Software Analysis and Test Technologies

Contract Number F30602-89-C-0082
(Data & Analysis Center for Software)

February 1992

Prepared for:

Rome Laboratory
RL/C3CB
525 Brooks Road
Griffiss AFB, NY 13441-4505

Prepared by:

Kaman Sciences Corporation
P.O. Box 120
Utica, NY 13503-0120

and

Center for Digital Systems Research
Research Triangle Institute
Research Triangle Park, NC 27709

The Data & Analysis Center for Software (DACS) is a Department of Defense (DoD) Information Analysis Center (IAC), administratively managed by the Defense Technical Information Center (DTIC) under the DoD IAC Program. The DACS is technically managed by Rome Laboratory (RL). Kaman Sciences Corporation manages and operates the DACS, serving as a source for current, readily available data and information concerning software engineering and software technology.
SOFTWARE ANALYSIS AND
TEST TECHNOLOGIES

Contract Number F30602-89-C-0082
(Data & Analysis Center for Software)

February 1992

Prepared for:
Rome Laboratory
RL/C3C
Griffiss AFB, NY 13441-5700

Prepared by:
Kaman Sciences Corporation
258 Genesee Street
Utica, New York 13502-4627

and

Center for Digital Systems Research
Research Triangle Institute
Research Triangle Park, North Carolina 27709
This report examines current software analysis and test technology and needs that should be filled by future technology. Analysis and testing of software includes all life cycle activities conducted to verify and validate the software product. These activities are undertaken with the goal of assuring the robustness of the development process and the integrity of the developed product throughout the life cycle. Successful strategies for analysis and test must provide decision support information to the acquisition manager, the certifying agent, and the field engineer. There is a need for quantitative empirical data to demonstrate when and where various techniques are most successful. There is a need for integrated development environments which include analysis and test support. This report also considers improvements in analysis and test required to support trends in formal methods, object oriented development, parallel programming, and system engineering.
# TABLE OF CONTENTS

1. **INTRODUCTION** ........................................................................................................... 3

2. **CURRENT TECHNOLOGY** .................................................................................................. 5
   2.1 Maturity of Current Technology .............................................................................. 5
   2.2 Maturity of the Industry in Using Analysis and Test Technology .............. 5

3. **ANALYSIS AND TEST PROCESS IMPROVEMENT** ............................................. 8
   3.1 Improving Early Life Cycle Analysis and Simulation Capabilities .......... 8
   3.2 Software Measurement and Life Cycle Analysis and Test Activities .... 8
   3.3 Strategies for Efficiently Allocating Test Effort .............................................. 11

4. **SOFTWARE ANALYSIS AND TEST TOOLS** .................................................. 16
   4.1 Development Support Tools ............................................................................. 17
   4.2 Maintenance Support Tools ............................................................................. 18
   4.3 Knowledge-Based Tool Support for Software Analysis and Test .......... 18

5. **INTEGRATION WITH ADVANCED SOFTWARE DEVELOPMENT TECHNOLOGY** .................................................. 20
   5.1 Formal Methods ................................................................................................. 20
   5.2 Object-Oriented Development ......................................................................... 20
   5.3 Analysis and Test of Parallel Software .............................................................. 22
   5.4 System Engineering Issues ............................................................................... 25

6. **SUMMARY AND RECOMMENDATIONS** .............................................................. 28

7. **REFERENCES** ................................................................................................................. 29

Appendix A: ACRONYMS ............................................................................................... 33

Appendix B: SOFTWARE ANALYSIS AND TEST TOOLS ........................................... 37

Appendix C: STANDARDS RELATED TO SOFTWARE ANALYSIS AND TEST ................. 43

Appendix D: ADDITIONAL READING ............................................................................. 48
## LIST OF TABLES

| Table 4-1: | Examples of Life-Cycle Validation Techniques and Tools | 16 |
| Table 5-1: | Some Factors in Analysis and Test of Parallel Software | 23 |
| Table 5-2: | Analysis and Test Activities for Parallel Software | 25 |

## LIST OF FIGURES

| Figure 1-1: | Critical Components of Quality Software Development | 4 |
| Figure 1-2: | Effect of Maintainability and Supportability Costs | 4 |
| Figure 2-1: | Software Analysis and Test Techniques | 6 |
| Figure 2-2: | Software Analysis and Test Activities at each Maturity Level | 7 |
| Figure 3-1: | Levels of Software Measurement and Assessment | 9 |
| Figure 3-2: | Overview of Software Measurement, Analysis, and Test | 10 |
| Figure 3-3: | A Risk/Criticality Driven Strategy | 12 |
| Figure 3-4: | A Fault Coverage Driven Strategy | 13 |
| Figure 3-5: | Data in Support of a Fault Coverage Driven Strategy | 14 |
| Figure 3-6: | Data in Support of a Fault Coverage Driven Strategy (cont.) | 15 |
corroborated by the 1987 survey on testing practices and trends [3].

<table>
<thead>
<tr>
<th>Safety and Security Evaluation</th>
<th>Risk Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>threat analysis</td>
<td>risk identification checklists</td>
</tr>
<tr>
<td>preliminary hazards analysis</td>
<td>decision driver analysis</td>
</tr>
<tr>
<td>fault-tree analysis</td>
<td>assumption analysis</td>
</tr>
<tr>
<td>failure modes &amp; effects criticality analysis</td>
<td>decision tree</td>
</tr>
<tr>
<td>system hazard cross-check analysis</td>
<td>network analysis</td>
</tr>
<tr>
<td>cause-effect graphing</td>
<td>compound risk analysis</td>
</tr>
<tr>
<td>operating support hazard analysis</td>
<td>requirements traceability</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Static and Dynamic Analyses</th>
<th>Measurement and Assessment</th>
</tr>
</thead>
<tbody>
<tr>
<td>formal verification</td>
<td>quality goal specification and assessment</td>
</tr>
<tr>
<td>structure analysis</td>
<td>design and code complexity analysis</td>
</tr>
<tr>
<td>inspections, reviews, walkthroughs</td>
<td>fault classification analysis</td>
</tr>
<tr>
<td>simulation</td>
<td>root cause analysis for defect prevention</td>
</tr>
<tr>
<td>system or acceptance testing</td>
<td>markov, semi-markov, and analytic system reliability modelling</td>
</tr>
<tr>
<td>(functions, performances, stress)</td>
<td>petri net analysis of performance</td>
</tr>
<tr>
<td>requirements-based functional testing</td>
<td>discrete event performance simulation</td>
</tr>
<tr>
<td>statistical or random testing</td>
<td>standardized &amp; application specific performance benchmarking</td>
</tr>
<tr>
<td>integration testing (top down, bottom up or big bang)</td>
<td>operational failure characterization</td>
</tr>
<tr>
<td>structural testing (path, branch, and statement testing)</td>
<td>modeling of software reliability growth during test</td>
</tr>
<tr>
<td>mutation testing</td>
<td>cost modeling</td>
</tr>
<tr>
<td>back-to-back testing</td>
<td>queueing models</td>
</tr>
<tr>
<td>error seeding</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-1: Software Analysis and Test Techniques
Figure 2-2: Software Analysis and Test Activities at each Maturity Level

<table>
<thead>
<tr>
<th>MATURITY LEVEL</th>
<th>1 &quot;Initial&quot;</th>
<th>2 &quot;Repeatable&quot;</th>
<th>3 &quot;Defined&quot;</th>
<th>4 &quot;Managed&quot;</th>
<th>5 &quot;Optimizing&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No orderly progress in process improvement</td>
<td>Repeatability of management control</td>
<td>Advanced technology can be introduced</td>
<td>Initiated, comprehensive process measurements</td>
<td>Foundation for continuous process improvement</td>
</tr>
</tbody>
</table>

- Variable debugging efforts by programmers
- User acceptance testing
- Quality assurance group
- End of life-cycle testing
- Possibly design and code reviews
- Test and analysis effort tracking and identification of resource requirements
- Defined role and procedures for V&V
- Development of V&V plan early in life cycle
- Design and code inspections
- Independent unit test
- Problem report tracking
- Tracking of test and analysis effectiveness and efficiency
- Use of objective, quantitative stopping rules
- Fault classification
- Use of design for validation techniques
- Life-cycle test and analysis strategies based on effectiveness and efficiency evaluations
- Root cause analysis for defect prevention
- Refined, comprehensive stopping rules
3. ANALYSIS AND TEST PROCESS IMPROVEMENT

Cost-effective strategies are critical to improving the software analysis and test process. These strategies should include early life cycle analysis and simulation, involve complementary techniques, be based on quality measurement, and efficiently allocate analysis and test effort [10]. For example, the Cleanroom approach which combines proofs of correctness with statistical quality control has had demonstrated success [11]. There are a number of issues remaining, however, with respect to improving the software analysis and test process. These include improving early life cycle analysis and simulation capabilities, incorporating software product and process measurement into life cycle analysis and test activities, and refining strategies for efficiently allocating test effort.

3.1 Improving Early Life Cycle Analysis and Simulation Capabilities

Software is an integral part of complex real-time embedded systems. Data has shown that tracing a problem or bug late in the software life cycle is more costly [12]. With this in mind, it becomes increasingly important to identify problems, whether they are requirements, design, performance, reliability, cost, or complexity related, early in the software development life cycle. Early life cycle analysis and test techniques can be used to assist developers in making system design decisions (e.g., hardware and software tradeoffs, performance and reliability tradeoffs).

Systems can be analyzed in many different ways. One approach is to separate functional requirements and quality attributes. Functional requirements define what the system is supposed to do, how it is expected to operate, and whether or not the system properly performs the functions specified. Quality attributes are divided into a variety of dependability categories, such as performance, reliability, safety, fault tolerance, security, testability, maintainability, and supportability [13].

Analyzing functional requirements and quality attributes prior to actually building the system requires the development of models. Requirements must translate into models in consistent, understandable, and standardized ways. Structured Design and Analysis techniques incorporated into Computer Aided Software Engineering (CASE) tools assist developers in graphically modeling system functional requirements and interactions at a high level which can be extended to include very minute, low-level process interaction and design information. What is lacking are methods and tools for additional integration of these functional models with models that are used to demonstrate and make tradeoffs among quality requirements [14]. For example, a need for safety-critical applications is to relate software fault-tree models to system performance models to functional design models. Once models are constructed, they must be "validated" if they are going to provide any useful information. Model validation suites used to test the model at this stage of development need to be expanded into test cases for each stage of the development process.

3.2 Software Measurement and Life Cycle Analysis and Test Activities

The integration of measurement and assessment with verification activities is necessary for building software with an acceptable quality level. Software measurement is defined to be the activity where attributes in both the software product and process can be quantified for specifying the level of quality of the software product, the productivity and effectiveness of the software process, or any specific goal defined in the early phases of the software process. Some progress has been made in the development and use of system, product, process, and acquisition metrics. More is needed [15].

Three levels of measurement exist as shown in Figure 3-1. The baseline or descriptive level provides measures of industry trends across many projects such as failure rate and fault density [16]. These measures are helpful when comparing where an organization stands in achieving product quality or understanding product quality across application domains. The next level of measurement provides decision support within a project. The use of objective stopping rules during analysis and test [17, 18] is an example of this level of measurement. Tools, such as the Assistant for Specifying the Quality of Software (ASQS), the Quality Evaluation System (QUES), the Tailoring an Ada Measurement Environment (TAME) system, and AMADEUS (an automated measurement and empirical analysis system), are emerging to support this type of measurement.

The third level of measurement deals with the assessment of method effectiveness. This assessment involves software engineering experimentation [19, 4] and provides technology transfer across projects. There are four general types of studies in use for addressing the effectiveness of current tools and techniques. These are descriptive evaluations, case studies, formal experiments, and quasi-experiments.
Descriptive evaluation studies use an objective approach to gather data and a qualitative approach to evaluate it [20]. An example of a descriptive evaluation is a study that sought to evaluate automated software cost-estimation models. This study evaluated seven cost-estimation models used by the US Defense Department software community. Each model's capabilities were graded, from having full capability to having minimal capability [21].

Case or multiple-case studies are empirical inquiries that investigate a phenomenon in its real-life context. These studies are used most effectively to address methods applied across the life cycle on actual projects [22]. A joint NASA-Langley/FAA study is using this approach for addressing the effectiveness of methods used in compliance with the DO-178A guidelines [23, 24]. One benefit to this approach is that it provides a realistic context for interpreting the results. A drawback is that it takes a long time to obtain meaningful results - a single study requires that you engineer a complete software system.

Formal experiments use formal statistical designs that let researchers make quantitative inferences about an observation [25]. For example, formal experiments have been used successfully to compare the effectiveness of test techniques [5, 26]. The findings of one formal experiment gave the researchers reason to be more optimistic about the effectiveness of reading code to find software errors.
Quasi-experiments are field investigations of, for example, the use of a new method or language [27]. They differ from formal experiments in that what you want to observe cannot be clearly delineated from other observations. Quasi-experiments differ from case studies in that the researcher is actively manipulating a change in the development process that did not exist before. An example is the study of a flight-dynamics simulator that was developed in both Fortran and Ada [28].

Figure 3-1 shows a proposed mapping of these levels to the DoD SEI software maturity framework.

Figure 3-2 provides an integrated overview of software measurement and analysis and test. Progress needs to be made at each of the three levels of software measurement described above. At the first level, more data (e.g., test effectiveness, coverage, effort, failure rate, fault density, and Mean Time Between Failure) needs to be collected to define an industry baseline. This baseline can provide insight, for example, into the relationship between analysis and test practices and achieved product quality.

Figure 3-2: Overview of Software Measurement, Analysis, and Test

At the second level of measurement, two activities are important. First, additional use and evaluation of decision support tools (e.g., ASQS, QUES, AMADEUS, TAME) for DoD mission-critical application software development needs to occur. Second, much of the current software analysis and test technology development efforts focus on the software developer’s perspective. Additional decision support measures and tools that address analysis and test from an acquisition specialist, certifying agent, and field engineer’s perspective are needed. To gain confidence that the software is of quality, a four-fold framework has been discussed:

- evaluation of analysis and test process
- measurement of end-product quality
- credentials of the developing organization
- past performance of the developing organization

The challenge is to define what information about the software analysis and test activities and end-product quality is needed for the independent evaluator to be convinced that the delivered product is robust.

The method effectiveness level of measurement addresses the view that by using good methods throughout the manufacturing process a quality software product will result. This is the view taken by DoD-2167A [29] and RTCA DO-178A [30] where documents at key points in the process are evaluated to see what the detailed processes are and how well the defined process is being followed. This level of
measurement builds on the decision support level of measurement. Specifically, the instrumentation of product quality measurement at analysis and test process milestones provides data for evaluating method effectiveness, as shown in Figure 3-1. These measures are used to gauge if additional analysis and test activities are warranted and raise the technical problem of specifying objective stopping rules for different methods. Note that this view represents a longitudinal or strategy oriented perspective on method effectiveness.

### 3.3 Strategies for Efficiently Allocating Test Effort

In the long term, insights gained from method effectiveness studies will provide better strategies for efficiently allocating analysis and test effort. Two strategies are currently being discussed. These are risk-driven, as shown in Figure 3-3, and fault coverage driven, as shown in Figure 3-4. Both these strategies fit within the quality goal specification and assessment framework shown in Figure 3-2.

In a risk driven strategy, such as that described in Boehm [31] and in Sherer [32], the goal is to allocate software analysis and test effort in a manner which demonstrates the absence of certain types of risks. For example, a preliminary hazard analysis may be conducted for identifying what hazards are to be avoided. This hazard analysis data will factor into design considerations for the software. It will also factor into the development of functional test cases which are then used to demonstrate that the hazard cannot occur. This type of strategy is sometimes referred to as a software safety analysis and can be carried forward through the life cycle, as is proposed in MIL-STD 882B [33]. Two issues remain before this strategy can be maximally effective. First, data on the frequency, type, and severity of hazards and how these hazards can be invoked by software specification, design, and code faults are typically unavailable. The second issue is how to effectively complement the safety analysis techniques with other non-functionally based black-box dynamic testing techniques (e.g., usage-based statistical testing techniques) and white-box dynamic techniques (e.g., data or control flow guided techniques).

The goal of a fault coverage driven strategy is to cover multiple classes of faults by complementing techniques. This strategy is the same as a risk-driven strategy if the fault classes are based on criteria related to a hazardous outcome (e.g., critical, serious, nonessential). If the classes are based on fault types (e.g., logic, data, interface, etc.), then this strategy results in a different allocation of test effort.

The fault coverage driven strategy is illustrated using data taken from the software test technique experiment summarized in [5]. The goal is to allocate effort by complementing test techniques so that the overlap in the faults found by these techniques is minimized. Figure 3-5 shows percent effectiveness for three dynamic and three static test techniques. Percent effectiveness is an average measure of the number of faults found by that technique divided by the total known faults in the software. Figure 3-6 shows this same data when the techniques are applied in pairs. Assuming that a fault class is of size 1, these data show that a meaningful fault coverage strategy is to combine the use of a static analysis technique with a dynamic test technique. Although the data are limited, this study suggests that combinations of static and dynamic strategies other than that chosen in the Cleanroom approach may be effective. However, two issues remain before a fault coverage driven strategy can be maximally effective. First, a framework for fault classification is needed. Second, additional data on which test techniques are better at finding which types of faults is needed.
Figure 3-3: A Risk/Criticality Driven Strategy
Key:

\{A\}  faults removed by technique A
\{B\}  faults removed by technique B
\{C\}  faults removed by technique C
\{D\}  faults removed by technique D

\(\omega\) : set of all faults in software product
\(\Omega\) : universe of faults

Figure 3-4: A Fault Coverage Driven Strategy
Figure 3-5: Data in Support of a Fault Coverage Driven Strategy
Figure 3-6: Data in Support of a Fault Coverage Driven Strategy (cont.)
4. SOFTWARE ANALYSIS AND TEST TOOLS

Computer Aided Software Engineering (CASE) tools now exist for all phases of the software life cycle, but particularly for the support of coding and debugging. Over the last few years, tools that support both the earliest stages of the software life cycle and the software maintenance process have become increasingly important. This is because a substantial body of empirical evidence shows that significant cost savings and higher software quality can be achieved if these two phases of the software life cycle can be improved.

Software analysis and test also benefit from existing tools and techniques, particularly in the areas of test case generation and test coverage analyzers. Table 4-1 shows some of the tools and methods that are available to support validation and verification activities during different parts of the life cycle.

Table 4-1: Examples of Life-Cycle Validation Techniques and Tools

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Requirements Evaluation:</strong></td>
<td></td>
</tr>
<tr>
<td>Error tracking</td>
<td>requirements-to-test-tracker</td>
</tr>
<tr>
<td>Reviews, walkthroughs, and audits</td>
<td>CASE/SA</td>
</tr>
<tr>
<td>Completeness, consistency checking</td>
<td>CASE/SA</td>
</tr>
<tr>
<td><strong>Design Evaluation:</strong></td>
<td></td>
</tr>
<tr>
<td>Error tracking</td>
<td>requirements-to-test-tracker</td>
</tr>
<tr>
<td>Reviews, walkthroughs, and audits</td>
<td>CASE/SD, consistency checkers</td>
</tr>
<tr>
<td>Design metrics</td>
<td>McCabe/ACT</td>
</tr>
<tr>
<td><strong>Implementation:</strong></td>
<td></td>
</tr>
<tr>
<td>Debugging</td>
<td>Symbolic Debuggers</td>
</tr>
<tr>
<td>Compile-time-analyses</td>
<td>Compiler options</td>
</tr>
<tr>
<td>Code metrics</td>
<td>AMS, MITS</td>
</tr>
<tr>
<td><strong>Test and Analysis</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Static Analysis</strong></td>
<td></td>
</tr>
<tr>
<td>Syntax/style</td>
<td>RXVP80, DEC/SCA, LDRA Testbed</td>
</tr>
<tr>
<td>Language/project standards</td>
<td>Logiscope</td>
</tr>
<tr>
<td>Reviews, walkthroughs, inspections, and audits</td>
<td>RXVP80, DEC/SCA, Logiscope</td>
</tr>
<tr>
<td>Structure/interface/data flow analysis</td>
<td>RXVP80, DEC/SCA, Logiscope</td>
</tr>
<tr>
<td>Code metrics</td>
<td>McCabe/ACT, AMS, MITS, Logiscope, LDRA Testbed</td>
</tr>
<tr>
<td>Formal verification</td>
<td>Theorem provers</td>
</tr>
<tr>
<td><strong>Dynamic Analysis</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Static Analysis</strong></td>
<td></td>
</tr>
<tr>
<td>Statement, branch, basis, path coverage</td>
<td>RXVP80, DEC/PCA, McCabe/ACT, Logiscope, LDRA Testbed, etc.</td>
</tr>
<tr>
<td>Statistical testing</td>
<td>random number generating routine</td>
</tr>
<tr>
<td>Functional testing</td>
<td>requirements-to-test-tracker</td>
</tr>
<tr>
<td>Mutation analysis</td>
<td>MOTHRA</td>
</tr>
<tr>
<td>Symbolic execution</td>
<td>custom hardware/software simulators</td>
</tr>
<tr>
<td>Run-time assertions</td>
<td>ATV5, assertion translators</td>
</tr>
<tr>
<td>Performance measurements</td>
<td>DEC/PCA, etc.</td>
</tr>
<tr>
<td>Regression testing</td>
<td>DEC/TM, etc.</td>
</tr>
<tr>
<td><strong>ALL PHASES:</strong></td>
<td></td>
</tr>
<tr>
<td>Requirements-to-test tracking</td>
<td>requirements-to-test-tracker</td>
</tr>
<tr>
<td>Configuration management</td>
<td>DEC/CMS, DEC/MMS, etc.</td>
</tr>
</tbody>
</table>

With the advent of electronic capture of software specification and design information, it has become easier to develop specialized software analysis and test tools. Parts of the software life cycle which could benefit from the development of additional tools include early life cycle analysis and
software maintenance activities. The development of knowledge based support for software analysis and test practice would also be of benefit.

4.1 Development Support Tools

Formal or semi-formal representations of a software system provide the basis for an emerging class of analysis tools, particularly in the earlier stages of the software life cycle. Formal representations include specification languages with a rigorously defined set of semantics. Z [34], VDM [35] and HOL [36] are three well-known examples.

Semi-formal representations include, for example, the Structured Analysis/Structured Design methods supported by the majority of commercial CASE tools. Others include various object-oriented design methodologies and design tools and methodologies for the support of Ada.

Formal methods are not widely used, mostly because of perceptions of difficulty in their use. However, certain high reliability or safety-critical systems have benefitted from the use of formal methods. Semi-formal methods enjoy a far wider following. Hence, the remainder of this discussion will be confined to semi-formal representations.

Integrated tool environments are a significant emerging trend. Such environments allow developers to build models using one type of modeling tool and perform another type of analysis on the model using another tool with little or no extra effort. An example of a hybrid toolset which performs this type of integration is a CASE tool, which describes a system in terms of a static structured analysis model, and then uses another related tool which can "execute" a real-time simulation of the modeled system by simply reading the static model from the CASE tool [6].

An additional benefit of an integrated environment is that all of the relevant design information can be contained in a central location. This makes the software maintenance process more cost-effective because all of the needed information is readily available. These environments are still evolving, and it will be some time before their benefits are fully realized.

Another example of an integrated software development environment is the Software Life Cycle Support Environment (SLCSE) [37] developed by the Rome Laboratory. This environment provides many of the support tools needed for developing and maintaining large embedded Ada programs.

Simulators, system architecture modeling tools, and software performance evaluation tools can assist developers in predicting the performance, reliability and behavior characteristics of a system by executing a "model" of a proposed system long before it is ever built. These simulators allow designers to change various parameters of a system and simulate the effect of the parametric modifications on the rest of the system and can assist in the development of software test cases for use later in the life cycle. Tools of this nature have been built around commercial CASE tools. Statemate and Teamwork both provide the ability to execute specifications developed within their respective environments. In addition, Teamwork has a performance evaluation capability. These types of analysis tools can help find design errors at a time in the life cycle when they are less expensive to correct.

Researchers at Research Triangle Institute (RTI) used an integrated toolset consisting of a CASE tool (that used structured analysis and real-time system specification techniques described by Hately [38]), along with an integrated performance modeling tool, to assist factory automation design engineers illustrate and identify performance bottlenecks and component interaction [39]. A benefit of using the tightly coupled toolset was that a change to the static structured analysis model automatically became part of the real-time simulation model as well. By reviewing information from the simulation model, researchers were able to evaluate the "correctness" of certain system activities by observing the simulation behavior, and when a problem was encountered in the simulation, the toolset forced changes to be made to the structured analysis model in order for the change to appear in the real-time simulation model. The models stayed completely consistent, as opposed to what might have occurred if the two tools required separate model forms (one for real-time simulation and one for structured analysis) in order to operate.

Test case generation and coverage analysis tools are also emerging which permit the identification of test cases based on a high-level specification of the system. For example, "T" [40], a test case generation tool, provides a specification language from which a minimal set of test cases can be derived. Without specialized tools, testing is a haphazard activity which is difficult to control. Testing tools allow test personnel to quantify and control the test process.
4.2 Maintenance Support Tools

Certain long-lived software systems continue to incur substantial costs after they have been fielded. These costs can represent a significant part of the total system life cycle cost. Software system modifications and improvements occur throughout the life cycle. Modifications are usually carried out by personnel who were not involved in the development of the software system. As a result, they are often faced with inadequate information about many aspects of the system's behavior and design.

To provide maintenance personnel with adequate information to maintain the system, new tools are being developed. These include tools for visualizing the structure of code and tools for navigating through a large volume of design information. For example, the Air Force is constructing a hypermedia system to provide maintenance personnel with a mechanism for navigating through a large set of system documents as part of the Modular Embedded Computer Software (MECS) for the Advanced Avionics Systems (MECS) program. The DoD is developing the Computer Aided Logistics System (CALS) to automate the collection and dissemination of design information throughout the life cycle.

Other tools which support the maintenance phase for complex software include high-fidelity hardware/software simulators and run-time data collection and monitoring systems which provide information that can be used to diagnose faults.

More research-oriented tools include visualization systems which provide a graphical representation of the system's behavior. This could include the behavior of individual programs or more global views of system operation in the case of a distributed system.

Perhaps the most important issue for the support of the maintenance portion of the software life cycle relates to determining what kinds of design information should be carried through to the maintenance phase and how this information should be represented for the best use by maintenance personnel. The experimentation needed to achieve the third level of measurement described in Section 3.2, the assessment of method effectiveness, should be used to explore this issue. Current approaches based on written documentation leave much to be desired.

4.3 Knowledge-Based Tool Support for Software Analysis and Test

The use of Artificial Intelligence (AI) technology is a current trend in the automation of software engineering technology. Knowledge-based tools, a type of AI technology, would also prove beneficial for the software test engineer. Knowledge-based tools should incorporate rules for testing gleaned from experimentation and the most effective testers. By thereby enhancing the ability of the average tester to approximate the abilities of the best, this type of tool support would reduce the variability in analyst/tester productivity and effectiveness. This support should include:

- Guidance for selecting test techniques based on detecting desired faults classes
- Building libraries of hazards and rules which check against these hazards
- Providing fault classes, rates, and severity as input for software risk analysis methods (For example, see Sherer [32])
- Procedural guidance, for example, statistical sampling support, data flow guided testing support
- Visualization tools for exploring the input domain and analysis tools for spanning/obtaining coverage of this domain

An example of a knowledge-based system for supporting software engineering is the Knowledge-Based Software Assistant (KBSA) [41]. Mid-term goals for the KBSA included automatic test generation. Knowledge based support was planned to assist in the generation of tests "based on specific test knowledge about the user and the application domain," "to increase the density of tests in areas of most relevance," and to track "a mixture of user-defined test cases, test cases generated by uniform, automatic procedures, and those generated from specific domain and design knowledge."

The long-term goals of the KBSA were much more revolutionary. The KBSA uses formal reasoning and formal specification throughout the life cycle. By use of a rapid prototyping style, the developer can ensure that a system meets the user needs. As each lower level of abstraction is developed (for example, preliminary designs, detailed designs, code), a formal proof is developed that ensures implementations and specifications are equivalent. These derivations are stored so a change to the specification can automatically generate the needed changes at lower levels. With this strong emphasis on requirements, prototyping, and formal verification, the need for testing as a separate phase at the end of development is much diminished. The long-term KBSA goal for testing was that testing...
would disappear as a separate activity. Testing was planned to be redistributed into the validation and development activities.

The actual development of the KBSA emphasized this long-term goal of integrating testing with other system validation activities. KBSA, then, is a demonstration of this report's central thesis, that testing should be considered just one of many analysis activities conducted throughout the life cycle. Knowledge-based testing tools, even if they do not all promote as radical a paradigm change as KBSA, can support a development style consistent with modern notions about analysis and test as a preventive, systems-oriented activity.
5. INTEGRATION WITH ADVANCED SOFTWARE DEVELOPMENT TECHNOLOGY

The technical challenge in advancing the state of software analysis and test techniques is made more difficult by advances in software development technology. Analysis and test techniques need not only to support traditional practices, but also to meet concerns that will develop as these advanced methods and architectures become more widely used. At present, the advanced development technologies of interest are formal methods, object-oriented development, artificial intelligence, and parallel and distributed systems. The growing importance and complexity of software also requires that software analysis and test deal with system engineering issues. These technologies are interrelated. However, the following sections examine the key issues for each technology area.

5.1 Formal Methods

Due to the increased application of software in high integrity applications and the development of associated standards (e.g., UK Defense Ministry MoD Standards 00-55 and 00-56), there has been a resurgence of interest in the use of formal methods of program specification and verification. Formal methods are techniques for rigorous reasoning about software properties. In the strictest view, formal methods require axiomatic reasoning and proofs of correctness based on the constructs and rules of mathematical logic. According to [42], a formal development effort consists of four steps:

1. Formalization of the set of assumptions characterizing the intended operating environment.
2. Formal characterization of the system specification.
3. Formalization of an implementation, where an implementation is a decomposition of the specification to a more detailed specification.
4. Proof that the implementation satisfies the specification under the assumptions for the operating environment.

While the benefits of formal specifications are being increasingly recognized and several languages exist, proofs of correctness have not yet proven practical for most systems. A strategy under development at NASA Langley Research Center addresses the use of formal methods by considering levels of its use [43], the lowest level being the development of a formal specification and the highest level being the use of a formal theorem prover. Thus, a critical issue is how to complement the use of formal methods, particularly formal verification techniques, with other less rigorous but perhaps more practical analysis and test techniques. This critical issue needs to be addressed in the context of defining a software process model that integrates the use of formal methods with other software analysis and test techniques.

Some examples of this integration are the Microelectronics and Computer Technology Corporation's (MCC's) definition and development of the SPECTRA environment, which facilitates communication between the developers of the formal models and the system user [44]; Mannering and Cohen's [45] work on integrating formal methods within a total analysis framework; and Mill's Cleanroom approach [11].

Key to defining this new software process model is the identification of the role that formal methods should play and the means of interfacing formal methods to other techniques. The appropriate role depends on which life cycle activities would benefit the most from formal methods and which system properties are better verified by proof than by testing. For example, security, safety, and temporal properties are not easily tested. As more parallel and distributed systems are developed, the temporal behavior becomes more complex and less amenable to testing. However, temporal logics show promise for proving properties of concurrency [46], [47], [48], [49]. The means of interfacing formal methods to other techniques may require the development of a formal semantics for those techniques. It would then be possible to reason about the correctness of their representation of system properties vis a vis a formal specification of the system.

5.2 Object-Oriented Development

The term object-oriented applies to many areas of software development technology. These areas include object-oriented specification, requirements analysis, design techniques, applications, programming, languages, and test strategies, to name a few. An object-oriented approach to software development can best be defined as the development of software systems structured as collections of Abstract Data Types (ADTs). Unlike traditional process-centered software development efforts, object-
oriented development centers around the representation, relationship, and manipulation of objects which "contain" both data and the methods (operations) which define how objects can be manipulated. One of the key differences in applying object-oriented methods to software problems is that the focus of the development effort shifts toward constructing more tangible product-centered objects and away from the abstract process-centered concepts.

In a recent study, a project conducted by Research Triangle Institute in conjunction with a team of students and a professor from a graduate-level software engineering class addressed how object-oriented design approaches differ from process-centered solutions. The project, nicknamed DAGOBHAH, involved the development of space vehicle guidance and engine control software that had previously been developed using extended structured analysis. In attempting to use object-oriented methods for the DAGOBHAH project, it was discovered that though the actual problem was a good application for object-oriented programming, researchers had to change their thinking about the entire problem and its solution.

The most formidable stumbling block encountered during the object-oriented development was that the specification was written with "processes" not "objects" in mind. One difficulty the DAGOBHAH development team encountered was the inability to reuse non-object-oriented external interface routines that already existed in code libraries. It is important to note that although some object-oriented languages (like C++) allow the inclusion of code written in other non-object-oriented languages which "solves" the problem of interfacing and using non-object-oriented code libraries with object-oriented code, such "flexibility" violates the whole purpose of developing an object-oriented solution and complicates the validation of the solution by mixing two separate approaches. With this in mind, the team decided that in order to achieve a completely object-oriented solution, all external interface routines needed to be rewritten as object-oriented routines. This problem, while of minor scale for this application, indicates that there would be considerable effort in the translation of larger scale non-object-oriented applications and tested code libraries. Object-oriented translations of existing applications and code libraries will have to be validated against the original non-object-oriented versions.

In general, the decomposition of a system using an object-oriented approach becomes, in the most elementary sense, a collection of abstract objects. Detailed information about the objects (their data and procedures) tends not to be available at the design stage and in fact is deferred almost to the coding stage because the hiding of information that is private to the object is considered the desired behavior of an object. Object-oriented programming is, for the most part, a bottom-up, iterative development effort opposed to the process-centered top-down approach.

Though the object-oriented paradigm is dramatically different than the more typical process-centered paradigms, many early life cycle analysis techniques can still be effective because certain aspects of object-oriented software development can be categorized into DoD-2167A-like phases, but the techniques applied at each phase need to be modified to map to the iterative object-oriented paradigm.

Requirements analysis, for example, is a technique that is important for both process-centered and object-oriented system development. If a requirement is not specified accurately and completely, an object (in the case of object-oriented) or a function (if process-centered) will not fulfill its desired purpose. One technique used in object-oriented system development to assist in the identification and behavior of objects is rapid-prototyping. Through these prototypes, objects, their behavior, and relationships with other objects can be identified and this information can then be folded back into the specification to provide more detailed requirements.

In an object-oriented system, analysis of objects, their relationship to other objects and the entire interaction of the system is similar to the process interaction analysis performed on process-centered systems. While quality assessment techniques such as performance modeling, reliability modeling, safety analysis, and security analysis can be used early in the life cycle of object-oriented systems, these analysis techniques need to be modified to address the analysis of objects, as opposed to functions found in traditional process-centered solutions.

In object-oriented approaches, encapsulation and information hiding cause us to modify our testing strategy. Object-oriented encapsulation impacts the way test designers view software testing. For

---

1The students, from Duke University, were enrolled in a software engineering course taught at the University of North Carolina by a UNC professor.
example, in an object-oriented application, a basic testable unit is no longer a subprogram. In fact in terms of object-oriented software, the smallest basic testable unit is a class (a collection of objects, an abstract data type). Because subprograms do not exist in the traditional sense in object-oriented applications, test designers need to modify strategies for integration testing. Test designers will be dealing with larger program units (e.g., a class) and will need to be concerned with two separate aspects of a class, the operation (the capability and external interface) and the method (the hidden internal algorithm that carries out the "operation"). Due to the fact that classes can "inherit" characteristics from other classes, the issue of testing some components as they are developed may not provide us with any useful information until the system is fully integrated. Only through careful planning and design will testers be able to avoid the "big-bang" integration effect. Information hiding also impacts the type of testing we can use on object-oriented programs. Objects tend to be "black-boxes" which carefully hide information from other parts of the system. Test strategies will need to emphasize the creation of test cases which explore the boundaries of the objects they are testing [54]. Structural (white-box) testing strategies tend to be difficult to apply since much of the internal working objects are not visible outside the object itself.

As object-oriented libraries are developed and objects are reused, test techniques need to be applied to both the old and the new objects in the system. Extensive testing of proven objects does not "excuse" an object from testing when it is used in a new system. Object-oriented systems comprised of many objects are difficult to test because every object in the system has the potential of being removed, replaced, or modified. This requires the development of strategies and tools for evolving test plans, procedures, and test cases during the frequent changes that may result from the highly iterative nature of object-oriented development.

5.3 Analysis and Test of Parallel Software

The development of software for parallel and distributed architectures presents analysis and test issues of greater magnitude and complexity than that of sequential software. While optimal performance and tolerance to faults are generally required of parallel systems, the effects of intertask communication and the match between inherent application task granularity and hardware architecture structure make it difficult to achieve these goals. Intertask communication and the need to match task granularity to architecture structure cause the parallel software engineering paradigm to differ from the existing software engineering paradigms in two primary ways. First, significantly more emphasis needs to be placed on the consideration of performance, reliability, and fault tolerance early in the life cycle during specification and design analysis activities. Second, the development and evaluation of parallel software requires the consideration of system and hardware issues to an extent that parallel applications programmers deal with issues in the parallel domain that are typically dealt with by systems programmers in sequential software development efforts.

Although parallel software analysis activities need to be conducted with knowledge both about the target architecture and the system reliability and performance requirements, many of the language-extension approaches that are in use today (e.g., Linda) ease part of the parallel software development burden by isolating the programmer from the target architecture. Fully utilizing these architectures, however, still requires an intimate knowledge of the target hardware's structure and behavior. Depending on software developers for this knowledge may not be realistic, particularly when using multiple target architectures. Fortunately, this knowledge does not have to be directly available to the parallel algorithm designers, software analysts, and programmers. It can be encapsulated in models, incorporated in language features, or hidden in expert systems and refined automatically as the system is developed. The use of complementary modeling and simulation methods to determine performance and reliability trade-offs for various algorithm decompositions, even at a low fidelity, provides a method for evaluating parallel software with respect to system requirements and hardware characteristics early in the life cycle.

The information contained in Table 5-1 shows some of the factors related to producing high-performance, high-quality parallel software at minimal cost. Even where these characteristics also relate to nondistributed systems, the problems in producing parallel software are more complex. For example, I/O rates affect both the performance of a sequential and parallel system. In parallel systems, however, I/O rates are of concern for both the interfaces between components within the system and the interface between the system and the external environment; only the latter concern exists for sequential systems. Language features are drivers of the quality of both sequential and parallel system. The ability to concisely express an algorithm in high-level terms relating to the application domain supports the
development of higher quality software. The compiler should encapsulate machine-level details such as the allocation of variables to memory locations and registers. How to hide such details is a much more contentious issue for parallel systems. Of course, many of the factors in Table 5-1 apply only to parallel systems.

Table 5-1: Some Factors in Analysis and Test of Parallel Software

<table>
<thead>
<tr>
<th>Performance</th>
<th>Quality</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Algorithm characteristics</td>
<td>- Design robustness</td>
<td>- Portability of Code</td>
</tr>
<tr>
<td>- Granularity of parallelism of problem and machine</td>
<td>- Fault detection, isolation, and recovery strategy</td>
<td>- Amount of life cycle tool support</td>
</tr>
<tr>
<td>- Degree to which machine and problem granularity matches</td>
<td>- Test effectiveness for concurrent, asynchronous, and real-time conditions</td>
<td>- Extent of code reuse</td>
</tr>
<tr>
<td>- I/O rates and limitations</td>
<td>- Data flow and control flow characteristics (e.g., complexity, data, volume, and distribution)</td>
<td>- Amount of automated code generation</td>
</tr>
<tr>
<td>- Data distribution, organization and management</td>
<td>- Processor workload utilization and balance</td>
<td>- Whether the application is new or existing</td>
</tr>
<tr>
<td>- Degree of data and/or function migration</td>
<td>- Amount of memory, processor, and interconnection resource contention activity</td>
<td>- Degree to which machine and problem granularity matches</td>
</tr>
<tr>
<td>- Fault detection, isolation, and recovery overhead</td>
<td>- Probability of deadlock, race, and starvation conditions</td>
<td>- Problem size</td>
</tr>
<tr>
<td>- Programming language features/compiler optimization</td>
<td>- Reproducibility of testing</td>
<td>- Number of processors</td>
</tr>
<tr>
<td>- Run-time environment support</td>
<td>- Language features</td>
<td>- Effectiveness and efficiency of life cycle activities</td>
</tr>
</tbody>
</table>

Effective parallel applications depend on more than providing the basic computational capacity. It is also not sufficient to break the algorithms into somewhat uniformly sized tasks and map the tasks to resources within the architecture. Effective decompositions are based on trade-offs between architectures and what has been termed "algotecture". That is, algorithms may need to be restructured to enhance opportunities for parallelism. Often the algorithm structure coupled with data or parameter dependencies may render a particular decomposition ineffective. For example, a mission planning algorithm may be broken down into a large number of independent integer programming problems. If these tasks were mapped to separate resources, some tasks would complete before others due to the specific data supplied to them. If all tasks must complete before other processors can start, the resources associated with all tasks, except the last one to complete, will remain idle until the last one completes. Similarly, decomposing search algorithms typically allocates different portions of a search tree to different processors. If one process determines that particular portions of a search can be terminated, that information may need to be communicated to the other tasks. Until the other tasks receive that information, they are likely to be performing unnecessary work. In both of the cases, the approach to parallel decomposition may result in poor utilization of resources, and thus in poor performance. Identifying the occurrence of structures with poor resource utilization is the first step in finding improvements.
The design of parallel software systems typically requires greater awareness of hardware details. One facet of the problem of matching software and hardware characteristics is the comparison of process and machine granularities for parallel applications. The level of parallelism in the algorithm required for a system may imply that vectorized computations are desirable for a specific set of calculations. If the hardware for the target architecture does not offer these facilities and their emulation cannot utilize the whole architecture, the resulting system will have periods of under-use while the vectors are processed. On the other hand, several complex and sparsely interacting execution processes would be practically unable to efficiently use most vector machines. In both of these cases, the hardware facilities must restrict the design space of the system's software to obtain the maximum system performance.

More than general knowledge about the type of hardware architecture is needed for the analysis of parallel software systems. Specific details, such as the number of processors, their memory capacities, and their interconnection topology, constrain the design space for replicating software tasks and assigning these tasks to processors, thus imposing restrictions on the grain size of each task. Typical goals of the assignment of software tasks to processors are to achieve performance or to maintain degraded performance levels upon processor failure through static and dynamic load balancing. Optimizing this assignment strategy requires knowledge about system reliability, performance, and fault-tolerant requirements and the target parallel architecture.

A mixed relationship exists between parallelism and fault tolerance. Parallelism implies well demarcated synchronization points, thus enabling the establishment of recovery points. Increased parallelism also implies smaller grain tasks, permitting incorporation of redundancy at very granular levels and frequent state recovery if needed. The effectiveness of fault-tolerant parallel programs cannot be ascertained, however, without considering the additional cost of the supporting hardware. For example, providing many parallel tasks with multiple recovery points while maintaining timelines may require special hardware, such as content addressable memory.

The analysis and test process is more difficult in parallel software. The interaction of multiple execution streams increases the frequency and complexity of the class of errors known as synchronization or "timing" errors. The asynchronous or loosely synchronous execution streams found in parallel software can exhibit deadlock, race, overflow, and starvation conditions, causing failures to propagate from execution stream to execution stream. Research has shown that addressing these types of errors in distributed, message-passing systems can be a non-trivial task [55]. Also, shared memory systems are particularly vulnerable to the situation in which a failed memory device or software element can contaminate a properly executing stream by feeding it with incorrect data. Testing is also complicated by execution-order variations among interacting software processes. Reproducibility is generally not a problem in the behavior of sequential software, but parallel software may not be as cooperative. Repeatability of execution order is not generally guaranteed in loosely coupled architectures. This feature dramatically increases the possible number of software states and actions, making their testing much more difficult.

Review of these factors suggests that a cohesive framework for the design, development, analysis and test of parallel software within a total systems context is needed if both near and long-term insight into parallel software engineering problems is to emerge. Table 5-2 identifies a list of activities that could be conducted at each life cycle phase within such a framework. Actual system developments should select appropriate analysis and test activities as part of upfront life cycle design based on system characteristics.

Due to being on the forefront of advanced computing technology, procedures for simultaneously addressing fault tolerance and performance requirements for parallel architectures are not well established. These procedures should rely on achieving complementary completeness through diverse models. That is, they should integrate and reconcile the diverse points of view necessary for parallel system design and evaluation, including fault-tolerant behavior, reliability, and performance. Reconciling and combining these diverse points of view is a present challenge.
5.4 System Engineering Issues

The increasing visibility and importance of software in modern systems, the more stringent system requirements being levied against software applications as a result, and the more complex nature of ever larger software applications necessitate addressing software issues in a system context. Systems engineering is a rapidly growing discipline for attacking these issues. After the very early life cycle phases, a system is partitioned into various component parts, some of which may be hardware and others software. Traditionally, these component subsystems are developed independently with very little attempt to keep an overall systems perspective. Systems engineering approaches provide this systems perspective, with consequent changes in the software life cycle and development methodologies. With such radical changes in approach and viewpoint, the demands on software analysis and test technologies are quite different.

Since parallel systems, by definition, are composed of several interacting components, many of the analysis and test issues discussed in Section 5.3 are concrete illustrations of more general system engineering problems. For example, the decomposition of a software application to take best advantage of the hardware structure is a concern in both parallel processing and systems engineering. In fact, one could consider the design of parallel systems to be a subfield of systems engineering.

Mission requirements establish criteria for various system characteristics such as functionality, reliability, testability, maintainability, computing performance, and life cycle cost. For complex systems, the design trade-offs between these attributes are not confined to isolated design areas such as hardware architecture, application algorithm structure, application software structure, operating systems.

### Table 5-2: Analysis and Test Activities for Parallel Software

<table>
<thead>
<tr>
<th>PHASE</th>
<th>ANALYSIS ACTIVITIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td>- Algorithm Specification and Analysis</td>
</tr>
<tr>
<td>and Design</td>
<td>- Data Distribution and Flow</td>
</tr>
<tr>
<td></td>
<td>- Control Flow</td>
</tr>
<tr>
<td></td>
<td>- Mapping to Architecture Models</td>
</tr>
<tr>
<td></td>
<td>- System Attribute Trade-off Analysis</td>
</tr>
<tr>
<td></td>
<td>- Reliability vs. Performance</td>
</tr>
<tr>
<td></td>
<td>- Task Replication Requirements</td>
</tr>
<tr>
<td></td>
<td>- Design Simulations or Walkthroughs</td>
</tr>
<tr>
<td></td>
<td>- Number of Processes Required</td>
</tr>
<tr>
<td></td>
<td>- Processor Utilization</td>
</tr>
<tr>
<td></td>
<td>- Workload Balancing</td>
</tr>
<tr>
<td></td>
<td>- Memory, Processor, Interconnection, Resource Allocation, and Contention</td>
</tr>
<tr>
<td></td>
<td>- Operation Sequencing</td>
</tr>
<tr>
<td></td>
<td>- Object-Oriented Analysis</td>
</tr>
<tr>
<td>Implement</td>
<td>- Manual and Automated Code Generation</td>
</tr>
<tr>
<td></td>
<td>- Software Reuse Analysis</td>
</tr>
<tr>
<td></td>
<td>- Static Analysis of Code</td>
</tr>
<tr>
<td></td>
<td>- Walkthroughs</td>
</tr>
<tr>
<td></td>
<td>- Set/use and Order of Operations</td>
</tr>
<tr>
<td></td>
<td>- Timing and Memory Requirements</td>
</tr>
<tr>
<td>Test</td>
<td>- Symbolic Debugging</td>
</tr>
<tr>
<td></td>
<td>- Functional and Usage Testing</td>
</tr>
<tr>
<td></td>
<td>- Failure Mode Testing</td>
</tr>
<tr>
<td></td>
<td>- Deadlock</td>
</tr>
<tr>
<td></td>
<td>- Race</td>
</tr>
<tr>
<td></td>
<td>- Starvation</td>
</tr>
<tr>
<td></td>
<td>- Reconfiguration</td>
</tr>
<tr>
<td></td>
<td>- Timing and Latency</td>
</tr>
<tr>
<td></td>
<td>- Operation Sequencing and Memory</td>
</tr>
<tr>
<td></td>
<td>- Addressing</td>
</tr>
</tbody>
</table>
architecture, or communications system architecture. Design decisions are dependent upon the effects they have on other system elements. Moreover, localized optimization of each system element does not, in general, lead to global optimization of system design. For example, the computing performance for application software optimized for a given hardware architecture and a given algorithm for a specified function may not meet requirements. Another algorithm for the same specified function and given architecture could lead to an application software structure that results in far better computing performance.

The trade-offs needed to develop an optimal system generally cannot be carried out solely by an algorithm designer, a hardware architect, or a software architect. Consequently, system design will involve multidisciplinary teams. Each team member must analyze the effects of design decisions on their portion of the design.

This multidisciplinary nature of system engineering introduces new requirements for analysis and test technology. There is a critical need for automated tools that manage the design complexity and provide appropriate design analysis and metrics across the various design disciplines. Reliability and performance are two areas of concern to system engineers where additional automated analysis support would be particularly valuable.

To satisfy extremely high reliability requirements, software must be developed that can both detect and correct hardware, software, and hardware-induced software faults. Some techniques have already been applied to these areas in both sequential and multiprocessor architectures, including the use of recovery blocks, check-pointing, atomic-actions, assertion-checking, and multi-version programming. Unfortunately, fault detection and correction are handled to different extents and to varying degrees of transparency in today's multiprocessor development tools. Many of the multiprocessing software support tools understand the limitations of their target architectures, particularly the constraints on the number of processors that can be made available, but fail to take advantage in applying them to provide fault tolerance. Multi-processor architectures should be able to utilize their innate redundancy by reconfiguring the assignment of functions and/or objects to processing elements or by selecting alternate communications paths in response to a component or subsystem failure. Few multiprocessor software tools in existence today address the software implications of their target hardware's fault tolerance capabilities.

System performance requirements may also constrain the design and implementation of the software for that system, particularly in real-time systems. Software abstractions that hide lower-level details may consume excessive systems resources, rendering them unusable for the given system. In fact, the strict deadline requirements of real-time systems can force a dramatic restructuring of software in order to meet the specified cycling rate or response time.

The performance of many systems is achieved by an effective utilization of one or more limited resources. Many times this limited resource is hardware, frequently processing power, storage, or bandwidth. In these situations, the proper matching between the demands of software and the hardware resources is critical. The choice of a particular algorithm, or software mechanism, can be the difference between a highly utilized system, a poorly performing one, and one that fails completely due to an insufficient resource.

These problems are best addressed in the early stages of system development so that specific performance requirements can be included in the software requirements and clear design constraints for achieving performance requirements can be included in the software specifications. One technique for performing the analysis necessary to develop the requirements and constraints is to use models of the application and target hardware components to examine system behavior. In particular, Petri net and directed graph models have been used to develop a system model from a computational model of the application and a structural model of the hardware. The Petri net or directed graph model can then be analyzed statically or dynamically, through simulation, to examine properties of the system and predict system performance. The computational model of the application captures the processing and communications workloads of application functions based on the types and sizes of data, the types of instructions, and the data and control flows necessary for processing the application. The system model captures the dependencies and interactions among elements of the computations model and their competition, or contention, for elements in the hardware structural model.

This discussion has highlighted certain enhancements that analysis and test technology needs to support system engineering:
• Analysis techniques that integrate local concerns with global systems views
• Tools supporting multidisciplinary analyses
• Software analysis tools and techniques that permit specification of all relevant hardware characteristics
• System level models.

Enhancements such as these will permit trade-off studies of alternative approaches, thus allowing the best system designs to be reflected in software requirements and specifications and allowing the identification of pitfalls to be avoided.
6. SUMMARY AND RECOMMENDATIONS

There is much room for improvement in software analysis and test technology. Improvement in the analysis and test process, integration of software analysis and test tools in software development frameworks, and approaches for advanced software development technology are a few areas where contributions can be made. By viewing software analysis and test activity from a systems perspective and by taking a preventive approach, this technology can be made more cost-effective.

A key recommendation is to develop a roadmap which addresses the following needs:

- life cycle integration of software analysis and test techniques with systems engineering analysis techniques
- integrated tools that enable analysis and testing of electronically captured specification and design information
- knowledge bases which provide data on error classes by application domain and which guide the development of strategies for effective analysis and test
- decision support tools which enable the acquisition specialist or certifying agent to assess the quality of the analysis and test process and of the resulting end-product

As components of this roadmap are developed, the cost/benefits and commercial availability of software analysis and testing technology will improve.
7. REFERENCES


Appendix A.
ACRONYMS

ACT Analysis of Complexity Tool
ADAS Architecture Design and Assessment System (A registered trademark of Research Triangle Institute)
AD/CYCLE Application Development/Cycle
AI Artificial Intelligence
AISLE Ada Integrated Software Lifecycle Environment
AMS Automated Measurement System
ANS American Nuclear Society
ANSI American National Standards Institute
ASA Automata and Structured Analysis
ASQS Assistant for Specifying the Quality of Software
ASTM American Society for Testing and Materials
ASSIST Abstract Semi-Markov Specification Interface to the SURE Tool
AT&T American Telephone and Telegraph
ATVS Automated Test and Verification System
BASE Boeing Applied Systems Environment
BSI British Standards Institution
CAFTA Computer Aided Fault Tree Analysis
CALS Computer Aided Logistics System
CARE III Computer-Aided Reliability Estimation
CASE Computer-Aided Software Engineering
CCC Change and Configuration Control
CISLE C Integrated Software Lifecycle Environment
CMORT Management Oversight and Risk Tree
CMT Configuration Management Tool
DARPA Defense Advanced Research Projects Agency
DEC Digital Equipment Corporation
DEC/CMS DEC Code Management System
DEC/MMS DEC Module Management System
DEC/PCA DEC Performance and Coverage Analysis
DEC/SCA DEC Static Code Analysis
DEC/TM DEC Test Manager
DG Data General
DoD Department of Defense
EUROCAE European Commission for Aeronautics
EWICS European Workshop on Industrial Computer Systems
FAA Federal Aviation Administration
FIPS Federal Information Publication System
HOL Higher Order Logic
HP Hewlett Packard
IBM-PC International Business Machines Personal Computer
IDA Institute for Defense Analysis
IEC International Electrotechnical Commission
IEEE Institute of Electrical and Electronic Engineers
I-P-O Input-Process-Output
IPT Inc. Integrated Program Technologies Incorporated
KBSA Knowledge-Based Software Assistant
MALPAS MALvern Program Analysis Suite
MAT Maintainability Analysis Tool
MCC Microelectronics and Computer Technology Corporation
MECS Modular Embedded Computer Software
MoD Ministry of Defense
MTBF Mean Time Between Failure
NASA National Aeronautics and Space Administration
NASA-LaRC NASA Langley Research Center
NATO North Atlantic Treaty Organization
O-O Object-Oriented
PAT Process Activation Table
PDL Program Design Language
POSE Picture Oriented Software Engineering
PVCS Portable Voice Communications System
QUES Quality Evaluation System
RADC Rome Air Development Center
RL Rome Laboratory
RTCA Radio Technical Commission for Aeronautics
RTI Research Triangle Institute
SA Structured Analysis
SAE Society of Automotive Engineers
SAIC Science Applications International Corporation
SAW Software Analysis Workstation
SD Structured Design
SDI/NTB Strategic Defense Initiative/National Test Bed
See Software Engineering Environment
SEI Software Engineering Institute
SLCSE Software Life Cycle Support Environment
SMARTS Software Maintenance and Regression Test System
SPADE Southampton Program Analysis and Development Environment
SPARK SPADE Ada Kernel
SQL Structured Query Language
SSADM Structure Systems Analysis Design Method
SSE Software Support Environment
STANAG NATO Standardization Agreement
STARS Software Technology for Adaptable Reliable Systems
START Structured Testing and Requirements Tool
SURE Semi-Markov Unreliability Range Evaluator
TAME Tailoring an Ada Measurement Environment
TCAT Test Coverage Analysis Tool

TDGEN Test file/Data Generator

UK United Kingdom

VDM Vienna Development Method

V&V Verification and Validation

WITS Westinghouse Information Tracking System
## Appendix B.
**SOFTWARE ANALYSIS AND TEST TOOLS**

### CASE TOOL DESCRIPTIONS

<table>
<thead>
<tr>
<th>Tool Name and Vendor</th>
<th>Product Description</th>
<th>Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyst/Designer Toolkit Yourdon Inc.</td>
<td>data flow, entity relationship modeling; structure, flow chart, or state transition design; design, rule &amp; consistency checks</td>
<td>PC</td>
</tr>
<tr>
<td>Auto-mate Plus Learmonth &amp; Burchett Management Systems</td>
<td>British SSADM method; data flow, entity relationship, or process dependence modeling; structure or flow chart-based design; design rule, consistency, &amp; cross-diagram checks</td>
<td>IBM-PC</td>
</tr>
<tr>
<td>DesignAid NASTEC Corp.</td>
<td>data flow, entity relationship, or process dependence/action modeling; structure or flow chart, Jackson, state transition, decision tree, or I-P-O hierarchy design; design rule, consistency, &amp; cross-diagram checks; multi-user</td>
<td>IBM-PC</td>
</tr>
<tr>
<td>Excelerator Index Technology</td>
<td>data flow, entity relationship modeling; structure, flow chart, or decision tree design; design rule, consistency, &amp; cross-diagram checks; parallel-users</td>
<td>IBM-PC</td>
</tr>
<tr>
<td>IEW Analysis &amp; Design Workstation KnowledgeWare</td>
<td>entity relationship, data flow, process action modeling; structure or action chart design; design rule, consistency &amp; cross-diagram checks;</td>
<td>IBM-PC (+host)</td>
</tr>
<tr>
<td>MicroSTEP Syscorp International</td>
<td>specification and data flow editor, data dictionary, and code generator</td>
<td>IBM-PC</td>
</tr>
<tr>
<td>POSE Computer Systems Advisors Inc.</td>
<td>data flow or process action modeling; structure or action chart design; design rule &amp; consistency checks</td>
<td>IBM-PC</td>
</tr>
<tr>
<td>Software through Pictures Interactive Development Environments</td>
<td>data flow, entity relationship, process action, or object modeling; O-O, state-transition, or structured design; design and dictionary consistency checks; requirements tracing; multiuser</td>
<td>DEC VAX, Apollo, Sun, HP9000</td>
</tr>
<tr>
<td>Teamwork Cadre Technologies Inc.</td>
<td>data flow, entity relationship, or process dependence/action modeling; structure or flow chart decision tree, state transition design; decomposition &amp; consistency checks; similar to mainframe version</td>
<td>IBM-PC, DEC VAX Sun, Apollo</td>
</tr>
<tr>
<td>vsDesigner Visual Software Inc.</td>
<td>data flow, entity relationship, or process action modeling; structure or action charts, state transition, Jackson, and other design; design rule &amp; consistency checks</td>
<td>IBM-PC</td>
</tr>
<tr>
<td>Tool Name</td>
<td>Vendor</td>
<td>Product Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ACT</td>
<td>McCabe &amp; Associates</td>
<td>static analysis; module complexity measurement; basis path analysis; specifies conditions for testing each code-path</td>
</tr>
<tr>
<td>AutoTester</td>
<td>Software Recording Corp.</td>
<td>automated test executive; capture and replay-type with comparator</td>
</tr>
<tr>
<td>CALLTEST</td>
<td>Logic Engineering Inc.</td>
<td>automated test executive; subroutine invocation-type for functional black-box tests</td>
</tr>
<tr>
<td>Check-mate</td>
<td>Cinnabar Software</td>
<td>screen and keyboard capture and comparison system; saves test input and output for regression analysis</td>
</tr>
<tr>
<td>Test Manager</td>
<td>DEC</td>
<td>organizes and automates tests; regression testing</td>
</tr>
<tr>
<td>lint-PLUS</td>
<td>IPT Inc.</td>
<td>static &amp; execution analyzer; Interactive debugger; allows tracing of execution flow and/or data updates</td>
</tr>
<tr>
<td>Logiscope</td>
<td>Verilog</td>
<td>static and dynamic analyzer; Halstead &amp; McCabe complexity; test coverage analysis; dead and untested code determination</td>
</tr>
<tr>
<td>SMARTS</td>
<td>Software Research Inc.</td>
<td>test manager, executive, and comparator; program structure-based test selection; reports regression discrepancies</td>
</tr>
<tr>
<td>TCAI</td>
<td>Software Research Inc.</td>
<td>segment-level test coverage analyzer for C, PASCAL, BASIC, et. al.; reports untested code</td>
</tr>
<tr>
<td>TDGEN</td>
<td>Software Research Inc.</td>
<td>test data generator; random, range-spanning, or selection modes</td>
</tr>
</tbody>
</table>
### REQUIREMENTS-TO-TEST TOOL DESCRIPTIONS

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Product Description</th>
<th>Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASA</td>
<td>requirements definition, editing, structuring; requirement allocation; specification simulation; automatic test scenario generation; consistency &amp; completeness checking</td>
<td>Apollo, DEC VAX, Sun</td>
</tr>
<tr>
<td>Verilog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTrace</td>
<td>requirements definition, editing, structuring; requirement allocation; multi-user with audit trail; SQL database; various reports</td>
<td>DEC VAX</td>
</tr>
<tr>
<td>NASTEC Corp.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>START</td>
<td>uses CASE-based data flow &amp; requirements PDL; computes requirements complexity; test generation for requirement control flows</td>
<td>SUN, DEC VAX</td>
</tr>
<tr>
<td>McCabe &amp; Associates</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T Programming</td>
<td>requirements definition, editing, and refinement; consistency &amp; completeness checks; some reverse-specification ability; requirement-to-test mapping; misc. reports</td>
<td>IBM-PC, DEC VAX, HP3000, AT&amp;T 3B</td>
</tr>
<tr>
<td>Environments Inc.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### SOFTWARE METRIC TOOL DESCRIPTIONS

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Product Description</th>
<th>Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-Stat</td>
<td>computes McCabe complexity for C software</td>
<td>IBM-PC, any UNIX</td>
</tr>
<tr>
<td>Software Research Inc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORTRAN-lint</td>
<td>static analysis of FORTRAN code; common block matching, argument/usage consistency checking, and code style evaluation</td>
<td>DG MV, DEC VAX</td>
</tr>
<tr>
<td>IPT Inc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MALPAS</td>
<td>control flow, data usage, and path analysis; some rule checks on design refinements; uses intermediate language only, but translators for PASCAL, Ada, etc. exist</td>
<td>DEC VAX</td>
</tr>
<tr>
<td>Rex, Thompson, &amp; Partners Ltd.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAT</td>
<td>static analysis of FORTRAN code; common block matching, argument/usage consistency checking, program cross-referencing, and maintainability metrics</td>
<td>PCs &amp; others</td>
</tr>
<tr>
<td>SAIC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC-METRIC</td>
<td>McCabe and Halstead code metrics; data reference distance; user-specified standards checking</td>
<td>IBM-PC</td>
</tr>
<tr>
<td>SET Laboratories</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASQS</td>
<td>metric database and adviser for management; assists in metric usage selection &amp; criteria definition</td>
<td>DEC VAX</td>
</tr>
<tr>
<td>Dynamics Research Corp.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUES</td>
<td>metric definition &amp; data collection/ presentation across life cycle; management window into metrics; interfaces to SLSCE</td>
<td>Sun 4, DEC VAX</td>
</tr>
<tr>
<td>Software Productivity Solutions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tool Name</td>
<td>Vendor</td>
<td>Product Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PCA</td>
<td>DEC</td>
<td>software execution monitor; <strong>performance analysis</strong>; statement-level test coverage</td>
</tr>
<tr>
<td>PAT</td>
<td>SAIC</td>
<td>test coverage and performance analyzer; minimal code instrumentation; <strong>module-level invocation count &amp; ranking</strong>; dead-code determination; FORTRAN only</td>
</tr>
<tr>
<td>SAW</td>
<td>MicroCASE</td>
<td>hardware-software execution monitor; execution history; <strong>instruction-level test coverage</strong>; performance analysis</td>
</tr>
<tr>
<td>CAFTA/ETA-II/ RBDA</td>
<td>SAIC</td>
<td>fault and event tree editing; failure rate, availability, and cut-set calculation; cut-set editing and threshold operations</td>
</tr>
<tr>
<td>CMORT/PC-TREE</td>
<td>EG&amp;G</td>
<td>fault tree editing; risk, failure, and cut-set calculation</td>
</tr>
<tr>
<td>CARE III</td>
<td>NASA-LaRC</td>
<td>Reliability evaluation</td>
</tr>
<tr>
<td>SURE</td>
<td>NASA-LaRC</td>
<td>Reliability evaluation</td>
</tr>
<tr>
<td>ASSIST</td>
<td>NASA-LaRC</td>
<td><strong>Markov model</strong> input language for SURE</td>
</tr>
<tr>
<td>ADAS</td>
<td>RTI</td>
<td>An engineering tool set for <strong>system level simulation</strong> supporting <strong>software-hardware co-design</strong></td>
</tr>
</tbody>
</table>
### REVISION CONTROL TOOLS

<table>
<thead>
<tr>
<th>Tool Name Vendor</th>
<th>Product Description</th>
<th>Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aide-De-Camp &amp; Development Systems, Inc.</td>
<td>code and document configuration management system with relational database; audit trails and automated builds</td>
<td>Misc. PCs, Minis, Mainframes</td>
</tr>
<tr>
<td>CCC Softool Corp.</td>
<td>generic configuration management system; audit trails, access control, component dependency tracking</td>
<td>Misc. PCs, Minis, Mainframes</td>
</tr>
<tr>
<td>CMT Expertware Inc.</td>
<td>code and document configuration management system; audit trails, revision reports</td>
<td>Misc PCs, Minis, Mainframes</td>
</tr>
<tr>
<td>CMS DEC</td>
<td>code and document configuration management system; audit trails, revision reports; integrated with VAXset</td>
<td>DEC VAX</td>
</tr>
<tr>
<td>Historian Plus OPCODE Inc.</td>
<td>code management system</td>
<td>Misc. PCs, Minis, Mainframes</td>
</tr>
<tr>
<td>PVCS POLYTRON Corp.</td>
<td>generic configuration management</td>
<td>IBM-PC, Macintosh, DEC VAX</td>
</tr>
</tbody>
</table>

### IMPLEMENTATION ANALYSIS TOOLS

<table>
<thead>
<tr>
<th>Tool Name Vendor</th>
<th>Product Description</th>
<th>Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPADE/SPARK Program Validation Limited</td>
<td>control &amp; data flow analyzer/translator; pre- and post-condition analyzer; proof checker; various language translators</td>
<td>DEC VAX</td>
</tr>
<tr>
<td>TeleUSE Telesoft</td>
<td>automated user-interface constructor</td>
<td>DEC VAX</td>
</tr>
</tbody>
</table>
## TOOL FRAMEWORKS (CASE)

<table>
<thead>
<tr>
<th>Environment</th>
<th>Developer</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLCSE</td>
<td>RADC</td>
</tr>
<tr>
<td>AD/CYCLE</td>
<td>IBM</td>
</tr>
<tr>
<td>Cohesion</td>
<td>DEC</td>
</tr>
<tr>
<td>AISLE/CISLE</td>
<td>Software Systems Design</td>
</tr>
<tr>
<td>BASE</td>
<td>Boeing</td>
</tr>
<tr>
<td>WITS</td>
<td>Westinghouse</td>
</tr>
<tr>
<td>Project East</td>
<td>France, Finland, Canada</td>
</tr>
<tr>
<td>STARS See (DARPA)</td>
<td>Unisys, IBM</td>
</tr>
<tr>
<td>SSE (NASA)</td>
<td>Lockheed</td>
</tr>
<tr>
<td>See (SDI/NTB)</td>
<td>Martin Marietta, IDA, and others</td>
</tr>
</tbody>
</table>
Appendix C.
STANDARDS RELATED TO SOFTWARE ANALYSIS AND TEST

ANS
(American Nuclear Society)

ANSI/ANS-10.4-1987 Guidelines for the V&V of Scientific and Engineering Computer Programs for the Nuclear Industry


NUREG. CR4640 Handbook of Software Quality Assurance Techniques Applicable to the Nuclear Industry


NUREG/CR4473 A Study of the Operation and Maintenance of Computer Systems to Meet the Requirements of 10 C.F.R. 73.55

ASTM STANDARDS
(American Society for Testing and Materials)

ASTM E1113-86 Standard Guide for Project Definition for Computerized Systems


ASTM E624-83 Guide for Developing Implementation Designs for Computerized Systems

ASTM E627-88 Standard Guide for Documenting Computerized Systems

ASTM E919-83 Specification for Software Documentation for a Computerized System

ASTM E1029-84 Documentation of Clinical Laboratory Computer Systems

ASTM E622-84 Generic Guide for Computerized Systems

ASTM E625-87 Guide for Training Users of Computerized Systems

ASTM E1246-88 Standard Practice for Reporting Reliability of Clinical
Laboratory Computer Systems

ASTM E1013-87  Standard Terminology Relating to Computerized Systems
ASTM E1206-87  Standard Guide for Computerization of Existing Equipment

BSI (British Standards Institution)

65A(Secretarial)96  Functional Safety of Programmable Electronics Systems (draft)
65A(Secretarial)94  Software for Computers in Application of Industrial Safety-Related Systems

European Military/Industry Standards

UK Health/Safety Executive - Programmable Electronic Systems (PES's) in Safety Related Applications
UK Interim Draft Defense Standards 00-55, 00-56
EWICS (European Workshop on Industrial Computer Systems) - variety of reference documents:
Guidelines for the Assessment of the Safety and Reliability of High Integrity Industrial Computer Systems
Attributes, Criteria and Measures: their definition and use in safety related projects
Draft Guidelines to Design Computer Systems for Safety
Draft Guidelines on Safety Related Measures to be used in Software Quality Assurance
Guidelines for the maintenance and modification of safety related computer systems
Safety Assessment and Design of Industrial Computer Systems - Techniques Directory

International Electrotechnical Commission (IEC)

Software for Computers in the Application of Industrial Safety-Related Systems

EUROCAE

ED-12A  Software Considerations in Airborne Systems and Equipment
(European equivalent of DO-178A)
FIPS STANDARDS
(Federal Information Publication System)


FIPS-PUB-101 Guideline for Lifecycle Validation, Verification, and Testing of Computer Software

FIPS-PUB-105 Guideline for Software Documentation Management

FIPS-PUB-106 Guideline on Software Maintenance

FIPS-Pub-132 Guidelines for Software Verification and Validation Plans

FIPS Special Pub 500-165 Software Verification and Validation: Role in Computer Assurance and Relationship with Software Project Management Standards

IEEE STANDARDS
(Institute of Electrical and Electronics Engineers)

IEEE Standards
IEEE Service Center
445 Hoes Lane
P. O. Box 1331
Piscataway, NJ 08855-1331 USA
1-800-678-IEEE


982.1-1988 Standard Dictionary of Measures to Produce Reliable Software

982.2-1988 Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software


990-1987 Recommended Practice for Ada as a Program Design Language

1002-1987 Standard Taxonomy for Software Engineering Standards

1008-1987 Standard for Software Unit Testing

1012-1986 Standard for Software Verification and Validation
1016-1987  Recommended Practice for Software Design Descriptions
1016.2-1990 Guide to Software Design Descriptions
1028-1988 Standard for Software Reviews and Audits
1042-1987 Guide to Software Configuration Management
1044-1989 Standard for Classification of Software Errors, Faults, and Failures
1045-1990 Standard for Software Productivity Metrics
1058.1-1987 Standard for Software Project Management Plans
1059-1990 Guide for Software Verification and Validation (June 1990)
1061-1990 Standard for a Software Quality Metrics Methodology (June 1990)
1062-1990 Recommended Practice for Software Acquisition (March 1990)
1063-1987 Standard for Software User Documentation
1074-1990 Standard for Software Life Cycle Processes

MIL-STD (Department of Defense Military Standards)

MIL-STD-2168 Defense Systems Software Quality Program
MIL-STD-2167A Defense Systems Software Development
MIL-STD-882B System Safety Program Requirements

NATO Standardization Agreement (STANAG)

AQAP-13 NATO Software Quality Control System Requirements
AQAP-14 Guide for the Evaluation of a Contractor's Software Quality Control System for Compliance w/ AQAP-13
RTCA/FAA
(Radio Technical Commission for Aeronautics and Federal Aviation Administration)

Advisory Circulars - Federal Aviation Administration
Public Inquiry Center, APA-230
800 Independence Avenue, SW
Washington, DC 20591

Advisory Circular 20-115A
Advisory Circular 25-1309-1B
Draft Verification Advisory Circular
Software considerations in the TSO Process
Checklists for DO-178A Documentation

SAE STANDARDS (Society of Automotive Engineers)

SAE
Department 362
400 Commonwealth Drive
Warrendale, PA 15096 USA

SAE ARP-1834, Fault/Failure Analysis for Digital Systems and Equipment
Appendix D.
ADDITIONAL READING


