Paper Session II-A - The Testability of Software for the Space Station Freedom Program

Lindell R. Small  
*IBM Federal Sector Division Space Station integration and Test Houston, TX*

Joseph W. Devlin  
*IBM Federal Sector Division Space Station integration and Test Houston, TX*

John G. McBride  
*Planned Solutions Inc. Houston, Tx.*

Follow this and additional works at: https://commons.erau.edu/space-congress-proceedings

**Scholarly Commons Citation**

This Event is brought to you for free and open access by the Conferences at Scholarly Commons. It has been accepted for inclusion in The Space Congress® Proceedings by an authorized administrator of Scholarly Commons. For more information, please contact commons@erau.edu.
The Testability of Software for the Space Station Freedom Program

June, 1991

Lindell R. Small
Joseph W. Devlin
IBM Federal Sector Division
Space Station Integration and Test
Houston, TX
John G. McBride
Planned Solutions Inc.
Houston, TX.

Abstract

The Space Station Freedom Data Management System consists of state-of-the-art hardware and software technology that exceeds the capabilities of earlier test tools and methods used to verify and certify man-rated space systems. New technologies and techniques are being developed to meet these challenges.

Introduction

Hardware and software technologies have advanced to a level that permits development of large and complex systems, such as the Space Station Freedom, which have a life of many years. The mission requirements over this period as well as maintainability dictate system architectures that are operationally flexible and accommodate technology insertion. In addition to crew/vehicle safety requirements for space systems, these systems may have other requirements that could not have been met cost effectively by older technologies.

Hardware technology for space applications has evolved to a level that allows reliable distributed processing over high speed LANs using commercial standards. These advancements provide much higher processing bandwidth, virtual memory, greater control and flexibility over the allocation and distribution of processing functions, and a growth path consistent with industrial directions.

It is now practical to develop real-time software upon a base of mostly commercial products. Using an operating system, such as UNIX*, can provide a standard execution environment that is compatible with other commercial products. This will allow a reduction in the cost of ground support and development environment. The use of newer high order languages, such as Ada, coupled with modern software engineering practices, allows software developers to tackle and control large software development efforts. Ada's high level of abstraction and strong typing can make the software program more understandable and prevent the introduction of many errors.

These advancements in technology, however, have outpaced our testing technology. Computer hardware designs may preclude the use of processor control and visibility tools used in previous manned space programs. Software technologies, designed to exploit the increased capabilities of hardware and to support software engineering principles, may use methods including dynamic memory allocations and optimization that can add to verification difficulties. There are also features of the Ada language that, if used improperly (but legally in an Ada sense), may lead to software that is difficult to test or perhaps even unstable.

This paper addresses these situations and presents testing strategies that are being validated for use on space systems software.

*UNIX is a registered trademark of UNIX System Laboratories in the United States and other countries.
Brief Overview of the Space Station Data Management System

The Space Station Data Management System (DMS) is comprised of both hardware and software elements that provide all of the communication, object management, system control, and user services to support the flight application software that controls the operation of the Space Station Freedom. A brief description of the architecture follows.

DMS Architecture

It was determined during the Phase B portion of the Space Station Freedom Program (SSFP) that the requirements of the DMS could be best satisfied by a distributed architecture which implemented a number of military and commercial standards in the data processing field. Figure 1 shows the DMS architecture chosen for the SSFP.

The DMS architecture consists of a core network and a payload global network implemented using a dual ring fiber optic topology that conforms to the Fiber Distributed Data Interface (FDDI) standards. The communications protocol that was chosen conforms to the Open Systems Interconnect standards (ISO/OSI). The core and payload networks are physically and functionally separated by a bridge. Local processing environments, required by the Space Station process control applications, are provided by processing nodes with Local Area Networks (LANs) for the dedicated command and control functions required. The LANs are implemented using the MIL-STD-1553B linear bus topology and communications protocol.

Figure 2 provides a functional depiction of the architecture. It can be clearly seen that the DMS provides three distinct data process domains as well as the characteristics of each domain. Both the information management and the process control domains consist of processing elements that are constructed from a standard inventory of cards. Figure 3 presents a more detailed...
view of the card types and their arrangement in the processing elements. Of particular interest are the Embedded Data Processors (EDPs) which are common to all the elements. It is within the EDPs that the DMS and applications software is executed.

The EDPs are the space application equivalent of the IBM PS/2 Model 80 personal computer. The processing unit of the EDP is the Intel 386 processor. The Intel 386 was chosen because it is a commercial standard used throughout industry; it provides a lasting technology that has a growth path, and it provides the processing flexibility required by the SSFP. The Intel 386SX processor was chosen for the multiplexer/demultiplexer (MDM) units.

The software architecture employed in the SSFP is presented in Figure 4. The architecture is implemented through a set of common system service functions used by all users and a set of applications software specific to the process control function being performed. The common system services provided by the DMS consist of the following items:

1) Runtime Services
2) Global Communications Services
3) Local Bus I/O and Applications Communications
4) Human User Interface
5) File Services
6) Hardware Management
7) Data Management
8) Operating System/Ada Runtime Environment (OS/ARTE)
9) Network Operating System (NOS)
10) Standard Services (STSV)
11) User Support Environment (USE)
12) Data Storage and Retrieval (DSAR)
13) Systems Management (SM)

To minimize development costs and provide a system consistent with future software technologies, Commercial-Off-The-Shelf (COTS) software products were chosen where feasible. As a result, a COTS UNIX operating system was chosen as the kernel for the OS/ARTE and modifications were made to the kernel to support specific interfaces. Additionally, Ada was specified by NASA as the programming language for SSFP.

Man–Rated Software Systems Testing

Testing of man-rated software systems is extremely critical because human lives are dependent on those systems. Strict procedures must be implemented and adhered to so that the system integrity will be maintained.

Fault Avoidance/Fault Tolerance

Two basic strategies exist to develop software to deal with faults for this environment. A fault is any failure in the system that leads to an error in the way information is processed. The first and most common strategy is fault avoidance. Software developed for earlier NASA spacecraft, such as the Space Shuttle, was developed with this strategy. This approach places a significant burden on software testing throughout the development phase. Each software product, from the requirements and design specification to the object code executing in the actual flight hardware, must be rigorously verified. This process can be expensive and can increase the relative cost of software from about 20 percent to as much as 40 percent.

The second strategy to handle faults is fault tolerance. This approach requires that faults be detected and recognized. The system must isolate the fault to prevent its effects from propagating to other parts of the system. The system must also provide some reconfiguration of the system to continue functioning until the fault can be eliminated. Unlike fault avoidance, this approach places a much greater burden on the design team to identify the classes of faults and design software to handle the detection, recognition, isolation, and reconfiguration functions. The fault tolerant approach is not used in lieu of fault avoidance, but in addition to it. While the fault avoidance strategy seeks to produce “error free” software, the fault tolerant strategy anticipates potential system failures such as communication channel failures, errors in man–machine interaction, and hardware component failures. It introduces another level of test concepts, policies, methods, and technology necessary to deal with the added complexity of the system.
DMS Integration and Test Methodology

The DMS integration and test methodology has evolved from previous manned space flight programs including Shuttle, Skylab, and Saturn/Apollo. The primary experience gained through the Shuttle program has been the major influence in the derivation of this methodology. It provides valuable information on not only the integration and test phase but also an organizational model to best ensure a complete and adequate test program.

The primary objectives of the integration and test methodology can be summarized in three key points.

1) Demonstrate that the DMS adheres to the letter of the customer's requirements
2) Assure the DMS performs and supports the users in accordance with the customer's operational expectations
3) Provide software that is "error free"

The first two objectives are quantifiable since they can be explicitly tested. Specific test plans and scenarios developed to cover all documented requirements can be implemented and a one-to-one mapping of tests to requirements can be established and maintained. The third objective is more nebulous and, as experience has shown, can only be approached asymptotically. How rapidly and efficiently this goal is reached is dependent on how effectively an organization is structured to address the following areas:

1) Early definition and application of programming standards and techniques
2) Establishment of tests, audits, and code inspections
3) Early definition and implementation of test tools, simulators, and other support software proven by a thorough test plan and maintained through configuration control
4) Configuration control of the build and integration of the evolving software system
5) Configuration control of the implementation and retest of the software changes resulting from requirements upgrades and discrepancy corrections
6) Step-wise integration and testing of the system

The DMS organization has been developed from the model provided by Shuttle. An organizational element independent of software development, called Integration, Test, and Verification (IT&V), has been established to focus on the integration and test of DMS. IT&V is cast in the role of the conscience of the project and fosters a definite but healthy adversary relationship between itself and the development organization.

The IT&V organization performs two major categories of tests on the DMS. The first category is the detailed requirements testing which covers each "shall" in the requirements document. A requirements cross reference matrix is developed and delivered to the customer which maps the test cases to the requirements tested. The second test category is system testing. System testing includes testing the system in a realistic operational environment and subjecting it to nominal and off-nominal stress conditions.

To accomplish both types of testing, the test tools and test environment must provide a number of basic capabilities. To control the conditions of the testing and establish the appropriate processing state, a capability must be provided to set the initial state parameters of the system. To acquire the proper data for evaluation of the test results, visibility into the data locations within memory must be provided. Additionally, knowledge of the system state over time must be provided as a potential triggering mechanism for setting the desired processing conditions or acquiring the desired data.

The test control, data visibility and access capabilities must be provided in a manner that supports both categories of testing being accomplished in a complete system configuration and in an "unaltered" state. An unaltered state simply means the system under test must be in the configuration expected to fly. Additionally, all controls of the system under test required to establish the test state, control the test conditions, and extract the required test data must not be detectable and must not affect the flight state. With these test constraints and system element features, it is clear that there are specific challenges that must be dealt with from a testing point of view.

System testing can be defined in terms of six different levels. These levels are described below.

Level 1 Testing (Unit)

During development, specific testing is done to ensure that mathematical equations and logic paths provide the results expected.
Level 2 Testing (Functional)
This development test activity is similar to Level 1 but expanded to include all interfacing modules required to satisfy a specific user input command.

Level 3 Testing (Subsystem)
Level 3 Testing demonstrates the ability of a particular subsystem to execute all of its nominal functions. Multiple functions are tested as well as timing.

Level 4 Testing (System)
Level 4 Testing exercises entire systems to test operational sequences and monitor system performance.

Level 5 Testing (Release Validation)
This activity involves repeating all of the Level 4 tests in the actual flight hardware environment.

Level 6 Testing (Design Validation)
Level 6 Testing (usually performed independently of prior levels) involves testing the system to design requirements including timing and performance requirements.

Space Station Testing Objectives
Figures 5 through 8 depict the testing objectives at each level for the DMS.

Typical Testing Methodologies Used
Black Box
Black Box testing implies the testing of software simply by exercising all external interfaces to the Box upon which the software is executing. The advantages of this testing method are that usually fewer, less complex test devices are needed and no instrumentation of the software under test is required. The disadvantages are limited visibility into the software execution and the extent of the test suite required to test the potential ranges of all interfaces.

White Box
White Box testing implies the testing of software by monitoring and controlling its execution on the Box with special test equipment. The advantage of this testing method is that more visibility of the software is allowed; however, completely nonintrusive test devices are usually costly and do not always solve all of the testing dilemmas.

Space Shuttle Program
Both the hardware and software technology used on the Space Shuttle Program were unlike that of today's Space Station systems. The primary onboard Shuttle computer system is centered around IBM AP-101 General Purpose Computers (GPCs). Testing of the Space Shuttle flight software typically involved simulators communicating with the GPCs to drive external interfaces. The execution of software on the GPCs was monitored and tested primarily by stopping the GPC during an execution run and dumping data out of the machine to be analyzed later.

Because the GPC contained a single processor, all software execution could be frozen during a simulation. Once the appropriate data was collected, the GPC was "started" again. Because of the controls allowed in the interface test equipment that stopped the GPC processor, software execution on the GPC was not affected. In short, the software never "knew" that it had been stopped.

Some flight software modifications were made to accommodate the test environment. To accomplish a GPC stop during certain execution states of the processor, elements of the flight software were instrumented to support the test environment. NASA felt that this "test scar" was acceptable because of the test capabilities provided.

The Space Shuttle software testing methodology is a good example of pure "white box" testing at all test levels.

Challenges to Test Space Station Software
The Space Station Freedom Program, as described previously, employs a distributed architecture for DMS services and application software. The software used to implement this system will be written in Ada and will execute in a Unix environment. The challenges of testing such a system are described below.
Language

In previous languages used on man-rated programs, limitations on program behavior ruled out many potential faults, since the language could not express certain types of processing. With Ada, many of these built-in restrictions have been eliminated. The Ada language has its own restrictions, recognizes certain abstractions explicitly, and provides a framework for dealing with them.

Ada Risk Areas

Ada's high level of abstraction, dynamic allocation, and optimization makes low level testing (i.e. use of compile time absolute addresses, memory dumps, and assembly debugging) impractical except for subprograms. Also, 100 percent deterministic paths may not be guaranteed in an asynchronous, multitasking program.

The Ada language provides the opportunity to improve program quality if it is properly used. Its features allow software engineers to develop programs that are more understandable, reliable, and maintainable. However, there are features that, if used indiscriminately in real time systems, can lead to software programs that are difficult to test. These are listed below.

1) Strong typing and subtype mechanisms may create inefficient execution making it more difficult to validate performance requirements
2) Subprograms formed by composite functions can lead to unpredictable storage use
3) Subprograms with global variables or side effects result in interfaces that are difficult to test
4) Use of generics may not be consistent with design intent
5) Use of generics may be inefficient or make it difficult to analyze time and space use
6) Exceptions may lead to inefficient execution
7) Exceptions may lead to more complex interfaces making them more difficult to test
8) Behavior of predefined exceptions can sometimes be unpredictable
9) Tasking may lead to nondeterministic behavior in time and space
10) Dynamic allocation of memory makes it more difficult to predict whether a program will meet its storage constraints

Software Technology

The UNIX Operating System (OS) and its associated Ada Run Time Environment (ARTE) provide a highly flexible, real-time platform upon which the DMS services and user applications can execute. However, the unpredictability of the way in which the OS/ARTE schedules and manages the various software tasks executing in this environment produces many difficulties in testing these software elements.

UNIX Risks

The UNIX environment allows for run-time reconfigurability and shuffling of OS resources by employing the concept of Virtual Memory. A few of the features of UNIX and its use of Virtual Memory that produce testing difficulties are listed below.

1) Lack of visibility into functions and operations being performed by the operating system
2) Inability to synchronize or stop all hardware elements and OS processes
3) Lack of determinism in task scheduling
4) Dynamic memory allocation

Hardware Technology

The distributed processing capabilities of the DMS are required to support the flexibility of multiple flight software applications and the various resources that they must access. Also, within each SDP, multiple processors are required to meet the throughput and processing requirements of these applications. The complexity of such a system only increases the difficulty in testing software that executes in that environment.

The major hardware architecture features of the DMS that induce risks in testing are:

1) Distributed Processors

   The major issues with testing software executing on a distributed processing system appear at the upper levels (4 - 6) of the test process.

2) Multiple Processors within an SDP

   The major issues of testing software that executes on multiple processors within a single SDP affect testing at all levels.
3) Pipelined Processors

The SDP primary processor is a pipelined processor. In simple terms, this means that it will "fetch" instructions from memory prior to their actual execution. This feature is necessary to achieve the processing power required of the SDP. Testing at all levels is affected due to the inability to control test operation based on actual instruction execution.

DMS Integration and Verification Methodology

The strategy for verifying the DMS requires a change in the verification paradigm. The old paradigm being that for the integration and test of Space Shuttle flight software. A discussion of the differences between the paradigms and recommended options for a modified verification methodology for Space Station DMS follows.

The Paradigm Shift

The former paradigm assumed a centralized processor, static allocation of memory, and a cyclic executive with high frequency control loops. All software was designed and tested for fault avoidance, and any flight software that only supported testing was minimized. Error free confidence in the software was principally achieved by demonstrating determinism by having system level visibility to all data. This was accomplished with various visibility tools through six levels of testing ranging from unit to system level. This was practical for the amount of flight software (less than 100,000 lines of code) on the Shuttle. While hardware and software technology used in the DMS may prevent the use of older verification techniques at the system level, even if it were technically feasible, the amount of flight software (approximately 900,000 lines of code of DMS software plus several hundred thousand lines of application software) and its distributed nature makes it cost prohibitive to use these techniques.

The current paradigm assumes a distributed processing system, dynamic allocation of memory, and an asynchronous executive with low frequency control loops. Besides application flight software, other mission software will be used to monitor and report the health and status of the system and to provide reconfiguration if a system fault occurs. Confidence in the software will be achieved by building upon a hierarchy of verification methods and maintaining consistency between the development and execution environments. While verification at the software unit level will rely on traditional and proven methods, higher levels of system testing will rely on trusted interfaces within applications, runtime, pre-runtime, and hardware systems. These interfaces specify the functional characteristics of the system. Once these interfaces are defined, verified, and baselined, they provide the context for higher levels of program execution. The features of the health monitoring software can provide access to system data to support nonintrusive verification.

The levels and techniques of verification planned for the DMS are described below. While the reliability of some of these has been proven on prior space systems, other techniques are being developed and validated.

Approaches to Solving Test Challenges

The challenges posed above are very new to the test requirements of man-rated software based programs. Some would consider this task to be much more challenging than that of the implementation of the system itself. Although these issues are still being addressed, a solid test approach is being implemented to effectively solve the issues with the lowest cost to the Space Station Program.

Test Philosophies

The general approach for testing Space Station software involves a combination of a black box and a white box methodology providing an extremely flexible environment that can be used to support each test level. Different elements of this environment are used to "certify" the system before stepping to the next level. Also, certain software implementation standards are being proposed that would force the software to be more "testable". A description of these tools and standards follows.

Test Tools

The basic environment for all levels of test is centered around a simulation host that will provide the necessary processing power to simulate the required interfaces at each test level. The simulation host is attached to the DMS elements via interface hardware that is of a similar architecture base as the DMS itself. Not only does this
reduce cost by utilizing reusable software in the interface equipment, but it provides for the capability to move functionality of the simulations down into the interface hardware to support throughput and processing requirements. Figure 9 depicts the test environment for Space Station software.

Simulation Host

The simulation host executes various models that are required to supply the missing elements in each test level. For example, at the lower test levels, more interface simulations and processing is required of the simulation host because only one SDP is used. At the highest test levels, the complete system is used in a fully integrated manner; therefore, minimal simulation equipment is needed.

The simulation host provides post-processing tools for analysis and correlation of simulated and real data.

The simulation host also provides the necessary tools to support the software symbolic referencing and cross debugging requirements.

Software Execution Monitoring/Breakpoint and Event Trapping

Software monitoring tools are provided for both the SDP and MDM elements of the DMS by the SDDU and MDM I/F subunits respectively. These elements provide the "white box" test capabilities.

System Development and Diagnostic Unit (SDDU)

The SDDU provides a passive monitoring capability of software executing in an SDP by "listening" to fetches that occur between the processor and memory. It can also be used to snap data directly from memory or patch over locations of memory. An "on-the-fly" capability is provided that allows data to be snapped or patched instantaneously based on a predefined event.

MDM Interface Unit

The MDM Interface Unit provides an internal debug interface to the MDM processor. It allows similar capabilities as the SDDU. The MDM Interface Unit also supports simulated sensor/effector I/O for up to five MDMs driven by host simulations.

Bus Monitors/Simulators

Bus monitors and simulators are provided for both the FDDI (global) and the 1553 (local) buses. These elements provide the "black box" test capabilities.

FDDI Network Monitor

This unit provides the capability to monitor traffic on the FDDI network based on triggers and filters specified by the simulation host.

FDDI Network Simulator

This unit can simulate multiple or single nodes on the FDDI network. The higher levels of the network communication protocols are tested in this unit as well.

Local Bus Subunit

This unit can respond to requests for data over the 1553 bus from an SDP as if a real MDM were connected. It can also simultaneously simulate a bus controller requesting data and a bus monitor recording traffic.

Test Approaches

The current proposed test approaches to meet each of the challenges described above utilizing these test tools are:

Ada

Some of the challenges of testing Ada could be solved through the selection of a high-quality optimizing compiler to eliminate unnecessary run-time type checking and to implement subprogram calls with minimal overhead.

Most of the challenges of testing Ada impact the lower test levels and could be met through the use of a resident Ada debugger. Specifically, non-performance related requirements place on lower elements within software functionality could be verified for proper code path execution and expected results. Also, detailed equations within applications could be verified for accuracy given all ranges of possible inputs. The system level or performance related requirements would be verified with the existing non-intrusive tools described earlier.

Although there is some risk with this method in that the debugger could actually impact the test environment, the visibility provided by a debugger far outweighs these risks.
Figure 9. SPACE STATION SOFTWARE TEST ENVIRONMENT
UNIX

Multiple approaches are being taken to address UNIX issues.

1) Again, a resident debugger is recommended for the lower test levels in areas that involve the Ada to UNIX interaction.

2) Deterministic path labels are recommended for performance measurements and OS monitoring.

3) Address resolution tools are being developed that allow the user to monitor software based on symbols. The virtual to physical addresses are resolved after the software is loaded.

Multiple Processors

The approaches to address testing of software in a multiprocessor environment are:

1) The SDDU read and write “on-the-fly” capability is being recommended to capture data in real time without having to stop software execution.

2) Applications are required to be self-starting and self-re-starting.

3) The simulation host will support a “checkpoint” of journal data so that even with distributed processing, the system state can be recreated for problem resolution.

Software Implementation Standards

Ada

The majority of the risks of testing Ada could be minimized through the use of proper education and programming guidelines. A few of the recommended guidelines are included below.

1) Reduce the use of global variables. All data should be passed to a package through parameters.

2) Document all expected use of storage.

3) Develop generic packages, rather than a generic subprogram, and document all assumptions made about the properties of the generic parameters. Select a compiler that allows the programmer to choose an implementation strategy for generics by means of a pragma.

4) Exceptions are part of a module’s interface and should be well documented. Exception handling and propagation should be planned as an overall strategy for maintaining system integrity. Exceptions should not be used for determining conditional processing in the normal execution of an algorithm.

5) Nondeterminism is inherent in any system with asynchronous events. The following guidelines are recommended for Ada tasks.
   - Tasks should not depend on other tasks except at synchronization points.
   - Task logic should not depend on the choice in a selective wait.
   - Tasks should employ deadlock detection schemes.
   - Use monitor tasks to provide access to shared data to avoid race conditions.
   - Packages containing hidden variables should only be used by one task to avoid race conditions, otherwise use a monitor task within the package.

6) Use representation clauses to limit run time allocation of storage and use an exception handler to initiate recovery upon pre-defined storage error exceptions.

Summary

The software testing objectives for the Space Station Freedom Program have initially been met by applying a philosophy of testing very similar to that of the Space Shuttle Program and through the use of a complex set of flexible tools and implementation standards.

The task of integrating and testing flight software for the SSFP is still in its early stages. Although many tools and capabilities exist or are planned, the difficulties encountered during these early stages will most likely dictate the need for additional support tools. In cases where tools are unavailable or are not cost effective, a change in the methodology of software testing and verification that deviates from the Space Shuttle methodology may be required.