. Bodson, J. P. Lehoczky, R. Rajkumar, Lui Sha, M. Smith, and J. Stephan, "Software fault-tolerance for control of responsive systems," Third International Workshop on Responsive Computer Systems, pp. 133-141, 1993.
3. P. Feiler, et al., "Simplex: A Technology for Rapid, Reliable Upgrade," Tutorial, Software Engineering Institute, 1998.
4. M. Gagliardi, et al., "Simplex Case Study: The Defensive Information Assurance/Warfare Prototype," Technical Report, Software Engineering Institute, 1999.
5. Ken Birman and Keith Marzullo, "The ISIS Distributed Programming Toolkit and the Meta Distributed Operating System," *SunTechnology 2,* 1, 1989.
6. B. Oki, M. Pfluegl, A. Siegel, and D. Skeen, "The Information Bus – An Architecture for Extensible Distributed Systems," *ACM Symposium on Operating System Principles,* 1993.
7. F. Cristian, "A probabilistic approach to distributed clock synchronization," *Distributed Computing 3*, pp. 146-158, 1989.
8. L. Sha, R. Rajkumar, and M. Gagliardi, "The Simplex Architecture: An Approach to Build Evolving Industrial Computing Systems," *The Proceedings of the ISSAT Conference on Reliability,* 1994.
9. Raj Rajkumar, Mike Gagliardi, and Lui Sha, "The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation," *Proceedings of the IEEE Real-time Technology and Applications Symposium*, June 1995.
10. R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach,* Kluwer Academic Publishers, 1991.
11. R. Rajkumar, L. Sha, J.P. Lehoczky, "An Experimental Investigation of Synchronization Protocols," *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, May 1988.
12. L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers,* pp. 1175-1185, September, 1990.
13. M.H. Klein, T. Ralya, B. Pollak, R. Obenza, and M.G. Harbour, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers, 1993.
14. T.A. Henzinger, "The theory of hybrid automata," *Proceedings of the 11th Annual Symposium on Logic in Computer Science,* pp. 278-292, IEEE Computer Society Press, 1996. Invited tutorial.
15. A. Chutinan and B.H. Krogh, "Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations," *Hybrid Systems: Computation and Control*, Second International Workshop, Lecture Notes in Computer Science, Springer-Verlag, 1999.
16. G. Lafferriere, G.J. Pappas, and S. Yovine, "A new class of decidable hybrid systems," *Hybrid Systems: Computation and Control, 2nd International Workshop, HSCC'99*, F.W. Vaandrager and J.H. Van Schuppen, editors, pp. 137-151, Berg en Dal, The Netherlands, Springer-Verlag, 1999.

17. A. Chutinan. and B. H. Krogh, "Approximate Quotient Transition Systems for Hybrid Systems," *Proc. 2000 American Control Conference*, Chicago, June 2000.
18. A. Chutinan and B.H. Krogh, "Computing polyhedral approximations to flow pipes for dynamic systems," *The 37th IEEE Conference on Decision and Control: Session on Synthesis and Verification of Hybrid Control Laws (TM-01)*, 1998.
19. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled, *Model Checking*. The MIT Press, ISBN 0-262-03270-8, January 2000.
20. Danbing Seto and Enrique Ferreira, "A Case Study on Development of A Baseline Controller for Automatic Landing of An F-16 Aircraft Using LMIs," August 1998.
21. D. Seto, B.H. Krogh, L. Sha, and A. Chutinan, "Dynamic Control System Upgrade Using the Simplex Architecture," *IEEE Control Systems*, pp. 72-80, August 1998.
22. A.A. Rizzi, "Hybrid Control as a Method for Robot Motion Programming," *IEEE Int'l. Conf. on Robotics and Automation*, Leuven, Belgium, pp. 832-837, May, 1998.
23. S. Kowalewski, S. Engell, J. Preußig, and O. Stursberg, "Verification of logic controllers for continuous plants using timed condition/event-system models," Automatica 35, pp. 505-518, 1999.
24. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HyTech: A Model Checker for Hybrid Systems," *Software Tools for Technology Transfer,* Vol. 1, pp. 110-122, 1997.
25. C. Daws, A. Olivero, S. Tripakis, and S. Yovine, "The tool kronos," *Hybrid Systems III: Verification and Control*, R. Alur, T. A. Henzinger, and E.D. Sontag, editors, pp. 208-219, Springer-Verlag, 1996.
26. Johan Bengtsson and Fredrik Larsson, *UPPAAL a Tool for Automatic Verification of Real-Time Systems*, DoCS Technical Report Nr 96/67, Uppsala University, ISSN 0283-0574, January 1996.
27. Akash Deshpande, Aleks Göllü and Luigi Semenzato, *The Shift Programming Language and Run-time System for Dynamic Networks of Hybrid Automata,* California PATH Research Report UCB-ITS-PRR-97-7, 22 pages, January 1997.
28. Zohar Manna and the STeP Group, "STeP: The Stanford Temporal Prover," Technical report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, 44 pages, July 1994.
29. M. Shaw and D. Garlan, "Software architecture: Perspectives on an Emerging Discipline," Prentice Hall, 1996, Upper Saddle River, N. J.
30. R. Allen, and D. Garlan, "Formalizing architectural connection," Sixteenth International Conference on Software Engineering, pp. 71-80, 1994.
31. D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Vera, D. Bryan, W. Mann, "Specification and analysis of system architecture using Rapide," *IEEE Trans. Soft. Eng.*, Vol. 21, pp. 336-355, 1995.
32. S. Vestal, and P. Binns, "Scheduling and communication in MetaH," Real-Time Systems Symposium, IEEE, pp. 194-200, 1993.
33. M. Klein, T. Ralya, B. Pollack, R. Obenza, M. Gonzalez Harbour, "A Practitioner's Handbook for Real-time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems," Kluwer, 1993, Boston.
34. D. E. Perry, "System compositions and shared dependencies," *Software Configuration Management*, ICSE'96 SCM-6 Workshop, pp. 139-153, 1996.
35. P. Feiler, J. LI, "Consistency in Dynamic Reconfiguration," *Proceedings of 4th International Conference on Configurable Distributed Systems*, May 1998, IEEE Computer Society Press.
36. P. Feiler, J. Li, "Managing Inconsistency in Reconfigurable Systems," *IEE Proceedings - Software*, Vol. 145, No. 5, October 1998.
37. L. A. Beladi, and P. M. Merlin, "Evolving Parts and Relations - A Model of System Families," Program Evolution, Academic Press, pp. 221-236, 1985.
38. M. Hiltunen, "Configuration Management for Highly-Customizable Services," *4th*

International Conference on Configurable Distributed Systems, Annapolis, MD, pp. 197-205, May 1998, IEEE CS Press.
39. K. G. Shin and H. Kim, "Derivation and application of hard deadlines for real-time control systems," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 22, No.6, pp. 1403-1412, 1992.
40. M. Gagliardi, et al., "Simplex Case Study: The Defensive Information Assurance/Warfare Prototype," Technical Report, Software Engineering Institute, 1999.
41. S. Bohner, R. Arnold, "An Introduction to Software Change Impact Analysis," *Software Change Impact Analysis*, pp.1-26, edited by S. Bohner and R. Arnold.
42. J. Li and P. Feiler, "Impact Analysis in Real-Time Control Systems," submitted to the International Software Maintenance Conference, 1999.
43. D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin, "On task schedulability in real-time control systems," *17th IEEE Real-Time Systems Symposium*, pp. 13-21, 1996.
44. S. Vestal, "Fixed-Priority Sensitivity Analysis for Linear Compute Time Models," *IEEE Transactions on Software Engineering*, Vol. 20, No. 4, pp. 308-317, 1994.
45. W. Tichy, "A Data Model for Programming Support Environments and Its Application," Automated Tools for Information System Design, pp. 31-48, North-Holland Publishing Company, 1982.
46. D. E. Perry, "Version Control in the Inscape Environment," *9th International Conference on Software Engineering*, pp. 142-149, 1987.
47. E. Ploedereder, and A. Fergany, "A Configuration management Assistant," *Second International Workshop on Version and Configuration Control*, Oct. 1989, ACM Press.
48. D. Garlan, R. Monroe, and D. Wile, "Acme: An Architecture Description Interchange Language," *Proceedings of CASCON'97*, pp. 169-183, Nov. 1997.
49. R. Monroe, "Capturing Software Architecture Design Expertise with Armani," Technical Report, CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, 1998.
50. C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," JACM 20 (1), pp. 46-61, 1973.
51. L. Sha and J. B. Goodenough, "Real-Time Scheduling Theory and Ada," Technical Report CMU/SEI-89-TR-14, Software Engineering Institute, 1989.
52. Mark H. Klein, John P. Lehoczky, and Ragunathan Rajkumar, "Rate-Monotonic Analysis for Real-Time Industrial Computing," IEEE Computer 27(1), pp. 24-33, 1994.
53. Lui Sha and Shirish S. Sathaye, "A Systematic Approach to Design Distributed Real-Time Systems," IEEE Computer 26(9), pp. 68-78, 1993.
54. Ben A. Calloni, et al., "INSERT: A COTS-based Solution for Building High-Assurance Applications," Gateway to the New Millennium. 18th Digital Avionics Systems Conference Proceedings (IEEE Cat. No.99CH37033), pp. 2.D.6-1, Oct 24-29, 1999.
55. Donald J. Bagert, and Ben A. Calloni, "An Icon-Based Environment for Teaching Computer Programming," *Journal of Epistecybernetics*, International Society of Epistecybernetics, pp. 49-61, Texas Tech University, Lubbock, Texas, September, 1999, Vol. 1 (01), 1999.
56. Ben A. Calloni, and Donald J. Bagert, "Iconic Programming Proves Effective for Teaching the First Year Programming Sequence," *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, pp. 262-266, San Jose, CA, 27 Feb.-1 Mar 1997.

List of Acronyms

ACRONYM	DESCRIPTION
A/C	Aircraft
ACTL	"All paths?" computation tree logic
ADL	Architecture Description Languages
ADT	Abstract data types
AFRL	Air Force Research Laboratory
AFTI	Advanced Flight Technology Integration
AMAS	Automated Maneuvering and Attack System
AMCOM	Army Aviation and Missile Command
ANSI/IEEE	American National Standards Institute/??
API	Application Programming Interface
AQTS	Approximate Quotient Transition System
AR	Analysis Region
ARC	Analytic Redundancy Component
ARU	Analytically Redundant Unit
ASEP	Advanced Software Execution Platform
ATM	Asynchronous Transfer Mode
CMU	Carnegie Mellon University
COMPOOLS	Common Data Pools for JOVIAL Language
COTS	Commercial Off the Shelf
CSCI	Computer Software Configuration Item
CTL	Computation Tree Logic
DARPA	Defense Advanced Research Projects Agency
DEC	Decision Module
DFIP	Data Fusion Integrity Processes
DM	Decision Module
EDCS	Evolutionary Design of Complex Software
EISR	Embedded Information Systems Re-engineering
FFFI	Form, Fit, Function, Interface
FSMBs	Finite State Machine Blocks
FT	Fault Tolerant

GAC	General Avionics Computer
GNU	GNU's Not UNIX
GRMA	Generalized Rate Monotonic Analysis
GUI	Graphical User Interface
HA	Hybrid Automata
HUD	Head Up Display
ICS	Initial Continuous Set
INS	Inertial Navigation System
INSERT	Incremental Software Evolution for Real-Time Systems
IPC	Interprocess Communication
LMI	Linear Matrix Inequality
LMTAS	Lockheed Martin Tactical Aircraft Systems
LOC	Lines of Code
MDU	Monitoring and Decision Unit
MMC	Modular Mission Computer
NDR	No Direct Replacement
NSSN	National Standards Systems Network
ODE	Ordinary Differential Equation
OFP	Operational Flight Program
OMD	Object Model Diagram
OO	Object-oriented
PCMCIA	Personal Computer Memory Card International Association
PIHA	Polyhedral Invariant Hybrid Automata
POSIX21	Portable Operating System Interface UNIX - Standard 21
PTHB	Polyhedral Threshold Block
QTS	Quotient Transition System
RMA	Rate Monotonic Analysis
RT	Real-time
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real-Time Operating System
SBC	Single Board Computer
SCSP	Switched Continuous System Block
SEER-SEM	Software Evaluation & Estimation of Resources – Software Estimation Model
SEI	Software Engineering Institute

STAT	System Timing Analysis Tool
UDP/IP User	Datagram Protocol/Internet Protocol
UML	Unified Modeling Language
WBS	Work Breakdown Structure