

Using Formal Specifications as Test Oracles for System-Critical Software

Jon Hagar
Martin Marietta Astronautics Company
P.O. Box 179, M/S H0512
Denver, CO 80201
Voice: (303) 977-1625
Fax: (303) 977-1472
INTERNET: hagar@den.mmc.com
and
Dr. James M. Bieman
Colorado State University
Computer Science Department
Fort Collins, Colorado 80523
(303) 491-5792
INTERNET: bieman@cs.colostate.edu

Abstract

The process used to validate, verify, and test flight avionics control systems has produced software that is highly reliable. However, ever greater demands for reliability require new automated tools to improve existing processes. We used the Anna (Annotated Ada) formal specification language and supporting tool set to develop a Test Range Oracle Tool (TROT) to automate the testing of equation execution. Our approach fits within the existing testing process, automates perviously manual analysis, and can increase the level of test coverage. The TROT approach also introduces the use of formal specification languages and supporting tools to an existing industry program. This approach supported production tests and is being expanded into other test support areas.

Keywords: Formal Specifications, Verification & Validation, Testing Process, Industrial Software, Test Oracle, Process Improvement, Ada, Annotated Ada (Anna) Spec Language

1.0 Introduction

Software application domains such as flight avionics control systems require extremely reliable software because the failure of these systems can have severe human and financial costs. The Aerospace industry has been successful in producing such reliable software. However, the cost of producing even small systems makes expansion of capabilities, new code or updated systems difficult. Our organization continually seeks to improve the validation, verification, and testing process. One improvement is to increase the automation of the testing process. We have developed a technique using the Anna Formal Specification language and tool set that supports the creation of simple test oracles to check the correctness of equation execution [3]. We call these simple oracles Test Range Oracle Tools (TROT).

A major challenge in our development effort is to design a TROT approach so that it fits into an existing, very successful approach used in the validation and verification (V&V) of software used for flight avionics control systems [4]. Similar V&V approaches are used in numerous programs at various operating elements of Martin Marietta and other Aerospace companies. The V&V approach has been successful in helping to produce a system with no catastrophic software failures during actual use. A zero failure rate does not mean that software errors do not remain within the system or have not been encountered during software use, only that mission loss due to a software failure has not been observed. The current approach includes functional, structural, requirements-based simulations, and mission simulation techniques. These complementary techniques are applied to software which is in a maintenance life cycle phase. Using requirements expressed in an executable form has been a normal part of the process associated with both the functional and structural testing. For example, FORTRAN programs model parts of the required behavior of a guidance system. The use of requirements expressed in executable form and the actual software under test, essentially operates very much like an N-version programming environment that supports V&V [3,5].

We have recently incorporated requirements in an executable specification language to support our testing. We started with the Anna specification language because it supports Ada, a common language for many of our projects, and because a supporting Computer Aided Software Engineering (CASE) tool set is available. We have used Anna and its CASE set to produce a test specific support aid, TROT, which as a simple test oracle [1] can judge correctness of software results.

The current approach and TROT versions support design level specifications that accomplish simple testing at an equation level, line of code, or small code segment. The TROT approach has been evaluated for incorporation within an existing and proven environment [9]. Most recently, TROT has been used in a series of tests on an modified segment of flight software that subsequently flew.

In this paper, we examine the introduction of formal specifications as used in TROT to the existing processes. We first describe our current V&V process; outline the basic concepts of TROT; and then we describe how TROT is being incorporated with and improving our current V&V process. We also examine the TROT results that have been gathered during our evaluation and first time use of this approach. The paper examines some of the problems encountered during the effort to incorporate formal specifications within an existing software development process. And, since this is work in progress, planned efforts are also reported.

2.0 Current V&V Practice Methodology

Our current V&V approach involves different levels of testing analysis. At all levels of testing we use a variety of supporting software tools and metrics.

The most basic testing level is structural-based verification testing conducted on a digital simulator or hardware system, such as an emulator. At this level, verification testing (note:

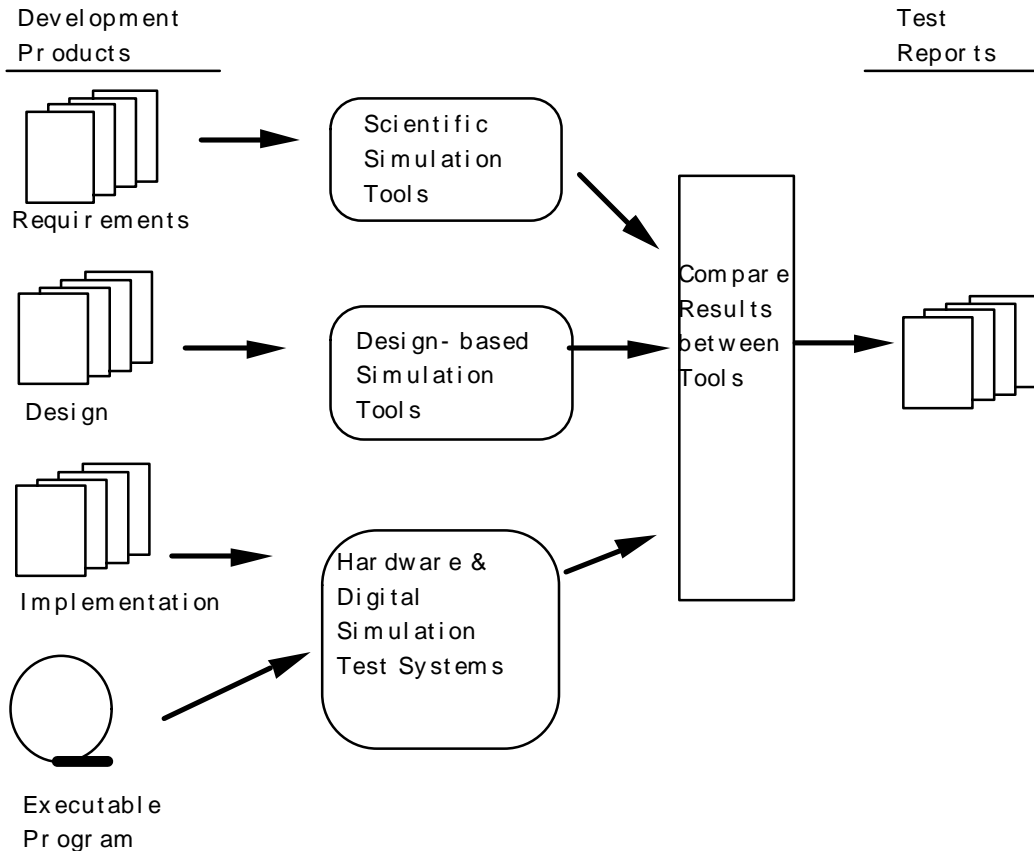
verification here is not formal verification by proof) is done to ensure that our code implements such things as software requirements, design information, and software standards. This testing is usually done at a module-level with small segments of the code being executed. Tools executing FORTRAN-like code generate comparable results that are available for analysis of individual equations and simple logic structure. The comparison and review of results is human intensive.

The next higher level, called integration testing within industry, uses tools and testing that cross module boundaries. These are design-based tools and, at this level, they simulate aspects of the system but lack some functionality of the total system. These tools allow the assessment of software for these particular aspects individually.

The next level is requirements-based simulation or what we call scientific simulation tools. These simulations are done in both a holistic fashion and individual functional basis. For example, a simulation may model the entire boost profile of a launch rocket with full 6-degrees of freedom simulation, while another simulation may model the specifics of how a rocket thrust vector control is required to work. This allows system evaluation starting from a microscopic level up to a "macroscopic" level.

At the system level, we test software with actual hardware in the loop. An extensive real-time, continuous simulation modeling and feedback system is used to test the software in a realistic environment. Realistic is defined here as the software being tested as a "black box" with the same interfaces, inputs, and outputs as an actual flight system. To test our real-time software system, we surround the computer with a first level of electrically equivalent hardware interfaces. We input signals into this test bed to simulate the performance of the system and hardware interfaces. The inputs stimulate the system under test which responds with the computed outputs. These outputs are read from the hardware into a workstation. The workstation software computes appropriate new inputs which are then fed back into the test bed. This arrangement forms a closed loop simulation environment that allows the software under test to be exercised in a realistic fashion. This realism is needed because the actual usage of the system is in space or a flight environment, such as zero gravity, that cannot be duplicated on the ground. In addition, unusual situations and system/hardware error conditions can be input into the software under test. The test system runs in actual real time, thus there is no speed-up or slow-down of the system.

A large number of the tools are simulations that are based on requirements or design information. The tools are stand-alone software programs (created by test engineers) that execute on separate platforms from the software under test. These tools take data that has been input to the software system under test, and produce computational similar outputs, which can then be compared to results generated by the actual software being tested. Some of the tools simulate individual equations or logic sequences, while other tools simulate aspects of the entire system. Scientific simulation-based tools provide success criteria or analysis capability that allow engineers to judge the success of the software under test without relying entirely on human judgment. These software testing tools simulate various levels of abstraction (see Figure 2.0-1).



2.0-1 Figure -- Tested Products

This system of simulation tools and executing software under test form the equivalent of an N-version programming system. In one of our project areas, we have a minimum $N = 3$, and a maximum $N = 8$ during various test cycles. Thus we compare and have check points of at least “N” different programs and/or levels.

No in-use catastrophic software failures have occurred when we have fully applied our basic V&V approach. (Note: Some software errors have been seen but software redundancy, fault tolerance, and "safing" which are required as part of the software, protected the system. In spite of our success, we are still looking for improvements in our processes and the way we do testing.)

Although successful, our approach has disadvantages including: humans must translate requirements into a standard executable programming language; no function exists within the programming language for automated compares; limited facilities existed within the standard programming language for semantic checks or formal verification; and the current translation and compare processes are prone to errors resulting in either missed problems or additional effort during test development. To address some of these, we became interested in expressing requirements in formal specification languages. Introducing formal specifications into a maintenance-phase production environment and process is being done slowly and incrementally. The risk of unproven tools, methods, and languages can be offset by a slow evolution and process improvement. The small incremental approach we are

following for incorporating formal specifications has allowed us to proceed without much controversy or cost impacts to the existing processes. In fact, we have actually saved money by practicing reuse and using commercial products. The incremental approach has as one ultimate goal the introduction of formal methods into common use during full development phases.

3.0 Introduction of Test Oracles

Our current V&V environment provides some level of automation in testing. Tools to automatically determine the correctness of program output during testing can dramatically increase the level of automation [1, 3]. Such tools are generally called "automated test oracles." An oracle is a tool or technique that will determine the correctness of execution results for input-process-output operations. Through TROT, we are investigating the use of formal specifications to produce test oracles that increase the test coverage levels and have the advantages of formal approaches, while reducing or keeping constant the "costs" of doing testing.

Using TROT, the correct execution of software at the equation level and statement level is automatically determined from a set of executable formal specifications. Formal specifications define "what to do" requirements in a clear and unambiguous fashion. Specifications express information about a program that is normally not part of the program [6 and others]. TROT code, which is written in Anna, takes the form of a special test script that, once run through the Anna tool set and an Ada compiler, forms a test oracle. TROT specifications themselves are derived from an English requirements document, but are check-able by the Anna Tool set.

Anna is a language extension to Ada that allows the formal specification of the intended behavior of programs [3]. A supporting tool set that was developed at Stanford University [7] is available for Anna and has sufficient capabilities to support TROT. The Anna tool provides a fairly robust environment and is capable of transforming Anna constructs into executable programs. The basis of Anna is in-line code annotations. Code annotations (Anna code) are essentially comment fields within Ada programs. An example of an Anna code fragment is shown in Figure 3.0-1. Annotations are converted by the supporting CASE tool set into Ada code that can be compiled by a normal Ada compiler. Code with the Anna-Ada code inserted into it, is called a "self-checking" program, because each time an annotated code sequence is encountered, it is checked for correctness to the associated code.

3.1 THE TROT SUPPORTING SYSTEM

The TROT supporting system we developed has two major components: the test environment and the analysis environment (see figure 3.1-1).

In the TROT approach, the software under test is executed on a hardware-based test system that has low-level CPU monitoring capability. In our current version, a Software Analysis Workstation (SAW) system [2] monitors the CPU while a second computer controls the CPU of the software under test. This test environment is made up of hardware

and software components that are detailed in [3]. Assembly language trace and result files are generated in the test environment on the SAW and transferred to a separate workstation analysis environment for post-processing by the TROT test case software. A trace file contains the following types of information:

- executed assembly language statements via a disassemble and non-intrusive hardware logic probe on the CPU bus;
- timing numbers (each statement is time tagged because the SAW's clock is faster than the system under test);
- memory addresses accessed and changed (memory inputs and output results); and
- computer status information.

This information is then available as an ASCII file for post processing by the TROT test oracle. A series of runs with different data and associated traces can be made and retained in this fashion.

The TROT test cases are created from English requirements by a tester using the Anna specification language and some supporting Ada code. TROT test cases are run through the supporting Anna tool set to produce a program capable of automatically analyzing the SAW trace files for correctness. Correctness is defined in terms of requirement expectations when associated with inputs that were initialized into the software under test.

An executable TROT test case is a program produced from an Anna specification and supporting Ada libraries. The TROT test case starts from a standard template that:

- initializes files used in processing;
- reads in needed files;
- provides access to the variable being tested;
- provides numeric conversion utilities;
- provides the base Anna requirement that is being tested;
- provides a reporting mechanism; and
- handles exceptions raised by the Anna logic.

```
.....Ada Software associated with the Specification.....
-- Names of months do not contain the letter x
--| for all X : MONTH range JAN .. DEC =>
--|   for all index_month : 1..MONTH'WIDTH =>
--|                                     MONTH'IMAGE(X)(index_month) /= 'X';
....More Ada Software....
-- Initialization of a Variable
--| B5_Guid_Update = 3.0
```

Figure 3.0-1 Anna Code Listing [7]

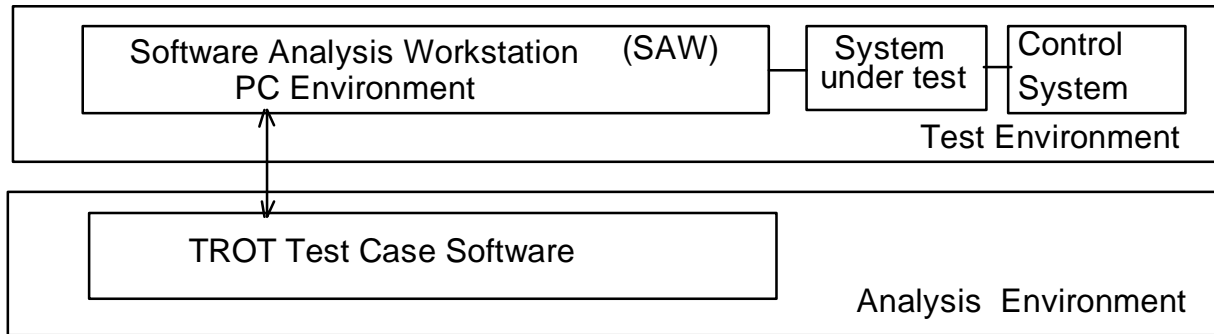


Figure 3.1-1 Major Components and Sub-components of the TROT System

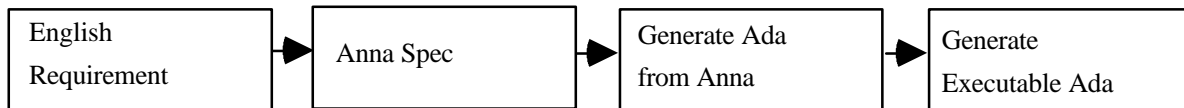


Figure 3.1-2 Production of a Test using Anna

The tester identifies code variables under test as part of the Anna specification. TROT logic automatically determines if the specified conditions of a variable are met. The compare capabilities are a normal part of what the Anna tool set provides. For variables that pass the annotation's specifications, a nominal report is issued at the end of TROT test case processing. However, if an annotation is violated, an Anna exception is raised. These are handled by Ada exception handlers which can trap the violation and issue a failed test report.

Conceptually, the TROT process can be viewed as extension of N-version programming practiced already in our V&V process, whereby two or more versions of the software are created and compared. However, in the TROT system, one version is "how-to-do" code and the other is "what-to-do" specifications. This is slightly different from the normal concept of N-version program, where two or more versions of software are created by different people or groups, possibly using different systems and languages. A criticism of N-version programming is that typically all programs use the same requirements[5], and little or nothing is done (in terms of formality or checking) with the requirements. And, if there is an error in the requirement, all "Ns" suffer the same fault. In TROT, the "what-to-do" Anna specifications, with supporting software, are transformed by the Anna system into an Ada program that can check the functionality of the "how-to-do" code under test. To do this, the TROT test program must be expressed with the required functions under test and the required constraints on these. This is an improvement over our current system since we express much of the requirements in a formal specification language rather than using a human process of deduction and interpretation which are needed during the normal programming process. Expression in formal specifications also allows some formal reasoning about the specification, which the Anna tool set supports to a limited extent, and was used in our evaluation. These include syntactical and semantical checks of the

specification, and limited formal verification. The addition of formal reasoning about the specification is an improvement over normal “N” version systems, alleviating some of the criticisms. Expansion by using additional Anna tools or different formal language systems will be a natural progression of our approach.

Finally, the TROT code does not impact the timing and memory of the system under test, since it is used after the software under test is executed. It is thus a non-interference method of test analysis. The Anna specifications are feasible in a test environment because they do not have to function with the same timing and memory constraints as the software under test, and thus can be inefficient in these areas, i.e., it would not be suitable for actual flight use.

4.0 EVALUATION, USE, AND RESULTS

We have gone through a process of development-prototyping, evaluation, test and finally simple use of the initial TROT implementation on actual flight software. Because of the high level of confidence in existing and proven methodology, new tools and techniques must be evaluated extensively before they can be incorporated and trusted. New techniques cannot be included into a proven methodology and applied to test new systems unless we can show that the new techniques will generate correct results and detect the same kinds of errors as tools they may be replacing. After a series of prototypes and a complete TROT development cycle, our evaluation period involved re-execution of already tested logic, testing of previously found errors, error seeding, and step-wise incorporation of TROT-Anna prototype constructs. First, we evaluated TROT in testing initialization routines, several guidance equations, simple logic structures (such as, “IF”) and math utilities. We did not intend for Anna-TROT to test more complex things at this stage of development. After evaluations of the system, we applied TROT to production test of actual flight software.

4.1 Evaluation of the Approach/System

The first phase of Anna-TROT development was to understand and demonstrate the basic capabilities of the Stanford Anna system. After the Anna system was installed and ported to a local computer system, a series of stand alone programs written in Anna and Ada code were generated/executed. Programs were written and the results generated without interfacing to the SAW/Control-System system. We did this to understand the basic capabilities of Anna and its tool set, as well as demonstrate that the Anna system was functional. We used examples and regression sets provided with Anna as well as test programs that we generated. When this testing proved satisfactory (see Table 4.1-1) and showed Anna's tool set was working on our computers, the next phase started. We then began to create interfaces between the Anna system and the SAW/ Control-System.

The Anna system is a prototype tool, not a fully functional system. The limitations of Anna, however, did not significantly encumber our work, and our initial evaluations indicated that it would support our limited analysis and future plans. More extensive use of Anna may require upgrades to the tool set.

The second phase of Anna-TROT prototypes used Ada code-logic to access trace data generated by the SAW/Control-System systems. SAW trace files were generated for a segment of code of the software under test using the Control-System to initialize and control the target computer. The SAW captured the results and saved this data to a disk file. This data could then be compared to the Anna specifications after it was transferred and read into the TROT test program.

We started with simple initialization routines captured from the SAW testing very simple Anna specifications, Ada program support libraries, and short SAW traces. The Anna/Ada test programs did “pass” a series of five separate SAW traces corresponding to five different initialization routines. This test demonstrated the logic needed to read and process SAW trace files.

After tests of the initialization routines succeeded, we tested a square root function and a simple math equation using SAW trace files of the software under test, which had been initialized to known states. The SAW and state information were passed to the Anna-TROT test case for evaluation. For a series of different data sets, TROT indicated the code met specifications, which was expected, since this code had been tested extensively and was shown to be very robust.

We next tested the capabilities of Anna-TROT to detect erroneous results from the software under test. But, since the software under test did not have any known errors (as previous testing, analysis, and use had revealed years ago), we “seeded” SAW results with errors.

We tested a square root function by seeding SAW traces with resultant values that were just above, below, and within the required resolutions. During this testing, all seeded errors were detected and acceptable values passed. We repeated this approach with a series of runs with a simple mathematical equation function and received similar results.

Once the error seeding approach had shown the feasibility of the Anna specification to detect numeric computation errors, we then sought actual past code problems that were found previously by V&V efforts. Documentation problems or problems that did not impact the requirements or the software code products were eliminated as candidates for Anna-TROT, since our approach is not intended to detect documentation errors. We identified a set of problems that were found by the tools that TROT was being built to replace. We used these past faults to generate trace and result information from the SAW. These were then submitted to Anna-TROT programs for analysis.

As seen in table 4.1-2, TROT detected all of the code-based errors, except a data dependent “divide-by-zero” condition. One error concerned program-compilation where an “IF” logic had been coded incorrectly. The expected computation of a variable was not produced in the results for a given set of inputs. This was detected by the Anna-generated logic and an exception was raised. Another error concerned a variable that was accessed by a wrong index. This caused the code to access the wrong memory location. Again, Anna-TROT was able to detect this memory access problem. A third problem concerned a

condition where the requirement and code did not match. This is a typical test-verification kind of problem and was exactly the type of problem Anna-TROT was designed to find.

Table 4.1-1 First phase of Anna-TROT Prototypes

Function Tested	Test Runs	Tool Success	Notes
Anna Regression test set	~ 300 sets	see notes	This set of tests demonstrated the functions needed by us were working, but showed some Anna functional areas were either not yet fully developed or functioning on our system. We determined that this was not a hindrance.
Variable Initialization	8	passed	Constraints placed on variables and tables being initialized to a constant value.
Numeric equation constraint	4	passed	Constraints placed on variables computed by a simple mathematical function with several inputs
Square root	8	passed	Constraints on a floating point math function, tests included 0 and min./max. numeric values
Sort function	4	passed	Numeric sort function tested with several different data sets, no erroneous results were tested (sort function/Anna Specification., provided with Anna tool set)
Membership function	1	passed	Member of set function tested with no erroneous result (function provided with Anna tool set)

Note: In each data set, at least two runs were made with Anna. Nominal, as well as error conditions, were executed. Success is defined as the tool detecting a difference between code and the specification. This condition raises an Anna-Ada exception, which can be propagated externally to the program or can be handled within the program.

Table 4.1-2 Second phase of Anna-TROT Prototypes

Function Tested	Test Runs	Tool Success	Notes
Initialization function	5 modules	Passed	Exact equality required
Square root function	10 data sets from SAW traces	Passed	Values as calculated by software under test
	10 data sets with values exactly equal to high resolution requirements	Passed	SAW trace file used with a seeded resultant
	10 data sets with values exceeding high resolution requirements	Passed -Error detected	SAW trace file used with a seeded erroneous resultant
	10 data sets with values exactly equal to low resolution requirements	Passed	SAW trace file used with a seeded resultant
	10 data sets with values exceeding low resolution requirements	Passed -Error detected by TROT	SAW trace file used with a seeded erroneous resultant
Simple Floating Point Numeric Equation	10 different data sets	Passed -SAW and TROT matched	Input conditions/values were picked by engineering analysis at high/low and significant data points
Incorrect IF logic	1 data set picked and ran with SAW trace that contained the error	Passed name exception raised	Old code error reproduced
Incorrect memory index (memory leak)	1 data set ran and demonstrated the error in the SAW	Passed -Anna exception raised	Old code error with a specification of variable result range and resolution requirements.
Guidance equation accuracy	2 data sets ran and demonstrated the error in the SAW	Passed -Anna exception raised	Old requirement and code problem replicated in Anna-TROT system
Data Input Precondition problem	4 old tests were run with an Anna-TROT test that had a Precondition specification	Passed -Anna exception raised	Input range values were constrained based on other inputs
Divide-by-zero equation problem	10 data sets (data from original tests) were executed on the SAW	Failed (see text)	Anna-TROT failed over several data sets to find problem

In these past history “error” based runs, we knew the problem and the “offending” conditions ahead of time and had test cases and data to expose them. Such a prior knowledge is not typical of the test process, however in our testing we replicated these inputs and conditions from earlier testing. We only needed to run one or two test cases for each of these tests. Once one test case detects a problem, we did not see any benefit of running additional cases. When a tester observes an anomaly detected by a test case, he or she stops the testing and analyzes the situation to determine the source of the problem. Finding one error-producing case is a sufficient stopping condition when doing error-based testing.

During this process of re-running prior error detecting tests, we also ran a test where we used Anna to establish input pre-condition specifications on a variable. Such pre-condition testing was not part of our original objectives, but we were able to flag an error based on invalid input states with a TROT test case.

We ran a final test using a previously found error that involved a divide-by-zero condition. The original tests on this code by the existing V&V tools failed to find this problem. A time related function had been assumed to have a hardware-based sampling rate that prevented the function from ever returning a zero value. The divide-by-zero problem was discovered during a complete hardware-software system test because under certain hardware conditions a zero can be returned. Since the prototype TROT approach does not employ random-based testing and was done with the same inputs as the original tests, the same inability to find this condition exists. However, TROT could be used to support a random-based testing scheme, employing a much larger set of input conditions. This should have the same probability of detecting an error of this type as did the system-level testing that by chance uncovered it.

4.2 Production Use of TROT

Before any large-scale use, we desired to accomplish an initial production evaluation. Thus, we applied TROT to a fairly simple production problem---a system modification completed in response to a problem report.

New logic was added to the software to correct a minor problem noticed during flight. A simple boolean expression was introduced which affected a subsequent “IF” test. Our testing was to demonstrate the correctness of the equation and the resulting path selections.

A sequence of 8 test cases based on operational use profiles were designed. The test cases corresponded to nominal and failure conditions of the hardware and represented a sampling of the expected conditions that the system could realistically be expected to encounter.

An Anna specification was created, and a TROT program derived from the problem report and resulting design change information. During initial development of the TROT program the Anna tool set detected and indicated data type problems in the software under test. The developer of the target assembly language software had mixed boolean and integer

types/operations (not directly a problem in the software but not good programming practices either). The TROT program was adapted to handle this problem, and the issue was reported to the developer and customer for possible correction. Once completed, the TROT specification defined the boolean computation and the expected path results. A partial sample of the TROT program is given in Figure 4.2. This code once compiled by Anna and Ada checks that the Anna Spec from a requirement document is met by SAW results. This code (for all...) requires that Anna function always be met by the output.

```
-- Ada Software associated with the Specification.....
--: function Req_equation return Integer is -- Anna from the B5 Requirements
--:begin
--:return Integer((((ZF4RCOM and LRWC) xor ZF4RSFB) and ZF4RSFB) or
JF4RSFA);
--:end Req_equation;
.....
-- All test results outputs from SAW must equal specified equation.
-- Last 2 Anna constraints on SAW output range is based on design range info.
--| for all Req_equation : Integer =>
--| (Output_Variable_Name.Numeric_Value = Req_equation) and
--| (Output_Variable_Name.Numeric_Value >= 0) and
--| (Output_Variable_Name.Numeric_Value <= 65535);
```

Figure 4.2-1 Anna Code

After successful creation and compilation of the TROT program, the eight tests were run using the SAW/test environment. The tests were initialized into computer memory and the tests executed on the module of code under test. SAW trace files were captured and then transferred in mass to the software analysis environment

TROT was then run against the eight SAW trace files. Each file was checked by the TROT program, confirmed matching requirements, and resulting test documentation was produced. Additional analysis confirmed the correctness of both the code and TROT.

Next, as part of on-going efforts to test TROT itself, the new code logic was mutated by hand. We introduced two different boolean structures. The eight data cases were then re-run resulting in different computational results. The TROT program was able to detect the errors in each mutation (an Anna exception message was raised).

Our success in using TROT in small-scale production applications and the evaluations gives us the confidence to proceed with more ambitious plans.

4.3 Problems During Anna-TROT Evaluation.

An odd problem is that our current software is "ultra" reliable. Even going back to the earliest versions of this software, the actual numbers of errors are relatively low. Thus we needed to seed errors to support our evaluation. Also, the collection of "real" test data and problems will be a very slow and time-consuming process. These delays in evaluation lead to delays in the introduction of formal specifications into our existing V&V process.

Introducing formal specification techniques to the engineering staff has taken much patience. Formal specifications are new to them. To use Anna, the staff required detailed training. In addition, management needed to be convinced that this new approach could save time and money while achieving equal or better levels of software testing. Our efforts to solve these problem have been aided by the slow and incremental approach.

The Anna tool set, while remarkably robust for "academic-ware", has had problems. These problems include some features, not used by current TROT versions, that are not fully implemented; some supporting tools which we could never fully make operational; and some functions that were misleading. The tool set is a possible candidate for re-engineering and upgrade to support a full industry environment, however, it has been sufficient to generate interest and support our efforts to date.

Although the introduction of TROT has been a success so far, several risks remain. The risks include: "scaleability" of the TROT concept to higher levels; continuing engineering staff acceptance; reliability of the Anna tool set and TROT test cases; the ability to automate additional aspects of the current TROT approach; and the application of the TROT concept to other programs and software areas. These remain topics for additional work.

4.4 Potential Benefits and Generalizations.

There are two main interests in software requirements conformance testing. One is that the software meets or conforms to the stated requirements. Second is that anomalies present in the software are detected and removed. The second is difficult, and should be a major emphasis of a complete test program. Our initial testing and evaluation suggests that the TROT system can replace the existing design to code, low-level verification tool which was used as part of a conformance testing process. The TROT approach should be able to:

- achieve the same or better coverage levels in testing;
- offer more automation;
- run and analyze results from the actual hardware versus digital simulations;
- base tests on formal specifications that are derived for requirements/design information (which is at a higher level of abstraction);
- avoid using any software instrumentation or intrusion;
- support formal reasoning about and verification of the requirements themselves; and
- offer detection of code anomalies by comparison to the specification/ requirement.

The errors we detected extend to both code and simple logic errors, as well as problems in requirement range/resolutions. Also, the increased rigor in test development proves useful in better testing. For example, during the square root testing, reviews of the tests using existing tools and processes exposed deficiencies in the old test cases. These were simply added and accomplished using the TROT version without any extra time in TROT test development. Expanded future versions of TROT or the TROT concept offer more coverage of requirements compliance testing, design verification, and anomaly detection.

A main contribution of this work is the use of a stand alone executable formal specification system to produce a test oracle and analyze software under test, where comments are executed separately, even on different computer systems. This is achievable in other test environments, since there was nothing unique or really customized in our system. The concept we outline in this paper should prove useful in testing other embedded and real-time computer systems where the “overhead” of self-checking code and embedded formal specifications are too great or risky. We used a generalized specification language, Anna, and a commercial system, the SAW. And, while some customization of software was needed, these customizations were made in library structures or small stand alone programs, which could be easily replaced, updated, and improved. Substitutes for hardware, software, and languages are quite possible. We completed the use of this system with a very simple production test which basically implemented a test that was very similar to our evaluation tests. We are now working to expand this approach to another new test environment which is currently under development.

Our demonstration that formal specifications can be used to generate test oracles should be of great interest to the software testing community, who have been looking for ways to improve the test process. We have also introduced formal specifications and oracles into existing industrial software test processes. Our experience here is that this kind of change and introduction will have to be done slowly and with small steps, if it is to be accepted. This is done by not introducing all the functions and features that formal specifications might offer right away.

TROT has also proven useful in finding problems in requirements. Many of the current English specifications are incomplete or designs have "derived" requirements information. During the development of several equation-based TROT test cases, these requirements/derivations were identified and clarified. In one case, clarification involved understanding the resolution provided by the actual code and then "backing" this into a requirement. Also, there was the identification of boolean-integer mixing. These were a natural out-growth of the detail needed to write an Anna Specification. In our case, the additional information was noted with the development team and test information, but did not result in any code changes. This contributes to improved longer-term maintenance of the software.

4.5 The TROT Approach Future

Our initial evaluation and use of TROT is based on a small set of test cases with very simple Anna constructs. This evaluation is limited, in part, due to limitations in the current TROT version [3] and our concern about quickly changing our existing V&V process. Thus, we planned a slow implementation, evaluation and then simple use of TROT. One of us (Hagar) is currently leading a group that is working on an expanded version of this technique.

We are developing plans to use TROT to evaluate a future software system. Future systems will have unknown errors and the TROT concept will be used to verify design requirements to code implementation of several hundred modules. This verification will be done at the equation level as well as expanded levels. Integration of commercial products,

for example Cadre Tools, Computer-hardware simulators, in-circuit emulators, and automated driver/stub generation tools, with the Anna or other formal specification languages is being pursued. We are considering the substitution of Anna with systems like ADL, Z, or other formal specification languages. Candidates are being studied now and will be selected on their ability to fit within our approach and on their supporting tool sets. Our approach is not fundamentally tied to Anna, and we have used object-oriented systems analysis to support development, so the substitution of other languages and systems should only require the modification of interface libraries.

We also plan to assess the impact of TROT on our existing V&V system. This quantitative analysis will be based on the time to prepare a test case and time to analyze test results.

Once we have the above measures, we can make comparisons to the historic numbers from the verification tool we are replacing. We will begin collecting data when we have actual new production code. Until then, evaluation and certification of TROT will continue to take place in preparation for its production use. We also expect that the use of formal specifications will increase the testing coverage level while reducing or keeping constant the "costs" of doing testing.

Once a TROT-like system is in use, we will look for improvements in our V&V process resulting from the introduction of formal specifications. Additionally, we anticipate improvements in our test process due to the use of formal specifications because they allow more expressive power than our current programming languages. This expressive power will allow our test and system experts to use TROT to formulate specifications that our current V&V techniques may have assumed or left as "derived" requirements. Such "unwritten requirements" can be formalized in oracle-specification so that the test process can include them. Thus, oracle-specification will support complete system testing and support the improvement of requirements. This should increase system reliability.

While the current TROT prototype is limited to range & resolution checking and limited path checking, we have shown that the basic concept is workable, and the approach has potential for expansion. Our old V&V approach relies on automation, human experts, and classical functional/structural testing.

Finally, we are considering the application of additional advanced error seeding techniques, such as mutation analysis, before more advanced production versions of TROT are completed.

5.0 CONCLUSIONS AND SUMMARY

The TROT system is a feasible, non-interfering, formal specification-based test system which can be used for automated verification to ensure that an implementation meets requirements. Our approach is unique in that we are applying formal specifications to an ongoing test process used to test software that is in a maintenance mode and that must be

highly reliable. TROT prototypes have shown a way to incorporate formal specification techniques in an industry setting which had not previously used formal methods as part of a test program. The tool supports faster and more complete testing based on requirements/design information. Our initial use and results support the conclusion that TROT can enhance an existing verification and validation process.

The initial TROT prototype is designed only to test variable range and resolution requirements of equations and math functions. A simple version of our approach has completed development and been used to support actual product testing. Work is underway to expand our approach into a full test system that is planned for extensive use in the near future. The expansion of the TROT concept and the long term evaluation of its impact on our ongoing V&V process is a natural outcome of this work. Because of the high reliability nature of our test environment and software under test, we can include new techniques like TROT slowly and only as part of an orderly transition. Our efforts to date support this process for adding an advanced technology like formal methods to existing test systems.

The TROT system is part of ongoing efforts to identify improvements to our existing V&V system. These improvements include:

- General hardware and software system upgrades;
- Test tool processing automation (of analysis results like TROT does); and
- Test process improvements including formal methods and test data selection techniques as in [8].

We plan to upgrade and continue to evaluate the TROT concept. This evaluation will include quantitative and qualitative measures of TROT's impact on the existing test process. We expect to see improvements in test capabilities due to the expressiveness of formal languages. We also expect to see improvements in time-to-test costs because of the increase in automation or self-analyzing capability of the TROT system. Thus, we expect more reliability in the software, because more testing will be possible without additional time or effort.

REFERENCES

- 1 Bieman and Yin, "Designing for Software Testability Using Automated Oracles", Proceedings International Test Conference, pp. 900-907, September 1992
- 2 CADRE, Workstation Environment Software, Manual Package - 02584-04033, CADRE Technologies Inc. 1988.
- 3 J. Hagar, J. Bieman "A Technique for Testing Highly Reliable Real-Time Software", Proceedings of Software Technology Conference, April 1994.
- 4 J. Hagar " A Systems Approach to Software Testing and Reliability", Proceedings of the 10th Annual Software Reliability Symposium, June 1992
- 5 J. Knight, P. Ammann, Testing Software using Multiple Versions, SPC publications, June 89.
- 6 David C. Luckham, Friedrich W. von Henke "An Overview of Anna, a Specification Language for Ada", IEEE Software, IEEE, 1985, pp. 19-22.
- 7 David C. Luckham, An Introduction to Anna, A Language for Specifying Ada Programs, Programming with Specification, Springer-Verlag, 1990 .
- 8 Tim Raske, "More Efficient Software Testing through the Application of Design of Experiments (DOE)", Proceedings of ISSRE 94.
- 9 J. Hagar, J. Bieman " Adding Formal Specifications to a Proven V&V Process for System-Critical Flight Software", Proceedings of Workshop in Industrial-strength Formal specification techniques - 95, April 1995.