Software Tech News

The DoD Source for Software Technology Information.

Software Testing Part 2

In This Issue:

Read additional Software Testing materials at:

www.dacs.dtic.mil/awareness/newsletters/listing.shtml

Testing Software Based Systems: The Final Frontier

by Thomas Drake, Coastal Research & Technology Inc. (CRTI)

Where are We? Setting the Stage

The increasing cost and complexity of software development is leading software organizations in the industry to search for new ways through process methodology and tools for improving the quality of the software they develop and deliver. However, the overall process is only as strong as its weakest link. This critical link is software quality engineering as an activity and as a process. Testing is the key instrument for making this process happen.

Software testing has traditionally been viewed by many as a necessary evil, dreaded by both software developers and management alike, and not as an integrated and parallel activity staged across the entire software development life cycle. One thing is clear - by definition, testing is still considered by many as only a negative step usually occurring at the end of the software development process while others now view testing as a "competitive edge" practice and strategy.

The best that can happen under the former perception is that no problems are detected in the software and that none exist for detection until after delivery. We know this is not the case in the real word of application development. In reality, it is testing that finds problems which trigger a feedback loop to development for resolution and retesting to make sure the fix works and has not created other problems. All of this activity invariably happens under extreme time constraints and with significant management visibility. But it is the kind of visibility that no one usually wants because everyone else above in the development food chain could slip.

continued on page 2

DACS

DoD Data & Analysis Center for Software
http://www.dacs.dtic.mil

Software Testing Series: Part 2

Note: This is the second in a series of newsletters devoted to Software Testing.

Testing Software Based Systems: The Final Frontier

Continued from page 1

The Real World of Software Development - A Sobering Perspective

The following scenario is not unusual and represents a composite perspective gleaned from this writer's 10 years of experience in the information technology industry and DoD environments.

A software test specialist is assigned to work on a multimillion dollar effort to develop a new system. The test specialist knows that the completion date for the program is unrealistic, given the scope and complexity of the development effort.

As a result of testing, the test specialist knows that there are some real serious technical difficulties impacting the software system's interface performance with a very large relational database system, as well as numerous bugs in the query routines for the graphical user interface.

After months of keeping growing concerns private, the test specialist decides to share these concerns with a fellow colleague. These concerns were not raised earlier because attempts to do so by others with management had resulted in management telling them not to rock the boat. They had learned that viewpoints perceived as negative were unwelcome and not wanted. Fellow colleagues had stopped giving feedback to management because they now felt their views would be ignored and were afraid that additional feedback of this type would affect their careers. So this test specialist told management what we thought they wanted to hear, that there were some minor problems with the software but nothing that could not be resolved in time for the projected delivery date.

However, as the release date loomed ever closer, it was becoming obvious that the software was overly complex, had a lot of functional problems, and most importantly, would not operate as promised at the point of delivery. The test specialist knew it would be a disaster if the system were delivered as scheduled.

The system went through a development test and evaluation (DT&E) period that was aborted, and the program was subsequently canceled by the acquisition organization after multiple tens of millions of dollars had been spent on development. The test organization was later disbanded because it was perceived as part of the problem.

Test professionals who find themselves in similar circumstances are faced with a difficult choice. What should this test specialist have done?

The Association for Computing Machinery (ACM) code of ethics states the following:

> *"The honest computing professional will not make deliberately false or deceptive claims about a system or system design, but will instead provide full disclosure of all pertinent system limitations and problems."*

And the biggest single obstacle is cultural. Testing is not generally viewed in our software development environments as where the real action is. The general perception is still the following in many development organizations - testers are software developers who could not make it and only real developers become programmers. Testers, in particular, are often regarded as second class citizens and rewarded accordingly. This often leads to high turnover, junior level experience, and no commitment to a comprehensive test program on the part of management.

However, becoming a good test engineer requires a skill set at least as equally complex as that of a good software developer. And the importance of testing is becoming more and more relevant with the dependencies we place on software creating "consequential damages" and legal quicksand when it does not work as advertised. What does it take?

Creating the Right Environment - The People Side of the Equation

Senior managers within information technology must create an environment and foster a professional climate in which their test and development engineers are encouraged to recognize and respond positively within a software development effort and where all project tasking is rigorously and regularly reviewed. It is the job of the tester to "tell it like it is."

We usually think of testing in software development as something we do when we run out of time or after we have developed code. However, the fundamental approach as presented here focuses on testing as a fully integrated but independent

continued on page 3

activity with development that has a lifecycle all its own, and that the people, the process and the appropriate automated technology are crucial for the successful delivery of the software based system. Planning, managing, executing, and documenting testing as a key process activity during all stages of development is an incredibly difficult process. By definition, it has to be comprehensive. And finally, who does the testing and the requisite commitment to testing is perhaps as important as the actual testing itself.

Software Quality Engineering - As a Discipline and as a Practice (Process and Product)

Software Quality Engineering is composed of two primary activities - process level quality which is normally called quality assurance, and product oriented quality that is normally called testing. Process level quality establishes the techniques, procedures, and tools that help promote, encourage, facilitate, and create a software development environment in which efficient, optimized, acceptable, and as fault-free as possible software code is produced. Product level quality focuses on ensuring that the software delivered is as error-free as possible, functionally sound, and meets or exceeds the real user's needs. Testing is normally done as a vehicle for finding errors in order to get rid of them. This raises an important point - then just what is testing?

Common definitions for testing - A Set of Testing Myths:

> *"Testing is the process of demonstrating that defects are not present in the application that was developed."*
>
> *"Testing is the activity or process which shows or demonstrates that a program or system performs all intended functions correctly."*
>
> *"Testing is the activity of establishing the necessary 'confidence' that a program or system does what it is supposed to do, based on the set of requirements that the user has specified."*

> *"Software implementation is a cozy bonfire, warm, bright, a bustle of comforting concrete activity. But beyond the flames is an immense zone of darkness. Testing is the exploration of this darkness."*
>
> - extracted from the *1992 Software Maintenance Technology Reference Guide*

All of the above myths are very common and still prevalent definitions of testing. However, there is something fundamentally wrong with each of these myths. The problem is this - each of these myths takes a positive approach towards testing. In other words, each of these testing myths represents an activity that proves that something works.

However, it is very easy to prove that something works but not so easy to prove that it does not work! In fact, if one were to use formal logic, it is nearly impossible to prove that defects are not present. Just because a particular test does not find a defect does not prove that a defect is not present. What it does mean is that the test did not find it.

These myths are still entrenched in much of how we collectively view testing and this mind-set sets us up for failure even before we start really testing!

So what is the real definition of testing?

> *"Testing is the process of executing a program/system with the intent of finding errors."*

The emphasis is on the *deliberate* intent of finding errors. This is much different than simply proving that a program or system works. This definition of testing comes from *"The Art of Software Testing"* by Glenford Myers. It was his opinion that computer software is one of the most complex products to come out of the human mind.

So why test in the first place? You know you can't find all of the bugs. You know you can't prove the code is correct. And you know that you will not win any popularity contests finding bugs in the first place. So why even bother testing when there are all these constraints? The

continued on page 4

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

Testing Software Based Systems: The Final Frontier

Continued from page 3

fundamental purpose of software testing is to find problems in the software. Finding problems and having them fixed is the core of what a test engineer does. A test engineer should WANT to find as many problems as possible and the more serious the problems the better. So it becomes critical that the testing process is made as efficient and as cost-effective as possible in finding those software problems. The primary axiom for the testing equation within software development is this:

"A test when executed that reveals a problem in the software is a success."

The purpose of finding problems is to get them fixed. The benefit is code that is more reliable, more robust, more stable, and more closely matches what the real end-user wanted or thought they asked for in the first place! A tester must take a destructive attitude toward the code, knowing that this activity is, in the end, constructive. Testing is a negative activity conducted with the explicit intent and purpose of creating a stronger software product and is operatively focused on the "weak links" in the software. So if a larger software quality engineering process is established to prevent and find errors, we can then change our collective mind-set about how to ensure the quality of the software developed.

The other problem is that you will really never have enough time to test. We need to change our understanding and use the testing time we do have, by applying it to the earlier phases of the software development life cycle. You need to think about testing the first day you think about the system. Rather then viewing testing as something that takes place after development, focus instead on the testing of everything as you go along to include the concept of operations, the requirements and specifications, the design, the code, and of course, the tests!

The Further Along You Are In The Software Development Life Cycle The More It Costs To Test!

Lesson learned - just test early. Test early and often. Test the design of the system before you build any pseudo-code. Test the specs before you actually code. Review the code during coding before you test the code, and then finally execute actual test cases. By doing the reviews and the code-level analyses during all phases of the development life cycle you will find many, if not most of the problems in the system before the traditional testing period even begins. These activities alone will greatly improve the quality of the delivered system.

Find out the cause of this effect, Or rather say, the cause of this defect, For this effect defective comes by cause.
- Hamlet (with thanks to DeMarco)

About the Author

Mr. Drake is a software systems quality specialist and management and information technology consultant for Coastal Research & Technology Inc. (CRTI). He currently leads and manages a U.S. government agency-level Software Engineering Knowledge Based Center's software quality engineering initiative. As part of an industry and government outreach/ partnership program, he holds frequent seminars and tutorials covering code analysis, software metrics, Object-Oriented (OO) analysis for C++ and Java, coding practice, testing, best current practices in software development, the business case for software engineering, software quality engineering practices and principles, quality and test architecture development and deployment, project management, organizational dynamics and change management, and the people side of information technology. He is the principal author of a chapter on "Metrics Used for Object-Oriented Software Quality" for a CRC Press Object Technology Handbook published in December of 1998. In addition, Mr. Drake is the author of a theme article entitled: "Measuring Software Quality: A Case Study" published in the November 1996 issue of IEEE Computer. Mr. Drake is listed with the International Who's Who for Information Technology for 1999, is a member of IEEE and an affiliate member of the IEEE Computer Society. He is also a Certified Software Test Engineer (CSTE) from the Quality Assurance Institute (QAI).

Author Contact Information

Thomas A. Drake
Coastal Research & Technology Inc.
5063 Beatrice Way
Columbia, MD 21044
Phone: (301) 688-9440
Fax: (301) 688-9436
tadrake@earthlink.net

Survivability as a Component of Software Metrics

by David L. Wells, Object Services and Consulting Inc. and David E. Langworthy, Langworthy Associates

Introduction

Software metrics provide estimates of software quality that are used to determine where to spend additional development testing resources or to determine the suitability of software for particular (often critical) applications. Metrics have traditionally focused on code quality. However, the trend toward constructing large, distributed applications as a collection of independent "services" interacting across a software backplane (e.g., CORBA), makes the process of configuring the application an important part of the development process. This affects the kinds of software metrics required, since perfect software, imperfectly deployed, or deployed in such a way that is vulnerable to failure or attack is of no more value than imperfect software that fails of its own accord. This paper describes metrics we developed [5] for measuring the *survivability* of software systems that can be applied to the more general realm of software metrics.

The Importance of Configurations

Service-based applications can have many physical configurations that provide the same (or approximately the same) logical functionality using identical services. Multiple configurations are enabled by the following:

- Clients and services may run on different platforms in differing combinations,
- Partial application failure (e.g., a client running, service down) is possible due to software or environmental factors,
- Interfaces that hide implementation details allow a service to have multiple implementations,
- Multiple objects to provide the same or equivalent services,
- Services can fail because of programming errors or because of a failure of the underlying resources (e.g., hosts or networks),
- Connections between clients and services are typically loose, which makes it possible in principle to change the connections on the fly.

A truly useful metric for distributed, service-based software must measure both the quality of the software itself (the traditional role) and the quality of its configuration vis a vis the underlying infrastructure and the kinds of threats to which the software and infrastructure are subject. In the real world, systems can fail for a variety of reasons other than code and specification errors (e.g., a virus might corrupt the file system that the software relies upon). Thus, rather than ask simply whether the specification and code are correct, it is necessary to ask how likely it is that the system will be to continue to provide the desired functionality, or failing this, something approaching it. A *survivable* system [1,2] is one in which actions can be taken to reconfigure applications in the event of partial failures to achieve functionality approximating the functionality of the original system. The usefulness of a survivable system can be judged in several ways: how useful is what it is doing now?; how useful is it likely to be in the future?; if it breaks, can it be repaired so that it can again do something useful?

Overview of Utility Theory

Utility theory is the study of decision making under risk and uncertainty among large groups of participants with differing goals and preferences [4]. A participant has direct control over the decisions he makes, but these decisions are only indirectly linked to their outcomes, which depend on the decisions of other participants and random chance.

Utility can be used to quantify the goodness of states and actions in a survivable system. System states can be compared using utility measures to determine which are preferred, and as a result, which survival actions should be taken in an attempt to move the system to a better state or avoid worse states. A key aspect of measuring the utility of a system state or administrative action is that utility depends on both the services that are currently running and the future configurations that can be reached. Future configurations need to be considered to differentiate between a rigid configuration that offers good current performance from a flexible configuration that offers slightly lower current performance but is more resilient to faults and is more likely to continue offering good performance. A balance must be reached between present performance and future performance. For example, for most systems the potential configurations a year in the future are not nearly as important as the configurations the system could reach during the next 12 hours.

Continued on page 6

Survivability as a Component of Software Metrics

Continued from page 5

Applying Utility Theory to Software Metrics

Every client receives a benefit from every service it uses, expressed as a *utility function, U,* that maps a description of the service being provided to a value received. The service to be received can be described in many ways, including using quality of service (QoS) concepts such as timeliness, precision, and accuracy of the results to be provided. Further, utility itself can have multiple definitions, depending on the overall goals to be achieved. For example, one utility function could value maximizing the work performed, another utility function could value minimizing the likelihood that the level of service provided falls below some threshold, and a third utility function could value minimizing the probability that information is divulged to an opponent. All are equally valid, and depending upon circumstances could in turn be valued to different degrees. This would result in a combined utility function that is some aggregation of the underlying utility functions.

The benefit a client receives from a service is accrued only if the service completes its task; i.e., an instantaneous, ephemeral connection to a service provides no value. Thus, every benefit function must include a *duration* over which the service must be provided in order to attain the specified benefit. Our analysis restricts the duration to fixed size discrete time intervals; a client receives the benefit only if the service is still being provided at the end of the interval. We define the *utility of a configuration, U(c)*, to be the aggregation across all clients in a configuration of the value of the services they receive. Because there can be multiple utility functions, we differentiate between them using subscripts when necessary; ergo, $U_{work}(c)$.

A configuration provides utility only for tasks it completes. Since a system that begins a time interval in some configuration c may end it in some other configuration that provides a possibly different utility, a more useful measure of utility is the expected *utility of a configuration c, EU(c)*. EU(c) measures the benefit of a collection of potential configurations, C, that can be reached from c in one time interval. It is the probability weighted sum of the utilities of each individual configuration that can be reached. The probability function, $P(c_i)$, is the probability of c_i being instantiated out of all the configurations in the set.

$$\boxed{\text{Expected Utility} = \text{EU(c)} = EU(C_T) = \sum_{c \in C_T} P(c) \times U(c)}$$

Expected utility allows us to compute the benefit expected to be obtained from a configuration even after considering the near term negative events that can cause the configuration to degrade. A second utility measure allows us to consider longer term changes to the system and to incorporate the ability to perform beneficial administrative transformations. We call this *net utility, NU(c)*. Net utility measures the fact that the long term desirability of a configuration depends upon the services that are currently running and the future configurations that can be reached.

$$\boxed{\text{Net Utility} = NU(c) = \sum_{t > now} D(t) \times EU\,(C_t)}$$

Net utility is thus a sum of future expected utilities. In general, not all time periods are of equal importance; the near term behavior of a system is usually valued more highly than behavior far into the future. To handle this, we introduce a *discount function, D(t),* which maps from time to an appropriate weighting factor. The discount function is related to net present value in finance.

The use of a discount factor has an additional benefit, since it allows us to discount far future states for computational as well as policy reasons. This has a practical advantage, since when one projects the configuration space further into the future, the computations rapidly become more expensive (due to state explosion) and the results rapidly become less precise (due to imprecise estimates of event probabilities). The benevolent myopia introduced by the discount factor allows us to ignore incomputable or dubious future states.

Utility Metrics

The meaning and power of the metrics defined above vary greatly depending on the precise definition of the base utility function U(c). As noted, the base utility function measures <u>what</u> is valued most highly. We introduce two very different utility metrics.

Utility of Value is based on a measure for aggregate performance. This work is developed from a market based, distributed resource allocation prototype.

continued on page 7

The goal of the market was to maximize the aggregate value of all the services provided by the system. End users or administrators would assign values to services. The resources, both hardware and software, would compete to offer the best service at the lowest cost. The resources' goal was to accumulate profits which would be gathered by the owners of the resources and allocated to end users and administrators, closing the loop. If users value a service highly, it will replicate itself to assure that it is highly available. If resources are removed from the system, the prices will rise and only the more valued services will obtain resources; if resources are added, prices will fall and lower priority services will run. Utility of Value implements a simple microeconomic model that tends toward Pareto Optimality, a local optimality criterion. If the Net Utility of Value is maximized, then future performance of the system will be maximized. There are many possible definitions of survivability, but a relatively straightforward one is that the system continues to offer good performance into the future.

Utility of Operation is based on a binary measure depending on whether the system meets some minimal level of operation over a given interval. This gives rise to a very different notion of survivability. Using this measure, EU(C) is itself a probability: the probability that the system is operational. Maximizing the Net Utility of Operation minimizes the possibility of some catastrophic failure in the future, possibly at the cost of optimal average case performance. This is arguably a better survivability metric than the Net Utility of Value, since the purpose of survivability is to avoid catastrophic failures. The two could be used in conjunction so that after a minimal level of service is guaranteed, performance is optimized for the normal case.

Examples: Replica Balancing

There are two services, A and B, and six hosts, 1 - 6. Each service can be replicated and each replica requires an entire host. There is a 10% probability of failure of each host during a period, so the probability of success of a service with n replicas is $1- 0.1^n$. In the initial configuration, C1, each service has three replicas: C1 = A{1, 2, 3}; B{4, 5, 6}. At step 2, B loses two replicas, so C2 = A{1, 2, 3}; B{4}. The third configuration, C3, is the result of a possibly automatic administrative action which trades a second backup from A to provide a single backup for B, C3 = A{1, 2}; B{4, 3}. This last transition is voluntary. The administrator or survivability service would take whatever action seemed best.

Table 1 calculates the expected utility for each configuration in the example. A bar over the service label in the State column (RC) indicates the service is not operational at the end of the period. The second column is the value of the configuration. The aggregation function is simple addition, so if both A and B are operational the value of the configuration is 2000. P(i) and E(i) show the calculation of the expected utility of Ci,EU(Ci), which is shown on the last row of the table.

In C1 everything is running fine. Out of a possible value of 2000 the expected utility is 1998, almost perfect. After the failures, the expected utility drops to 1899 because of the uncertainty that B will complete. C3 reflects the administrative action of taking a replica from A and giving it to B. This increases the expected utility to 1980, a dramatic improvement considering that no resources were added.

Utility of Value vs. Utility of Operation

The following illustrates the difference between utility of value (which optimizes for performance) and utility of operation (which optimizes for stability). Service A now has two levels of operation,

continued on page 8

Table 1. Expected Utility

RC	Value	P(C1)	E(C1)	P(C2)	E(C2)	P(C3)	E(C3)
$A\overline{B}$	2000	.9980	1996	.8991	1798	.9801	1960
$\overline{A}B$	1000	.0009	1	.0999	100	.0099	10
$\overline{AB}$	1000	.0009	1	.0009	1	.0099	10
AB	0	.0000	0	.0001	0	.0001	0
			1998		**1899**		**1980**

Survivability as a Component of Software Metrics

Continued from page 7

high and low. The high level offers a value of 2000 and requires 3 hosts to run. The low level is required for a minimal level of operation and offers a value of 1000 but requires only 1 host to run. If the high level of service cannot be maintained, it automatically drops to the low level of service. In the example A starts out at the high level of QoS. If A loses a host, it drops to the low level of QoS with one replica. The probability that A completes the period at the high level is the probability that all three hosts complete. The probability that A completes the period at the low level is the probability that any single host completes minus the probability that A completes at the high level. There are now 6 possible outcomes. B is still worth 1000, so if A completes at the high level along with B the value is 3000.

Table 2 calculates the utility of value. In the initial configuration all hosts are operational and the expected Utility of Value is nearly optimal at 2739. After the failures, the expected value drops by about 150 reflecting B's instability. C3 evaluates the administrative action of removing a host from A to increase B's stability. In this case, the action does not appear to be desirable and would not be taken. The reason is that removing a host from A would cause it to drop from a high level of QoS to a low level of QoS at a cost of nearly 1000.

Utility of Value metric maximizes perceived performance and maintaining A at a high level of QoS is consistent with this goal. However, the survivability of the system is sacrificed by this choice as Table 3 using Utility of Operation shows.

In the initial state all hosts are operational and A is operating at the high level. After the failures, B is reduced to one replica and the expected Utility of Operation drops to .8991. A is still operating at the high level, but this is not reflected in the binary operational metric. Step 3 reflects the administrative action of taking a host from A. This causes A to drop from the high level to the low level and increases the stability of B. As a result the expected operational utility increases to .9801.

Conclusions

The metrics presented allow measurement of the useful work that is likely to be done by software as actually deployed and subject to the various kinds of attacks and failures that exist in the real world. These metrics can be combined with more traditional software metrics that measure the likelihood of failure due to software or specification failure to produce a combined metric that measures both the quality of the code and its expected long-term behavior in a realistic environment.

About the Authors

David Wells, is Vice President of Object Services and Consulting Inc. and the head of software research. Wells received his D. Eng. degree in Computer Science from the University of Wisconsin-Milwaukee in 1980. He was Assistant Professor in the Computer Science Deptartment at Southern Methodist University from 1980 to 1986 where he conducted research in databases, computer security, and computer graphics.

Dr. Wells was the Principal Investigator on the DARPA/ITO project *Survivability in Object Services Architectures*. He was

continued on page 9

Table 2. Utility of Value

RC	Value	P(C1)	E(C1)	P(C2)	E(C2)	P(C3)	E(C3)
A_hB	3000	.7283	2183	.6561	1968	.0000	0
A_lB	2000	.2697	539	.2430	486	.9801	1960
$\overline{A}B$	1000	.0010	1	.0729	73	.0099	10
$A_h\overline{B}$	2000	.0007	14	.0270	54	.0000	0
$A_l\overline{B}$	1000	.0003	0	.0009	1	.0099	10
$\overline{AB}$	0	.0000	0	.0001	0	.0001	0
			2739		**2582**		**1980**

Table 3. Utility of Operation

RC	Value	P(C1)	E(C1)	P(C2)	E(C2)	P(C3)	E(C3)
A_hB	1	.7283	.7283	.6561	.6561	.0000	0
A_lB	1	.2697	.2697	.2430	.2430	.9801	.9801
$\overline{A}B$	0	.0010	0	.0729	0	.0099	0
$A_h\overline{B}$	0	.0007	0	.0270	0	.0000	0
$A_l\overline{B}$	0	.0003	0	.0009	0	.0099	0
$\overline{AB}$	0	.0000	0	.0001	0	.0001	0
			.9980		**.8991**		**.9801**

previously PI on the DARPA funded *Open OODB* and *Open OODB II* projects at Texas Instruments where he was the principal architect of a modular object-oriented database that seamlessly added persistence to programming objects. Those projects produced many of the ideas of flexible service binding used in this software survivability work. Wells has also done work in cryptography for databases and risk assessment. Wells holds 5 patents and has published over 20 technical articles in journals and conferences.

David E. Langworthy performs experimental computer science research in both academic and industrial contexts. He has ten years experience in the design of scaleable distributed systems with a focus on object oriented database technology.

Langworthy received his PhD from Brown University in May of 1995.

While completing his PhD, Langworthy was a consultant at Microsoft, and designed the Information Retrieval system for the Microsoft Network. This system scaled to thousands of queries per second using parallel arrays of NT servers. The work resulted in fundamentally new technology for combined query evaluation.

Other accomplishments include: teaching a course in Object Oriented Analysis and Design, developing courseware, and consulting for Semaphore and the Trilogy Development Group.

Author Contact Information

David L. Wells
Object Services and Consulting, Inc.
Baltimore, MD
Phone: (410) 318-8938
Fax: (410) 318-8948
wells@objs.com

David E. Langworthy
Langworthy Associates
del@onr.com

References

[1] "Survivability in Object Services Architectures - 1998 Annual Report," David Wells, Object Services and Consulting Inc., www.objs.com/Survivability/, 1998.

[2] DARPA/ITO Information Survivability Website, Defense Advanced Research Projects Agency, www.darpa.mil/ito/research/is/, 1998.

[3] "Lazy Replication: Exploiting the Semantics of Distributed Services," R. Ladin, B. Liskov, L. Shrira, Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec, 1990.

[4] Game Theory in the Social Sciences: Concepts and Solutions, Martin Shubik, MIT Press, Cambridge Massachusetts, 1982.

[5] "Survivability is Utility," David Langworthy and David Wells, Object Services and Consulting, Inc., www.objs.com/Survivability/Utility.doc

Using Models for Test Generation and Analysis

Mark R. Blackburn, Software Productivity Consortium

Introduction

Systems are increasing in complexity. More systems perform mission-critical functions, and dependability requirements such as safety, reliability, availability, and security are vital to the users of these systems. The competitive marketplace is forcing companies to define or adopt new approaches to reduce the time-to-market as well as the development cost of these critical systems. Much focus has been placed on front-end development efforts, not realizing that testing accounts for 40 to 75 percent of the lifetime development and maintenance costs [3; 11]. Testing is traditionally performed at the end of development, but market-driven schedules often force organizations to release products before they are adequately tested. The long-term effect is increased warranty costs due to product's poor reliability and poor quality.

Model-based development tools are increasing in use because they provide tangible benefits by supporting simulation and code generation, in addition to the traditional design and analysis activities. These tools help users develop requirements and design models of target systems. Certain tools are based on formal models, and the underlying models are represented using specification languages. Such formal specifications provide a basis for test case generation. However, the underlying development models are generally not represented in a form that supports automatic test case generation. The key challenge is to translate development-oriented modeling languages into a form that is suitable for automated test vector generation, specification-based test coverage analysis, requirement to test traceability, and design-to-test traceability.

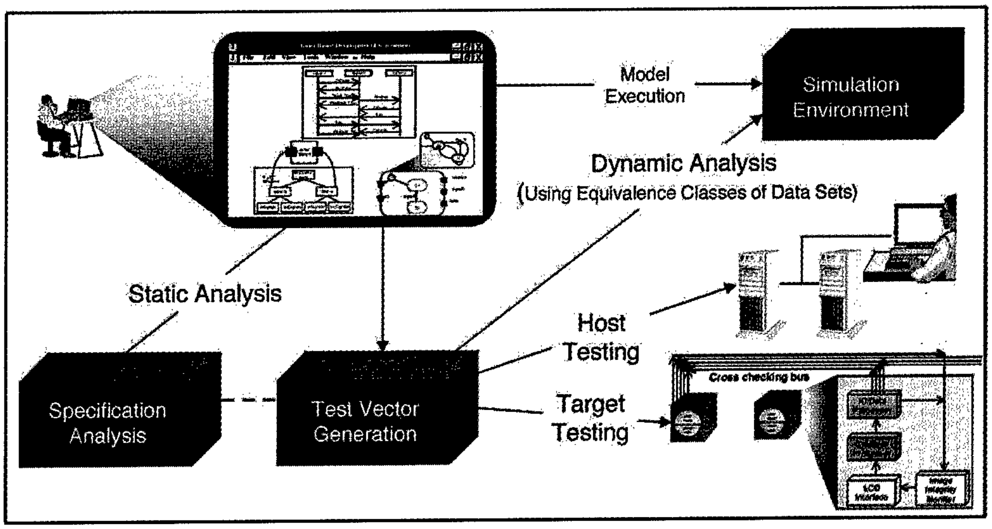


Figure 1. Using Models for Test Generation and Analysis

Using Models for Testing and Analysis

Figure 1 illustrates a conceptual view for using models to support test generation and analysis. Models and their associated tools typically provide various views of the system under development. When modeling tools are based on precise semantics, user models can also support:

- *Test Vector Generation.* A test vector includes inputs, expected outputs, and an association with the specification from which it was derived.
- *Static Analysis.* Typically used to determine if there are contradictions in specification.
- *Dynamic Analysis.* Analysis based on execution of the model.

Modeling tools are beginning to support simulation and code generation. Simulation of a model can help developers assess the correctness of the model with respect to user requirements; however, it can be time consuming to develop the simulation data required for thorough dynamic analysis. Automatically generated test vectors can provide a cost effective way to exercise a model in a simulator using the boundary values associated with the constraints of a model specification; it is at the boundaries where model anomalies are typically discovered. In addition, these same test vectors can also be used to test the code in a host or target environment.

Scope

This article describes the use of automated test generation and analysis from specification models. Through the integration of commercial off-the-shelf (COTS) model development and test generation tools, a process has been

continued on page 11

developed that eliminates most of the traditional testing activities. This approach has been demonstrated to identify many types of specification errors prior to any implementation. This article is based on experiences in developing two model translators [4; 5] supporting:

- Software Cost Reduction (SCR) [12] /Consortium Requirements Engineering Method (CoRE) [18] for modeling requirements
- Real-Time Object-Oriented Modeling (ROOM) [17] method for analysis and design

For each respective method and associated tool, the translators produce a specification that is used by the T-VEC tool system to generate test vectors and perform specification-based test coverage analysis. The model transformation process is briefly described using a specification example. The article summarizes the results of applying the process and tools to industrial applications.

Models and Specifications

Formal specifications provide simple abstract descriptions of the required behaviors describing what the software should do. Because formal specifications have, in the past, been considered difficult to use, they have not been widely used. Recent advances in visual model-based development tools provide the basis for developing formal specifications while hiding the formalism.

It has been commonly accepted that formal specifications provide a basis for test case generation. Goodenough and Gerhart may have been the first to claim that testing based only on a program implementation is fundamentally flawed [8]. Gourlay developed a mathematical framework for specification-based testing [10]. Figure 2 graphically represents Gourlay's mathematical framework for testing and the key relationships between specifications, tests, and programs. Given a **specification** that describes the requirements for some system, there are one or more **programs** that implement the specification. **Tests** are derived from the specification; if every test executed by a program computes the appropriate expected results (i.e., passes every test), there is some level of confidence that the program satisfies the specification.

In Figure 2, the specification symbol (i.e., rounded rectangle) is generically used to represent requirement, design, or test specifications. Certain specification languages have tool support that helps in developing complete and consistent specifications. Such tools provide the syntactic and semantic rigor that is required for transforming specifications into a form suitable for test vector generation. Model-based specification methods that support functional, state transition, and event based techniques are increasing in popularity and use because the tool support has helped make them easier to use[1].

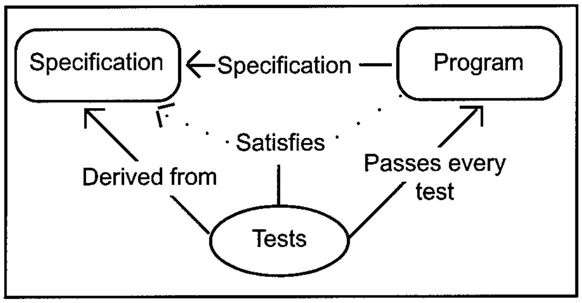


Figure 2. Testing Model and Relationships

A **model-based specification** approach constructs an abstract model of the system states and characterizes how a state is changed by abstract or concrete operations (paraphrased from Cohen et al. and Cooke et al. [7; 6]). Operations in the system are specified by defining the state changes or events that affect the model using existing mathematical constructs like sets or functions. **State transitions** define relationships between sequences of states based on conditions of the system state. **Event specifications** define certain conditions related to a change in the system state[1].

A **test specification model** is defined by a set of test specification elements, as shown in Figure 3 on page 12. A **test specification element** is an input-to-output relation and an associated constraint defined by a conjunction (i.e., logically ANDed) of Boolean-valued relations that define constraints on the inputs associated with the input-to-output relation.

Given a specification element, a **test vector** is a set of test input values derived from the constraint, and an expected output value derived from the input-to-output relation with respect to the test input values [1]. Informally, from a test generation perspective, a specification is **satisfiable** if at least one test vector exists for every specification element [2].

continued on page 12

[1] Zave and Jackson [21] identify potential implementation bias of model-oriented techniques but support the claim that model-oriented techniques are gaining in popularity.

Using Models for Test Generation and Analysis

Continued from page 11

Model Transformation

Model transformations are typically required to transform model-based specifications into a form to support test generation. Hierons describes rewriting rules for Z specifications [13] to support test case generation, but does not address specifications composed of combinations of specification techniques, particularly specifications composed using event specification techniques. In general, model transformation to support tool interoperability is an important area of investigation [9].

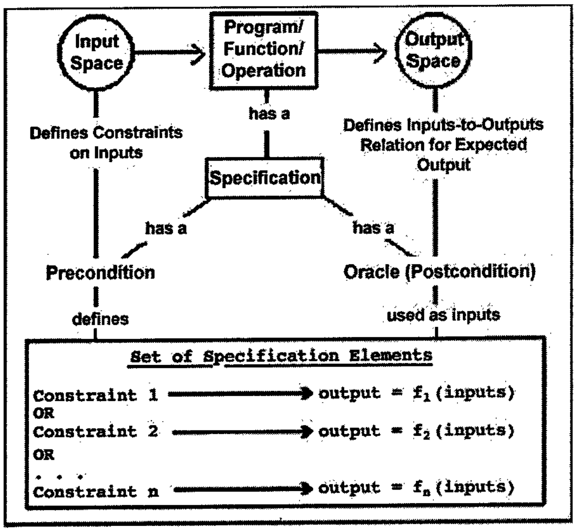

Input Space
Program/ Function/ Operation
Output Space
has a
Defines Constraints on Inputs
Defines Inputs-to-Outputs Relation for Expected Output
Specification
has a
has a
Precondition
Oracle (Postcondition)
defines
used as inputs
Set of Specification Elements
Constraint 1 → output = f1(inputs)
OR
Constraint 2 → output = f2(inputs)
OR
. . .
Constraint n → output = fn(inputs)

Figure 3. Representation of Test Specification Model

Blackburn [5] describes a tool based approach for transforming a model-based specification into a form that supports test vector generation. The model-based specification supports composition using function, state, and event specifications. A translator implements rules for transforming SCR model specifications into a language used by the T-VEC test vector generation tool. The development of the prototype translator and evaluation environment helped identify shortcomings in the rules described in prior work that was presented at the 1997 Computer Assurance Conference [2].

Similar model transformation efforts, not described in this article, were performed for the ROOM method using the Object Time Developer tool set as part of the validation environment[4].

Evaluation Environment

Figure 4 identifies generic tool types that are related to the elements of the test model shown in Figure 2. Such tools use or produce the three primary types of system artifacts (i.e., specifications, programs, and tests). A specific instance of this model was created to support the model transformation approach using the SCR tool (referred to as SCR* - pronounced SCR star) as the source for model-based specifications and the T-VEC tool system as the tool that supports test generation and specification-based coverage analysis.

SCR*, developed by the Naval Research Laboratory, supports modeling and analysis of requirement specifications using a formal modeling language (i.e., a language with well-defined syntax and semantics).

T-VEC, developed by T-VEC Technologies, Inc. supports:

- *Test Vector Generation. A* test vector generator produces test vectors from test specifications.
- *Specification-Based Coverage Analysis.* This tool analyzes the transformed specification to determine whether all specification elements have a corresponding test vector. This is the mechanism used to assess satisfiability of the transformed specifications.

Applications and Results

The remainder of this article describes a simple example to illustrate the use of this approach for model analysis and testing. Consider the example of an electronic regulator, shown in Figure 5. The requirements for the regulator are:

- When the temperature reaches the High zone (i.e., 180 degrees), the valve opens.
- The amount the valve opens is a function of the temperature from 120 degrees (closed) up to 300 degrees (fully open).

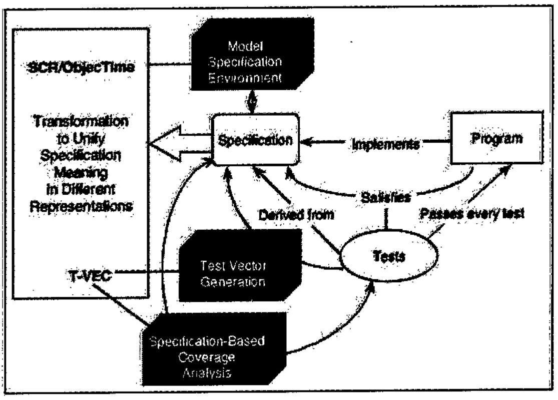

SCR/ObjecTime
Model Specification Environment
Transformation to Unify Specification Meaning in Different Representations
Specification
Implements
Program
Satisfies
Derived from
Passes every test
Tests
T-VEC
Test Vector Generation
Specification-Based Coverage Analysis

Figure 4. Tools of the Evaluation Environment

continued on page 13

- Once the valve is open, it remains open until the temperature reaches the Low zone (i.e., 120 degrees).

The specification is described in the SCR tabular notation. Heitmeyer, et al. [14] describes the SCR method. The specifications are defined in two parts. The **first part of the specification** defines the relationships between the temperature and the associated modes that relate to the temperature zones. This is referred to as the Sensor Mode Table shown in Figure 5. The system can be in one of three modes: LOW, READY, and HIGH. At the time when the temperature becomes greater than the constant Low (i.e., 120 degrees), the system transitions into the mode READY. The formal expansion of the event is:

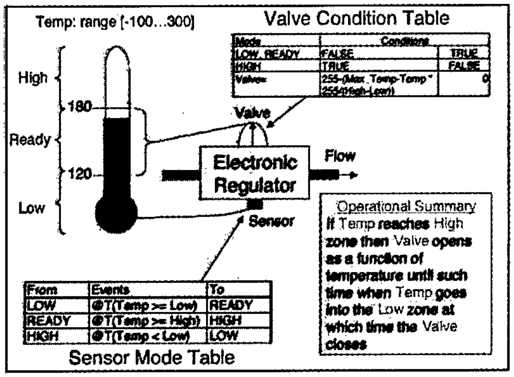


Figure 5. Example of Electronic Regulator

```
@T(Temp >= Low) means:

if the previous value of Temp
denoted NOT(_Temp >= Low) and the
new value of Temp >= Low then the
event is true and the mode
transitions from LOW to READY
```

Table 1. Relationship of Translated Constraints

Events	Translation
@T(TemD >= Low)	(Temp >= Low) AND (Temp < Low)
@T(Temp >= High)	(Temp >= High) AND (Temp < High)
@T(Temp < Low)	(Temp < Low) AND (_Temp >= Low)

Table 2. Tests for Each Translated Constraint

	Output	Inputs		
Translation	Sensor	Sensor	Temp	_Temp
(Temp >= Low) AND (Temp < Low)	READY READY	LOW LOW	120 179	-100 119
(Temp >= High) AND (Temp < High)	HIGH HIGH	READY READY	180 300	120 179
(Temp >= Low) AND (Temp < Low)	LOW LOW	HIGH HIGH	-100 119	180 300

Table 1 shows the translated meaning for each event specification of the Sensor Mode Table. For each constraint, there is a minimal set of tests as shown in Table 2. The T-VEC test generation system uses a test selection heuristic based on domain testing theory where low-bound and high-bound values are selected for each constraint[2]. For example, the first test selects the low-bound value for the previous state value of_TEMP[3] (-100), which is less than the constant Low, and selects a value of 120 for the next state value of TEMP. For the high-bound selection, the value of 119 (i.e., one less that the constant Low) is selected for _TEMP, and 179 for TEMP (i.e., one less than the constant High).

The **second part of the specification** defines the constraints and functions for the Value Condition Table shown in Figure 5. This table depends on the Sensor Mode Table. The Valve Condition Table is interpreted as follows:

```
if Sensor mode = High then
  Valve = 255-(Max_Temp-Temp
        * 255/(High-Low))
else if Sensor mode = LOW
        or Sensor mode = READY then
          Valve = 0
endif
```

Each SCR output variable and associated function map to a T-VEC functional relationship of an output variable with respect to the constraints on the input variables. The SCR model does not necessarily define a system state strictly in terms of constraints on the input variables as is required for T-VEC. For example, the Sensor mode is defined in terms of a mode transition table.

continued on page 14

[2] White and Cohen proposed domain testing theory as a strategy to select test points to reveal domain errors [19]. Their theory is based on the premise that if there is no coincidental correctness, then test cases that localize the boundaries of domains with arbitrarily high precision are sufficient to test all the points in the domain. When there is a strong correlation between the specification constraints and implementation paths, the selected test data should uncover computation and domain errors. As defined by Howden and refined later by Zeil, a computation error occurs when the correct path through the program is taken, but the output is incorrect due to faults in the computation along the path. A domain error occurs when an incorrect output is generated due to executing the wrong path through a program [15; 20].

[3] An underbar (_) precedes the variable name to indicate that the variable represents the previous state variable before the event versus the next state variable after the event.

Using Models for Test Generation and Analysis

Continued from page 13

This results in table dependencies as illustrated in Figure 6. The mode variables and the associated table relations must be transformed into constraints on the input variables. Figure 6 provides a perspective of the example SCR specification represented in a form that is compatible with the test model shown in Figure 3. The constraint for the Valve Condition Table includes the conditions of the Valve table and the Sensor Mode transition table. This means that the constraint:

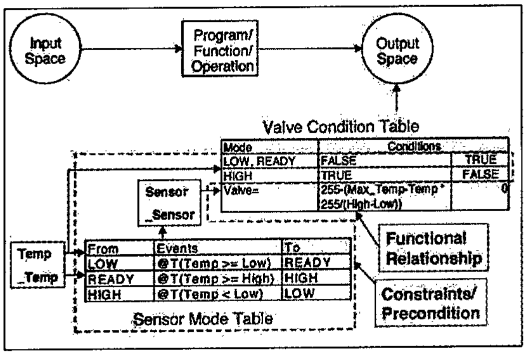


Figure 6. Dependency Relationship

```
(Temp >= High) AND (_Temp < High)
```

must be satisfied (i.e., the Sensor mode is HIGH) as a requirement for the value to be computed using the functional relationship:

```
1)  255-(Max_Temp-Temp
2)  *  255/(High-Low))
```

In general, mode transition tables can have dependencies on other terms and modes. Events for modes and terms create the need to identify the previous and next state variable dependencies. As shown in Figure 6, the Sensor Mode table depends on both the previous and next state input value of Temp; similarly the condition table Valve depends on the previous and next states of variables Temp and Sensor.

A test specification requires the constraints of a specification to be defined strictly in terms of the input and output variables. A model-based approach defines states that are relations of inputs, terms, or state variables (e.g., Sensor, _Sensor). This allows the constraint/precondition and functional relationship (defined in terms of a Condition Table) to be defined as a relation on inputs, states, or terms. This approach typically simplifies the task of specifying behavior, but it is the key reason why a model transformation process is required.

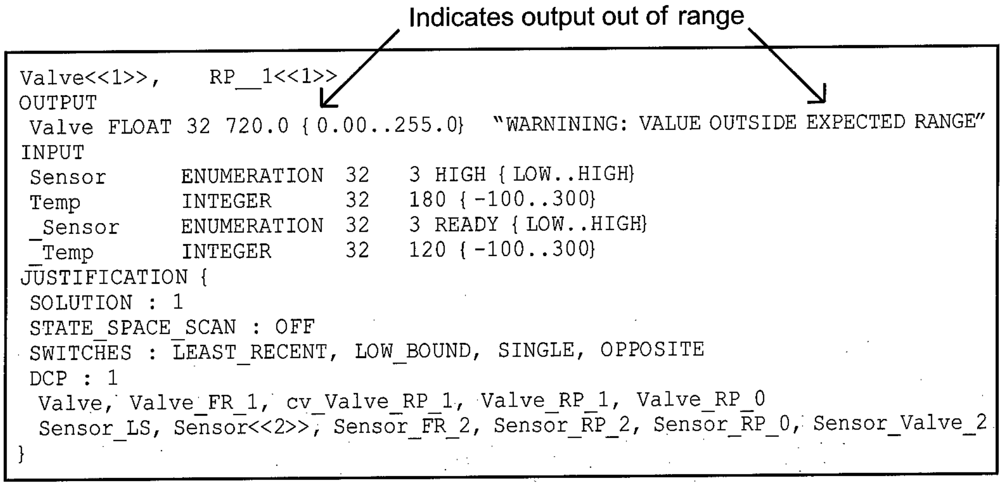
Static Analysis

Static analysis helps determine whether there are contradictions in the model without executing the model. Contradictions exist if constraints cannot be satisfied. This is typically the most common problem, especially when the dependencies of specifications become large. This example is a simple 2-level dependency problem, but typical systems can have 10 or more dependency levels. It is also possible to identify functional relationships that specify values that are inconsistent with the domain of the output variables. These are analogous to computation errors in the code.

Consider the function to compute the Valve function. The requirements are that as the temperature reaches the maximum temperature (i.e., 300 degrees), the valve should be completely opened, and when the value reaches the constant Low (i.e., 120 degrees), the valve should be closed. Electronically controlled devices typically use some type of digital value to represent a fully open valve (in this case 255 - an 8-bit unsigned integer), and the value should be 0 when the valve is closed. It is common for implementors to make errors in scaled arithmetic conversions. To illustrate this point, the computation has two errors.

Figure 7 shows a sample test vector that has identified a problem in the computation. A warning is appended to the expected output because the computation is out of range. This is

continued on page 15


Indicates output out of range

```
Valve<<1>>,    RP__1<<1>>
OUTPUT
 Valve FLOAT 32 720.0 {0.00..255.0}  "WARNINING: VALUE OUTSIDE EXPECTED RANGE"
INPUT
 Sensor      ENUMERATION  32   3 HIGH {LOW..HIGH}
 Temp        INTEGER      32   180 {-100..300}
 _Sensor     ENUMERATION  32   3 READY {LOW..HIGH}
 _Temp       INTEGER      32   120 {-100..300}
JUSTIFICATION {
 SOLUTION : 1
 STATE_SPACE_SCAN : OFF
 SWITCHES : LEAST_RECENT, LOW_BOUND, SINGLE, OPPOSITE
 DCP : 1
  Valve, Valve_FR_1, cv_Valve_RP_1, Valve_RP_1, Valve_RP_0
  Sensor_LS, Sensor<<2>>, Sensor_FR_2, Sensor_RP_2, Sensor_RP_0, Sensor_Valve_2
}
```

Figure 7. Internal Form of Test Vector with Warning

typically an indication that there is a computation error in the specification or that there are missing constraints on the inputs. The original expression (line 1 of the functional relationship under Figure 6) is missing parentheses around Max_Temp-Temp. In line 2, the subtraction should be Max_Temp-Low rather than High-Low. The correct computation is as follows:

```
255-((Max_Temp-Temp) * 255/
(Max_Temp-Low))
```

Identifying this type of problem is time consuming. In addition, it is well known that identification and removal of errors in the implementation or integration phase is much more costly than it is during the requirements phase.

Figure 8 provides a summary of a minimal set of test values for the translated condition table for Valve. In this figure, the associated test selection mode (i.e, LOW_BOUND, HIGH_BOUND) is also shown.

Sample Results

Table 3 shows some sample results on the application of this approach to other systems. Each specification originally had one or more specification problems or anomalies. As seen in Figure 5, the electronic regulator problem is very small (two tables, five functional relationships, six constraints, and a maximum depth of two table dependencies). A flight guidance system is a real-world industrial problem [16]; it has 78 tables, 423 functional relationships, 7,349 constraints, and a maximum dependency depth of 12. The results on this project are planned for publication in the next year.

			Output	Inputs				Test Selection
Mode	Conditions		Valve	Sensor	_Sensor	Temp	_Temp	Mode
HIGH	TRUE	FALSE	85.00	HIGH	READY	120	-100	LOW_BOUND
			255.00	HIGH	READY	179	119	HIGH_BOUND
LOW, READY	FALSE	TRUE	0.00	LOW	HIGH	-100	180	LOW_BOUND
			0.00	LOW	HIGH	119	300	HIGH_BOUND
			0.00	READY	LOW	-100	180	LOW_BOUND
			0.00	READY	LOW	119	300	HIGH_BOUND

Figure 8. Test Vectors for Valve Condition Table

Summary

Software testing will play a role in the development of software systems for some time to come. Although testing can account for 40 to 75 percent of the lifetime development and maintenance costs, the results summarized in this article provide promising evidence that the use of test automation to support the manually intensive test generation and model-based analysis is feasible and practical.

There is a great need to demonstrate and integrate new and advanced technologies. This article describes an environment developed to validate the use of model-based translators on real-world applications. The environment integrates model-based development tools with a specification-based test vector generator and specification-based coverage analyzer.

As modeling tools and associated methodologies continue to evolve, these results provide the basis for building translators for other modeling tools. This allows new tooling technology to be integrated with existing tools and has the indirect effects of reducing the cost and time of specialized training and tool expenditures.

The ability to integrate front-end development tools with back-end testing tools fosters the use of model-development tools, and such tools can significantly reduce the maintenance phase of a product, which typically consumes 70 percent of the product life cycle. Maintenance typically requires minimal development effort but typically large efforts in testing. Because the original developers usually are not available to assist in maintenance and evolution efforts, test automation can significantly minimize reverification efforts

continued on page 16

Table 3. Sample Results Statistic

System/ Projects	Condition Table	Event Table	Mode Table	Functional Relationship	Constraint	Level
Temperature regulator	1		1	5	6	2
Safety injection	1	1	1	10	68	3
Electronic flight instrumentation system	37	5	0	88	389	3
Elevator system	10	6	0	38	90	3
Flight Guidance system	49	15	14	423	7349	12

Using Models for Test Generation and Analysis

Continued from page 15

because the designer's requirement and design knowledge is captured in model specifications.

About the Author

Mark R. Blackburn, Ph.D., is the President of T-VEC Technologies, Inc. and co-inventor of the T-VEC system, an advanced specification and verification environment. Blackburn has eighteen years of software systems engineering experience in development, project leadership and applied research in software systems engineering experience in development, project leadership and applied research in specification-based testing, object technology, requirement and design specification, formal methods, and formal verification.

Mark is currently Chief Technologist at the Software Productivity Consortium where his current assignment includes the development of a specification-based test automation framework; he is developing a generalized specification-based testing model and language that is being used to support translators for four specification-based methods and associated tools (two requirement specification methods, a real-time OO design specification method, and a hybrid structured/object-based design method). He has also been involved in applied research and advanced technology demonstrations in web-based knowledge engineering, domain engineering, reverse engineering of programs to specifications, object technology, formal methods approaches to high assurance, requirement specification and model-based verification.

References

[1] Blackburn, M.R., R.D. Busser, "T-VEC: A Tool for Developing Critical System," *Proceeding of the Eleventh International Conference on Computer Assurance*, Gaithersburg, Maryland, pages 237-249, June, 1996.

[2] Blackburn, M.R., R.D. Busser, J.S. Fontaine, "Automatic Generation of Test Vectors for SCR-Style Specifications," *Proceeding of the 12th Annual Conference on Computer Assurance*, Gaithersburg, Maryland, pages 54-67, June, 1997.

[3] Beizer, B., *Software Testing Techniques*, New York, New York: Van Nostrand Reinhold, 1983.

[4] Blackburn, M.R., J.S. Fontaine, "Specification Transformation to Support Automated Testing," TR SPC-97036-MC, Version 02.00.01, Software Productivity Consortium, March 1998.

[5] Blackburn, M.R., "Specification Transformation and Semantic Expansion to Support Automated Testing," Ph.D. Dissertation, George Mason University, 1998.

[6] Cooke, D., A. Gates, E. Demirors, O. Demirors, M. Tankik, B. Kramer, "Languages for the Specification of Software," *Journal of Systems Software*, 32:269-308, 1996.

[7] Cohen, B., W. T. Harwood, M.I. Jackson, *The Specification of Complex System*, Addison-Wesley, Great Britain, 1988.

[8] Goodenough, J. B., S. L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Transactions on Software Engineering*, 1(2):156-173, 1975.

[9] Gill, D. H., "Formal Methods for Software Evolution, Solicitation," Defense Advanced Research Projects Agency, BAA 98-1O, November 1997

[10] Gourlay, J.S., "Introduction to the Formal Treatment of Testing, Software Validation," *Proceeding of the Symposium on Software Validation*, 1983.

[11] Ghiassi, M., K.I.S. Woldman, "Dual Programming Approach to Software Testing," *Software Quality Journal*, 3:45 58, 1994.

[12] Heninger, K., "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Transactions on Software Engineering*, 6(1):2-13, 1980.

[13] Hierons, R. M., "Testing from a Z Specification," *Journal of Software Testing, Verification and Reliability*, 7:19-33, 1997.

[14] Heitmeyer, C., R. Jeffords, B. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM TOSEM*, 5(3):231-261, 1996.

[15] Howden, W.E., "Reliability of the Path Analysis Testing Strategy," *IEEE Transactions on Software Engineering*, 2(9):208-215, 1976.

[16] Miller, S., "Specifying the Mode Logic of a Flight Guidance System in CoRE and SCR," Accepted to the Second Workshop on Formal Methods in Software Practice (FMSP'98), Clearwater Beach, Florida, March, 1998.

[17] Selic, B., G. Gullekson, P.T. Ward, *Real Time Object-Oriented Modeling*. New York, New York: John Wiley & Sons, 1994.

[18] Software Productivity Consortium, *Consortium Requirements Engineering Guidebook*, SPC-92060-CMC, version 01.00.09. Herndon, Virginia, 1993.

[19] White, L.J., E.I. Cohen, "A Domain Strategy for Computer Program Testing," *IEEE Transactions on Software Engineering*, 6(3):247-257 May, 1980.

[20] Zeil, S.J., "Perturbation Techniques for Detecting Domain Errors," *IEEE Transactions on Software Engineering*, 15(6):737-746, 1989.

[21] Zave, P., M. Jackson, "Four Dark Corners of Requirements Engineering," *ACM Transactions on Software Engineering and Methodology*, 6(1): 1-30, 1997.

Task-Based Software Testing

Daniel G. Telford, MacAulay Brown, Inc.

Introduction

There is a plethora of software testing techniques available to a development team. A survey by Zhu, et. al.[1] identified over 200 unit testing techniques. However, for the services' operational test agencies, there has been a continuing, unanswered question of how to test software's impact on a system's mission effectiveness. I propose a task-based approach as part of an integrated test strategy in an effort to answer this long-standing question.

Why Test?

From a speech by Lloyd K. Mosemann II, at the time the Deputy Assistant Secretary for the Air Force (Communications, Computers, and Support Systems) [2], a customer's concerns are:

> *They want systems that are on-time, within budget, that satisfy user requirements, and are reliable.*

A report from the National Research Council[3] refines the latter two concerns in his statement by presenting two broad objectives for operational testing:

1. to help certify, through significance testing, that a system's performance satisfies its requirements as specified in the ORD and related documents, and
2. to identify any serious deficiencies in the system design that need correction before full rate production

Following the path from the system level to software, these two reasons are consistent with the two primary reasons for testing software or software intensive systems [4,5,6]. Stated generically, these are:

1. test for defects so they can be fixed, and
2. test for confidence in the software

The literature often refers to these as "debug" and "operational" testing, respectively [4]. Debug testing is usually conducted using a combination of functional test techniques and structural test techniques. The goal is to locate defects in the most cost-effective manner and correct the defects, ensuring the performance satisfies the user requirements. Operational testing is based on the expected usage profile for a system. The goal is to estimate the confidence in a system, ensuring the system is reliable for its intended use.

Task-Based Testing

Task-based testing, as I define it here, is a variation on operational testing. It uses current DoD doctrine and policy to build a framework for designing tests. The particular techniques are not new, rather it leverages commonly accepted techniques by placing them within the context of current DoD operational and acquisition strategies.

Task Analysis

Task-based testing, as the name implies, uses task analysis. Within the DoD, this begins with the Uniform Joint Task List [7] and, in the case of the Air Force, is closely aligned with the Air Force Task List (AFTL) [8]. The AFTL *"...provides a comprehensive framework for all of the tasks that the Air Force performs."* Through a series of hierarchical task analyses, each unit within the service creates a Mission Essential Task List (METL). The Mission Essential Tasks (METs) are *"...only those tasks that represent the indispensable tasks to that particular organization."*

METLs, however, only describe "what" needs to be done, not "how" or "who." Further task decomposition identifies the system(s) and people required to carry out a mission essential task. Another level of decomposition results in the system tasks (i.e. functions) a system must provide. This is, naturally, the level in which developers and testers are most interested. From a tester's perspective, this framework identifies the most important functions to test by correlating functions against the mission essential tasks a system is designed to support.

This is distinctly different from the typical functional testing or "test-to-spec" approach where each function or specification carries equal importance. Ideally, there should be no function or specification which does not contribute to a task, but in reality there are often requirements, specifications, and capabilities which do not or minimally support a mission essential task. Using task analysis, one identifies those functions impacting the successful completion of mission essential tasks and highlights them for testing.

Operational Profiles

The above process alone has great benefit in identifying what functions are the most important to test. However, the task analysis above only identifies the mission essential

continued on page 18

Task-Based Software Testing

Continued from page 17

tasks and functions, not their frequency of use. Greater utility can be gained by combining the mission essential tasks with an operational profile—an estimate of the relative frequency of inputs that represent field use. This has several benefits:

1. *"...offers a basis for reliability assessment, so that the developer can have not only the assurance of having tried to improve the software, but also has an estimate of the reliability actually achieved."* [4]
2. *"...provides a common base for communicating with the developers about the intended use of the system and how it will be evaluated."* [3]
3. *"When testing schedules and budgets are tightly constrained, this design yields the highest practical reliability because if failures are seen they would be the high frequency failures."* [3]

The first benefit has the advantage of applying statistical techniques, both in the design of tests and in the analysis of resulting data. Software reliability estimation methods such as those in [5] and [9] are available to estimate both the expected field reliability and the rate of growth in reliability. This directly supports an answer to the long-standing question about software's impact on a system's mission effectiveness as well as answering Mr. Mosemann II's fourth concern a customer has (is it reliable).

Operational profiles are criticized as being difficult to develop. However, as part of its current operations and acquisition strategy, the DoD inherently develops an operational profile. At higher levels, this is reflected in such documents as the Analysis of Alternatives (AOA), the Operational Requirements Document (ORD), Operations Plans, Concept of Operations (CONOPS), etc. Closer to the tester's realm is the interaction between the user and the developer which the current acquisition strategy encourages. The tester can act as a facilitator in helping the user refine his or her needs while providing insight to the developer on expected use. This highlights the second benefit above the communication between the user, developer, and tester.

The third benefit is certainly of interest in today's environment of shrinking budgets and manpower, shorter schedules (spiral acquisition), and greater demands on a system. Despite years of improvement in the software development process, one still sees systems which have gone through intensive debug testing (statement coverage, branch coverage, etc.) and "test-to-spec," but still fail to satisfy the customer's concerns as stated by Mr. Mosemann II. By involving a customer early in the process to develop an operational profile, the most needed functions to support a task will be developed and tested first, increasing the likelihood of satisfying the customer's four concerns.

Task-Based Software Testing

Task-based software testing, as defined herein, is the combination of a task analysis and an operational profile. The task analysis helps partition the input domain into mission essential tasks and the system functions which support them. Operational profiles, based on these tasks, are developed to further focus the testing effort.

Integrated Testing

Operational testing is not without its weaknesses. As a rather obvious example of this, one can raise the question, "What about a critical feature that is seldom executed?" Operational testing, or task-based testing as defined herein, does not address such questions well. Debug testing, with the explicit goal of locating defects in a cost-effective manner, is more suited to this.

Debug Testing

Debug testing is *"...directed at finding as many bugs as possible, by either sampling all situations likely to produce failures (e.g., methods informed by code coverage or specification criteria), or concentrating on those that are considered most likely to produce failures (e.g., stress testing or boundary testing methods)."* [4] Zhu's, et. al. [1] survey of unit testing methods are examples of debug testing methods. These include such techniques as statement testing, branch testing, basis path testing, etc. Typically associated with these methods are some criteria based on coverage, thus they are sometimes referred to as coverage methods. Debug testing is based on a tester's hypothesis of the likely types and locations of bugs. Consequently, the effectiveness of this method depends heavily on whether the tester's assumptions are correct.

If a developer and/or tester has a process in place to correctly identify the potential types and locations of bugs, then debug testing may be very effective at finding bugs. If a "standard" or "blind" approach is used, such as statement testing for

continued on page 19

its own sake, the testing effort may be ineffectual and wasted. A subtle hazard of debug testing is that it may uncover many failures, but in the process wastes test and repair effort without notably improving the software because the failures occur at a negligible rate during field use.

Integration of Test Methods

Historically, a system's developer relied on debug testing (which includes functional or "test-to-spec" testing). Testing with the perspective of how the system would by employed was not seen until an operational test agency (OTA) became involved. Even on the occasions when developmental test took on an operational flavor, this is viewed as too late in the process. This historical approach to testing amplifies the weaknesses of both operational and debug testing. I propose that task-based software testing be accelerated to a much earlier point in the acquisition process. This has the potential of countering each respective method's weaknesses with the other's strengths. This view is supported by the current philosophy in the test community, to develop a combined test force spanning contractor, developmental, and operational test (CT/DT/OT).

Summary

Task-based software evaluation is a combination of demonstrated, existing methods (task analysis and operational testing). Its strength lies in matching well with the DoD's current operational strategy of mission essential tasks and the acquisition community's goal to deliver operational capability quickly. By integrating task-based software testing with existing debug testing, the risk of meeting the customer's four concerns (on-time, within budget, satisfies requirements, and is reliable) can be reduced.

Caveat

The success of the process presented herein, like so many of the processes presented in the software engineering community, is only a proposal at this point. However, as pointed out earlier, many of the individual components of task-based software testing are not new and have been shown effective both in the literature and in the author's personal experience. Task-based software testing is an approach of taking established methods and techniques and matching them against the current DoD operations and acquisition strategy.

About the Author

Daniel G. Telford is a systems engineer for MacAulay Brown, Inc., a support contractor for the U.S. Air Force. Prior to beginning a second career at MacAulay Brown, he completed a career as an officer in the U.S. Air Force with experience in field operations, operational test, and acquisition.

Author Contact Information

Daniel G. Telford
MacAulay Brown, Inc.
11728 Linn Ave. NE, Suite B
Albuquerque, NM 87123

dan.telford@macb.com

References

[1] Hong Zhu, Patrick A.V. Hall, and John H.R. May, "Software Unit Test Coverage and Adequacy," *Communications of the ACM*, Volume 29, #4, December 1997.

[2] Lloyd K. Moseman II, Deputy Assistant Secretary of the Air Force for Communications, Computers, and Support Systems. Speech to the Software Technology Conference, Salt Lake City, UT, 1994.

[3] Michael L. Cohen, John E. Rolph, and Duane L. Steffey, editors, *Statistics, Testing, and Defense Acquisition: New Approaches and Methodological Improvements*, National Academy Press, Washington, D.C., 1998.

[4] Phyllis Frankl, Dick Hamlet, Bev Littlewood, and Lorenzo Strigini, "Evaluating Testing Methods by Delivered Reliability," *IEEE Transactions on Software Engineering*, 24 (8), 1998.

[5] Michael A. Friedman and Jeffrey M. Voas, *Software Assessment: Reliability, Safety, Testability*, John Wiley & Sons, 1995.

[6] *Little Book of Testing*, Vol I and II, Computers and Concepts Associates, under contract to the Software Program Managers Network, 1998.

[7] "Chairman of the Joint Chiefs of Staff Manual (CJCSM) 3500.04A, Universal Joint Task List."

[8] "Air Force Doctrine Document 1-1, Air Force Task List," 12 August 1998.

[9] Michael R. Lyu, editor, *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, McGraw-Hill, 1996.

Thread-Based Integration Testing: Lessons Learned from an Iterative Approach

William M. Borgia, Neil J. Hrdlick, Northrop Grumman Corporation

Introduction

Our organization has recently completed the development of a large-scale command and control system through the implementation and formal qualification phases of the project. This development involved over eighty software engineers developing roughly 1.5 million source lines of code using multiple languages and platforms. In order to deliver the product within the projected schedule, parallel development and rapid integration occurred over many related software functional areas. To facilitate the decomposition of our design into manageable components we chose the concept of a "functional thread" as the elementary building block for integration. In this context, a "functional thread" is defined as a logical execution sequence through a series of interfacing software components resulting from or ending in the receipt of a message, event or operator interaction.

Threads not only serve as the basis for integration, they also tend to drive the entire software development effort from scheduling to status reporting. Each thread itself represents a microcosm of the system in that each has a documented definition and general execution path, an internal design and an associated test. Thread definition intends to communicate functional background and execution details between developers and from developers to testers. More importantly, the desired independence of threads supports incremental integration and system testing while the corresponding thread definition substantiates the results. Finally, since all system development activity progresses in relation to threads, management has an accurate method of judging the status of individual tasks, functional areas and requirements.

Threads

Keeping the goals of iterative development and testing in mind, each thread has its own lifecycle with autonomous states and a formal process for state transitions (see Figure 1). Individual team leaders usually decompose general requirements into groups of threads at the beginning of formal, six month software builds and assign threads to developers. Developers maintain ownership of their threads and are responsible for documenting a scenario under which an integrator can verify the basic functionality, providing rudimentary definition to the thread. Following implementation and unit test, the developer releases the corresponding software components to a daily integration build, at which point the thread enters a "testable" state. After verifying the functionality in the integration build, the developer marks the thread "ready" for an integrator who performs more extensive testing and eventually "integrates" the thread and corresponding software components into the system. At the end of each formal build, a team of key engineers in conjunction with quality assurance checks all threads against requirements as a regression test and "finalizes" those threads which pass.

While the development team originally tracked threads manually, we quickly developed a shared database application to serve as a

continued on page 21

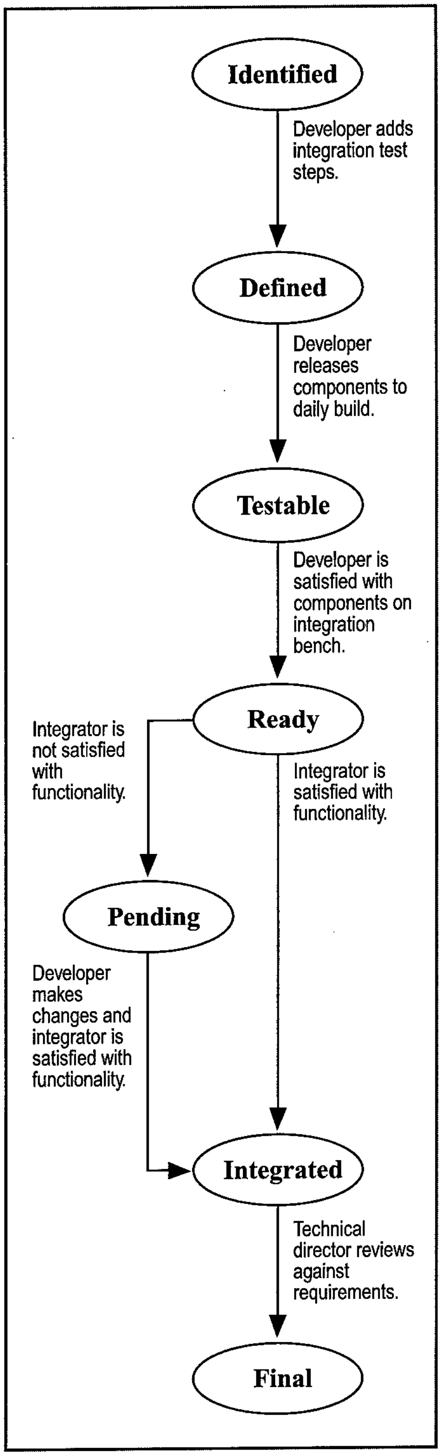

Identified
Developer adds integration test steps.
Defined
Developer releases components to daily build.
Testable
Developer is satisfied with components on integration bench.
Ready
Integrator is not satisfied with functionality.
Integrator is satisfied with functionality.
Pending
Developer makes changes and integrator is satisfied with functionality.
Integrated
Technical director reviews against requirements.
Final

Figure 1. Thread State Transition Diagram

central repository for thread development, maintenance and tracking. The database provides a formal mechanism for defining and documenting threads, changing thread status and reporting status to project management. Moreover, the database manages references between threads: threads can serve as preconditions to other threads and developers may incorporate thread test steps from previous threads. Most importantly, the interface helps enforce the process by demonstrating the autonomy of thread status and establishing clearly defined responsibilities among developers and testers.

Thread Test Steps

Thread test steps and other background information from the database serve as a contract between developers and integrators. Integrators use thread test steps as a simple scenario to identify the scope of a thread rather than as a rigid test case that may only rubber-stamp a developer's unit test. Consequently, the integrators are responsible for developing several execution scenarios within the boundaries of the thread and applying appropriate testing mechanisms such as known exceptional cases and boundary checking. Furthermore, the integration team often stresses exercising subsystem interfaces during integration testing, which was an area that thread steps often overlooked.

In addition to helping formalize the implementation process, the thread testing approach standardizes the integration testing process as well. As a result, the number of detected coding errors increased almost 250 percent over three formal builds after thread testing had been introduced. Although errors attributable to integration doubled during the first formal build during which our group used threads, that number has subsequently dropped to almost fifty percent below the level at which we started using threads.

While thread-based development certainly contributes greatly to the main goals of early, rapid integration and iterative development, we have also identified several potential areas of further process improvement. Perhaps most notably, developers and testers shared concerns that thread scope lacked uniformity among subsystems. At times, thread definitions were far too specific and a conscientious integrator could verify the basic functionality in fewer steps than the developer identified. Likewise, developers sometimes defined threads at too high a level, requiring the integrator to seek further information from the developer to ensure a meaningful test. A thread review process, perhaps as part of a design walk through, may answer this problem. Likewise, we recommend requiring completion of a code walk through as a prerequisite to thread completion due to the implications of walk through initiated design and code changes.

Thread Maintenance

A related area of improvement is thread maintenance. While the process encouraged (and the database supported) threads referencing other threads, maintaining consistency was not always an easy task. Furthermore, while software that composes a thread often changes after a thread has been integrated, there is no formal update process for the thread. The changes to process here are obvious and one could modify the tool to help enforce these concerns. For example, the tool would benefit from the ability to attach references to source code units so that changes to code might trigger the need for associated thread changes.

In this project the thread process focused on the integration activities rather than the full development lifecycle. This is certainly the main difference between our thread-based approach and use-case analysis. The thread database requires references to user interface specifications where applicable, but the process did not link the thread directly to the requirements database. Thus software testing and overall system testing were somewhat disjoint in that system testers found it difficult to use the thread database as a reference when creating test cases. Though it might be desirable to shift thread definition to the requirements analysis phases of the project, such analysis usually occurs at a higher level than what we had used for our threads and almost always span subsystem boundaries. Instead we suggest a more hierarchical approach to thread definition rooted in requirement-based parent threads.

This would directly link the software thread repository to system requirements and better facilitate a similar iterative approach to system-wide testing. Finally, by linking threads directly to requirements, project management would have better insight about the status of entire requirements.

Since threads drove the software efforts and status, developers viewed threads as the most visible formal

continued on page 22

Thread-Based Integration Testing

Continued from page 21

process in place. The simplicity of the process, accurate status and integration efficiency contributed to the development team's acceptance of the process and enthusiasm to suggest improvements. In addition, the empirical results suggest that the introduction of thread-based testing exposed design and coding errors earlier and attributed fewer errors to the integration process itself, probably due to the enhanced communication between developers and testers. In short, our method appears to have synchronized the notion of task completion among developers, testers and management.

Summary

Thread-based integration testing played a key role in the success of this software project. At the lowest level, it provided integrators with better knowledge of the scope of what to test, in effect a contract between developers and testers. At the highest level, it provided a unified status tracking method and facilitated an agreement between management and the developers as to what would be delivered during each formal build. Furthermore, instead of testing software components directly, it required integrators to focus on testing logical execution paths in the context of the entire system. Because of this, it strongly supported the goals of early, rapid integration coupled with an iterative development approach. In summary, the thread approach resulted in tangible executable scenarios driving development and integration while the autonomous, well-defined thread states strengthened the use of threads as an accurate method of scheduling and tracking status.

About the Authors

William M. Borgia received a B.S. in computer science from Truman State University in Kirksville, Missouri.

He currently serves as a software engineer for command and control systems at Northrop Grumman Electronic Sensors and Systems Sector in Baltimore, Maryland.

Neil J. Hrdlick received a B.S. in computer science from the University of Maryland and a M. S. in computer science from The Johns Hopkins University.

As a fellow engineer at Northrop Grumman Electronic Sensors and Systems Sector in Baltimore, Maryland, he serves as software technical director for command and control systems.

Author Contact Information

William M. Borgia
Northrop Grumman Corporation
Electronic Sensors &
Systems Sector
Mailstop B320
Box 17320
Baltimore, MD 21203
(410) 993-2875
Borgia@acm.org

Neil J. Hrdlick
Northrop Grumman Corporation
Electronic Sensors &
Systems Sector
Mailstop B320
Box 17320
Baltimore, MD 21203
(410) 765-1578
Neil_j_hrdlick@mail.northgrum.com
http://sensor.northgrum.com/

The *Software Tech News* is a publication of the DoD Data & Analysis Center for Software (DACS). The DACS is a DTIC Sponsored Information Analysis Center (IAC) and the DoD Software Information Clearinghouse. The DACS is operated by ITT Industries, Systems Division and technically managed by Air Force Research Laboratory - Information Directorate (AFRL/IF).

DACS: (800) 214-7921 or www.dacs.dtic.mil

DoD DACS Products & Services Order Form

Name: Position/Title:

Organization: Acronym:

Address:

City: State: Zip Code:

Country: E-mail:

Telephone: Fax:

Product Description	*Format*	*Quantity*	*Price*	*Total*
The DACS Information Package	*Note: All Disks are available in PC or Mac			
□ Including: 2 recent Software Tech News newsletters, and several DACS Products & Services Brochures	Documents		FREE	FREE
Empirical Data				
□ Architecture Research Facility (ARF) Error Dataset	Disk		$ 50	
□ NASA / Software Engineering Laboratory (SEL) Dataset	CD-ROM		$ 50	
□ NASA / AMES Error/Fault Dataset	Disk		$ 50	
□ Software Reliability Dataset	Disk		$ 50	
□ DACS Productivity Dataset	Disk		$ 50	
Technical Reports				
□ A Business Case for Software Process Improvement	Document	FREE with Spreadsheet →	$ 25	
□ ROI from Software Process Improvement Spreadsheet	Disk		$ 40	
□ A History of Software Measurement at Rome Laboratory	Document		$ 25	
□ An Analysis of Two Formal Methods: VDM and Z	Document		$ 25	
□ An Overview of Object-Oriented Design	Document		$ 25	
□ Artificial Neural Networks Technology	Document		$ 25	
□ A Review of Formal Methods	Document		$ 25	
□ A Review of Non-Ada to Ada Conversion	Document		$ 25	
NEW! □ Using Defect Tracking & Analysis to Improve SW Quality	Document		$ 50	
□ Software Design Methods	Document		$ 25	
□ Distributable Database Technology	Document		$ 25	
□ Electronic Publishing on the World Wide Web: An Engineering Approach	Document	SALE Item! →	$ 5	
NEW! □ Object Oriented Database Management Systems (Revisited)	Document		$ 50	
□ Software Analysis and Testing Technologies	Document		$ 25	
□ Software Design Methods	Document		$ 25	
□ Software Prototyping and Requirements Engineering	Document		$ 25	
□ Software Interoperability	Document		$ 25	
□ Software Reusability	Document		$ 25	
NEW! □ Understanding & Improving Technology Transfer in Software Engineering	Document		$ 50	
Bibliographic Products				
□ DACS Custom Bibliographic Search	Disk		$ 40	
□ DACS Software Engineering Bibliographic Database (SEBD)	CD-ROM		$ 50	

Number of Items Ordered □ *Total Cost* □

Method of Payment:

□ Check □ Mastercard □ Visa

Credit Card # ______________________ Expiration Date ______________

Name on Credit Card ______________________ Signature ______________________

Mail this form or: Phone: (315) 334-4905, Fax: (315) 334-4964
E-mail: **cust-liasn@dacs.dtic.mil**

This form is also on-line at: www.dacs.dtic.mil/forms/orderform.shtml

---fold here---

---fold here---

____

____

____

Affix
postage
here

DoD Data & Analysis Center for Software
Attn: DACS Customer Liaison
PO. Box 1400
Rome, NY 13442-1400