(2)

IDA PAPER P-2493

A STRATEGIC DEFENSE INITIATIVE ORGANIZATION SOFTWARE TESTING INITIATIVE

Bill R. Brykczynski, *Task Leader*

Reginald N. Meeson
Christine Youngblut

October 1990

Prepared for
Strategic Defense Initiative Organization

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 90-036099

91

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE October 1990	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE A Strategic Defense Initiative Organization Software Testing Initiative	5. FUNDING NUMBERS MDA 903 89 C 0003 Task T-R2-597.21
6. AUTHOR(S) Bill R. Brykczynski, Reginald N. Meeson, Christine Youngblut	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Institute for Defense Analyses (IDA) 1801 N. Beauregard St. Alexandria, VA 22311-1772	8. PERFORMING ORGANIZATION REPORT NUMBER IDA Paper P-2493
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Strategic Defense Initiative Organization (SDIO) SDIO/ENT Room 1E149, The Pentagon Washington, D.C. 20301-7100	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release, unlimited distribution: 24 June 1991.	12b. DISTRIBUTION CODE 2A

13. ABSTRACT (Maximum 200 words)

This paper provides a top-level description of an initiative designed to develop critically needed software testing capabilities prior to a Strategic Defense System (SDS) Milestone II decision. It identifies several SDI software testing requirements and examines the state of the art and practice in this field. Deficiencies were found between needed Strategic Defense System (SDS) capabilities and the expected available technology. Aspects of SDS software testing that should be addressed are described, including the ability to assure the correctness of SDS software, the types of software to be tested, and the ability to perform required testing within cost and schedule constraints. The document contains several detailed example project descriptions, and provides a brief overview of the software testing technology horizon.

14. SUBJECT TERMS Software Testing, Technology Transition, State of the Art, State of the Practice			15. NUMBER OF PAGES 78
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

IDA PAPER P-2493

A STRATEGIC DEFENSE INITIATIVE ORGANIZATION SOFTWARE TESTING INITIATIVE

Bill R. Brykczynski, *Task Leader*

Reginald N. Meeson
Christine Youngblut

October 1990

INSTITUTE FOR DEFENSE ANALYSES
Contract MDA 903 89 C 0003
Task T-R2-597.21

PREFACE

This paper was prepared by the Institute for Defense Analyses (IDA) for the Strategic Defense Initiative Organization (SDIO), under contract MDA 903 89 C 0003, Subtask Order T-R2-597.21, "SDS Test and Evaluation." The objective of the subtask is to assist the SDIO in planning, executing, and monitoring software testing ana evaluation research, development, and practice.

In support of this objective, IDA identified several Strategic Defense Initiative (SDI) software testing requirements and examined the state of the art and practice in this field. Deficiencies were found between needed Strategic Defense System (SDS) capabilities and the expected available technology. The subject of this paper is a high-level description of a research and development initiative that addresses deficiencies in required SDS software testing technology.

This paper was reviewed by the following members of IDA: Robert Atwell, James Baldo, Michael Bloom, Dennis Fife, Karen Gordon, Audrey Hook, Richard Ivanetich, Terry Mayfield, and Richard Wexelblat.

EXECUTIVE SUMMARY

The ability to develop and demonstrate sufficient confidence that Strategic Defense System (SDS) operational software can achieve its mission objectives has been identified as a key technical issue many times throughout the program's history. An activity that contributes to establishing system confidence is software testing. Complicating SDS software characteristics and the current inadequate state of software testing practice combine to pose a significant challenge in building the SDS. This paper describes a plan to develop software testing capabilities that must be available prior to an SDS Milestone II decision.

Historically, 50% to 80% of the cost of software development is spent in testing. There is no reason to expect that testing will play a lesser role in SDS software development. Testing technology must do more than simply identify software defects. It must provide the Strategic Defense Initiative Organization (SDIO) with the ability to:

a. Relate the effect of latent software defects to the operational fitness of SDS software in order to assess the probability that the SDS will achieve its mission.

b. Determine what level of testing to perform and provide the information needed to control the testing process.

This report concludes that conventional testing methods sufficient for small-scale sequential software will not be adequate for testing SDS software. These methods are largely ad hoc and unlikely to scale up to the levels required by the SDS. Existing state-of-the-art techniques must be evaluated and inserted into use as appropriate. Furthermore, the real-time, concurrent, distributed, and fault-tolerance characteristics of SDS software compound the traditional challenge of testing. Here there is little technology ready for evaluation and transition, and ongoing research efforts must be supported.

SDIO cannot rely upon other organizations to produce the technology needed to test SDS software. The Department of Defense is funding virtually no research, demonstration, or transition efforts in this area. While the National Science Foundation is funding a few research projects related to SDS testing needs, its effort is minimal. This report recommends that SDIO sponsor a software testing initiative. Such an initiative would span 5 years and cost on the order of 20 million dollars. The activities in this initiative would consist of research and development projects, experiments to evaluate promising technology, and the transition of proven technology into practice. The initiative cannot be performed independently of other SDIO activities. Instead, it must be closely tied to near-term SDS software development efforts to demonstrate the capabilities of testing technology prior to a Milestone II decision. Since the lead time to produce effective testing techniques and the tools to support them will be between three and eight years, appropriate action must be taken now.

This report describes the software testing initiative at a high level. An introduction to software testing and its importance to the SDS is presented. Key aspects of SDS software testing and the challenges in addressing these aspects is also discussed. A description of three classes of projects needed for the initiative is provided with preliminary cost and schedule estimates. The report also presents a preliminary technical description of three sample projects representing each initiative class, and provides detail on the technologies from which projects in the initiative can be drawn.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Purpose

This paper presents a top-level description of an initiative designed to develop critically needed software testing capabilities prior to a Strategic Defense System (SDS) Milestone II decision.

1.2 Scope

Four key components contribute to building confidence in SDS operational software. These are modeling and simulation, engineering discipline, software testing, and operational evaluation (see Figure 1). Modeling and simulation build confidence in system requirements and design decisions. Engineering discipline attempts to avoid construction errors. Software testing attempts to detect errors so that they can be corrected before a system is deployed. Operational evaluation determines how successful the other three efforts were, but only after development is completed. This paper and the initiative it describes focus on software testing and the underlying technologies necessary to support its contribution to confidence.

The objectives of the proposed initiative include providing the Strategic Defense Initiative Organization (SDIO) with the ability to:

a. Relate the effect of latent software defects to the operational fitness of SDS software in order to assess the probability that the SDS will achieve its mission,

b. Track technology drivers; that is, assess the impact of advanced software development technologies on the ability to effectively test SDS software, and

c. Determine what level of testing to perform and, based on the extent of additional assurance that would be gained by further testing effort, determine when to stop software testing.

It is vital that the initiative be tightly integrated with near-term SDS software development efforts to provide the necessary demonstration of the capability to test SDS

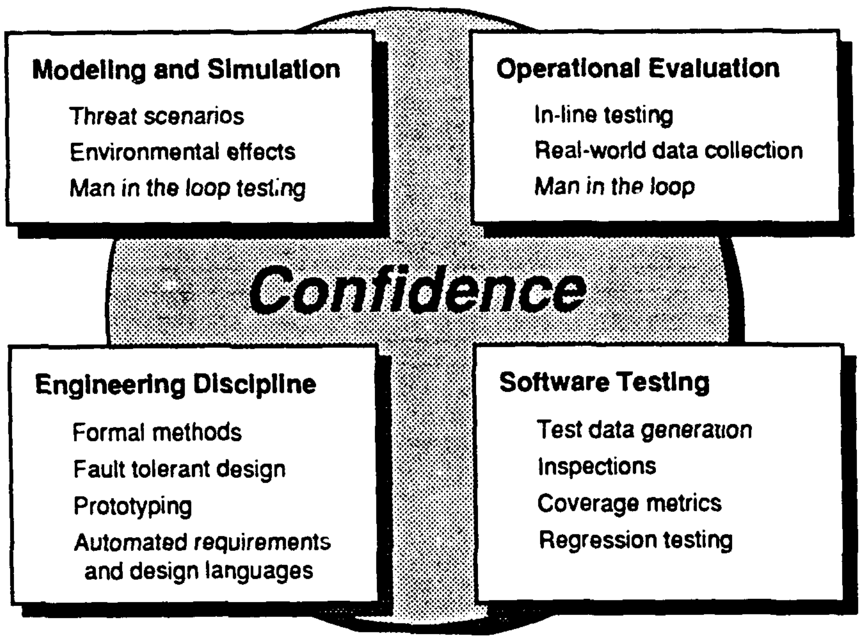

Figure 1. Key Contributors to Software Confidence

software. Development of systems such as the Level 2 System Simulator (L2SS) can also provide highly useful and visible vehicles for illustrating the benefits of improved testing technology to encourage its introduction into widespread use within the Strategic Defense Initiative (SDI) community. Moreover, as technology makes transitions into SDI practice throughout the course of the initiative, not only will a body of expertise in advanced testing practices be established, but the quality of Phase 1 software will be increased.

1.3 Background

The ability to develop and demonstrate sufficient confidence that SDS operational software can achieve SDS mission objectives has been identified as a key technical issue many times during the history of the SDI [Fletcher 1983, Cohen 1985, Parnas 1985]. Software testing will play a critical role in demonstrating SDS confidence.

Software testing involves the application of tools, techniques, and methodologies to determine the presence of defects.[1] These defects exist not just in code, but also in requirements, specifications, designs, documentation and all products of the software development process. To date, the majority of software testing research and development

1. A *defect* is an error in software code, design, or requirements that might cause faults. A *fault* is a manifestation of a defect. A *failure* is a serious fault that cannot be recovered from and prevents a system from achieving its mission.

has focused on relatively small-scale, sequential software. Unfortunately, even this available technology is seldom applied. This was illustrated by an effort sponsored by the Department of Defense (DOD) in 1981 to document current testing practices. This study found a fifteen year gap between the state of the art in software testing and the state of DOD practice [DeMillo 1987]. This gap has not closed in recent years. Moreover, techniques that can cost-effectively find defects in small pieces of software typically do not scale-up well for large systems. In addition, today's weapons systems exhibit characteristics which severely complicate the testing process. Technology for testing such software has not kept pace with the demands for this software and remains a research topic.

Testing is typically an ad hoc, labor-intensive effort only poorly supported by automated tools. Testing is usually applied as an independent activity towards the end of the development process to fix code that was not developed properly. As a result, defects are not found until long past their point of injection, resulting in rework that only adds to the high cost of software. As shown in Figure 2, data collected during the development of a number of early, large software systems (e.g., SAGE, NTDS, GEMINI, SATURN V, and IBM OS/360) reveal that software unit, integration, and system testing alone represent approximately one half of the software development effort [Boehm 1970, Alberts 1976]. Indeed, 80% of the total software development effort was applied to testing activities in the National Aeronautics and Space Administration (NASA) Apollo system [Dunn84]. There is no reason to believe that the proportion of effort applied to testing is reducing with time.

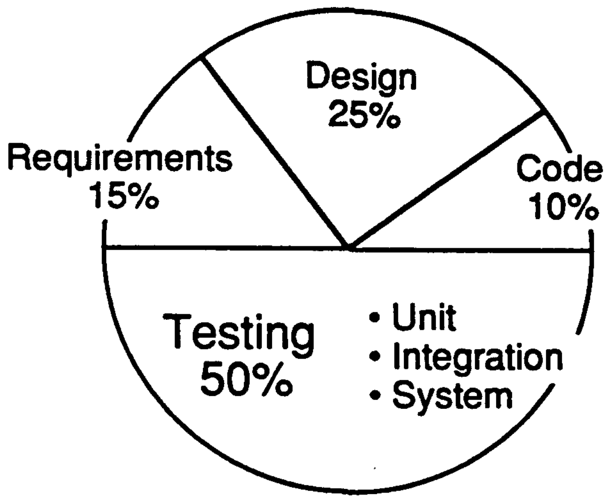

Figure 2. Breakdown of Software Development Activities

2. IMPORTANT ASPECTS OF SDS SOFTWARE TESTING

Three aspects of testing technology are important to the SDI: the ability to assure the correctness of SDS software, the types of software to be tested, and the ability to perform required testing within cost and schedule constraints. In this section of the report each of these aspects, as well as the difficulties associated with them, is discussed.

2.1 Establishing the Reliability of Software

Testing will be the primary means by which confidence in the correct operation of SDS software is gained during its development. In addition to defect detection, testing can identify potential problems areas, focus corrective actions, and support planning of subsequent testing activities. Testing not only addresses software functionality and performance directly, but guides the application of fault-tolerance techniques to maximize continued functionality and performance in the event of hardware and software faults, and to ensure continued safe operation. Testing also provides valuable insight into the appropriateness and effectiveness of development activities themselves which can be used to fine-tune the development process.

Facing the testing challenge requires a realistic assessment of the capabilities of available technology. Testing cannot be used to demonstrate the absence of *all* defects in software [Myers 1979, pp. 9-11, Boehm 1970, pp. 25] and it is generally accepted that all but the most trivial software will contain defects. Nonetheless, systems can be developed that achieve their mission objectives in spite of residual software defects. What is required is to distinguish between defects which may give rise to tolerable faults and those which may result in intolerable faults. The primary intent of SDS software testing must be to ensure that intolerable faults cannot occur and minimize the impact of other faults. This cannot be achieved after the bulk of the software has been implemented. Analysis of the overall SDS design must include assessments of the impact that software faults can have on SDS operation. The same argument can be made for analysis of the software system architecture and lower-level software subsystem designs to assess the potential impact of faults in lower-level software components.

Conventional measures of software reliability, such as mean-time-to-failure, are based on statistical modeling techniques originally developed for hardware systems.

There are, however, fundamental differences between software and hardware. For example, software does not wear out or fail randomly; if software fails, given the exact same conditions, it will produce exactly the same incorrect results again and again. Software reliability in this sense usually means "What is the probability of conditions arising that will make the software fail?" Hence, this is more a measure of the software's operating environment than an independent measure of the software itself. If the environment changes (e.g., if the distribution of inputs changes), the apparent reliability of the software can also be expected to change.

This is part of the larger question of understanding the results of testing. It arises because currently the only measures of testing effectiveness are indirect ones, typically based on the proportion of the software control structure that has been executed. Consequently, most testing is devoted to revealing the presence of defects, instead of their absence. The estimated number of defects remaining in the software, the potential conditions under which faults might arise, and the potential impact software faults might have on overall system operation contribute to confidence assessment. There are techniques for roughly estimating the number of remaining defects based on the defects already found. These techniques need refinement to allow increased accuracy by reflecting the number and types of tests run in finding these defects.

Meanwhile, there is reason for hope. There are a few state-of-the-art techniques which can conclusively demonstrate the absence of particular, *limited types of software* defects. These techniques provide for a high degree of confidence in certain aspects of the software.

2.2 Addressing Problematic Software Characteristics

Conventional testing methods sufficient for simple, sequential software are not adequate for the SDI. SDS software will be highly real-time, concurrent, distributed, and fault-tolerant. Each of these characteristics imposes particular testing problems, as noted in the following paragraphs.

a. **Large-scale software** consists of a million or more lines of source code. The complexity of such software systems, the diversity of possible inputs they must handle, and the potential conditions under which faults might arise increase dramatically with the size of a system. It is futile to expect that these systems can be tested by hand. Sophisticated testing tools are necessary to handle the many details.

b. **Real-time software** must execute within strict time requirements and a missed deadline is considered a fault even if computed values are correct. Consequently, testing must verify timing behavior as well as functionality. Analytical models and queuing theory can be used to analyze simple systems, but the simplifying assumptions that need to be introduced as complexity increases degrade model fidelity. As system complexity increases, it also becomes disproportionately more difficult to establish and control test conditions under which time-critical software must be exercised. Moreover, "invasive" testing techniques commonly used to check functiona ..y (e.g., executable assertions and probes that collect coverage data during execution) can distort performance such that the operational software cannot be examined.

c. **Concurrent software** consists of multiple tasks or "threads" that execute in parallel on multiple processors or asynchronously on a single processor. Cooperation between tasks is managed by operations that allow two tasks to synchronize and exchange information. Defects in concurrent software include all those that can appear in single-thread sequential programs, plus defects in synchronization. The difficulty of testing concurrent software arises from the indeterminism inherent in synchronization events, which means that two executions of the same software with the same inputs may not behave similarly. For static analysis, this results in high computation costs incurred by the need to consider all possible synchronization sequences and the traditional problem of determining path feasibility is exacerbated. In the case of dynamic testing, special actions are required to enable repeating a particular execution to aid in identifying the cause of a software fault, or to ensure the correctness of a modification made to remove a defect.

d. **Distributed software** is made up of cooperating processes that execute on multiple processors separated physically and connected via communications channels. These processes can be modeled using the same techniques used for concurrent software. The main difference is that synchronization requires communication between processors, which typically introduces additional latency and uncertainty into the timing of synchronizations. This uncontrollable variability increases the difficulty of systematically repeating faulty behavior. External communication channels also represent additional points of potential failure that concurrent tasks on single processors and closely-coupled multiprocessors typically do not have to address.

e. **Fault-tolerant software** is designed to ensure that intolerable faults cannot occur. Tolerance of hardware failures is usually achieved by the provision of redundant system components. Tolerance of software faults tends to focus on the use of independently-developed software versions (e.g., N-version programming), or fault detection and recovery mechanisms (e.g., recovery blocks). Testing such software involves the introduction of hardware and software faults in order to evaluate the response. The problem here is that the introduced defects may not be realistic. There are also problems specific to particular fault-tolerance techniques. For example, rigorous testing of N-version software must include validation of the underlying assumption of the independence of faults in each version.

The only methods in current practice for testing these types of software are ad hoc and cannot be relied upon to scale-up to the size and complexity anticipated for SDS software. The problems are, however, well recognized and technology to address them is being investigated. For example, in the case of reproducing the execution of concurrent software, basic support technology is under development which will provide a framework in which some existing testing techniques for sequential software can be applied.

2.3 Maximizing Effectiveness to Meet Budget and Schedule

There are two avenues for increasing the effectiveness and productivity of software testing. A significant step forward will be achieved by the provision of automated testing tools for both software development and support. There is a wealth of advanced prototypes that are waiting to be transformed into production-quality tools, or to be taken from a specialized development environment to one applicable for SDS. Rather than simply refining or rehosting available prototypes, careful planning for tool packaging would offer even greater improvements in productivity. For example, many testing techniques have common processing elements, such as an initial static analysis pass, to build an internal representation of the software under test. Productivity in testing activities can be increased by separating out these common elements into individual tool fragments so that the initial static analysis need only be performed once for each version of the software, instead of once for each application of each dynamic testing technique. Tool fragments can also increase productivity in tool production since each tool fragment only need be developed once.

Tools must be integrated with the planned SDS Software Engineering Environment (SEE). At the very least, a set of basic testing utilities such as tests drivers, debuggers, instrumentors, and coverage monitors must be provided. These utilities will form a common framework in which all testing will be performed, and with which additional tools must interface appropriately.

A second avenue for increasing the software testing effectiveness and productivity involves the cost effectiveness of available testing techniques. One of the classic questions that remains unanswered by current software testing technology is, how much testing is needed? Related questions include which techniques should be applied and to what extent? Testing budgets and schedules are typically based on experience gained in building similar systems and, all too often, testing simply stops when the budget or schedule so dictates. The effectiveness of various software testing techniques range from detecting all instances of particular, narrow but well-defined classes of defects (e.g., 100% of parameter type defects), to detecting smaller percentages on wider ranges of defects (e.g., 60 to 90% of all defects via inspections). A thorough understanding of the cost effectiveness of available techniques, including areas where techniques overlap, is needed to adequately plan and control testing activities for SDS software.

Theoretical approaches to the effectiveness of particular testing techniques are being pursued, but are unlikely to yield practical results in the next five years. In the meantime, data collected from the demonstration of techniques on near-term SDS software efforts will go a long way to resolving this shortcoming. This data will facilitate the establishment of cost effectiveness profiles and their refinement prior to Full Scale Development (FSD).

3. DESCRIPTION OF AN SDIO SOFTWARE TESTING INITIATIVE

This section of the paper describes three main classes of projects that the initiative will include. The costing for these projects is discussed, as well as the schedule for the initiative and methods for funding. Finally, as this paper does not provide all of the information necessary to plan and develop the initiative, a description of additional required work is presented.

3.1 Classes of Projects

Projects in the initiative can be broadly classified into three main classes. First, technology transition projects will emphasize the building of robust, production quality tools; training in the technology being inserted; and development of methods for moving the technology into the SDI state of practice. Production quality tools will reduce the amount of labor involved in the testing process, in addition to increasing software quality. Such tools will be identified as a part of the initiative, and then competitively procured. Training in the technology being inserted can be accomplished by those currently within the SDI community. It may take the form of instruction in the use of a tool or establishing guidelines for the use of non-automated testing techniques. Small-scale demonstration projects will be used to emphasize the benefits to be gained from new technology. Examples of topics for transition are:

a. *Formal software inspections applied to requirements, designs, code, and test plans and test cases.* Inspections have been proven in practice to be capable of finding 60% to 90% of software defects, while reducing total development costs by as much as 25% [Fagan 1986]. While many software developers do use some kind of structured review techniques, the rigor necessary to achieve these types of benefits is rarely applied.

b. *Basic support for dynamic analysis of sequential code products at the unit and integration levels.* Test drivers, debuggers, instrumentors, coverage monitors, and profilers are required to support all dynamic testing. While independent examples of each of these tools are available, their inclusion as an integrated set of basic testing utilities in the SDS software engineering environment will greatly facilitate testing effectiveness.

c. *Alternative criteria for control flow testing of sequential code products.* The only test data adequacy criteria in current use reflect the coverage of all program statements, branches, or paths by test data. Additional criteria reflecting coverage between the weak all-branches and the infeasible all-paths criteria will support better assessment of the effectiveness of dynamic testing.

Second, evaluation projects will focus on promising technology not yet demonstrated as cost effective on large systems. Technology of interest here includes the following:

a. *Fault tree analysis applied to sequential, concurrent, and distributed designs and code.* Identifies potential safety-related faults to guide the placement of fault-tolerance techniques and critical areas for additional testing focus.

b. *Interface analysis for sequential design and code products.* Allows increasing software confidence by demonstrating the absence of specific types of defects by analyzing the consistency and completeness within and among system components.

c. *Intra and interprocedural data flow testing applied to sequential code.* Provides additional measures of test data adequacy that is more defect discriminating than traditional control flow based measures.

d. *Symbolic evaluation of code products.* Serves as an independent static analysis technique as well as providing capabilities such as path analysis to support other testing activities.

e. *Control flow testing for concurrent code products.* Provides basic measures of test data adequacy to assess the effectiveness of dynamic testing of concurrent software.

Evaluation projects will require the development of prototype tools, techniques, and methods and the conduct of experiments to assess the cost effectiveness of various methods. They will require contact between the intended user of the technology and the demonstrator/evaluator of the technology. The user would be from an SDS development project, such as the L2SS, or other representative DOD software development. The demonstrator/evaluator would be from an organization possessing the resources to develop or modify small-scale prototype tools, having access to experts involved in the development of the technology, and familiar with combining these functions into a complete project.

Finally, research and development projects will strive for practical rather than theoretic solutions to the special problems of testing large-scale, real-time, distributed, concurrent and fault-tolerant software. Sample technologies in this case include:

a. *Animation of requirements products.* Allows a developer to gain insight into required behavior to support validation. Also useful for providing feedback on the interplay of design elements and operation of code.

b. *Behavior analysis of concurrent and distributed code products.* Supports use of self-checks to identify anomalous behavior both during testing and software operation.

c. *Deterministic execution testing for concurrent and distributed code products.* Supports reproducing an execution of parallel software to allow diagnosing the cause of a fault, or retesting software after it is changed.

d. *Performance analysis of concurrent and distributed designs.* Facilitates predicting software performance attributes to allow designers to make timely decisions about design alternatives.

e. *Regression testing of modified code products.* Minimizes the expense of retesting modified software by careful reuse of previous tests.

f. *Static concurrency analysis of concurrent and distributed code products.* Allows determining the presence and, in some cases, absence of synchronization faults such as deadlock.

g. *Timing analysis of real-time, concurrent, and distributed design and code products.* Supports comparison of expected and actual timing behavior against required behavior.

A preliminary technical description of three sample projects which represent each class is given in Appendix A. Appendix B provides more detail on the above mentioned technologies and some of the other technologies from which projects can be drawn. Additional information on the state of the art in software testing can be found in [Youngblut 1989].

3.2 Cost

Detailed cost planning for the initiative should include consideration of factors such as the criticality and status of technology, limits on how funding can be spent effectively, coordination opportunities within the SDI community, and the amount of available funding. Prior to thorough planning, only rough cost estimates can be given. These are based on 1) a general understanding of what types of tasks will be necessary to address the SDS software testing aspects described in the previous section, 2) the level of effort required for each project type, and 3) a sense for the number of projects that could be funded effectively. Table 1 summarizes estimates developed in their studv These results are used primarily to convey the order of magnitude of needed funding and its distribution among the classes of projects.

Table 1. Initiative Cost Data

Number of Projects	Type of Project	Avg. Cost Per Project	Total
40	Research and development projects	$150K	$6.0M
20	Technology evaluation projects	$450K	$9.0M
15	Technology transition projects	$300K	$4.5M
			$19.5M

The estimate of $150,000 per research project used in Table 1 is based on recent National Science Foundation (NSF) data. The NSF funds almost all government supported research in software testing. In 1988, only $600,000 was spent on projects in the area of software testing and fault-tolerance [NSF 1988], although related projects were funded in areas such as requirements specification, tool development, distributed computing, and formal verification. In the area of program testing, NSF awards average about $100,000, and range in duration from 1 to 3 years. The number of projects, 40, emphasizes the need for development of testing technology, much of which is currently in the research domain.

The estimate of $450,000 per technology evaluation project is based on past demonstrations and experiments with testing technology. Little software testing technology has been applied on actual development projects. Thus, there is a shortage of quantitative evidence on the effectiveness of promising tools, techniques, and methods. Demonstrations and experiments to provide this data will be performed during development of SDI testbeds (e.g., the L2SS and the Communications Network Testbed) and element software. Some projects will likely require multiple independent developments of the same software to produce statistically significant data. Developers will be trained in the use of the tool/technique that is being examined. Academic researchers will assist in developing experiments. In addition, an organization or group of people will be identified as the focal point for each experiment. All of these factors will impact the cost of an experiment.

The estimate of $300,000 per technology transition project is based on continuing the work of evaluation projects by identifying cases where proven tools can be inserted into SDIO use. Production quality tools must be developed. The existence of such tools, however, does not guarantee their use and real world data will be collected showing SDI program managers how the technology can benefit them.

3.3 Schedule

Without a detailed analysis comparing SDS software testing requirements against projections of what the research community could provide, it is difficult to develop any realistic schedules. However, two factors are assumed in this paper. First, the SDI Milestone II decision will occur in the 1995 timeframe and the majority of results should be known or accomplished by this time. Care will be taken to facilitate a continual infusion of technology into SDIO use throughout the initiative. Second, the type of projects described in the initiative are multi-year efforts, many of which will build upon the results of the previous efforts. Thus, for planning purposes, a 5-year time span for the initiative is used.

3.4 Funding

Given the breadth of project types within the initiative, a variety of programmatic funding issues arise. These issues include the methods used to fund and monitor individual projects, SDIO involvement in the initiative, and the oversight needed to ensure the appropriate coordination, selection, and evaluation of projects. For example, some of the research projects will require grants to the academic experts who are best qualified to produce useful results in a timely fashion. While national laboratories, service laboratories, and various government agencies have provided funding for research in the past, NSF is by far the most experienced organization in handling long-term research and possesses the infrastructure necessary to ensure successful direction of research funds. It is recommended that SDIO develop a cooperative agreement with NSF to assist in obtaining effective research towards fulfillment of SDS testing requirements.

In the case of evaluation projects, organizations such as the national and service laboratories, NASA, the Software Technology for Adaptable, Reliable Systems (STARS) program, and the Defense Advanced Research Projects Agency (DARPA) can assist SDIO by providing candidate projects whereby testing technologies can be evaluated.

However, it must be noted that SDIO cannot rely upon other organizations to produce the technology needed to test SDS software. For example, while NSF is funding a few research projects related to SDS needs, its effort is minimal. In general, the DOD is funding virtually no research, demonstration, or transition efforts in software testing.

3.5 Next Steps

Three primary tasks are required to fully define the initiative. First, criteria must be developed to identify those technologies of immediate interest to SDS and clearly rule out areas not applicable or useful. These criteria will form the basis for a comprehensive examination of promising technology. A prioritization of promising technology will be prepared in cooperation with support from industry and academia to determine current research focuses.

Second, the projects comprising the initiative will be selected. For the evaluation projects, candidate SDIO (or other DOD) projects must be identified to provide a testbed for evaluation. As these types of projects are tied to programmatic plans, evaluation projects will require careful coordination.

Finally, as the initiative will be managed by SDIO, detailed planning identifying, for instance, funding agents (e.g., NSF, Service R&D agencies, and SDIO) and schedules for each project must be developed. Coordination with other agencies is also needed. For example, the Office of Naval Technology (ONT) and NSF are currently exploring a collaborative software testing technology effort. SDIO should keep abreast of such activities and identify opportunities to cooperate with those efforts.

4. REFERENCES

Boehm, Barry W. 1970. *Some Information Processing Implications of Air Force Space Missions: 1970-1980.* Memorandum RM-6213-PR, Rand Corporation

Cohen, Danny, study chairman. 1985. Eastport Study Group, Summer Study 1985.

DeMillo, R.A., W.M. McCracken, R.J. Martin, and J.F. Passafiume. 1987. *Software Testing and Evaluation.*

Dunn, R.H. 1984. *Software Defect Removal.* McGraw-Hill.

Fagan, M.E. 1986. "Advances in Software Inspections." *IEEE: Transactions on Software Engineering,* 12/7, July, 744-751.

Fletcher, James C., study chairman. 1983. *Report of the Study on Eliminating the Threat Posed by Nuclear Ballistic Missiles,* SDIO.

Myers, Glenford J. 1979. *The Art of Software Testing.* John Wiley & Sons.

National Science Foundation. 1988. *Summary of Awards.*
Division of Computation Research. Directorate for Computer and Information Science and Engineering.

Parnas, David L. 1985. *Software Aspects of Strategic Defense Systems.* American Scientist, Sept.-Oct., pp. 432-440.

Youngblut, C., B. Brykczynski, K. Gordon, R.N. Meeson, and J. Salasin. February 1989. *SDS Software Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation With Recommended R&D Tasks.* Alexandria, VA: Institute for Defense Analyses. P-2132.

5. ACRONYMS

DARPA	Defense Advanced Research Projects Agency
DOD	Department of Defense
IDA	Institute for Defense Analyses
FSD	Full Scale Development
L2SS	Level 2 System Simulator
NASA	National Aeronautics and Space Administration
NSF	National Science Foundation
ONT	Office of Naval Technology
R&D	Research and Development
SDI	Strategic Defense Initiative
SDIO	Strategic Defense Initiative Organization
SDS	Strategic Defense System
SEE	Software Engineering Environment
STARS	Software Technology for Adaptable, Reliable Systems

APPENDIX A: SAMPLE PROJECT DESCRIPTIONS

This appendix describes three sample software testing technology projects. The descriptions convey the type of projects needed in areas of technology transition, demonstration of promising technology, and research and development.

A.1 Transitioning Software Inspection Technology into Practice

A.1.1 Introduction

This section outlines a project for increasing software quality and reducing development costs by transitioning software inspections into SDIO-wide practice. The longer software defects remain undetected, the more expensive they become to correct. For example, the correction of a requirements defect during coding activities may incur 70 times the cost of correcting that same error during requirements specification [Dunn 1984]. Unfortunately, the types of static and dynamic analysis techniques used to detect code defects are rarely applicable to requirements and design products. Consequently, these defects are difficult to detect and a prime contributor to the high cost of software development and subsequent corrective maintenance. Since inspections can be applied at each stage of software development, they provide for timely defect detection and can be expected to yield significant savings in development and maintenance costs.

In the case of SDS, the expected use of real-time, concurrent, and fault-tolerant software will exacerbate and extend the usual difficulties in testing requirements and design products to code products. Software inspections are one of the few techniques that can be used for testing critical software of this kind.

The following section provides a brief overview of software inspection technology. The final section describes the necessary project steps, along with their associated estimated schedules and costs.

A.1.2 Background

Software inspections are a manual formal review technique. The software inspection process was developed in 1972 by Michael Fagan at IBM Kingston, NY [Fagan 1976]. Inspections can be applied to all types of software products, regardless of implementation issues such as real-time or concurrency. This allows inspections to be performed much nearer the point of injection of defects than most other forms of testing and, therefore, requires less resources for rework following the detection of a defect. In terms of defect detection, inspections have proven to be capable of finding 60-90% of software defects [Fagan 1986]. In practical development efforts they have been found to reduce total development costs by as much as 25%, and have actually resulted in the delivery of defect-free software.

One of the fundamental concepts for the inspection process (usually lacking in practice) is the definition of the software development process in terms of activities and their *exit criteria*. The purpose of an inspection is to verify that the output product of each activity satisfies the activity's exit criteria; essentially exit criteria define the output requirements of an activity. They are the standard against which inspections measure completion of the product at the end of an activity, and verify the presence or absence of quality attributes. The inspection process itself comprises six well-defined steps. The first three (*planning*, *overview*, and *preparation*) proceed the actual *inspection*. The final two (*rework* and *follow-up*) focus on correcting defects found in the inspection and ensuring that bad fixes are not included in a product. Participants in inspections play well-defined roles as *author*, *reader*, *tester*, and *moderator*.

The inspection process is well understood. Studies have isolated its key characteristics and fostered improvements such that inspections may routinely yield a very high defect detection efficiency. In [Fagan 1986], Fagan reports these improvements in the inspection process, along with data which demonstrates the value of software inspections in increasing software quality while reducing software costs. He also provides a me ısure for the defect detection efficiency of inspections. Statistical analysis and empirical data have revealed the prime contributors to inspection quality and indicators of quality inspections have been developed. This allows management to influence the quality, and the development and maintenance costs, of inspected products.

In addition to timely defect detection, there is substantial empirical evidence of many other benefits from the software inspection process. It promotes defect prevention by providing feedback that helps programmers to avoid making errors in future work. When applied to test plans, it provides a mechanism for improving the defect detection efficiency of other forms of testing. In the case of code products, there is a directly proportional relationship between the inspection detected defect rate and the defect rate found in a piece of code by subsequent testing. This allows inspection results to be used to identify defect prone code which requires special handling. Since exit criteria are the standard against which inspections measure completion of the product at the end of an activity, inspections provide checkpoints which facilitate process management. Finally, although not to be used for performance measurement of software developers themselves, the inspection process provides data useful for performance measurement of tools and techniques in individual development activities. Indeed, with respect to inspections themselves, inspection results initially should be analyzed to drive process improvement. Once a stable point is reached, results should be used for process control.

A.1.3 Description of Project

Experience has shown that a positive philosophy and set of attitudes is essential for the successful transition of any technology. Consequently, this project shall start with a demonstration of software inspections on an SDS software development effort. The results of this demonstration shall be widely disseminated among the SDIO community to encourage a good reception to required program-wide use of inspections. The subsequent activities required to introduce inspections into SDIO-wide usage shall exploit any lessons learned from the demonstration.

Step 1: Demonstration of Software Inspections. The inspection process shall be applied on a near-term, relatively short SDIO software development effort. The purpose of this application shall be to provide a highly visible vehicle for demonstrating the practical benefits of the inspection process. Candidate show cases should be selected on the basis of several characteristics, for example, their visibility to the SDIO community, the likelihood of timely results, and the breadth of inspection benefits that can be demonstrated. Since inspections are well proven in practice, their use is not anticipated to impose significant risk to the demonstration effort. The L2SS Build 1 is one example of a suitable candidate.

Prior to demonstration, project managers will be provided with 1 day of training in the operation and effects of inspections. In general, software engineers will receive 1 day of training for inspection participants, 5% of the software engineers, however, will receive 3 days of training in moderating inspections. An expert in software inspections will support the development organization in defining their software development process and the associated exit criteria and defect checklists. The development organization will also be provided with a defect reporting system (ideally, this will be an automated system) and training in the use of this system. Requirements for data collection above that provided by the defect reporting system will be defined and, again, the development organization provided with any necessary tools and training. Since the chief objective will be to demonstrate the cost-effectiveness of inspections, rather than compare inspections against other forms of testing, these additional data requirements should be minimal.

The inspection expert should be available on call to assist with any problems that may arise. This expert should also make periodic visits to the development effort to ensure that the demonstration is going as planned and to validate sample portions of the data being collected.

Interviews with representative participants will be conducted once the demonstration is completed to provide information on the human reaction to the inspection process. Accumulated defect reports and other collected data will be analyzed to determine the degree of defect detection efficiency achieved by the inspections. Where information is available from previous software development efforts performed by same organization, the extent of achieved quality and productivity improvements will be assessed. This information will be extrapolated to estimate the benefits of software inspections for the development of SDS software as a whole. As appropriate, various mechanisms may be used for the dissemination of demonstration results through the SDIO community. Examples of such mechanisms include briefings at SDIO conferences and memos to project managers and other suitable people.

The commencement of this step will depend upon the selection of a near-term SDS software development effort as a suitable demonstration vehicle. After the necessary set-up, the demonstration shall proceed in parallel with the chosen development effort. Data collection shall terminate with the completion of inspections on integration and system testing plans and cases, or inspections on code products, whichever is later. This step can be conducted independently of any other testing efforts. The cost of this step will include (1) costs incurred by the development organization who will be applying inspections (training costs, set up costs, and data collection costs), and (2) costs incurred in data analysis and dissemination of demonstration results.

Step 2: Bringing Inspections into Common SDIO Practice. Education has been found to be the key prerequisite for transitioning software inspections into practice. Consequently, a training program for managers, inspection moderators, and other inspection participants shall be established.

Organizations preparing to use inspections for the first time shall be provided with guidance to help in defining the organization's software development process and the accompanying exit criteria. Guidance is also required for developing checklists for each type of product to be inspected, together with the establishment of a process for continually refining these checklists based on inspection results. Ideally, initial checklists will be based on available quality control information such as the identification of frequently occurring errors. Although inspections themselves require no automated support, identified defects should be entered into a defect reporting system. Finally, guidance in establishing a database which shall record inspection results so that they may be used for quality control purposes and to support assessment of the effectiveness of the inspection process itself shall be provided.

Management should guide the application of inspections in the planning of a software development effort. For example, one typical effect is a slight front-loading in the commitment of people resources for software development. Guidance for exploiting inspection results to facilitate development process control shall also be prepared. Finally, the policy necessary to support the introduction of inspections on a consistent, SDIO-wide basis shall be drafted.

This step can commence on the completion of the demonstration step above. It should be coordinated with the introduction of a formal defect reporting system.

References

Dunn, R.H. 1984. *Software Defect Removal.* McGraw-Hill.

Fagan, M.E. 1976. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal*, Vol. 15, No. 3, 182-211.

Fagan, M.E. 1986. "Advances in Software Inspections." *IEEE: Transactions on Software Engineering*, Vol. SE-12, No. 7, 744-751.

A.2 Development of a Data Flow Testing Tool

A.2.1 Introduction

This section outlines a project to develop, and demonstrate, a production-quality tool for the application of data flow testing to Ada software. Current testing practices are insufficient for a critical application such as the SDS which must operate successfully on its first activation. Available statistics indicate that 54% of all defects are not found until acceptance testing or later [Boehm 1975]. Indeed, the software community has been unable to identify a single major application that operated correctly on its first use. One of the objectives of data flow testing is to increase software reliability by increasing the detection of software defects during unit and integration testing.

Regardless of the effectiveness of available testing techniques, the cost of current testing practices pose a significant problem for the SDS. The Air Force Systems Command has reported that approximately 50% of software development costs are typically spent in unit, integration, and system testing alone. The major proportion of these costs are incurred by the human-intensive activities of selecting test data and examining test results required for dynamic testing of code products. This problem is exacerbated by the difficulty of determining when to stop testing. In theory the ideal stopping point is when the software is defect-free. However, because current dynamic testing approaches are based on detecting the presence of defects, rather than their absence, there are no absolute measures for this point. Instead, indirect measures of test data adequacy based on the proportion of the control structure of a program that has been exercised are used. Data flow testing will help to reduce testing costs by supporting the automated generation of test data, providing test data adequacy measures based on data flow, and allowing easy determination of the minimum set of tests needed to verify the implementation of a software modification.

A brief overview of data flow testing technology is provided. This is followed by a description of the necessary steps for the project, supported by schedule and cost estimates.

A.2.2 Background

Data flow testing is a dynamic analysis technique that embodies a set of criteria that can support both test data generation and the assessment of test data adequacy [Frankl 1988]. These criteria reflect the intuition that the path from a variable assignment to its use must be executed to provide confidence that the correct value was assigned to that variable. They provide better support for automated test data generation than criteria based on control structure [Ural 1988]. They are also more discriminating; for example, Girgis reports that the all-c-uses criterion could detect 48% of faults and all-p-uses could discover 34%, giving a total of 54%-70% for these two data flow criteria, while all-branches only could detect 34% [Girgis 1986]. The *selectivity* of criteria is another pertinent issue. Selective criteria are those which never require more, and may require less, test paths to achieve some testing goal. While data flow criteria are not more selective than control flow criteria in general, they are more efficient for some practical testing goals, namely, with respect to the selection of simple transference paths, and differentiation from constant values [Zeil 1988].

Data flow testing is based on the data flow analysis originally used in compiler optimization. Initially, data flow analysis only examined the data dependencies that exist within a program unit; this is termed intraprocedural testing. Recently, algorithms for analyzing data dependencies that exist between program units have been developed, supporting not only interprocedural data flow testing, but also more accurate intraprocedural testing [Harrison 1989]. Incremental data flow analysis has been another focus of recent research. Here the objective is to reduce the cost of maintenance activities, which currently are estimated to consume 30-80% of the total cost of a system, by controlling the scope of retesting needed after a software change has been made. Data flow analysis can be used to identify the changes in program paths from the definition to the use of a variable that result from a modification and, thus, limit retesting to only the modified and new associations. Figures on the time and space complexity of such incremental data flow testing are given in [Taha 1989].

A.2.3 Description of Project

Development of a production-quality data flow testing capability shall consist of three steps. It should be noted that there are several data flow testing prototype tools in existence. The developers of the key prototypes shall be encouraged to share their experience in the form of "lessons learned." In particular, the Ada-based prototype recently developed as part of the Testing, Evaluation, and Analysis Medley (TEAM)[2] project shall be carefully examined to determine whether there are existing tool fragments that can be exploited for the current effort.

Step 1: Develop Tool Specification. Develop a detailed specification of the desired functional and performance capabilities of a production-quality tool that supports the incremental application of data flow criteria for both test data generation and test data adequacy evaluation for Ada software at the unit and

2. This project is being undertaken by the Arcadia consortium which is headed by the University of Massachusetts and the University of California at Los Angeles.

integration levels. This requires the resolution of several issues. One such issue is how higher quality test data should be traded-off against the difficulty of accurate data flow analysis. This requires, for example, determining whether array references will be disambiguated and what degree of infeasible path elimination will be provided. Not all tool issues can be resolved through consideration of data flow testing in isolation. For example, many testing techniques have common processing elements such as an initial static analysis pass to build an internal representation of the software under test. Productivity in testing activities can be increased by separating out these common elements into individual tool fragments so that the initial static analysis need only be performed once for each version of the software, instead of once for each application of each dynamic testing technique. Tool fragments can also increase productivity in tool production since each tool fragment only need be developed once. The specification shall detail the required use of tool fragments.

The specification shall provide a quantifiable basis for monitoring the tool development effort and assessing its results. Additionally, it shall specify requirements for data collection to support cost-effectiveness analysis of the implemented technology.

This task can commence immediately. It can be conducted independently of any other testing efforts.

Step 2: Implement Tool and Integrate into SEE Testing Support Toolset. The tool shall be implemented in accord with the requirements laid out in the SDIO Directive 3405, Software Policy. The final products shall be integrated into the basic testing support toolset as appropriate (see Section B.2). In particular, the instrumentor and coverage monitor shall be extended to facilitate the use of data flow test data adequacy criteria, and the test driver shall support execution of tests employing the outputs of a data flow test data generation component. Appropriate support for data collection to support cost effectiveness analysis of data flow testing shall be provided.

This step may commence once the tool specification becomes available. It must follow, or be carefully coordinated with, the development of SEE basic testing support toolset.

Step 3: Evaluate Products Via Application on SDS Development Effort. The products of the implementation step shall be applied on the coding and unit test and the integration phases of a near-term SDS software development effort. This evaluation exercise shall serve several purposes. For example, it serves as a demonstration vehicle for data flow testing preparatory to introducing this technology into SDIO-wide use. It shall also allow the collection of data on the practical cost/effectiveness of data flow testing as implemented. The users of the technology shall be trained in the application of data flow testing, use of the automated support, and necessary data collection. Expert assistance shall be made available throughout the course of the demonstration. On completion of integration testing activities, the data collected shall be used to develop decision criteria that aid in the selection of the most appropriate data flow criteria for particular development efforts.

The evaluation step must be coordinated with a suitable SDS software development effort. Such an effort should be relatively short, and be highly visible to the SDIO community. It will last through the code and unit test and the integration phases of the chosen development effort.

References

Bieman, J., and J. Schultz. 1989. "Estimating the Number of Test Cases Required to Satisfy the All-du-paths Testing Criterion." In *Proceedings ACM 3rd Symposium on Software Testing, Analysis, and Verification*, 13-15 December, Key West, Florida, 179-186, 1989.

Boehm, B.W., R.K. McClean, and D.B. Urfig. "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software." *Proceedings International Conference on Reliable Software*, published in *SIGPLAN Notices*, Vol. 10, No. 6, (June 1975):105-113.

Frankl, P.G., and E.J. Weyuker. Oct 1988. "An Applicable Family of Data Flow Testing Criteria." *IEEE: Transactions on Software Engineering,* 14/10, 1483-1498.

Girgis, M.R., and M.R. Woodward. 1986. "An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria." In *Proceedings Workshop on Software Testing*, 15-17 July, Banff, Canada, 64-73, 1986.

Harrold, M.J. January 1989. *An Approach to Incremental Testing*. University of Pittsburgh. TR-89-1.

Taha, A.-B., S.M. Thebaut, and S.-S. Liu. 1989. "An Approach to Software Fault Localization and Revalidation Based on Incremental Data Flow Analysis." In *Proceedings 13th Annual International Computer Software and Applications Conference*, Orlando, FL, September 20-22, 527-534.

Ural, H., and B. Yang. "A Structural Test Selection Criterion." *Information Processing Letters,* 28/4 (Jul 1988):157-163.

Zeil, S.J. 1988. "Selectivity of Data-Flow and Control-Flow Path Criteria." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 216-222. IEEE Press.

A.3 Research Into Deterministic Execution Testing and Debugging

A.3.1 Introduction

This section outlines a project to determine the feasibility and cost-effectiveness of applying deterministic execution testing to concurrent Ada software. Dynamic analysis is the predominant defect detection method for concurrent software, that is, software designed as parallel processes executing on a single processor, multi-processor, or distributed processor. While 35-50% of programmer effort is spent in defect removal for sequential software, the difficulties in testing concurrent software necessitate significantly more effort [Dunn 1984].

The prime cause of difficulties in the dynamic analysis of concurrent software is the inherent execution indeterminism, which means that two executions of the same software with the same inputs may behave differently. Consequently, the traditional dynamic approach of executing a program once with each input can fail to detect many concurrency-related defects. Moreover, an execution which does reveal a defect cannot be repeated to facilitate determining the cause of that defect and, subsequently, to ensure that it was corrected properly. Deterministic execution testing provides a way to capture the synchronization behavior of a concurrent program and to reproduce this behavior as necessary. As such, it provides a foundational framework for concurrent software testing, particular test data generation, and test data adequacy assessment schemes that can then be employed within this framework. To this end, this technology should, ultimately, be included in the SDS software engineering environment (SEE) basic testing support toolset.

The following section provides a brief technical overview of deterministic execution testing. The final section describes the steps necessary to develop prototype tools and investigate some outstanding technology issues.

A.3.2 Background

Deterministic execution testing is based on the concept of *SYN-sequences* [Tai 1989]. A SYN-sequence is the set of synchronization events that occur in the execution of a concurrent program and allows unambiguous specification of that execution. A *feasible* SYN-sequence is one that can be exercised during an execution of a program P. A *valid* SYN-sequence is one expected to be exercised according to the specification of P. Tai has developed a formal definition of correctness for concurrent programs which forms the basis of deterministic execution testing; in essence, if the validity of feasible SYN-sequences is not examined, some synchronization defects will not be detected. The basic elements of deterministic execution testing are SYN-sequence collection, determining SYN-sequence feasibility, and forcing SYN-sequence replay.

SYN-sequence collection is achieved by transforming a program P into an equivalent program P', except that during an execution of P' the SYN-sequence of this execution is collected. Determination of SYN-sequence feasibility starts with transforming P into another equivalent program P* which will produce the same results as P if the SYN-sequence is feasible. This second transformation introduces a special control task and causes each rendezvous event to be replaced with entry calls to this task. The control task reads in a SYN-sequence that comprises a sequence of entry call arrivals and rendezvous events such that arrivals of entry calls occur as late as possible (LAR-sequence). It then tries to force the given LAR-sequence to be exercised by returning from entry calls when the next event to occur in the given LAR-sequence is the execution of a rendezvous event by the task. If the sequence is feasible, P* issues a completion message indicating that the sequence was exercised and P* terminated as expected. Otherwise, P* issues either a timeout message indicating that the time interval between the most recent synchronization events and the current time is longer than that allowed, or an unexpected event message occurred. SYN-sequence replay forces execution in a similar way SYN-sequence feasibility determination, although here the SYN-sequence can be simplified.

Deterministic execution testing is still under development. It is, however, a fundamental technology for effective testing of concurrent software that is critically needed.

A.3.3 Description of Project

This project is, potentially, the first in a series of projects designed to bring deterministic execution testing into SDIO practice. It requires consolidating existing technology and experience in distributed execution testing to evaluate its near-term feasibility and potential cost effectiveness. These investigations shall result in a set of recommendations for productization of the basic technology, and for any necessary additional research into outstanding technology areas.

Step 1: Develop Prototype Tools. Evaluation of the feasibility of deterministic execution testing shall be based on the development of prototype tools applying SYN-sequence collection, feasibility evaluation, and replay to concurrent Ada software operating on single, multi-, and distributed processors. These prototypes shall be applied to available real-world software to assess the likely cost-effectiveness of this technology

The step can begin now. It can be performed independently of any other testing efforts.

Step 2: Investigate Integration of SYN-Sequence Replay with Debuggers. In order to achieve the maximum benefit of SYN-sequence replay for debugging concurrent software, debugging techniques should be directly integrated with deterministic execution testing. In point of fact, SYN_sequence replay tools may need to be integrated with several debuggers, since there are significant differences between debugging concurrent software operating on a single processor, a multiprocessor, or distributed processors. For example, one of the major difficulties in debugging distributed software is the need to halt all processors simultaneously. Moreover, such a debugger may be centralized or itself distributed among the various machines. This step shall integrate the prototype SYN-sequence replay tool with sample available debuggers for concurrent software.

This task will proceed in parallel with step 1 above. It can be conducted independently of any other testing efforts.

Step 3: Investigate Support for Incremental Testing. A significant portion of software costs are incurred in retesting software after it has been changed. The identification of new or modified SYN-sequences can be used to determine the actual scope of a change and so reduce the cost of retesting by limiting it to the affected parts of the software. The prototype tools shall be extended to support the identification and execution of new or modified SYN-sequences.

This task will proceed in parallel with Step 1 above. It can be conducted independently of any other testing efforts.

Step 4: Investigate Support for Target-Host Reproducibility. Process-control or embedded software is often developed on a host machine which provides the development support not available on the intended target machine. While the majority of testing occurs on the host, some testing must also be performed on the target. If defects are detected during target testing, there may be no facilities to support diagnosing the cause of the defect. This step shall involve determining the feasibility of modifying the prototypes to develop tools capable of reconstructing an erroneous target execution on the host machine [Taylor 1982].

This task will proceed in parallel with steps 1 through 3 above.

References

Dunn, R.H. 1984. *Software Defect Removal.* McGraw-Hill.

Tai, K.C., R.H. Carver, and E.E. Obaid. 1989. "Deterministic Execution Debugging of Concurrent Ada Programs." In *Proceedings 13th Annual International Computer Software and Applications Conference*, Orlando, FL, September 20-22, 102-109.

Taylor, R.N. November 1982. *Debugging Real-Time Software in a Host-Target Environment.* University of California at Irvine. TR-212.

APPENDIX B: SOFTWARE TESTING TECHNOLOGY HORIZON

This appendix provides an overview of the current state-of-art in software testing. It provides a series of descriptions on selected technology areas. While by no means addressing all possible topics, it illustrates how state-of-art technology might benefit the testing of SDS software. As can be seen, each area may contain technology ready for transition or evaluation and demonstration, and needed research. Consequently, there is not necessarily a one-to-one mapping between technology areas and potential projects in a testing initiative.

There are some general points that apply to all the technology areas, or all instances of one type of initiative project. For example, quantitative data on the cost-effectiveness of each testing technique is urgently needed. This data would support both test planning and the monitoring and control of test activities. Additionally, it would guide software testers so that each technique is applied appropriately under given circumstances. Every evaluation and demonstration project shall collect data to facilitate the establishment of cost-effectiveness profiles. Where appropriate, transition projects shall also establish necessary data collection activities so that the cost-effectiveness profile of a particular technique can be extended and refined prior to FSD development.

It is important that planning for the development of tools to support testing activities shall be considered with respect to the testing initiative as a whole, not on a project by project basis. For example, many testing techniques have common processing elements such as an initial static analysis pass to build an internal representation of the software under test. Productivity in testing activities can be increased by separating out these common elements into individual tool fragments so that the initial static analysis need only be performed once for each version of the software, instead of once for each application of each dynamic testing technique. Tool fragments can also increase productivity in tool production since each tool fragment only need be developed once. It is also useful to conceptually separate basic testing utilities from specialized testing techniques. This basic testing support toolset then forms a common framework in which all testing will be performed, and with which additional tools must interface appropriately.

B.1 Animation

The advent of high-resolution graphics workstations and pointing devices makes possible the creation and display of drawings whose elements can be continuously moved for realistic animation. At the program level, this animation provides the user with visual feedback that both increases the user's understanding and aids in defect detection [Feldman 1989]. Indeed, it is becoming an integral part of the new generation of graphical debuggers. Program data structures are animated by abstract, pictorial representations or textual views showing the contents of variables, whereas algorithms are animated by graphical maps of the state of the computation domain of the program. Animation of concurrent software is subject to the usual problems arising from the indeterminism in process scheduling. Solutions to this problem are focusing on graph-based representations that represent the full set of possible, correct event-orderings for a set of concurrent processes. The absence of a global clock presents additional difficulties when concurrent processes are distributed across several machines. Here an independent animation of concurrent programs on each site may be used, or a global site-transparent animation.

Animation is also being proposed for interpretation and validation of requirements specifications. Current approaches include translating a specification given in a formal language such as VDM into a logic programming language [Bloomfield 1986]. This not only facilitates animation but provides a diverse implementation for use in back-to-back testing. Alternatively, facilities may be provided for the selection and execution of a transaction to reflect the specified behavior of a particular scenario given in the specification [Kramer 1988]. In a broader application, animation is being investigated as the basis for a visual paradigm capable of supporting the representation and refinement of interactions among the conceptual entities in a system design [Moriconi 1983].

Experimentation with prototype tools for program animation have shown that a visual view of the operation of concurrent processes is instrumental in quickly identifying software defects that would otherwise be difficult to detect [Socha 1988]. The ability to interact with an animation by, say, changing data values has also been demonstrated to be a valuable tool for algorithm design and analysis, whereas role playing certain actions can support the validation of requirements specifications.

Although some prototype tools for program animation may be ready for evaluation, much of this technology is still in early stages of development.

B.2 Basic Dynamic Testing Support Toolset

Dynamic testing refers to the activity of executing a software unit, or subsystem, with a set of input data and comparing the actual outputs against expected outputs to identify possible defects. If an fault does occur, the execution may be halted or repeated, or tracing information collected to aid in isolating the defect which caused the fault to manifest. Usually, the test run is monitored to collect data on the number of times particular control structure elements are executed. Regardless of the types of test data generation used, all dynamic testing activities have some common elements. For sequential software, the primary common elements are test drivers, debuggers, instrumentors, and coverage monitors. There are many commercial tools, and prototypes, available in each case.

It is vital that these common elements be provided as a basic testing support toolset in the SDS Software Engineering Environment (SEE). This will be the framework within which all dynamic testing of sequential software will be performed.

The same type of support is urgently needed for dynamic testing of concurrent and real-time SDS software. While the same support functions are required, these tools must address some special concerns not applicable for sequential software. Concurrent software, for example, contains additional control mechanisms that must be reflected in coverage measures, the interaction of the concurrent processes must be addressed by the test driver, and a debugger needs to be able to halt all processes at the same time. (The problem of indeterminism that results in a lack of execution reproducibility is dealt with in Section B.7). The primary issue for real-time software is that of interference. The addition of extra code for instrumentation or debugging purposes will impact the timing characteristics of the code under test and, therefore, the 'real' software is not being tested. This is an area requiring additional research and development. While some of the elements needed to support the testing of of concurrent software are ready for evaluation and demonstration, research is required before a full set of basic testing support for concurrent software can be made available.

In addition to the necessary support for dynamic testing of SDS software products, an automated problem reporting system is required to support all testing activities. A problem reporting system serves several purposes. Its primary usage is to record the faults or failures that occurred in all testing activities, together with the defect(s) that gave rise to each fault or failure. This information is typically supported by a defect classification and the status of the activities associated with correcting the underlying defect and subsequent retesting of the product in question. Problem information also is valuable for monitoring overall software quality. For example, the Air Force Systems Command

(AFSC) recommend that the trend in software defect density be used to assess readiness to proceed with the development phase, or for government acceptance of software, and to identify defect-prone software components that require special handling [AFSC 1987].

In addition to recording and reporting on defects, general information about testing activities should be kept to support controlling the testing process. For example, the AFSC propose using data on the tests scheduled, the tests passed, and unresolved problem reports to monitor testing progress [AFSC 1986]. Information on prior tests also is needed to support regression testing (see Section B.17). Finally, testing data and problems reports provide a source of insight into the effectiveness of the software development and testing activities that can be used to identify weak points in the development process and to evaluate new techniques.

B.3 Behavior Analysis

In general terms, behavior analysis is concerned with comparing the intended software behavior with the actual behavior. Static forms of behavior analysis for sequential and concurrent software are discussed in the sections on sequence analysis and static concurrency. Here the discussion focuses on dynamic behavior analysis where information ranging from functional requirements to background knowledge (such as the domain of values a program unit operates on) is captured to serve a variety of purposes throughout the software development process.

In the case of SDS, for example, behavior analysis could be used to support software requirements analysis by constructing formal specifications for software components from informal requirements and analyzing these for consistency, completeness, and compatibility with the specifications of other components. If formal specifications are expressed in a language that is executable or symbolically interpretable, they can support rapid prototyping for early prediction of software behavior. During implementation, verification techniques can be used to check code products for consistency with their formal specifications, which may serve as an initial implementation to be refined into executable code. Another advantage of formal specifications lies in their ability to be automatically transformed into runtime consistency checks. Such checks offer the potential to increase the effectiveness and productivity of software testing by automating certain types of software testing and debugging. They also can be used to automate the construction of self-testing software to promote software fault-tolerance.

One of the most advanced research efforts into behavior analysis is being performed by the Program Analysis and Verification Group of the Computer Systems Laboratory at Stanford University. This effort began in 1980. It has focused on the development of wide-spectrum languages for describing the intended behavior of both system software and hardware and the implementation of that behavior. Three languages have been developed. Anna is an extension of the Ada programming language and is used at the program unit level. The Task Sequencing Language (TSL) provides for specifying sequences of interactions between tasks in Ada software, these are included with the Ada text and monitored during software execution. The Hardware Design Language (HDL) combines concepts from Anna for software specification with features in the VHSIC Hardware Description Language for hardware specification. Prototype tools to support application of both Anna and TSL are available.

B.4 Control Flow Testing

The only 'formalized' testing technique in wide-spread use is structural, control flow testing. It provides test data adequacy criteria as a measure of the coverage of dynamic testing. The only criteria in common use correspond to a requirement to execute all program statements, branches, or paths at least once in the course of testing. They apply to the units of a sequential program. Unfortunately, while the all-statements and all-branches criteria are not very discriminating in terms of defect detection, the all-paths criterion is infeasible. Several alternative control structure criteria have been defined to provide a hierarchy of criteria which include coverage measures lying between the all-branches and all-paths criteria. One example of such an alternative is the set of criteria based on Linear Code Sequence and Jump (LCSAJ) blocks within a program [Hennell 1976, Ural 1988, Wu 1987]. Various studies have demonstrated the increased cost-effectiveness offered by such alternative criteria. Even so, they have yet to be brought into common practice. In addition to unit testing, test data adequacy criteria provide a valuable mechanism for guiding integration testing. Although various sets of criteria corresponding to the requirement to execute the parameters in each unit interface with different values have been proposed, these have also failed to make the transition into common practice.

Research is being conducted into control-based test data adequacy criteria to support the dynamic analysis of concurrent software [Taylor 1986]. In this case, criteria are typically based on the notion of concurrency states, where a concurrency state displays the next synchronization-related activity to occur in each task. A synchronization activity history, the concurrency graph, is given by the sequence of states that may occur for a class of program executions. The coverage criteria focus on concurrency state coverage, state transition coverage, and synchronization coverage. Requisite support includes a static concurrency analyzer, and either a program transformation system or a powerful run-time monitor for noting and recording the concurrency states encountered.

Also helpful is a controllable run-time scheduler so that the execution of specified concurrency paths can be guaranteed. This research is only beginning. As yet no prototype tools are available.

Production-quality tools supporting the use of effective control-based test data adequacy criteria for SDS software testing are a necessity. These will provide the basic means for monitoring the extent of dynamic testing of sequential SDS software that is a prerequisite for effective management of testing activities. To this end, prototypes allowing the comparison of alternative criteria on realistic sequential software should be developed. Advanced research on control-based testing for concurrent software should be performed to provide the SDIO with a basic capability for monitoring the extent of dynamic testing on concurrent SDS software. Care should be taken to exploit the potential consistency with static concurrency analysis, dynamic behavior analysis, and reachability analysis.

B.5 Data Flow Testing

Data flow testing reflects the intuition that the path from a variable assignment to its use must be executed to provide confidence that the correct value was assigned to that variable [Frankl 1988]. In a similar way to traditional control structure criteria, data flow criteria can be used to assess the adequacy of test data or to generate test data. Theoretical and empirical studies have shown, however, that certain data flow criteria are able to detect more defects than the control-based all-branches criterion. Data flow testing is a well developed technology that can be applied to detect defects both within a program unit and between units. It can be applied in an incremental fashion which allows highly efficient application during regression testing. The theoretical and empirical cost of various data flow criteria has been investigated [Bieman 1989]. Over recent years several prototypes tools have been developed to apply data flow testing to Fortran, Pascal, COBOL, and sequential Ada code.

Data flow testing provides additional measure of test data adequacy that offer more reliable SDS software than can be acquired through structural control-based testing alone. A production-quality tool to apply intra- and interprocedural, incremental data flow testing to sequential Ada software should be developed. Research to extend data flow testing to concurrent Ada software should be undertaken.

B.6 Dependency Analysis

Dependency analysis is a static analysis approach to capturing the dependencies between different entities in design and code products. These dependencies reflect both direct and indirect relationships between program units and may be represented by, for example, a call tree, a call matrix, and a transitive closure of the matrix. In addition to dependencies at the program unit level, dependencies between the blocks within a program unit may be identified. Dependency information is primarily used to support ripple effect analysis (see Section B.18), and to monitor the testing of modified programs.

This a basic capability that will be useful to support SDS software testing. One major effort in the development of tools to present and analyze software dependencies is being performed as part of the Maintenance Assistant project at the Purdue/Florida Software Engineering Research Center.

B.7 Deterministic Execution Testing

The prime cause of difficulties in the dynamic analysis of concurrent software is the inherent execution indeterminism which means that two executions of the same software with the same inputs may behave differently. Consequently, the traditional dynamic approach of executing a program once with each input can fail to detect many concurrency-related defects. Moreover, an execution which does reveal a defect cannot be repeated to facilitate determining the cause of that defect and, subsequently, to ensure that it was corrected properly. Deterministic execution testing provides a way to capture the synchronization behavior of a concurrent program and to reproduce this behavior as necessary. As such, it provides a foundational framework for concurrent software testing, particular test data generation and test data adequacy assessment schemes can then be employed within this framework.

Some insight into the collection of synchronization information and its use to replay a particular execution is being performed [Tai 1989a, Tai 1989b], though as yet no prototype tools are available. More research is needed in several areas. For example, to determine how careful identification of the impact of a software modification on the synchronization behavior might support efficient regression testing.

The ability to reproduce an execution as required is a fundamental capability for any testing of concurrent software. Moreover, with this capability, some existing techniques for dynamic analysis of sequential software can be applied to concurrent software. As such, this technology should, ultimately, be included in the SEE basic testing support toolset.

B.8 Domain Testing

Domain testing strategy has been developed primarily to detect errors in software control flow [White 1986]. The control flow statements in a program partition the input space into a set of mutually exclusive domains, each of which corresponds to a particular program path and consists of input data points which cause that path to be executed. Domain testing uses geometric analysis to generate test points to examine whether a domain error has occurred. If so, one or more of these boundaries will have shifted, or else the corresponding predicate relational operator has changed.

Domain testing offers an advantage over many dynamic analysis techniques in its ability to test the absence of certain defects, rather than merely their presence. As such, it would enable the confidence in particular, limited aspects of SDS software correctness to be stated in terms of a known error bound.

This technology has not yet advanced to the point of a practical testing technique. One of the difficulties is the high computation cost incurred. Recent research has investigated making cost-effectiveness trade-offs in the accuracy of error bound measures by adjusting the number of testing points, and their geometric positions, that are used [White 1988].

B.9 Fault-tree Analysis

Software Fault Tree Analysis is a static analysis technique for safety-critical, real-time software. In traditional, system Fault Tree Analysis a hazard is specified and the system is then analyzed in the context of its environment and operation to find credible sequences of events that can lead to this hazard. The general approach is to assume a safety-related fault has occurred and then work backward to determine the set of possible causes [Leveson 1983]. A fault tree built down to the software interface defines the high level requirements for software safety in terms of the software behavior that could adversely affect system safety. Software Fault Tree Analysis is then applied at the design and code level to identify safety critical items and conditions under which fault-tolerance and fail-safe procedures should be initiated.

With a system the projected size of the SDS, it would be a formidable task to identify all possible unsafe states. Yet Software Fault Tree Analysis may offer substantial benefits for critical SDS software components. In the current phase of the SDI program, it can be used in conjunction with a system simulator to examine the interfaces of the software fault tree to determine appropriate simulation states and events. For actual SDS software, it can guide the application of fault-tolerance procedures such as N-version software and recovery blocks, and the use of run-time checks to detect hazardous software states. It also can support testing by identifying critical functions and test cases.

This technology is ready for evaluation and demonstration. Prototype tools for generating software fault trees from Ada code are being developed as part of Murphy [Rolandelli 1986], a product of the UCI Safety Project.

B.10 Functional Testing

Functional testing refers to a class of dynamic analysis techniques that usuallycan be applied to all types of software regardless of implementation characteristics such as concurrency. They do not consider the internal software structure, but base test data generation on the functional requirements. Although functional testing techniques can be applied at all levels of software testing, their importance largely derives from the fact that they are one of the few approachessuitable for system and acceptance testing.

The most widely used technique is equivalence partitioning. This technique addresses the problem of trying to select the subset of all possible inputs thathave the highest probability of finding the most faults [Myers 1979. The software input domain is partitioned into a finite number of equivalence classeswhere, it is assumed, a test of a representative value of each class is equivalent to a test of any other value. The minimal set of test data covering all equivalence classes is developed by selecting test cases that invoke as many different input conditions as possible. Boundary value analysis is a variation on equivalence partitioning where values in both input and output equivalence classes are selected to test the edges of each class. Another variation, cause-effect graphing, explores combinations of input conditions by partitioning the output domain into causes corresponding to particular effects. The different classes, and links between them, are expressed in a combinatorial logic network called a cause-effect graph. The dependencies thus revealed are used to derive appropriate test cases.

Functional testing is needed for SDS software. Although there is some evidence of the use of tools to support its practical application, see [Solis 1985], more automated support is needed to facilitate effective and productive functional testing. The introduction of formal specification languages offers one avenue for increased automated support. Here some researchers are looking at the use of formal grammars developed from a formal specification to allow random generation of implementation-independent test cases or the application ofsystematic testing strategies such as boundary value analysis [Duncan 1981]. Others are investigating the use of algebraic specifications [Gannon 1981, Choquet 1986].

B.11 Interface Analysis

Describing the major modules and their interaction is the primary concern of architectural software design, while maintaining correct and consistent interfaces is a crucial part of coding and maintenance activities. Even programming language like Ada, which allows more detailed specification of program unit interfaces than is usually available, do not support describing the relationships among components with sufficient accuracy. Additional mechanisms for describing the relationships between software components are needed to support effective analysis on the consistency and completeness within and among system components, and on changes to the interface relationships resulting from a software modification. Recent advances have developed techniques for effective interface analysis which, unlike traditional approaches which must be delayed until the entire system is completed, can be performed on incomplete products.

For systems such as the SDS, controlling component relationships will be an extremely challenging and critical task and one that requires substantial automated aid. Tools to provide this aid must be provided. The Precise Interface Control (PIC) system is one example of a promising approach. AdaPIC is a particular instantiation of PIC for Ada-based development environments [Wolf 1986].

B.12 Mutation Analysis

Mutation analysis is a fault-based dynamic analysis technique capable of demonstrating the absence of a specified set of code defects [DeMillo 1989]. It is based on the "competent programmer hypothesis," which assumes that if the program being testing is not correct, it differs from a correct program by, at most, a few small errors. Mutation analysis allows the user to determine whether a set of test data is adequate to detect these defects. The technique involves constructing a collection of *mutants* of the software under test. Each mutant is identical to the original program except for a single syntactic change (e.g., replacing one operator by another). The mutants are executed on the same set of test data. Those which produce different output than the original program are said to be "killed," that is, the test data was adequate to find the errors that these mutants represent. Live mutants indicate that either the test data is inadequate or the mutant is equivalent to the original program. More test data can be added in an effort to kill nonequivalent mutants. The adequacy of a set of test data is measured by a score based on the percentage of nonequivalent mutants killed.

The value of mutation analysis lies in its ability to assure the absence of certain defects, not merely their presence. In terms of SDS software, this capability allows better understanding of the effectiveness of some testing activities and, hence, increased confidence in the software.

Prototype tools for mutation analysis are available for both FORTRAN and C. The major problem with this technique arises from extensive computational resources needed to execute each mutant on each test case, where the number of mutants required for a test program is bounded by the square of the number of references to variables and constants. In an effort to control this high cost, researchers are investigating the uses of parallel machines to support its implementation [DeMillo 1988, Krauser 1986]. Researchers are also investigating the application of mutation analysis for automated test data generation.

B.13 Performance Analysis

During design activities, performance analysis can be used to predict performance attributes of software prior to its implementation [Gilkey]. This requires modeling the significant structural elements of a system and it is usually necessary to consider the effect of environmental factors such as data dependency, competitive effects, and memory contention. Methods for the performance analysis of designs have been based on timed Petri-nets, abstract data types extended with performance information, and state models supported with a probabilistic grammar-based model of the input. For time critical concurrent processes, a rapid simulation can be developed to provide quick identifications of time critical regions and an understanding of model behaviors that allows the designer to tailer the model for the best timing performance prior to its implementation.

The key uses of performance analysis of code products are to guide program optimization and to assess the maximum execution time of a program [Lopriore 1989]. Although a fundamental theorem in the theory of computability implies that it is not possible to determine an upper bound on the running time of a program in general, it can be achieved for broad categories of interesting programs such as loop programs. A variety of approaches for performance analysis of programs have been proposed. One of the early ones uses a mechanized analysis for deriving a closed-form expression of program execution time expressed in terms of size of input. Later extensions to this approach consider the problem of finding efficient ways to determine, given the values for the input variables, the values of various program performance indices. In another approach the notion of a call-return tree is used to describe the dynamic calling relationship of the procedures and functions in a program execution, then to compute the live times and execution times of the various calls. It can also be used to compute additional behavioral metrics such as the depth and height of a call and the number of direct and indirect calls generated from any point [Kundu 1986]. Alternatively, an engineering-oriented performance model of a computation can be developed by extending the concept of a computation structure to cover the performance costs appropriate to software modeling. Such a model can cater for multiprocessors configurations, and the evaluation of both time and space parameters for alternate realizations.

In the limited domain of compiler performance analysis, benchmarking is a common method of evaluating performance. It allows comparing the relative efficiencies of compilers and their operating environments, and comparing the size and execution speed of generated machine code. Halstead's theory of software science has been used to describe the compilation process and generate a compiler performance index [Shaw 1989]. Here a nonlinear model of compile time is estimated with a fundamental relation between compile time and program modularity. The research suggests that the discrimination rate of a compiler is a valuable performance index preferable to average compile time statistics.

The primary importance of performance analysis for the development of SDS software will be in helping software engineers to make decisions about design alternatives. It is critical that these decisions are made in a timely manner, efforts to force an implemented system to meet performance requirements not allowed for in a faulty design are not only wasteful of resources, but are rarely fully successful. Of course, performance analysis is also valuable during coding activities, and to assess the potential performance impact of proposed modifications (see Section B.19).

While technology for performance analysis of sequential code products is ready for transition into practice, its application to concurrent software is still under research. While some prototype tools for performance analysis of designs have been developed, these are mainly used as research vehicles and few are ready for evaluation.

B.14 Profiling

Dynamic program measurement, commonly known as profiling, is a valuable tool in understanding software efficiency [Bentley 1987]. By measuring the actual behavior of programs during execution, tuning efforts can focus on those components that account for the significant part of the execution time. Traditionally this is done by inserting counters into programs either before compilation, during compilation, or during assembly. Timing information is then collected via interrupt-driven sampling or, alternatively, a per-process high precision timer in the underlying architecture. Another type of profiling involves monitoring the execution of a program to gather relevant statistics. This is particularly useful for analyzing a program's interprocess dependencies and providing a picture of what happened during the execution of distributed software. Perhaps the most advanced research in this areas is the investigation of profilers for large-scale, multiprocessors, such as shared-memory and hypercube machines, where a dynamic, fine-grain characterization of parallel program executions based on a partial order of accesses to shared objects has been developed [Francioni 1988].

Profiling is a basic capability that should be available to support SDS software testing. Profiling tools should be integrated with tools such as an interactive debugger, a graphical execution browser, and performance analysis packages to provide powerful support for interactive analysis of the behavior of SDS software.

B.15 Random Testing

Random testing is one of the primary techniques for acceptance testing, the other being functional testing, see Section B.10. It is based on the idea of sampling for faults and can both lead to the detection of defects and providea software reliability measure. In this latter case, the number of failures in a set of test data is related to a reliability measure via a probability distribution function. The probability distribution function depends on the way test data is chosen, which is randomly generated from either a uniform distribution of the software input domain or from the operational profile. Although the effectiveness of random testing has been questioned, recent studies have demonstrated its value [Duran 1984]. In particular, random testing has been shown to be as effective, if not superior, in defect detection ability to methods based on input partitioning [Hamlet 1988]. Additionally, once the input domain or operational profile is determined, generation of test data is usually much easier and cheaper than with other test data generation strategies.

Random testing will undoubtedly be used in the course of testing SDS software. Its value, however, is dependent on the accurate identification of the input domain or operational profile which, in turn, depends on the ability to predict the actual operating environment. This is a difficult task at best and, in some cases, may not be possible. The use of random testing should be supported by some method of assessing the accuracy underlying the selection of test data. This accuracy can then be confirmed, or updated, as data is collected from actual SDS operation.

B.16 Reachability Analysis

Reachability analysis is a form of static concurrency analysis where the behavior of a set of concurrent processes is represented by a state-transition graph. The analysis starts with constructing an abstract representation of the system which is used to construct a representation of the set of possible behaviors, called a reachability graph. The reachability graph is checked against a specification of acceptable behaviors, and violations of the specification are reported. Like many techniques in this class, reachability analysis can be computationally expensive since the size of the global model usually grows as the product of the sizes of individual process models. These models typically highlight the synchronization structure at the expense of other execution details which may be essential to the correctness of the software and, as a result, the analysis may report spurious errors.

The potential value of reachability analysis for testing of SDS software lies in the verification of properties of the synchronization structure of concurrent software. In this context, it provides the same level of assurance as formal verification. Thus, it offers a valuable degree of confidence in the correctness of this aspect of concurrent software.

Some prototype reachability analysis tools are ready for evaluation. One promising example is provided in the Concurrency Analysis Tool Suite (CATS) being developed as part of the Arcadia effort [Young 1989]. Here analysis is applied on state-transition graphs derived from Ada code. CATS provides checking of two types of sequence constraints. The first type is freedom from deadlock. The second type consists of additional constraints explicitly specified by user by embedding temporal logic assertions in the software.

B.17 Regression Testing

Regression testing is the activity of rerunning previous tests on a piece of software to assure that a modification to that software has not introduced any new defects. In fact, regression testing is the one area where software reuse has been successfully practiced for many years. Of course, reuse of tests can only be accomplished if the necessary details are recorded (see Section B.2).

Although the ability to apply existing tests to a piece of modified software is very valuable, it is not generally necessary to apply *all* previous tests. If advances in data flow and path analysis which enable accurately determining the scope of effect of a modification are exploited, then only those tests that address the affected scope of a modification need be applied (for example, see Section B.6). To this end, for each modification, existing test cases can be grouped into three basic classes: reusable, obsolete, and changed. Of course, new test cases may be required as well.

Regression testing undoubtedly will be widely used in the course of both developing and supporting SDS software. The availability of efficient regression testing tools promises substantially savings in these testing costs.

Tools for basic regression testing have been available for over a decade. Although some of the prototype tools for particular, advanced testing techniques support the identification of the scope of a software modification, as yet there are no general tools for efficient regression testing. Some ongoing research which is addressing the test case selection problem, and the related problem of updating test plans, is being performed at the University of Alberta [Leung 1988]. This work also proposes a direct measure for software modifiability in terms of a regression number that reflects the number of test cases affected by a single instruction change.

B.18 Ripple Effect Analysis

It has become apparent that software maintenance cost dominates the total expense of computer software. Among the many activities of software maintenance, identification of the logical ripple effect and elimination of inconsistencies resulting from a software modification are of significant importance. Logical ripple effect analysis can be applied in two ways. First, it identifies those portions of a program that are potentially impacted by a software modification and which must be retested after a modification has been performed [Wilde 1987]. Secondly, it provides a basis for quantitative estimation of program quality in terms of logical stability, which represents the resistance to the potential logical ripple effect. Ripple effect analysis may be applied to source code directly or a graph model of a program.

Ripple effect analysis offers valuable assistance for SDS software development and support. While ripple effect analysis traditionally identifies the potential ripple effect on the logical aspect of program behavior, it can also be applied to assess the effect on performance behavior [Chen 1987]. This would be extremely useful for distributed SDS software.

Technology for basic logical ripple effect analysis is ready for evaluation and demonstration. There are techniques to identify all potential performance ripple effects during maintenance activities, but research into developing techniques to determine the actual effect of an identified ripple effect on a system is only just beginning. Investigations into measures of logical stability that can be applied during design to provide an early indication of likely modifiability are also in their early stages.

B.19 Sequence Analysis

The discussion here focuses on static sequence analysis of sequential software. Static sequence analysis for concurrent software is reviewed under Static Concurrency Analysis, whereas dynamic methods for examining the sequencing of events are discussed under Behavior Analysis.

Sequence analysis was initially introduced as a method of examining software for a fixed set of anomalies in the sequence of events. The leading example of this early work is DAVE, an automated tool which used static data flow analysis to search for erroneous, or otherwise interesting, sequencing in the definition and handling of data variables [Osterweil 1976]. More recently, technology has advanced to allow flexible sequence analysis where the user can specified the sequence constraints of interest. The primary example here is the Cesar system [Olender 1989]. Again based on data flow analysis, the user specifies sequence constraints in the language Cecil, Cesar then analyzes a code product for conformance to these constraints. Currently providing programmable, inter and intraprocedural sequence analysis for Fortan programs, additional tools to support Ada and C are under development.

Although static analysis is unable to detect the range of defects that can be identified through dynamic testing, and is often computationally expensive, it is invariably requires fewer human resources than dynamic analysis. Another advantage of static analysis is its ability to detect the absence of certain defects, not just their presence. As such, sequence analysis may provide for reducing the cost of SDS software testing by allowing the detection of some program defects before resorting to more expensive dynamic testing and increasing software quality.

Technology for flexible sequence analysis still needs refining. Evaluation of the prototype Cesar system would, however, provide useful insight into the likely near-term cost-effectiveness of this technology.

B.20 Software Inspections

Software inspections are a manual, formal review technique that improves software quality while, at the same time, reducing development costs. They can be applied to products at all stages of software development, regardless of design issues such as concurrency. Inspections have proven to be capable of finding 60-90% of software defects. In practical development efforts they have been found to reduce total development costs by as much as 25%, while leading to the delivery of defect-free software [Fagan 1986].

The value of software inspections for testing of SDS software cannot be questioned. In some cases, they will be the only well-defined testing technique that can be applied to a particular SDS software product. Moreover, since they can be applied to all software products, inspections are able to detect defects as close to their point of insertion as possible, thus minimizing required rework after a defect is identified.

The inspection process is ready for transition into daily SDS software development practice. It should be required for all requirements, design, and coding products, including test plans and test cases.

B.21 Static Concurrency Analysis

The term concurrency analysis refers to examining software to determine whether particular anomalous sequencing events, such as deadlock, may occur. This type of analysis is usually performed statically to provide identification of all possible occurrences of the given sequencing errors. The difficulty in determining whether a particular set of sequencing events is a feasible set, however, may result in the detection of spurious sequencing errors. (Reachability analysis is a special-case of static concurrency analysis, see Section B.16.)

Static concurrency analysis is by no means new [Apt 1983c]. One group of researchers is investigating a version of this analysis termed constrained expression analysis [Dillon 1988c]. Here a design is translated into formal representations, constrained expression representations. The behavior of concurrent software is treated as sequences of events. By associating an event symbol with each event the possible behavior can be regarded as a string over a alphabet of event symbols. The analysis then proceeds by determining whether a particular event symbol, or pattern of event symbols, occurs in a string representing a possible behavior. Prototype tools for the Ada-based Constrained Expression Design Language (CEDL) have been developed as part of the TEAM effort. These tools are intended to support analysis of both design and code products. Static concurrency analysis based on a Petri-net representation of software are also being investigated [Shatz 1988].

Static concurrency analysis is prerequisite for rigorous testing of concurrent SDS software. It is the only way of guaranteeing that particular concurrency-related faults will not occur. Unfortunately, existing prototypes are largely research vehicles. It is doubtful that any prototype tools ready for evaluation and potential productization are currently available. Further research to extend static concurrency analysis to identify additional types of scheduling and timing problems is needed [Avrunin 1989b], in particular for analysis of concurrency processes that are distributed across a number of machines.

B.22 Symbolic Evaluation

Symbolic evaluation is a necessary element of many testing activities [Clarke 1985]. The basic idea is to allow numeric variables to take on 'symbolic values' as well as numeric values. A symbolic value is either an elementary symbolic value or an expression in numbers, arithmetic operators, and other symbolic values. The symbolic evaluation of a path is carried out by symbolically evaluating the sequence of assignment statements in the path. This can be used to reconstruct the logic and computations used in a program to aid in determining the correctness of outputs. The branch conditions which occur in conditional statements can be symbolically evaluated to form symbolic predicates. The symbolic system of predicates for a path can be constructed by symbolically evaluating both assignment statements and branch predicates during evaluation of the path and consists of the sequence of symbolic predicates that are generated by the evaluation of the branch predicate. It can be used to assist the user in constructing test data since it describes the subset of the input domain that causes that path to be executed. The symbolic system of predicates also can be used in formal verification of program correctness.

The basic technology for symbolic evaluation is ready for transition. It offers the most benefit, however, when integrated with other testing techniques. While some applications are ready for evaluation and demonstration, others require additional research. An example of the first case is the integration of symbolic evaluation with static concurrency analysis [Young 1988]. The use of symbolic evaluation for fault-based testing is an example of an application requiring further investigation [Morell 1988]. The development of a highly efficient symbolic evaluation system that can be integrated appropriately with various testing approaches offers increased effectiveness for many areas of SDS software testing.

B.23 Timing Analysis

In a real-time software system, it is necessary to ensure that stringent timing requirements are satisfied. Since most time-critical software systems interact with physical processes in the external world and a model of the external world is included, at least implicitly, in the software, it is also necessary to guard against an imperfect execution environment which may violate these design assumptions. The analysis of real-time system depends on the specification of expected real-time behavior, against which the behavior in the real execution environment can be compared. This comparison can begin as soon as the system specification is available and be refined through subsequent design and coding activities.

SDS software will be subject to timing constraints. Indeed, it is probably the largest real-time system ever planned for development and the capability to test the timing behavior of the software throughout its development process is critically needed.

While researchers have begun pay attention to the problems of timing analysis over the last few years, much work remains to be done. At the software specification and design levels, program level, few languages include support for explicit specification of timing constraints. Consequently, some researchers are examining the use of timing assertions to allow the derivation of performance conjectures which are proved, or analyzed, against the system specification [Auernheimer 1986, Jahanian 1986]. Programming languages are similarly lacking. Here researchers are investigating the use of extended forms of temporal logic to express the necessary information so that the expected timing behavior can be compared with the actual behavior in the practical execution environment [Razoukk 1988].

References

Air Force Systems Command. January 1986. *Software Management Indicators.* AFSC Pamphlet 800-43.

Air Force Systems Command. January 1986. *Software Quality Indicators.* AFSC Pamphlet 800-14.

Apt, K.R. 1983. "A Static Analysis of CSP Programs." In *Proceedings Workshop on Program Logic*, June, Pittsburgh, PA.

Auernheimer, B., and R.A. Kemmerer. "RT-ASLAN: A Specification Language for Real-Time Systems." *IEEE: Transactions on Software Engineering,* 12/9 (Sep 1986):879-889.

Avrunin, G.S., L.K. Dillon, and J.C. Wileden. 1989. *Constrained Expression Analysis of Real-time Systems.* University of Massachusetts. TR-89-50.

Bentley, J. "Profilers." *ACM: Communications of the ACM,* 30/7 (Jul 1987):587-592.

Bieman, J., and J. Schultz. 1989. "Estimating the Number of Test Cases Required to Satisfy the All-du-paths Testing Criterion." In *Proceedings ACM SIGSOFT '89 3rd Symposium on Software Testing, Analysis, and Verification (TAV3)* December 13-15, Key West, FL, 179-186. ACM Press.

Bloomfield, R.E., and P.K. Froome. "The Application of Formal Methods to the Assessment of High Integrity Software." *IEEE: Transactions on Software*

Engineering, 12/9 (Sep 1986):988-993.

Chen, J.L. 1987. *Analysis of Distributed System Software in Maintenance Phase Based on Timed Petri Net Model.* Ph.D. Diss., Northwestern University. Choquet, N. 1986. "Test Data Generation using a Prolog with Constraints." In *Proceedings Workshop on Software Testing,* July 15-17, Banff, Canada, 132-141. IEEE Press.

Clarke, L.A., and D.J. Richardson. "Applications of Symbolic Evaluation." *Journal of Systems and Software,* 5/1 (1985):15-35.

DeMillo, R.A., E.W. Krauser, and A.P. Mathur. August 1988. *Using the Hypercube for Reliable Testing of Large Software.* Purdue University. SERC-TR-24-P.

DeMillo, R.A. February 1989. *Test Adequacy and Program Mutation.* Purdue University. SERC-TR-37-P.

Dillon, L.K., G.S. Avrunin, and J.C. Wileden. "Constrained Expressions: Toward Broad Applicability of Analysis Methods for Distributed Software Systems." *ACM: Transactions on Programming Languages and Systems,* 10/3 (Jul 1988):374-402.

Duncan, A.G., and J.S. Hutchison. 1981. "Using Attribute Grammars to Test Designs and Implementations." In *Proceedings 5th International Conference on Software Engineering,* March 9-12, San Diego, CA, 170-178. IEEE Press.

Duran, J.W., and S.C. Ntafos. "An Evaluation of Random Testing." *IEEE: Transactions on Software Engineering,* 10/4 (Jul 1984):438-444.

Feldman, M.B., and M.L. Moran. "Validating a Demonstration Tool for Graphics-Assisted Debugging of Ada Concurrent Programs." *IEEE: Transactions on Software Engineering,* 15/3 (Mar 1989):305-313.

Francioni, J.M. 1988. "Communication Profiles of Distributed Parallel Process." In *Proceedings ACM SIGPLAN-SIGOPS Workshop on Parallel and Distributed Debugging,* May 5-6, Madison, WI, 301-303. ACM Press.

Frankl, P.G., and E.J. Weyuker. "An Applicable Family of Data Flow Testing Criteria." *IEEE: Transactions on Software Engineering,* 14/10 (Oct 1988):1483-1498.

Gannon, J.D., P.R. McMullin, and R.G. Hamlet. "Data Abstraction Implementation,Specification and Testing." *ACM: Transactions on Programming Languages and Systems,* 3/3 (Jul 1981):211-223.

Gilkey, T.J., J.R. White, and T.L. Booth. *Performance Analysis as a Software Design Tool.* University of Connecticut.

Hamlet, D., and R. Taylor. 1988. "Partition Testing Does Not Inspire Confidence." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 206-216. IEEE Press.

Hennell, M.A., M.R. Woodward, and D. Hedley. "On Program Analysis." *Information Processing Letters*, 5/5 (Nov 1976):136-140.

Jahanian, F., and A.K. Mok. "Safety Analysis of Timing Properties in Real-Time Systems." *IEEE: Transactions on Software Engineering*, 12/9 (Sep 1986):890-904.

Kramer, J., and K. Whitehead. May 1988. *TARA: Tool Assisted Requirements Analysis*. Imperial College of Science and Technology.

Krauser, E.W., and A.P. Mathur. 1986. "Program Testing on a Massively Parallel Transputer Based System." In *Proceedings ISMM International Symposium on Mini and Microcomputers and their Applications*, November 10-12, Austin, TX, 67-71.

Kundu, S. "The Call-Return Tree and Its Application to Program Performance Analysis." *IEEE: Transactions on Software Engineering*, 12/11 (Nov 1986):1096-1098.

Leung, H.K., and L.J. White. September 1988. *A Study of Regression Testing*. University of Alberta. TR-88-15.

Leveson, N.G., and J.L. Stolzy. "Safety Analysis of Ada Programs Using Fault Trees." *IEEE: Transactions on Reliability*, R-32/5 (Dec 1983):479-484.

Lopriore, L. "A User Interface Specification for a Program Debugging and Measuring Environment." *Software Practice and Experience*, 19/5 (May 1989):437-460.

Morell, L.J. 1988. "Theoretical Insights into Fault-Based Testing." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 45-62. IEEE Press.

Moriconi, M. 1983. "PegaSys: An Environment for Displaying, Animating, and Reasoning About Graphical Descriptions of Systems." In *Proceedings Symposium on Software Validation*, H.-L. Hansen (ed.), September, Darmstadt. North-Holland.

Myers, G.J. 1979. *The Art of Software Testing*. John Wiley & Sons.

Olender, K., and L. Osterweil. 1989. "Cesar: A Static Sequencing Constraint Analyzer." In *Proceedings ACM SIGSOFT '89 3rd Symposium on Software Testing, Analysis, and Verification (TAV3)* December 13-15, Key West, FL, 66-74. ACM Press.

Osterweil, L.J., and L.D. Fosdick. "DAVE — A Validation Error Detection and Documentation System for Fortran Programs." *Software Practice and Experience,* 6/4 (Oct-Dec 1976):473-486.

Razouk, R., and M. Gorlick. 1989. "Real-Time Interval Logic for Reasoning About Executions of Real-Time Programs." In *Proceedings ACM SIGSOFT '89 3rd Symposium on Software Testing, Analysis, and Verification (TAV3)* December 13-15, Key West, FL, 10-19. ACM Press.

Rolandelli, C., T.J. Shimeall, C. Genung, and N. Leveson. February 1986. *Software Fault Tree Analysis Tool User's Manual.* University of California at Irvine. TR-86-06

Shaw, W.H., J.W. Howatt, R.S. Maness, D.M. Miller. "A Software Science Model of Compile Time." *IEEE: Transactions on Software Engineering,* 15/5 (May 1989):543-549.

Socha, D., M.L. Bailey, and D. Notkin. "Voyeur: Graphical Views of Parallel Programs." In *Proceedings ACM SIGPLAN-SIGOPS Workshop on Parallel and Distributed Debugging*, May 5-6, Madison, WI, 206-215. ACM Press.

Solis, D.M. 1985. "AutoParts— A Tool to Aid in Equivalence Partition Testing." In *Proceedings SoftFair II: 2nd Conference on Software Development Tools, Techniques, and Alternatives,* December 2-5, San Francisco, CA, 122-125. IEEE Press.

Tai, K.C. 1989. "Testing of Concurrent Software." In *Proceedings 13th Annual International Computer Software and Applications Conference*, Orlando, FL, September 20-22, 62-64. IEEE Press.

Tai, K.C., R.H. Carver, and E.E. Obaid. 1989. "Deterministic Execution Debugging of Concurrent Ada Programs." In *Proceedings 13th Annual International Computer Software and Applications Conference*, Orlando, FL, September 20-22, 102-109. IEEE Press.

Taylor, R.N., L. Clarke, L.J. Osterweil, J.C. Wileden, and M. Young. 1986. "Arcadia: A Software Development Environment Research Project." In *Proceedings IEEE Conference on Ada Applications and Environments,* April 8-10, Miami Beach, FL. IEEE Press.

Taylor, R.N., and C.D. Kelly. 1986. "Structural Testing of Concurrent Programs." In *Proceedings Workshop on Software Testing,* July 15-17, Banff, Canada, 164-169. IEEE Press.

Urai, H., and B. Yang. "A Structural Test Selection Criterion." *Information Processing Letters,* 28/4 (Jul 1988):157-163.

White, L.J., and I.A. Perera. 1986. "An Alternative Measure for Error Analysis of the Domain Testing Strategy." In *Proceedings Workshop on Software Testing,* July 15-17, Banff, Canada, 45-62. IEEE Press.

White, L.J., and B. Wiszniewski. 1988. "Complexity of Testing Iterated Borders for Structured Programs." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis,* July 19-21, Banff, Canada, 231-237. IEEE Press.

Wolf, A.L., L.A. Clarke, and J.C. Wileden. September 1986. *The AdaPIC Toolset: Supporting Interface Control and Analysis Throughout the Software Development Process. IEEE: Transactions on Software Engineering,* 15/3 (Mar 1989):250-263.

Wu, L., V. R. Basili, and K. Reed. 1987. "A Structure Coverage Tool for Ada Software Systems." In *Proceedings Joint Conference of 5th National Conference on Ada Technology and Washington Ada Symposium,* March 16-19, Arlington, VA, 294-302. ACM Press.

Wilde, N., and B. Nejmeh. September 1987. *Dependency Analysis: An Aid for Software Maintenance.* University of Florida. SERC-TR-13-F.

Young, M., and E.C. Epp. "Combining Static Concurrency Analysis with Symbolic Execution." In *IEEE: Transactions on Software Engineering,* 14/10 (Oct 1988):1499-1511.

Young, W., R. Taylor, K. Forester, and D. Brodbeck. 1989. "Integrated Concurrency Analysis in a Software Development Environment." In *Proceedings ACM SIGSOFT '89 3rd Symposium on Software Testing, Analysis, and Verification (TAV3)* December 13-15, Key West, FL, 200-209. ACM Press.

Distribution List for IDA Paper P-2493

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Lt Col James Sweeder Chief, Computer Resources; Engineering Division SDIO/ENT Room 1E149, The Pentagon Washir gton, DC 20301-7100	2
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dr. Dan Alpert, Director Program in Science, Technology & Society University of Illinois Room 201 912-1/2 West Illinois Street Urbana, Illinois 61801	1
IDA	
General Larry D. Welch, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Mr. Bill R. Brykczynski, CSED	30
Ms. Ruth L. Greenstein, HQ	1
Ms. Anne Douville, CSED	1
Dr. Richard J. Ivanetich, CSED	1
Mr. Terry Mayfield, CSED	1
Dr. Reginald N. Meeson, CSED	1
Ms. Katydean Price, CSED	2
Dr. Richard L. Wexelblat, CSED	1
Ms. Christine Youngblut, CSED	1
IDA Control & Distribution Vault	3