TRW Systems Engineering &
Development Division

TRW-TS-90-02

Incremental Software Test Approach for DoD-STD-2167A Projects

Michael Springman

January 1990

TRW Technology Series

TRW Technology Series
TRW Technology Series
TRW Technology Series
TRW Technology Series
TRW Technology Series
TRW Technology Series
TRW Technology Series

...chnology Series

Incremental Software Test Approach for DOD-STD-2167A Ada Projects

Michael C. Springman
TRW Defense Systems Group
Redondo Beach, California

Abstract

One of the key challenges for large Ada software projects is to define and execute an efficient, cost effective, bounded test program that results in a quality product that meets all customer software requirements. This is particularly challenging in the current climate of Government fixed price contracts. The Command Center Processing and Display System Replacement (CCPDS-R) project is being developed entirely in Ada by TRW for the U.S. Air Force on a fixed price basis. To mitigate downstream test risks, TRW has defined an incremental test approach that satisfies TRW and Government objectives for informal development/integration testing and for formal requirements verification. Features of Ada are employed to create a software architecture that supports an incremental test philosophy and contributes to reduced integration effort and risk. The resulting test approach conforms to DOD-STD-2167A standards.

Project Background

The CCPDS-R system will provide display information used during emergency conferences by the National Command Authorities; Chairman, Joint Chiefs of Staff; Commander in Chief North American Aerospace Command; Commander in Chief United States Space Command; Commander in Chief Strategic Air Command; and other nuclear capable Commanders in Chief. It is the missile warning element of the new Integrated Attack Warning/Attack Assessment System Architecture developed by North American Aerospace Defense Command/Air Force Space Command.

The CCPDS-R project is being procured by Headquarters Electronic Systems Division (ESD) at Hanscom AFB and was awarded to TRW Defense Systems Group in June 1987. The project consists of three separate subsystems of which the first, identified as the Common Subsystem, is 24 months into development. The Common Subsystem consists of approximately 350,000 source lines of Ada with a development schedule of 40 months. When software development for all three subsystems is complete in 1992, over 600,000 Ada source lines plus developed tools and commercial off-the-shelf (COTS) software will have been delivered to the Air Force. CCPDS-R is characterised as a highly reliable, real-time distributed system with a sophisticated user interface and stringent performance requirements. All CCPDS-R software is being developed using DEC's VAX Ada compiler on DEC VAX/VMS machines, augmented with Rational's R1000 Ada environment. The software will execute on a network of DEC mainframes and workstations.

CCPDS-R was planned and bid prior to the establishment of DOD-STD-2167A [2167A], so the software is being developed using a heavily tailored DOD-STD-2167. The 2167 tailoring was done in parallel with the formulation of DOD-STD-2167A, which has resulted in a CCPDS-R methodology and documentation set that is consistent with DOD-STD-2167A.

CCPDS-R exhibits the characteristics of a typical large 2167/2167A Ada development project, including:

1. Large number of software requirements (approximately 2,000)
2. Multiple CSCIs (6 for the Common Subsystem; 15 total)

3. Large number of 2167A components (approximately 7,000 CSCs/CSUs) and architecture objects (30 VAX/VMS processes, 110 Ada tasks)

4. Informal test of individual components to test all nominal, off-nominal and boundary conditions

5. Informal integration of tested components into working capability strings

6. Formal requirements verification per Government-approved test plans and test procedures

Test Program Objectives

A successful Ada development project must have an efficient (i.e., cost-effective) test program that results in a reliable, tested product that meets all customer requirements. The best-designed Ada system in the world that doesn't meet a customer's test expectations will have difficulty being sold or fielded. The test program must have clear bounds to the scope of testing; it cannot be open-ended. Both the contractor and the customer must know when testing is complete. A well-defined, complete, consistent requirements set that specifies required functionality is essential, and should *not* specify design solutions (ref. [Grauling 89]). A cooperative effort by the contractor and the customer is necessary to plan the test approach early in the program, to execute the plan, and to iterate the plan as needed to incorporate lessons learned and more efficient techniques. This paper describes an approach being used by CCPDS-R and provides recommendations for establishing a large-scale Ada test program with the following specific objectives:

1. Maximize test efficiency by using test cases for multiple purposes (e.g., integration, software installation, regression, formal qualification)

2. Reduce integration schedule and risk in order to concentrate on the test portion of "integration and test"

3. Formally verify all requirements to enable timely selloff of the system to the customer

4. Formally verify requirements incrementally to reduce the risk of a single monolithic FQT period

Definition of DOD-STD-2167/2167A Terms

DOD-STD-2167 terminology is being used to describe the static CCPDS-R software structure. The CCPDS-R definitions are:

Computer Software Configuration Item (CSCI): A collection of TLCSCs, LLCSCs and Units that can be allocated to a single functional organization (i.e., skill center) to implement. For example, CCPDS-R has display, communications, system services, test and simulation, and algorithm CSCIs.

Top Level Computer Software Component (TLCSC): A component which maps directly to Ada library units or collections of functionally cohesive Ada library units. A TLCSC may contain nested LLCSCs and Units, and must be separately testable (termed "standalone test", or SAT). For management purposes, a logically related collection of TLCSCs within a CSCI is termed a "TLCSC Group".

Lower Level Computer Software Component (LLCSC): A program unit declared within a program unit (which could be either a TLCSC or a higher level LLCSC) that is sufficiently complex to require standalone testing prior to its inclusion in the standalone testing of its parent.

Subordinate Unit: A component of an LLCSC or TLCSC whose standalone test is wholly provided by the standalone test of its parent program unit. A Unit may also be defined as a library unit as long as its services are not shared across TLCSC boundaries.

The equivalent 2167A terms are **CSCI**; **Computer Software Component (CSC)**, which equates to a TLCSC Group or TLCSC; **Computer Software Unit (CSU)**, which equates to a TLCSC or LLCSC; and **Subordinate Unit**. Throughout the rest of this paper, 2167A terminology will be used, including software document names.

Development Approach Overview

The CCPDS-R software development approach is the initial application of TRW's "Ada Process Model" [Royce 1989-2], which is based on early definition, demonstration, implementation and test of incremental capabilities termed builds. DOD-STD-2167 has been tailored for CCPDS-R to accommodate this process model, including the incremental generation and review of the design and documentation products. A subsystem build consists of a collection of CSCs from one or more CSCIs which are integrated to form an incremental set of subsystem capabilities. Each CSCI is developed incrementally, with each CSCI build having its own preliminary design, detailed design, code and test cycle.

The builds are defined so that the foundation architecture components that are relatively independent of the required System Specification capabilities are developed, integrated and tested as early as possible, while the generally more volatile, application-specific components are allocated to later builds. The Ada Process Model requires that software capabilities be demonstrated at informal design walkthrough milestones and at formal review milestones to provide tangible evidence of design progress. Such reviews involving capability demonstrations provide a much sounder basis than traditional paper/viewgraph reviews for the customer and the contractor to assess readiness to proceed with subsequent development activities.

The CCPDS-R software design is described in terms of DEC VAX *nodes*, VAX/VMS *processes*, Ada *tasks*, and intertask communications *circuits* and *sockets*. The Software Architecture Skeleton (SAS) is defined and baselined early, and consists of the top level executive structure for all processes and tasks and their interconnecting circuits and sockets. The process and task executives are all instantiated generics, with the Ada source code produced by a tool which has all the architecture objects described in a database. The SAS concept enables rapid construction of a complete functioning network, which facilitates early discovery of design, interface and integration problems [Royce 1989-1].

The primary advantage of Ada in supporting incremental development as defined above is its support for partial implementations. Separation of specifications and bodies, packages, sophisticated data typing and Ada's expressiveness and readability provide powerful features which can be exploited to provide an integrated, uniform development approach. The uniformity gained through the use of Ada throughout the software development cycle as a representation format is also useful for providing consistent and insightful development progress metrics for continuous assessment of project status from multiple perspectives.

The software development phases of the Ada Process Model are:

- **Top level architecture design** of the foundation software components, resulting in definition of the System Global Interface (SGI) packages and the Software Architecture Skeleton (SAS). Also produced is the allocation of software for each CSCI to specific incremental builds to maximise early availability of functionality and minimise downstream breakage. Preliminary Design Walkthroughs (PDWs) are conducted during this phase for the contractor and the Government to periodically review the evolving top level design.
- **Top level design** for each applications build, which refines the overall top level architecture design and iterates the SAS/SGI architecture as the design progresses. An applications oriented PDW culminates this phase.
- **Detailed design** for each build, culminating in a Critical Design Walkthrough (CDW).
- **Implementation and informal standalone test** of all build components.
- **Turnover** of completed build components to the I&T organisation for formal baselining and test activities. The turnover process involves a significant amount of integration by the developers and testers as the software is built into a functioning configuration.

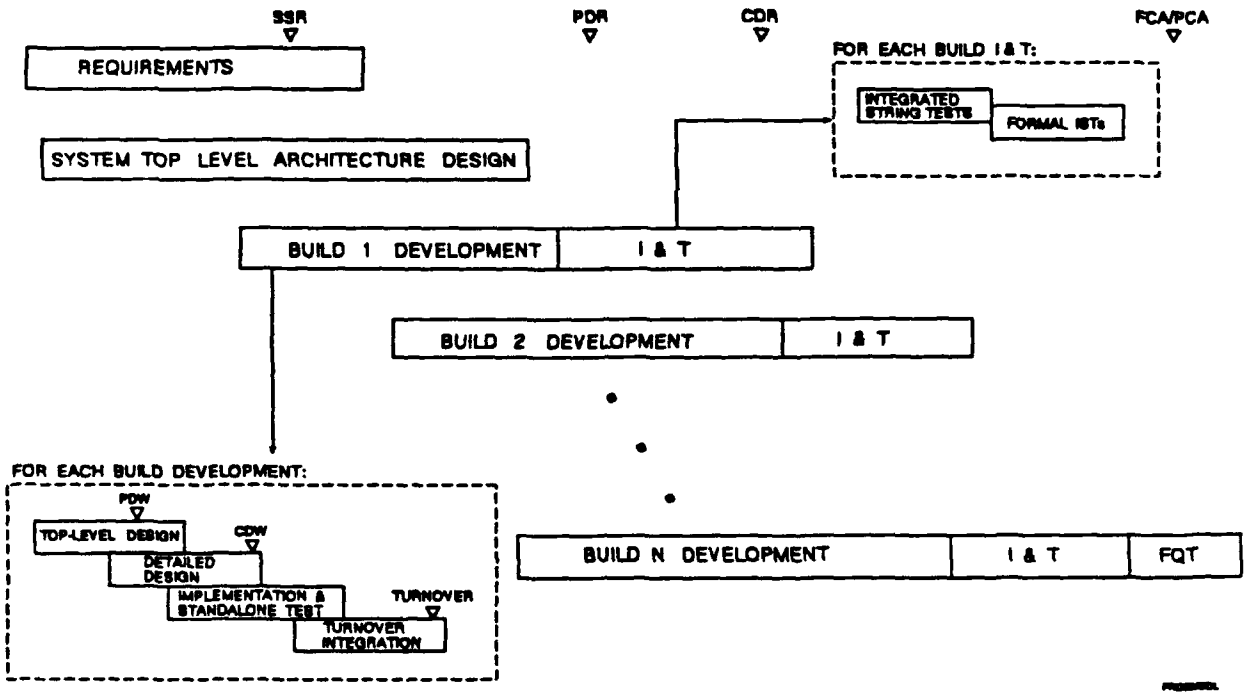

Figure 1: Ada Software Development Process Model

Test Approach Overview

The CCPDS-R software test approach maps directly into TRW's incremental software development methodology described above. Complete testing is performed per DOD-STD-2167A, featuring formal testing at multiple levels and by FQT of integrated CSCIs. The software test program consists of informal and formal testing (Figure 2). Informal testing is performed by developers and integrators to ensure (1) that individual components function correctly in a standalone mode and (2) that the integrated components function correctly in capability strings. Formal testing is the responsibility of an independent formal test organisation that verifies all software requirements are met. All testing is performed within a hierarchical structure termed the Ada Testbed (or simply the "testbed").

Ada Testbed Concept

The Ada Testbed provides the software execution environment and the software control environment for the physical control of all developed and test support software. The testbed is an environment where developers and testers can work in parallel against an established baseline. Its structure is designed to eliminate duplication of software among testbed users, minimise the software needed by each testbed user, and establish a uniform set of controls as the software moves from developer to baseline. The testbed is hierarchical, and consists of a predefined directory structure at each level, testbed build procedures, and support tools (Figure 3). Each procedure works within the hierarchy to find source, objects and

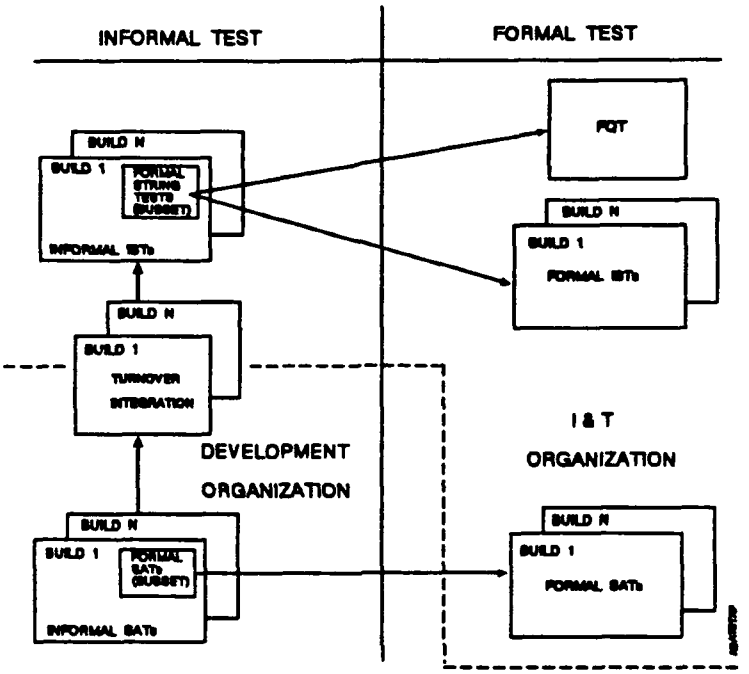

Figure 2: Software Test Approach

executables at the lowest level of the testbed. When a user wishes to access the latest copy of a file, the testbed will start at the lowest level and look upward until the file is found.

The build procedures utilise the VAX Ada Compile System (ACS) to provide a uniform method for compiling, linking and executing all software. ACS is structured around a library system. Every module that is compiled is placed in an Ada library. The Ada library is used to form shells around Ada objects to determine compile order and when modules need to be recompiled. The ACS COMPILE and RECOMPILE commands are used to automatically compile a module and all other modules that are associated with that module. This enables partial builds of the system instead of costly, time consuming full builds.

The software development organisation controls the lowest levels of the testbed. The lowest levels are where a developer implements and tests individual components, using higher level components brought down to the developer's levels as needed. The next higher levels are for turnover integration, and are controlled by a single individual who coordinates the turnover sequence of components from individual developer areas and orchestrates the compilation and preliminary integration of the total set of components. These levels serve as a staging area for the formal turnover of the software to the configuration managed baselined upper levels.

The testbed provides the physical configuration control of the baselined software. All changes to software in the upper baselined levels of the testbed are strictly controlled by the software configuration control board and a paper trail managed by the configuration management organisation. There is a single individual authorised to perform testbed builds for new software configurations. All formal turnovers

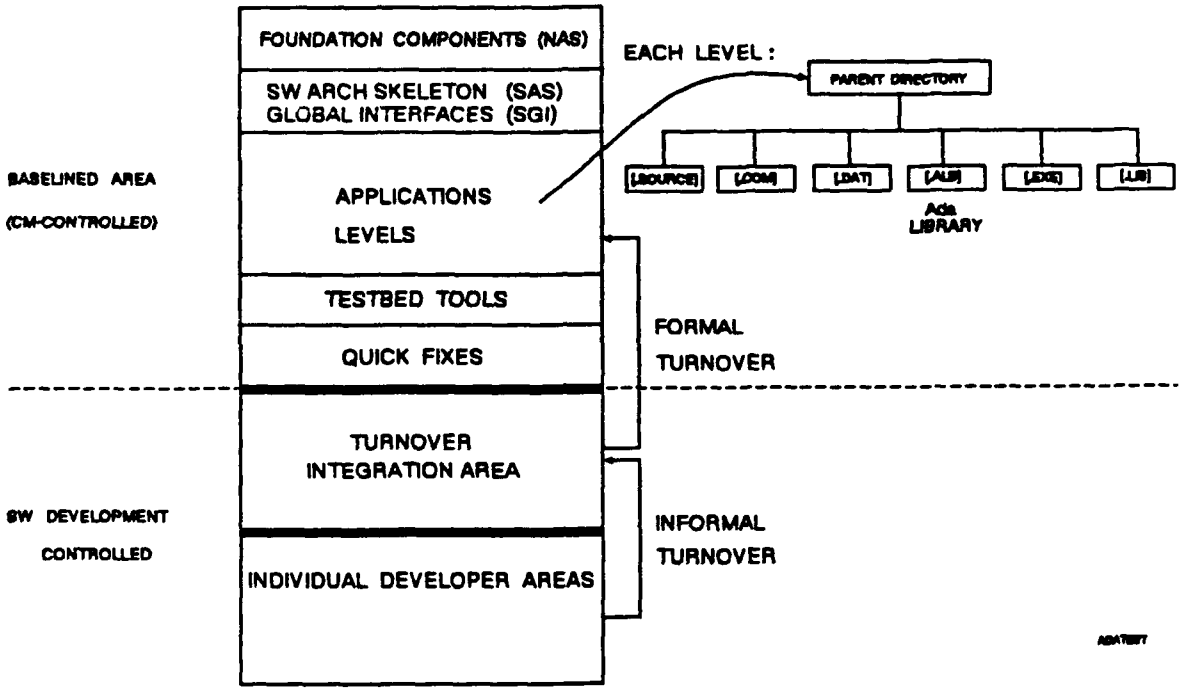

Figure 3: Ada Testbed Approach

and the products of a testbed build are audited by quality assurance personnel using various verification tools and manual analysis methods.

Informal Tests

Informal tests are performed by software developers and testers to debug individual components, check functionality at low levels of the architecture, and integrate components into functioning strings. The emphasis of informal testing is on thoroughly exercising the code through as many possible logic paths as possible. In a design such as CCPDS-R's message-based architecture, this involves inputting messages into each Ada task that cover the spectrum of possible input values. These are well-defined in the Ada System Global Interface (SGI) packages which are constantly visible to all developers and testers to support local testing. The informal test phases are:

Standalone Test (SAT). Standalone tests are the lowest level of test and are performed by the software developers. The term "standalone test" was created because the definition of "CSU test" as used in DOD-STD-2167A is ambiguous in a hierarchy of Ada "program units". Also, the distinction between "CSU test" and "CSC integration" as used in DOD-STD-2167A is difficult to define in the Ada process model. Standalone tests are performed on a CSC or CSU, each of which may be composed of multiple subordinate program units that are tested in the context of their parent CSC or CSU.

Standalone tests informally verify requirements and test off-nominal and boundary conditions in the developer's environment. The test procedures are written in Ada (wherever possible), with the procedures and test results included in the CSC Software Development Files (SDFs) [Springman 1989].

Turnover Integration. The bulk of what is traditionally called software integration is performed during the process of compiling, building and checking out the software in the turnover integration area (Figure 4). This results in a completely integrated, fully functioning set of software being turned over to the I&T team for exhaustive string testing. The Ada process model requires that CSC and CSU interfaces be defined and baselined early. These are maintained in System Global Interface Ada packages that are withed by interfacing CSCs and CSUs. As a build progresses, the developers are constantly compiling against the SGI packages as prototypes are built, the Ada Design Language (ADL) evolves into Ada, and formal and informal demos are integrated.

Throughout the Ada process model, there is a constant design integration which eliminates an entire class of interface errors that are normally not found until the I&T team attempts to integrate the software. This is facilitated by a combination of: (1) using Ada as the design representation as well as the implementation language; (2) rigid interface control through Ada type checking; and (3) the demonstration-oriented Ada process model.

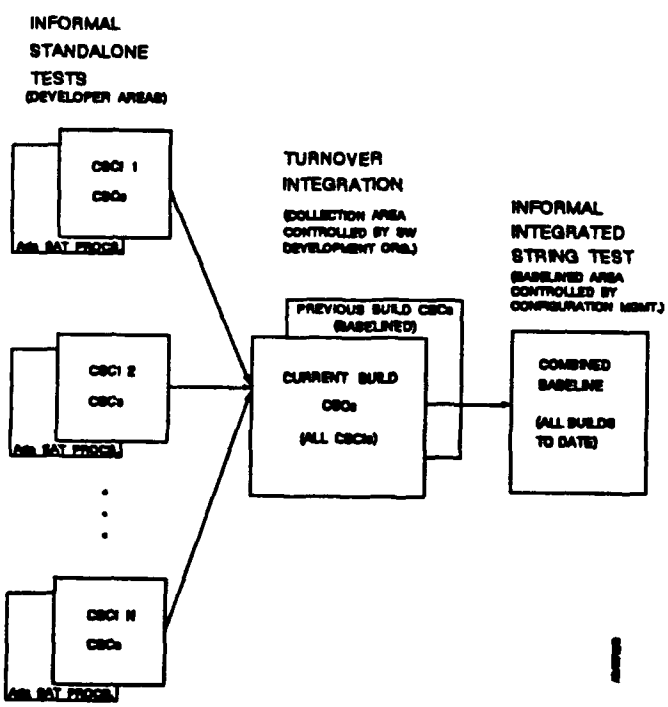

Figure 4: Software Integration Stages (Each Build)

Integrated String Test (IST). ISTs are performed after the build components have successfully completed informal SAT testing, have been informally built and integrated in the turnover integration

area by the development team, and are turned over to the independent I&T team. These ISTs are informal, and exercise strings comprised of components from multiple CSCIs that represent a required system capability. IST includes off-nominal, boundary and stress testing.

Software Reliability Assessment. In parallel with other integration and test activities, software that is already integrated and functioning is used as the basis for assessing the reliability of the system's software. This activity concentrates on the foundation components (e.g., Network Architecture Services and Software Architecture Skeleton for initial assessments, and then adds application components as they are completed. The goal is to execute the software in a stress environment using varying input scenarios to thoroughly exercise the logic over extended periods of time (e.g., overnight). Such testing will uncover errors that are difficult to detect in normal, human-attended testing, such as errors dependent on timing or input sequencing. By the time FQT has occurred, the software, especially the foundation components, will have been thoroughly stress tested, to provide a high degree of confidence in the reliability of the product.

Informal test procedures and results are documented in Software Development Files (SDFs) for standalone tests and in Test Data Files (TDFs) for integrated string tests and reliability tests. Integration testing is governed by a "Build Schedule and Content Plan" which defines the specific software contents of each build and the functional strings to be tested during each IST phase. This plan is closely controlled by the project so that the test organization is fully prepared for a software turnover.

Formal Tests

Formal tests are the responsibility of an independent test organisation. The purpose of formal software testing is to verify all software requirements. Ideally, all software requirements should be testable or demonstrable at Formal Qualification Test (FQT) using operationally produced outputs as success criteria. However, requirements definition is generally far from ideal, and waiting until FQT to verify the full requirements set is risky for both the contractor and the customer.

Recognizing that requirements vary in level of detail and that a single FQT is too unwieldy for a major program, an incremental requirements verification approach is being used on CCPDS-R. This approach verifies requirements at three levels (Standalone Test, Integrated String Test, and FQT), dependent upon the components and data needed to verify a requirement. In addition, the concept of *implicit* testing of lower level requirements at higher level string tests is being employed. The formal test levels are:

Formal Standalone Test: Verifies requirements at an individual CSU level (e.g., intermediate algorithm results, results not readily observable via operational displays, or detailed design requirements). The scope of verification at this level is highly dependent on the amount of detail in the requirements documents. Formal SAT cases are the responsibility of the independent test organization. They are specific informal SAT cases whose test procedures are provided to the customer for approval. For efficiency, they are executed by the developers of the software being tested, in a formally configured environment managed by the CM organization, and while being witnessed by I&T, QA and customer personnel.

Formal Integrated String Test: Formal ISTs verify requirements satisfied by multiple CSCs and CSCIs that can be tested or demonstrated using functional strings and operationally produced outputs for success criteria. These tests are performed by the independent I&T organization when an aggregate of software capabilities has completed informal SAT and IST tests. Formal ISTs specifically pertain to SRS requirements and are a subset of the informal ISTs. Formal ISTs are run in a formally configured environment and are fully witnessed.

Formal Qualification Test (FQT): FQT verifies all software performance requirements and other requirements not allocatable to prior SAT and IST levels. FQT test cases are generally ISTs rerun in the FQT configuration. FQT is run using the complete software and hardware configuration in a formally configured environment and are fully witnessed.

Implicit Test: For requirements that are associated with the specifics of the design (e.g., the method used to access a file that produces outputs visible on a display) or are purely specification entities (e.g., internal function-to-function interfaces), explicit verification is generally impractical and unnecessary. Such requirements are verifiable by executing a test that must perform processing associated with those requirements in order to complete successfully. The requirements allocated to implicit testing are verified at formal IST and FQT. For test traceability, the test case name that verified a requirement implicitly is sufficient for test audit purposes.

All formal testing is fully documented in accordance with DOD-STD-2167A. The Software Test Plan defines the scope of formal testing. Individual Software Test Description and Software Test Report documents are provided for each formal test level. Requirements are allocated to one of the three levels described above for verification and are assigned to specific test cases within each level. Test traceability is maintained in comprehensive Test Verification Matrices, which use traceability information generated automatically from the Software Requirements Specifications. Each SRS "shall" requirement is uniquely labeled by the documentation tools, and the traceability tools carry these labels down through the design and test documentation. As the SRS traceability changes, so does the rest of the traceability trail, ensuring that traceability information is always current and consistent.

An overview of a generic development and test schedule is shown in Figure 5. This schedule shows the time phasing of the development and test activities.

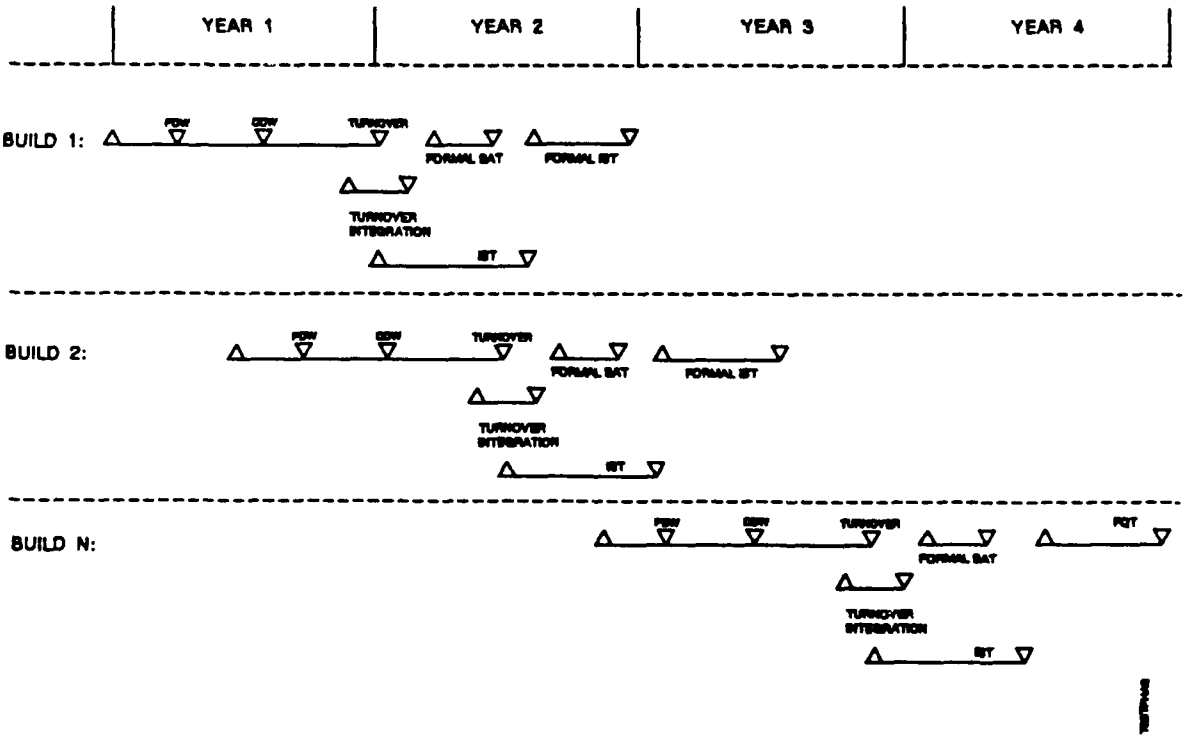

Figure 5: Generic Test Activity Schedule

Test Metrics

On a large Ada project with thousands of components and software requirements, it is essential to define and maintain metrics that: (1) bound the scope of the test program; (2) are readily reportable to management and the customer; and (3) are easily understandable. The test metrics should complement any development metrics used on the project. Examples of test metrics include:

- **Informal standalone test progress**, which measures the number of CSCs that have been standalone tested by the developers. This metric is part of the overall development progress metric (the "Tested" columns in Figure 6), which measures the software development team's progress in completing a defined set of software. The development is complete when: (1) all Ada Design Language (ADL) has been transformed into Ada ("Designed"); (2) all standalone testing is executed ("Tested"); and (3) all documentation has been generated ("Documented"). The "Total KSLOC" is determined by an Ada metrics tool that counts completed Ada source lines in the specification and body parts [Boehm/Royce 1988] and "TBD" (To Be Determined) Ada lines identified in ADL statements [Royce 1989-1]. A CSC contributes to the calculation of percent complete for "Tested" or "Documented" only when informal SAT testing or SDF documentation is 100% complete for the CSC. It's contribution is weighted by its size in Ada source lines. No partial test or documentation status is maintained because of the subjectivity of such status assessments.

- **Informal IST progress**, which measures the number of IST test cases and test steps that have been successfully completed against the plan (Figure 7).

- **Software requirements verification progress**, which summarizes the actual versus planned number of requirements verified at each formal test level. Figure 8 shows the CCPDS-R plan and status, with the six CSCIs as the columns. As shown, the first two SAT phases and the first IST phase have been completed, with some percentage of the requirements allocated to each of those phases accepted by the customer (e.g., 69 out of 76 for Build A2 SAT).

- **Overall test program status**, which provides a comprehensive status summary based upon the metrics described above and the cost/schedule earned value assessments for each test activity. Figure 9 shows each test phase for which the independent test organization is responsible, including informal ISTs and formal SATs, ISTs and FQT. The metric indicates the percentage of test cases prepared/executed and reported, with a composite assessment at the bottom. The vertical dashed line identifies the current date, against which progress is measured. The example indicates that informal IST3 is slightly behind in test prep/execution and SAT2 is approximately 2 months behind, resulting in a composite assessment of on schedule for informal IST and approximately 1.5 months behind in the formal verification activities.

- **Software Problem Report (SPR) summaries and history**, which indicate areas where test resources should be applied and where problem trends should be addressed. SPR summaries by test level also indicate the relative value of SAT vs. IST vs. formal testing, which can be used to adapt the test program as trends are discovered.

It is important to track metrics status against a plan. It is progress against the plan that determines whether or not management attention is required. This requires a realistic plan, which is not always easy to determine at the start of a test program. The plan must therefore be updated as required to enable accurate and meaningful status assessment.

CCPDS-R Ada Test Experience and Recommendations

Use of Ada as the CCPDS-R implementation language has had both positive and negative effects from a test perspective (Table 1). CCPDS-R's Network Architecture Services and Software Architecture Skeleton [Royce 1989-2] rely upon Ada generics, Ada tasking and Ada interface packaging to create an architecture that enables early development, integration, demonstration and test of foundation capabilities. Interface problems are discovered earlier in the development cycle. The Ada compiler identifies obsolescent modules requiring recompilation due to changes to other modules, which speeds up the change checkout process. Self-documentation features of the Ada language (assuming good naming practices) result in more documentation being included in the source and in Ada test procedures/drivers.

CSCI	Designed						Tested		Documented	
	Total SDFs	Total KSLOC	Complete (Ada)	TBD (ADL)	%	CM ADL ⇒ Ada	Tested KSLOC	%	Complete SDFs	%
NAS	47	18.6	18.6	0	100%	-.2	18.6	100%	47	100%
SSV	46	182	182	0	100%	33.8	142.6	78%	22	47%
DCO	17	40.4	39.6	.8	98%	.7	25.8	63%	12	70%
TAS	16	9.6	8.3	1.3	86%	.7	9.6	100%	16	100%
CMP	21	10.2	10	.2	98%	-.3	3.8	37%	8	38%
CCO	9	75.5	57.8	17.7	76%	2.2	12.6	16%	2	22%
Total	156	336.3	316.3	20	94%	36.9	213	63%	107	68%

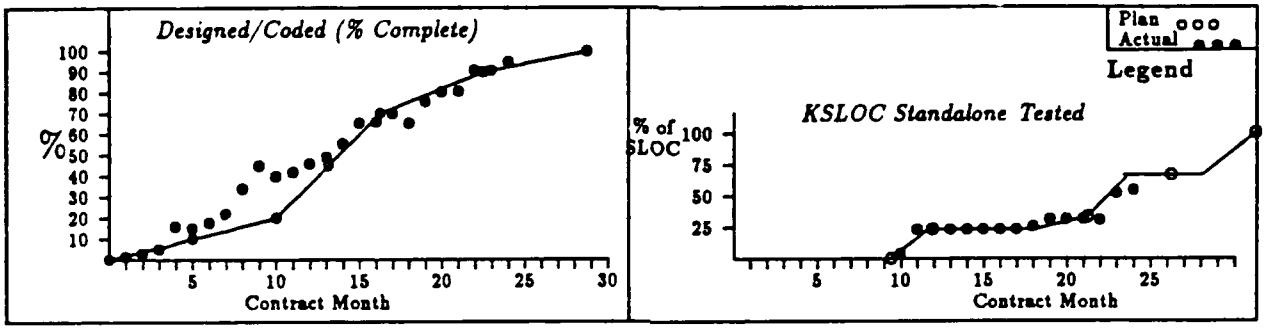

Figure 6: Overall Development Progress Metric

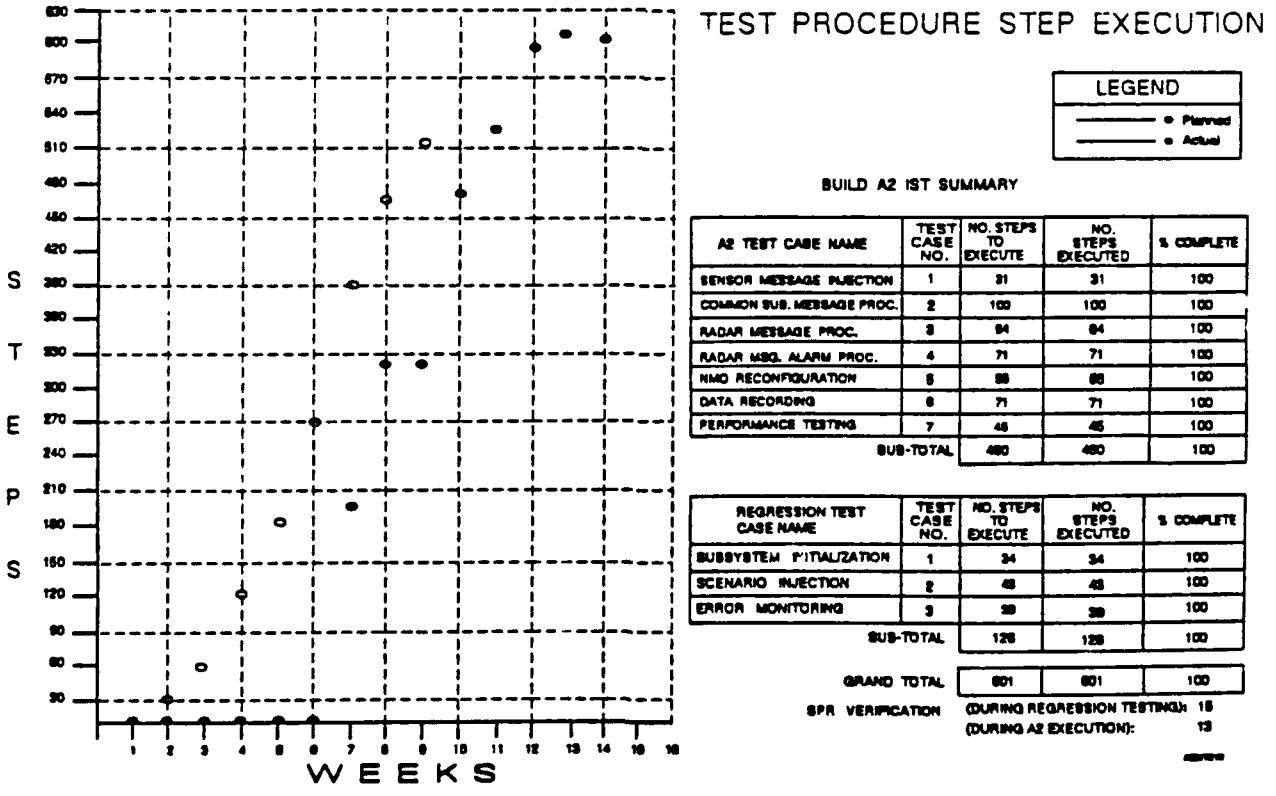

Figure 7: Informal Integration Progress Metric

TEST SOURCE	Completed/Total planned						
	NAS	SSV	DCO	TAS	CMP	CCO	TOTAL
Build A0/A1 SAT	48/49			5/5			53/54
Build A2 SAT		8/8	10/10	15/15	33/39	3/4	69/76
Build A3 SAT	14	126	58	14	56	4	0/272
Build A4 SAT			58		57	28	0/143
Build A5 SAT						5	0/5
IST 1	75/75		10/10	21/21			106/106
IST 2	59	42	64	82	6	8	0/261
IST 3	16	150	185	40	29	61	0/481
FQT	25	164	217	48	39	82	0/575
TOTAL	238	490	602	225	226	192	228/1973

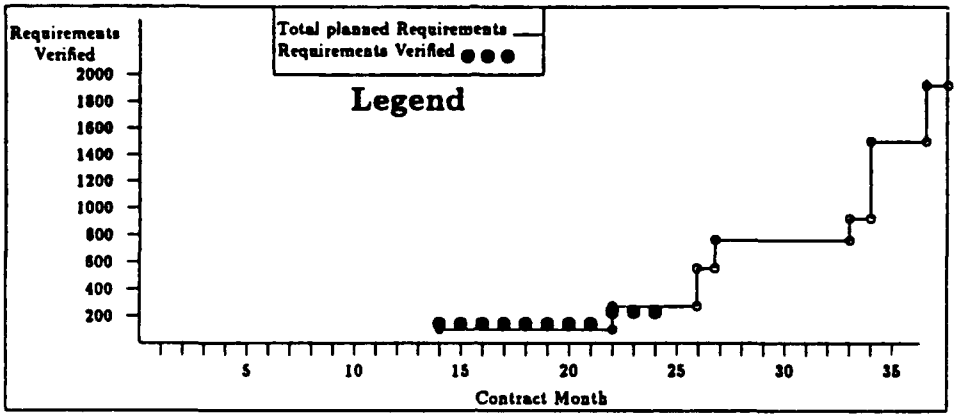

Figure 8: Formal Requirements Verification Progress Metric

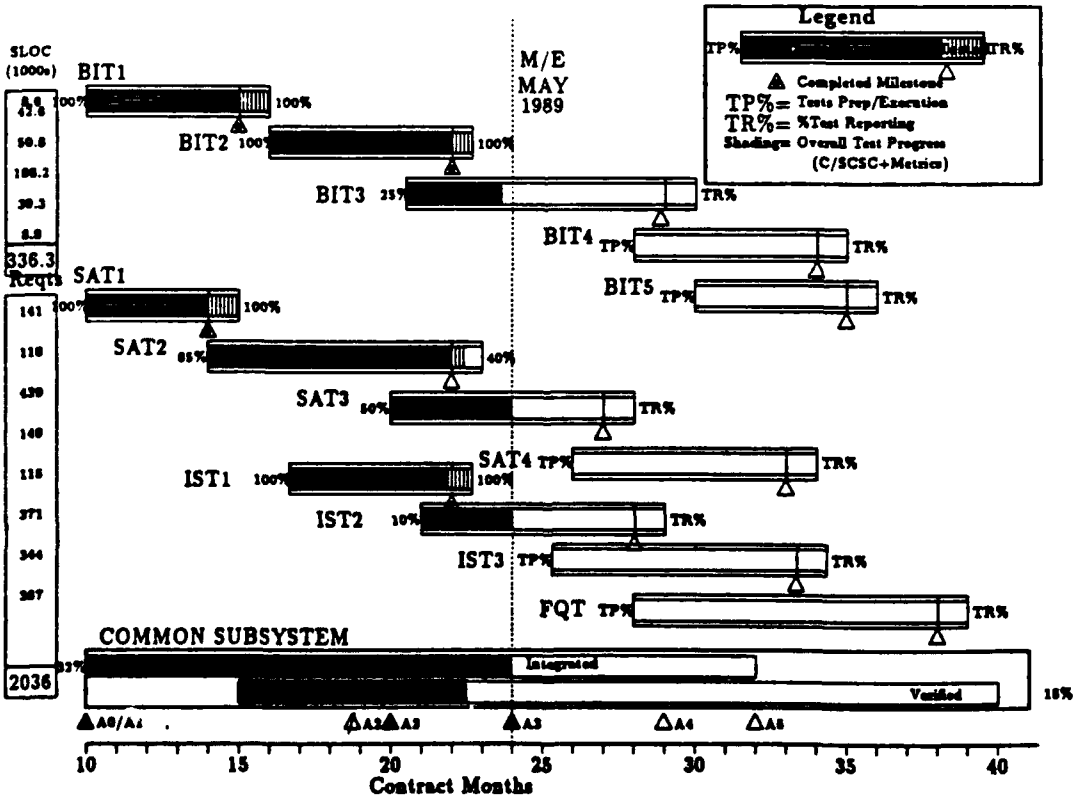

Figure 9: Overall Test Program Progress Metric

Table 1: Ada Testing Pros/Cons

Ada Testing Pros	Ada Testing Cons
1. Early identification of interface inconsistencies (compiler type checking)	1. Ada training required for testers and customer reviewers
2. Notification of obsolesced program units requiring recompilation	2. Extensive dependencies may result in frequent recompilation in individual user areas
3. Ada source code and Ada test procedures self-documenting	3. Significant disk space requirements for source and compiled products
4. Shorter integration timeline (see 1.)	4. Significant software build times (compilation)

Integration occurs constantly as the design evolves and the developers compile their ADL and Ada against the established global interface packages. Assuming the foundation architecture components (i.e., NAS and SAS) are integrated and baselined in early builds, applications components in subsequent builds are integrated relatively easily into the SAS. On CCPDS-R, a small team of 3-4 developers familiar with the overall software architecture has been able to integrate each build over a period of 1-2 months to a point where the test organization can begin working with a functioning testbed. This small team draws upon specific developer and tester expertise as needed to solve problems that arise during the integration. The latest build, consisting of over 150,000 Ada source lines, was brought into a functioning state in about 5 weeks.

On the negative side, Ada consumes significant resources for building and maintaining software configurations (i.e., testbeds). As incremental builds are completed and the number of files increases, the build time becomes lengthy and the disk space requirements grow rapidly. Currently, the CCPDS-R testbed consists of 270,000 source lines, over 6,000 files, testbed tools and build procedures, and associated standalone test drivers and files. The testbed currently takes 15-20 hours of CPU time on a VAX 8800 (using a single CPU) to build the complete software architecture, and requires over 600,000 disk blocks (512 bytes each) for storage. Only a portion of the build time (approximately 70%) is used for compiling and linking source code; the rest is for hierarchical directory searches, file difference checks, checksum compares, and other testbed functions. Experience to date with the DEC Ada compiler in the CCPDS-R testbed environment has shown compiler performance in the range of 500-900 source lines per CPU minute. This performance range reflects the complex and numerous dependencies among the software components for a system such as CCPDS-R. Testbed improvements are in progress to improve resource utilization.

Because Ada is used as a uniform representation vehicle throughout the design, implementation and test of the software, testers and reviewers (including the Customer) must be trained to read and understand Ada. This may result in higher up-front costs and initial inefficiencies as individuals become trained. Once trained, though, these individuals are ready to step into any subsequent Ada project with no problem.

An Ada implementation for a large application has extensive dependencies among modules based upon how the software is packaged. Care must be taken to avoid constant recompilation of individual developer and tester components as changes are made to higher level components. While developers and testers always want the latest software, constant changes to baselined software result in constant recompiles at lower user levels, which takes time and CPU resources.

The CCPDS-R software architecture involves many requirements and design components. With a total estimated size of 350,000 Ada source lines and 2,000 individual software requirements to be verified, the CCPDS-R test program is definitely an area of cost and schedule uncertainty. Recognizing this, both TRW and the Government have sought to ease the verification burden and spread the Government review/approval load by verifying requirements incrementally. For detailed design-oriented requirements, CSU standalone testing using detailed Ada test procedures is necessary, which requires a certain level of Ada expertise of the Government reviewers. Schedules have been adjusted to accommodate lengthened

review/analysis timelines. Also, the concept of "implicit" verification of lower level requirements at higher level string tests has been defined. The following recommendations and lessons learned have resulted from CCPDS-R's experience:

- Define the test approach early, and make it a major topic of the Software Development Plan.
- Get the customer involved early in the test program definition and get the customer to commit to a cost effective, bounded test program. Solicit early feedback and incorporate lessons learned into the test approach.
- Keep SRS requirements at a true requirements level. The more detailed and design-oriented the requirements, the more detailed the tests must be, and the more time required of the contractor and the customer to generate and review/approve the tests.
- Employ a design/development methodology that enables early and continual visibility by test and customer personnel into the software product. This can be accomplished through: (1) demonstrations of functionality; (2) early definition/baselining of system products (e.g., displays, report forms); and (3) incremental testing of software to enable early customer feedback on the adequacy and scope of testing.
- Devise an Ada software architecture that enables early prototyping and incremental demonstration of functional capability and integrated string testing. For example, instantiation of generic task and process executives with task-to-task interfaces enables rapid construction of a working SAS which can be used to demonstrate applications software.
- Define early software builds to: (1) baseline foundation components; (2) enable test of complete capabilities early; (3) minimize potential for breakage of earlier builds as later builds are implemented and tested; and (4) set a precedent using a small early build before attempting the larger later builds.
- Establish a comprehensive configuration management process that supports developers' and testers' rapid response needs, as well as the project's need to maintain strict configuration control of all baselined software.
- Prepare test personnel (both contractor and customer) for an Ada test program so that they can generate/review detailed Ada test procedures and the software under test.
- Use test metrics to define the scope of the test program and to measure progress against a realistic plan.
- Define the standards and procedures to be used for the development and test of the software as early as possible, preferably within 1-2 months of contract start. These should include documentation formats, naming conventions, header standards, and annotation standards.

Summary

This paper has discussed a test approach that is being used successfully on CCPDS-R, a large software project developed completely in Ada. The test approach has been modified and enhanced significantly as both TRW and the customer better understand the test requirements and implications of specific test techniques. Tailoring of the approach is necessary as experience is gained from the earlier test phases, and must be encouraged to achieve timely closure of the test process and enable cost and schedule targets to be met. Ada has proven to be a significant benefit in the incremental development/integration/test approach, particularly in enabling rapid integration of software from multiple CSCIs. This enables earlier, useful integrated string testing as each software build completes development and is turned over to I&T, rather than forcing the I&T team to undergo an extended period of debugging before exercising integrated string tests. More time is available for formal requirements verification activities, a traditional area of cost and schedule risk on major government software programs. Ada itself does not guarantee rapid integration. Sound engineering discipline is still required to define and control the software architecture and interfaces that are expressed in Ada.