Computer Science
Technical Report

Software Reliability Through Fault-Avoidance and Fault-Tolerance

Reports 7-9 (3/1/92-9/1/93) on Research Supported through NASA Grant NAG-1-983

by

Mladen A. Vouk and David F. McAllister

North Carolina State University

Box 8206
Raleigh, NC 27695

Table of Contents

Summary of Accomplishments (NAG-1-983 - 1992/1993)

This summary is the synthesis of reports #7, #8 and #9. The summary covers the period from March 1, 1992 through August 31, 1993. Appendices contain the published papers. The general topic of research was:

Strategies and Tools for Testing, Risk Assessment and Risk Control of Dependable Software-Based Systems

The goals of the efforts undertake during the reported period were several. The primary one was to provide empirical and theoretical information on fault avoidance and fault elimination and control strategies that would be suitable for use during development of software based systems that need to be highly dependable. In parallel we explored the dual-use possibilities of the NASA supported research through cooperation with research programs supported by the National Science Foundation (e.g., testing of concurrent software), special N.C. State supported programs (e.g., information fault-tolerance in the area of high-speed networks and ATM communication that may affect aerospace applications), and through applied research supported by the industry in the areas of software reliability and safety (e.g., modeling of telecommunications outages that can impact safety of the U.S. air traffic) and software process modeling (e.g., use of risk management techniques and reliability models during different software process phases).

Sub-topics were:

1. Testing and Reliability: Investigation of structure based testing, error removal and reliability growth modeling suitable for development and evaluation of dependable and safety-related software systems. Part of this work consists of studies that would enable transfer of the investigated technology to industry, for example to very large DEPENDABLE telecommunications systems.The work on this topic is in progress and results obtained so far are available as reports and papers.
2. Software and Process Risk Assessment, Control and Safety: Risk assessment and control techniques suitable for software intended for dependable systems. This includes study of appropriate software reliability and availability models, use of state-based testing, formal methods in the areas of requirements and design, design for testability, run-time fault-tolerance, etc. Part of this work consists of studies that would enable transfer of the investigated technology to industry, for example to small to medium safety-conscious systems (e.g., appliances under microprocess control, computer-based medical devices, etc.).The work on this topic is in progress and results obtained so far are available as reports and papers.

Testing and Reliability

The objective of this part of the work was to continue development of code coverage based reliability and test effectiveness models in order to improve fault-avoidance and fault-elimination during software production, and to study issues and models that apply in the case of large dependable systems.

The coverage-based models relate the quality of testing, as measured through metrics such as branch coverage, path coverage, definition-use pair coverage, etc., to the residual defect levels and reliability of the software. They are intended to efficiently guide the testing process, as well as offer insight into operational reliability of the product. An existing tool for computing different software

code coverage measures was extended to include some new and promising metrics encompassed by the term "condition testing". The theory of coverage based testing was extended to include more complex models.

Early reliability prediction, based on software development metrics, was studied in the context of a large dependable telecommunications system. Several metrics that can serve to guide process were identified and simple risk-based models for multi-phase prediction were formulated.

The work was coordinated with similar efforts supported by National Science Foundation (with K.C. Tai), and by the telecommunications industry (BNR, Inc.). The latter cooperation is particularly important from the perspective of dual-use technology, and industrial matching of the federally supported research funds. Several conference presentations of the work were made, two are in preparation. Four papers were published. Part of the work will be reported in the *Handbook of Software Reliability Engineering*,.

Break-down by sub-topic

* **Condition-Based Testing: We conducted further theoretical investigations of Boolean and Relational Operator (BRO) testing strategy, and extened the software structure analysis tool BGG to fully incorporate several variants of the BRO metric.** The strategy is promising since it appears to require a relatively small number of test cases to achieve the results similar to or better than some more demanding strategies, such as mutation testing. The BRO theory was refined to cover typed and more complex predicate expressions.

  Accomplishments:
  An updated version of the Basic Graph Generation and Analysis tool (BGG) for dynamic and static analysis of Pascal code was developed. The tool now has an X-window user interface and incorporates a number of additional condition-based metrics (see Appendix IV).

  Further experimental investigation of Boolean and Relational Operator (BRO) testing strategy using different software sources including RSDIMU is in progress. This work will involve comparison of the BRO strategy with other structure based strategies such as branch testing, definition-use path testing, etc. Fault detection capabilities of the strategy will also be compared with black-box techniques and statistical (random) testing.

* **Single-phase and multi-phase software reliability, availability and risk models, suitable for use during development of highly dependable software are, being developed and evaluated.**

  The goal is to provide additional theoretical and empirical basis for estimation of the reliability and availability of large highly dependable software. These models include coverage based and time-based models, and risk-based multi-phase models.

  Accomplishments:
  Reliability and availability models suitable for use with very large critical multi-component telecommunications systems were investigated [Cra92]. Particular attention was directed at multi-state models which can be used to account for a variety of system failure types, as well as for hardware/software interaction. The knowledge gained may be used in building appropriate models for the highly-dependable aerospace applications. In addition, critical rare events, such as FCC reportable network outages were studied and modeled along with the use of models which allow early prediction of the field performance of software based

on the software development and testing indicators (e.g., testing effort per line of code) [Vou93] (also see Appendix II and Appendix IV)

The work was reported at several conferences and was in part published.

* **Papers and reports.**

1. Vouk, M.A., and Coyle, R.E., "BGG: A Testing Coverage Tool," Proc. Seventh Annual Pacific Northwest Software Quality Conference, Lawrence and Craig, Inc., Portland, OR, pp212-233, September 1989.
2. Borger D, "BGG User's Manual", NCSU Department of Computer Science, 1990. (available on request)
3. Vouk, M.A. and McAllister, D.F., "Software Reliability through Fault-Avoidance and Fault-Tolerance", NAG-1-983 presentation, NASA-LaRC, Hampton, May 16, 1990.
4. Vouk, M.A. and McAllister, D.F., "Software Reliability through Fault-Avoidance and Fault-Tolerance", NAG-1-983 presentation, NASA-LaRC, Hampton, January 15, 1991.
5. Vouk, M.A. and Tai, K.C "Software Testing and Reliability", Summary of the Presentation Prepared for the Workshop on Issues in Software Reliability Estimation, Purdue University, May 21, 1991.
6. K.C. Tai, "Theory of Condition-Based Software Testing", Draft Paper, September 1991

(1992 - 1993)

7. Vouk, M.A., "Modeling Software Reliability and Fault Removal During Structure Based Testing," 9th Quality and Productivity Research Conference, Corning, New York, **June 1992**.
8. Cramp R., Vouk M.A., and Jones W., "On Operational Availability of a Large Software-Based Telecommunications System," Proc. Thrid Intl. Symposium on Software Reliability Engineering, IEEE CS, pp. 358-366, **October 1992.**
9. Kenney G.Q, and Vouk M.A., "Measuring Field Quality of Wide-Distribution Commetcial Software," Proc. Thrid Intl. Symposium on Software Reliability Engineering, IEEE CS, pp. 351-357, **October 1992.**
10. Vouk, M.A., "Engineering of Telecommunications Software", *High-Speed Communications Networks*, editor H. Perros, Plenum Press, New York, pp. 227-237, **October 1992**.
11 Vouk M.A., "Using Reliability Models During Testing with Non-Operational Profiles," Proc. Second Workshop on Issues in Software Reliability Estimation, **October 12-13, 1992**, Bellcore, Livingston, N.J.
12. K. C. Tai, A. Paradkar, H.K. Su, and M. A. Vouk, "Fault-Based Test Generation for Cause-Effect Graphs", accepted for CASCON '93, **October 1993 (see Appendix IV)**
13. M. A. Vouk and K.C. Tai, "Some Issues in Multi-Phase Software Reliability Modeling," accepted for CASCON '93, **October 1993 (see Appendix I)**
14. Vouk M.A., and Jones W., "Software Reliability Field Data Analysis," Chapter 10 in *Handbook of Software Reliability Engineering*, McGraw Hill, editor M. Lyu, 1994. (work in progress)

Software and Process Risk Assessment, Control and Safety

The objective of this part of the work was to study and evaluate risk assessment and control techniques suitable for software intended for highly dependable systems. This includes study of appropriate software specification and of the development process. As part of this work we have participated in an international study organized by the International Invitational Workshop on the

Design and Review of Software Controlled Safety-Related Systems. North Carolina State University (Raleigh, NC) developed several prototypes of the GPE Boiler Control and Monitoring (BCM) program. The NCSU solution apart from meeting the immediate objectives of the study allowed us to explore some issues related to the development of small to medium sized safety-critical software (such as the nature of the software process, use of object-oriented state-based testing, and design for testability), and to further explore the issues related to multi-version software experiments. The software is also expected to be part of the kernel test-bed that would be used to investigate formal specification issues that can enhance software testability.

The work was coordinated with similar efforts supported by National Science Foundation (with K.C. Tai). Several conference presentations and papers are in preparation.

* **Software process and risk management models.**

  The goal of this part of the study is to extend theoretical and empirical basis for risk management of highly dependable software.

  Accomplishments:
  A highly iterative prototype based model for software process and risk management model was developed and used in the context of the "Boiler" program exercise described in Appendix V. The model is expected to provide guidance for risk-based process and software design and provide risk evaluation tools such as software reliability and availability estimation modeling, fault-tree analysis, schedule analysis and statistical decision making. Evaluation of the model is in progress. We believe that and avanced version of the model may be applicable in the case of small safety-related applications (e.g., houshold appliances, medical devices).

* **Investigation of the issues related to designing software for testability.**

  The intention is to provide empirical and theoretical information on fault avoidance and fault elimination properties of different approaches and metrics for designing software for testability. We would eventually like to be able to design a coverage metric mix that would provide some assurances concerning the level to which the code Could be tested. We believe that application specific design of metric mixes for the highest testability and sensitivity to high potential loss (risk sensitive) faults is particularly important when predicting behavior of highly dependable software.

  Accomplishments:
  As part of this study we have been investigating the use of cause-effect graphing for software specification and validation (Appendix I). We are in the process of conducting an experiment using the "boiler control" software described in Appendix V.

* **Run-time software fault tolerance.**

  The objective of this part of the study was to investigate advanced software fault-tolerance models in order to provide alternatives and improvements in situations where simple software fault-tolerance strategies break-down.

  Accomplishments:
  Two papers describing work supported in part through this grant have been reprinted, one has been accepted for journal publication. The work in this area is nearing completion. An

overview of the work will be published in the *Handbook of Software Reliability Engineering.*

* **Relevant papers and reports.**

1. Athavale A., "Performance evaluation of hybrid voting schemes", North Carolina State University, Department of Computer Science, M.S. Thesis, December 1989.
2. M.A. Vouk, and D.F. McAllister, "Preliminary Report on Consensus Voting in the Presence of Failure Correlation" in *Software Reliability Through Fault-Avoidance and Fault-Tolerance*, NASA grant NAG-1-983 Progress Report #2 (9/1/89-3/31/90), 1990.
3. Vouk, M.A. and McAllister, D.F., "Software Reliability through Fault-Avoidance and Fault-Tolerance", NAG-1-983 presentation, NASA-LaRC, Hampton, May 16, 1990.
4. D.F. McAllister and R. Scott, "Cost Modeling of Fault Tolerant Software", Information and Software Technology, Vol 33 (8), pp 594-603, October 1991.
5. D.S. Borger. M.A. Vouk, "Modeling the Behavior of Large Software Projects", NCSU Center for Communications and Signal Processing, Technical Report TR-91/19, June 91.

(1992-1993)

6. Vouk, M.A., Paradkar, A., and McAllister, D., "Modeling Execution Time of Multistage N-Version Fault-Tolerant Software," **Reprinted** in *Fault-Tolerant Software Systems: Techniques and Applications*, ed. Hoang Pham, IEEE Computer Society Press, pp 55-61, 1992.
7. Eckhardt D.E., Caglayan A.K., Kelly J.P.J., Knight J.C., Lee L.D., McAllister D.F., and Vouk M.A., "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability," **Reprinted** in *Fault-Tolerant Software Systems: Techniques and Applications*, ed. Hoang Pham, IEEE Computer Society Press, pp 72-82, 1992.
8. Vouk M.A., McAllister D.F., Eckhardt, D.E., and Kim K., "An Empirical Evaluation of Some Software Fault-Tolerance Schemes in the Presence of Failure Correlation," **to appear** in the Journal of Computer and Software Engineering Special Issue on Reliable Software, **1993**.
9. K. C. Tai, A. Paradkar, H.K. Su, and M. A. Vouk, "Fault-Based Test Generation for Cause-Effect Graphs", accepted for CASCON '93, **October 1993 (see Appendix IV)**
10. A. Paradkar, I. Shields, and J. Waters, "The NCSU Solution to the Generic Problem Exercise: Boiler Control and Monitoring System,", NCSU, May 1993.
11. M. Vouk, and A. Paradkar, "Report on the The NCSU Solution to the Generic Problem Exercise: Boiler Control and Monitoring System," NCSU, May 1993, **to appear** in the Proc. of the International Invitational Workshop on the Design and Review of Software Controlled Safety-Related Systems, **June 1993 (see Appendix II).**
12. Vouk M.A., and McAllister D.F., "Software Fault-Tolerance Engineering," Chapter 15 in *Handbook of Software Reliability Engineering*, McGraw Hill, editor M. Lyu, 1994. (work in progress)

Appendix I – Some Issues in Multi-Phase Software Reliability Modeling*

M. A. Vouk[1] and K.C. Tai[1]

Abstract

During early software testing phases, testing profiles are often very different from operational profiles. Consequently, assessment of operational software quality during these non-operational testing stages is difficult, and is open to interpretation. The paper discusses some issues related to this. Software is assumed to be a large system composed of components that evolve in parallel. The focus is on early identification of software components that in operation may be excessively error-prone. The approach involves definition of states based on static and dynamic propositions about the verification and testing history of the software, and the use of that information in models that span multiple testing phases. An example based on a risk model is presented.

1. Introduction

The use of software reliability engineering (SRE), in organizations with advanced software processes, is on the rise. But, some practical obstacles still remain.

For example, SRE requires testing based on an operational profile. An operational profile is a set of relative frequencies of occurrence of the operations associated with the software during its use in the field [Mus93]. Interpretation of many software reliability models assumes failure detection based on operational profiles [Mus87]. Since this assumption is usually violated during early software testing phases (for example, during unit-testing and integration-testing), assessment and control of software quality growth during non-operational testing stages is difficult and open to interpretation.

Another confounding factor can be the (necessary) discontinuities that different testing strategies introduce within one testing phase, or between adjacent testing phases. For instance, unit-testing concentrates on the functionality and coverage of the structures within a software unit; integration-testing concentrates on the coverage of the functions and links that involve two or more software units, etc. It is not unusual to observe an apparent failure-intensity decay (reliability growth) during one of the phases, followed by an upsurge in the failure-intensity in the next phase (due to different types of failures). This oscillatory effect can make reliability growth modeling difficult, although several different approaches have been suggested [e.g., Mus87, Lyu92] .

In an organization that constructs its final deliverable software out of a number of components that evolve in parallel, an added problem can be the variability of the quality across system components

The need to recognize these problems early, so that appropriate corrections can be undertaken within one software release frame, is obvious. How to achieve this is less clear. In general, it is necessary to link the symptoms observed during the early testing phases with the effects observed in the later phases, such, as early operational phase. Several authors have published models that attempt to relate some early software metrics, such as, the size of the code, Halstead length, or cyclomatic number, to the failure proneness of the program [Mun92, Bri93].

This paper is concerned with some issues related to the use of software reliability engineering (SRE) indicators available during early software testing of a large multi-component software system to:

i) **Quantify component quality** expressed, for example, as the number of failures (or problem reports) expected during initial operational deployment of the component;
ii) **Identify problem-prone software components**, that is, components that might show an increased propensity to failure during initial operational deployment (e.g., number of field problem reports);
iii) **Guide software testing process** to minimize the number of failures that can be expected from the components in the future phases of their life cycle.

The software system model we consider here is a system that consists of a collection of software products, or software components, arranged in a certain hierarchy of usage. The components that support the basic functionalities

*Research supported in part by NASA Grant No. NAG-1-983 and NSF grant CCR-8907807
[1]Computer Science Department, Box 8206, North Carolina State University, Raleigh, NC 27695-8206, USA

of the system form the system basis, or root. The interaction among different components and their hierarchy can be described using the system call tree structure [Mus93].

The issues presented in this paper are based, in part, on our experiences with several large commercial systems which, for proprietary reasons, are not identified. The results are exploratory in nature, and are intended to continue the dialogue and provide incentive for further research and input on these topics.

Section 2 discusses the SRE metrics that may be of interest. Section 3 presents a simple multi-phase model for problem identification. The model serves to illustrate the approach, but needs to be extended and made more robust. Section 4 discusses the role of reliability growth models during non-operational testing and the possibilities this offers in terms of software process control.

2. Metrics

We distinguish: a) metrics collected in order to quantify the current software process and quality, and b) the parameters that describe the quality of software in later stages, and that are estimated using the collected data. We call the first group "process and product quality descriptors", or input **drivers**, and the latter "(future state) quality estimators", or **estimators**. We illustrate the metrics through several example drivers and estimators.

2.1 Estimators

2.1.1 Number of Failures

A metric that may be readily available in many organizations is the number of failures observed per product, or component. The number of failures may be chosen because:

i) It has intuitive relationship with the quality of software. For example, one should be uneasy about the software that exhibits more than a certain number of problems immediately upon release to its operational environment.

ii) It has practical value in terms of the amount of attention (and work) the software will receive due to the reported problems.

The number of observed problems will depend on how often and in what manner the system, or a component, is used. This suggests that the data also needs to be collected on the software usage, size, and any other relevant metric. Distinction needs to be made between unique failures, repeated failures, and the underlying faults [Mus87].

2.1.2 Failure-intensity

Failure-intensity is a classical SRE metric [Mus87]. It can be defined as the rate of change of the mean value function, or the number of failures per unit time. The mean value function is the average cumulative number of failures at a point in time. In the context of the failure-intensity, "time" is the exposure time of software to use, or to testing. It can be the CPU time, the calendar time, the expended testing or debugging time (or effort), and even the number of test cases run.

For example, we distinguish time-oriented failure-intensity (that is, failures per unit time), and test-case intensity (that is, number of failures per test case). In the case of operational testing, an excellent exposure metric is the CPU execution time [Mus87, Jon91, Cra92], but this may not be the case in a non-operational testing environment.

In addition to the instantaneous (classical) failure-intensity, we have found it useful to compute the following three failure-intensity metrics: the average failure-intensity expressed in terms of failures per unit time (total failures over total time); "final failure-intensity", which is the failure-intensity expressed in terms of failures per unit time averaged over the last 10% to 20% of the reported testing; and "test-case failure-intensity", which is average failure-intensity expressed in terms of failures per test case, or the number of unique failures per unique test case.

2.2 Drivers

A number of factors may influence the observed field quality of software. For example, the quality of the verification, validation and testing efforts, the frequency of usage of the component, the size of the changes made in the software, and the number and location of the residual defects.

In order to relate the drivers and the estimators, it is necessary to find either a direct analytical, or a tabulated, relationship which connects many different, possibly continuous, levels of conditions (or causes) with the final effect (reliability). The approach where the input conditions are discrete states requires determination of threshold values for these variables (or metrics). When an input metric exceeds (or is below, depending on the metric) a threshold, transition to the next state of quality may take place.

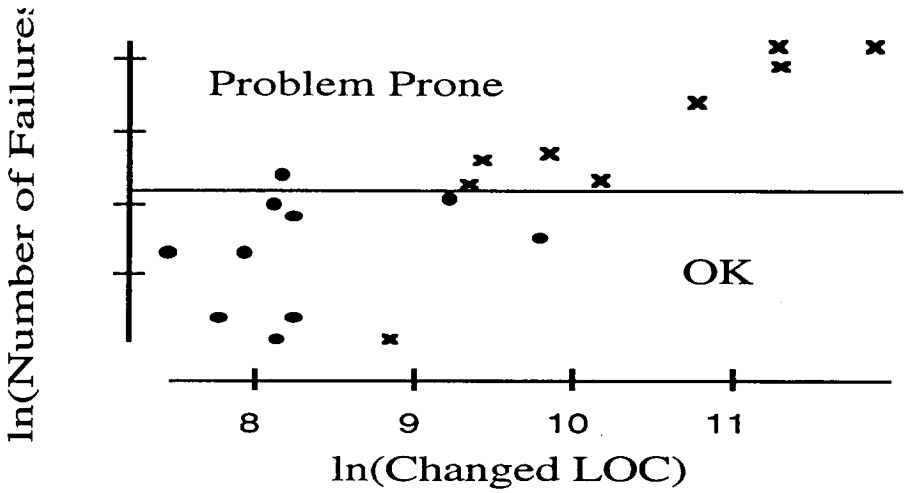

Figure 2.1 The number of reported software failures may correlate with the change experienced by the software.

2.2.1 Change Level

This metric may indicate the number of faults introduced, or activated, by the changes in the software. A larger change in the software might be expected to result in a proportionally larger number of operational failures. Figure 2.1 illustrates the relationship that might be observed between early operational failures (over the same operational period) and the extent of the changes that occurred in a component. The changes can be expressed as the number of added and modified lines of code, or similar. The relationship shown, although undesirable, is not unexpected. A detailed causal analysis of the observed problems may be needed to identify precisely the development phases where the problems have originated, and where they could have been, but were not, detected.

2.2.2 Usage

The more often the product is used, the more likely it is that defects will be found in it. A full implementation of SRE requires determination of operational profile(s) [Mus93], and analysis of observed problems in that context. For example, it should be established whether the cases showing large numbers of reported problems owe that to very frequent usage of a component that has an average residual fault density, or to an excessive residual fault density in a product that is being used at the rate typical for most other products.

Definition and use of the appropriate operational profile(s) is essential for accurate evaluation of the testing process and its effects. Lacking actual operational usage information, it may be possible to estimate it. One possibility is to use the **dynamic** operational deployment information and figures [Mus93]. Another, less accurate way, is to **statically** analyze the product component call graphs. The call graph is a tree-like structure which describes the interactions between different product components and their hierarchy [Mus93]. In that context we consider two metrics. The component "Importance" and the component "Call Usage".

The "Importance" metric attempts to indicate the importance, and, indirectly, possible usage frequency of a component (or function). The "level 1", or root, components are most important and are likely to be most frequently used because all components above them use them to some degree. The set of components that connect directly to the this level are "level 2" components. The level of a component can be computed by counting the number of graph edges that exist between the node (component) of interest and "level 1", and adding 1 to that count. An alternative is to count the number of nodes between the "level 1" component set (root) and the higher level component. If there are two or more paths between the root and the desired node, the average is taken. The "Importance" is related to usage through a transformation function. For example, assuming that seven levels have been defined (level = 1 to 7), the function (8-level) implies that "level 1" components are most frequently used, and that usage reduces linearly with the distance from the system root.

A more accurate description of the static usage frequency might be the count of the total number of components that **use** a component. With that in mind we constructed the metric we named "Call Usage". This metric is "1 plus the total number of distinct siblings that can call a given component on the system call-tree". The 1 is added to account for the execution (self-use) of the component itself.

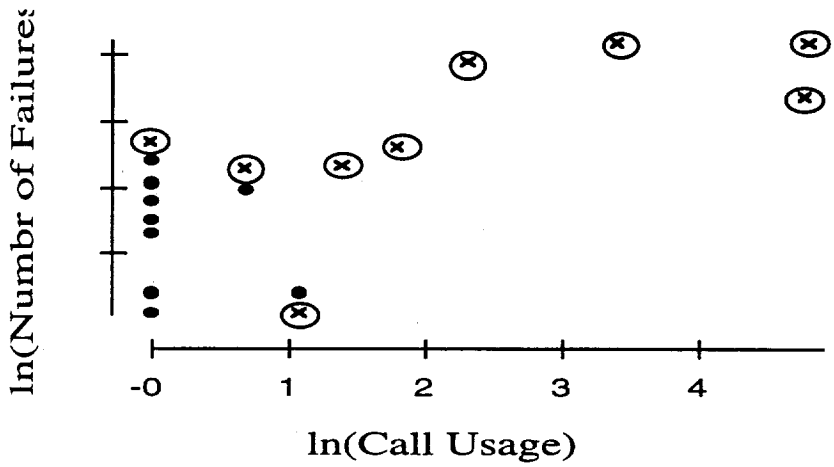

Figure 2.2 The number of operational failures may correlate with the total number of product call-tree siblings.

Figure 2.2 illustrates a possible relationship between the number of failures, and the usage level. The entries marked with a large circle illustrate components that may have experienced high levels of change (for example, over 10,000 lines of code). The horizontal axis is the logarithm of "Call Usage", and the vertical axis is the logarithm of the total number of field problems reported during, for example, an early operational period lasting X units of time. We see that, in this illustration, high "Call Usage" components show increased incidence of problem reports, and may have experienced a higher change level.

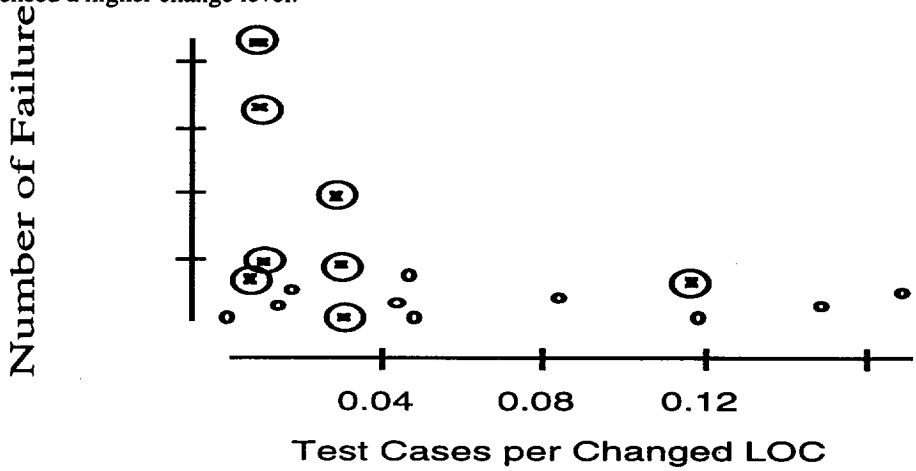

Figure 2.3 Possible relationship between the number of failures and the "Testing Effort Coverage".

The assumption, made above, that each sibling contributes equally to the load on the root node is not likely to be entirely true in practice, and this may introduce distortions into parameter estimates. The contribution may vary depending on the sibling, platform, deployment site, customer, user, etc.

2.2.3 Effort

The usage information, although quite helpful, may be incomplete and not adequate as far as prediction of the problem-proneness of components is concerned. For example, usage usually does not reflect prior knowledge about the component evolution, such as its change level, or verification and testing history.

A driver that attempts to combine two historical variables is the "Testing Effort Coverage". It is meant to be an indicator of the effort invested into testing. The larger the effort, the smaller would be the residual number of faults one would expect. The general metrics is the "number of unique test cases per unit of change". The particular metric may be the "number of unique test cases per changed line of code", or the "number or test cases per decision node". The assumption is that the testers select the test cases based on some internal importance criteria related to the operations a component performs, and that they attempt to "cover" as many of the elements (or functions), that make up the component, as possible. The metric implies that there may be a threshold which would indicate the effectiveness, or the quality, of testing and, perhaps, guarantee a level of freedom from errors of a certain type.

A possible relationships is shown in Figure 2.3. We see that an analysis of this type can yield target values for the "Testing Effort Coverage". For instance, components that have not received at least one test case per 10 to 20

lines of changed code (0.1 to 0.05 test cases per line of code) may show problems in the later phases. In our experience, the false alarm rates for this type of metric alone may be of the order of 30-40%. Although, the "Testing Effort Coverage" appears to be an indicator of the problem-potential of a component, its false alarm rate may be unacceptably high, and, for practical use, it may need to be combined with other metrics.

2.2.4 Failure-intensity

Failure-intensity can also serve as a driver. However, our experience is that in the context of non-operational testing, classical time-based failure-intensity appears to have a very limited meaning. To be useful, the failure-intensity must be associated with the particular testing strategy and effort used, the coverage of the product operations and functionalities, and, if possible, it must be corrected for the distortions induced by the non-operational nature of the testing profile.

Part of the problem may stem from the fact that, during component and integration testing, different components of a software system may execute at different speeds (for example, they may run on different processors). This may mean that different test cases may use different amounts of execution time, and, unless corrections are applied, a typical integration test-suite for one component may accumulate considerably more time per test case, than a similar test-suite for another component of the system. Normalization of the failure-intensities across different components may be necessary [Mus87]. One possibility is to use instantaneous or average "test-case failure-intensity", which is expressed in terms of failures per test case, or the number of unique failures per unique test case.

3. Risk Modeling

This section addresses the use of early testing information in identification of components that may be problem-prone in the field. Several approaches have been proposed [Kho90, Mun92, Bri93]. Highly correlated nature of the early software verification and testing events may require the use of a more sophisticated, time-series, approach [Sin92]. We illustrate some of the issues through a risk model [Ehr85, Boe89].

3.1 Process States

At the end of a non-operational testing phase an indication of the state of the software may be available as a set of conditions. In the most general form, the conditions will be compound, and will reflect the evolution process and history of software. The conditions will derive from quantification of a number of factors that may influence the operational quality of software. For example, the quality of the verification, validation and testing efforts, the operational usage profile, the size of the changes, and the number and location of the residual defects.

What is of interest here is the likelihood that the current state will lead to an unsatisfactory final state. For example, a state where the component exhibits high failure rate during its operational use. A specific state may be defined by providing a set of propositions (e.g., conditions that are true) that describe the achieved characteristics of the software, and its progress through the development process. For example,

$$P_i(S_f \mid M_1 > m_1,\ M_2 > m_2, M_3 = m_3,\ \ldots M_n \geq m_n)$$

might indicate the **conditional** probability that state, S_i, will lead to failure (for example, state S_f which is described by the condition: "initial operational failure intensity is **more than** 0.000006 failures per minute"). This transition probability is conditioned on the truth of propositions for M_1 through M_n. These propositions capture the development history, the process, and, possibly, the operating environment. They indicate the conditions that have been met during the verification and testing process. Some of the metrics derive from the past, some from the current, and some from the expected **future** phases. For example, M_3 may be "Coding Inspection Effort Intensity" [Chr90], M_2 may be the current "Testing Effort Coverage", and M_1 may be the "Call Usage". Of course other drivers should also be considered.

Each identified state should be evaluated for the probability that it occurs, P(S), and for its capability to indicate (influence) what the future state of the process could be (for example, by its a posteriori probability, $P_i(S_f \mid S)$). This information can be used to define risk models.

When defining states and computing probabilities we need to account for the fact that many conditions are not mutually independent by considering their joint probabilities. We illustrate the issues using the following conditions (events):

i) A component has experienced **high level of change**.
ii) A component has **high usage level.**
iii) A component has **not been "covered"** with sufficient number of test cases.
iv) A component exhibits **high test-case failure-intensity.**

v) A component has **not been "covered"** with sufficient number of test cases **and** it exhibits **high test-case failure-intensity**.

Table 3.1 A comparison of the indicator metrics with respect to their capability to identify problem-prone products

Input Variable		Change Level	Importance Level (level)	Test Cases per Changed Line of Code TC/ULOC	Unique Failures per Unique Test Case UF/UTC	UF/UTC **and** TC/ULOC	(8-level) * (UF/UTC) **and** TC/ULOC
Number of Components	**N**	19	19	17	19	17	17
Total number that is considered Problem-Prone (i.e., in state S_f)	**NP** $P(S_f) \cong \frac{NP}{N}$	9	9	9	9	9	9
Condition (C)		**> 10,000** LOC	**≤ 3**	**$\leq \frac{2}{25}$** (≤0.08)	**$> \frac{1}{25}$** (> 0.04)	UF/UTC **> 0.04 and** TC/UPS **< 0.08**	(8-level)* (UF/UTC) **> 0.15 and** TC/UPS **< 0.08**
Number of Components Satisfying the Condition	**NC** $P(C) \cong \frac{NC}{N}$	9	9	13	16	12	11
Number of Problem-Prone Components Satisfying the Condition	**NPC**	8	7	9	9	9	9
Number of Problem-Prone Components **NOT** Identified	**NP-NPC**	1	2	0	0	0	0
Number of Components **MIS**-Identified as Problem Prone **(False Alarms)**	**NC-NPC**	1	2	4	7	3	2
Problem Recognition Ratio	$P(C \mid S_f) \cong \frac{NPC}{NP}$	8/9	7/9	9/9	9/9	9/9	9/9
Conditional Problem Identification Ratio	$P(S_f \mid C) \cong \frac{NPC}{NC}$	8/9	7/9	9/13	9/16	9/12	9/11

vi) A component has **not been "covered"** with sufficient number of test cases **and** it the product of its "Importance" and test-case failure-intensity exceed a threshold. Note that the test-case failure-intensity is adjusted for usage of the component by multiplying it by the (8-level). Seven levels are defined, and it is assumed that "level 1" components will be most frequently used so that the failure intensity observed during non-operational testing will be magnified in operation by factor (8-level).

An example is given in Table 3.1. The table contains columns for individual, as well as joint events. The conditions and frequencies would be derived from experimental (historical) information. For example, we see that the "Change Level" and "Importance Level" metrics can recognize 8 and 7 of the 9 problem-prone components, respectively. The "Change Level" has a false alarm rate[2] of 1/9, while "Importance Level" has a false alarm rate of 2/9. The other two individual conditions recognize all problem-prone components, but also exhibit higher false alarm rates. Joint events show improved false alarm rate, as well as better capability to identify problem prone components (NPC/NC). In general, we want the two ratios listed at the end of Table 3.1 to be as close to 1 as possible. We also want the false alarm rate to be as close to zero as possible.

Table 3.2 Example Risk Models

	Model
A	**if** ('test cases per line of code'< 0.08) and ('failed test cases per test case' > 0.04) then 2 **else if** ('failed test cases per test case' > 0.04) or ('test cases per line of code' < 0.08) then 1 **else** 0
B	**if** ((('test cases per line of code') < 0.08) and ('(8-level)*(failed test cases per test case)') > 0.15) then 9/11 **else if** ('(8-level)*(failed test cases per test case)') > 0.1 then 9/14 **else if** ('test cases per line of code') < 0.08 then 9/13 **else** 0

Any quantitative estimation should be given as an interval rather than as a point estimate (for example, upper and lower 95% confidence bounds).

3.2 Simple Model

We define risk as the probability that an undesirable event takes place and causes an operational loss, multiplied by the magnitude of the loss it causes [Ehr85, Boe89]. We consider the risk given that we know what the current state of software is. Let the undesirable event be the problem-proneness of the software in the field due to one or more categories of faults (for example, the event is transition to state S_f described earlier). The loss, L_f, can be the severity of the resultant class of failures, or any other appropriate measure. Then, given state i, the risk is

$$R(S_f) = P_i(S_f \mid \ldots.)*L_f$$

If the appropriate information is available, the analysis may be broken down into N failure classes that contribute to the overall problem, that is

$$R(S_f) = \sum_{j=1}^{N} P_{ij}(S_{fj} \mid \ldots)*L_{fj}$$

The computed risk value can be used directly, or it can be weighted to reflect some other concerns.

Table 3.2 two summarizes very simple risk models. The score for model 'A' is based on the count of condition events. The model 'B' score is an estimate of the a posteriori probability.

Figures 3.1 and 3.2 show examples of component clustering that might be offered by these risk models. We see that, using the chosen target values, the risk models successfully identify problem-prone products, but also generate some false alarms (items in the rightmost column and below the threshold line).

4. Software Process Control

Advanced software reliability engineering requires active guidance of the testing process based on the quality growth. Many different reliability indicators can be used to establish test stopping criteria, and guide the testing strategy [Dal90, EhP93]. But, many available reliability models are not well suited for evaluation of systems under

---

[2] false alarm rate = $\frac{NC-NPC}{NC}$

test with other than their operational profiles. The formulation of software reliability models, the estimation of their parameters, and the accuracy of their predictions are somewhat controversial issues [Bro92]. Common estimation procedures include maximum likelihood, least-squares and Bayesian approaches [Mus87]. However, the highly correlated nature of the early testing process may require the use of advanced "time-series" analyses to evaluate the reliability growth.

This section briefly examines the role of reliability growth models in the context of multi-phase software process control.

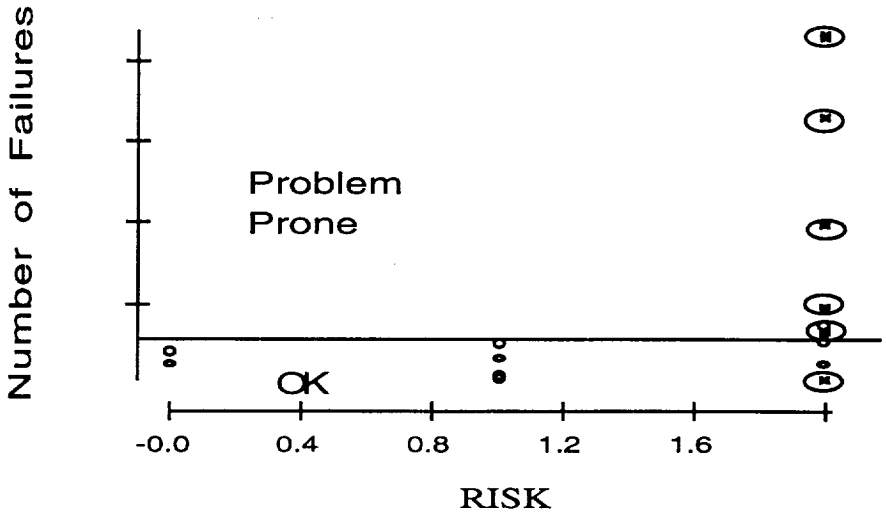

Figure 3.1 The capability of risk model "A" to identify problem-prone components

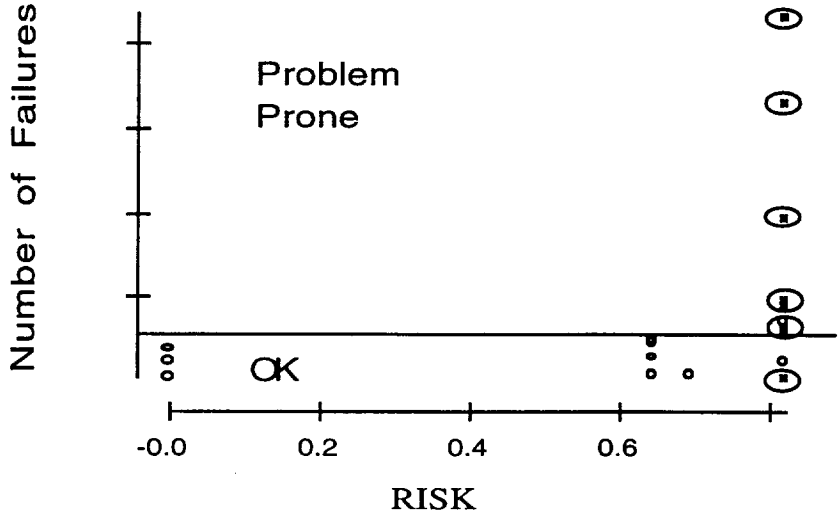

Figure 3.2 The capability of risk model "B" to identify problem-prone components

4.1 Exposure "Time"

The choice of the "exposure metric" is important. Time is the usual measure. An alternative may be the count of the executed test cases. The underlying assumption is that non-operational testing concentrates on low-level software "operations" which may be better represented by test cases than by the execution time. This metric may be easier to normalize since the number of planned test cases is usually available early in the testing process. The number of unique **test cases** successfully executed may be combined with the information on the "planned" number of test cases to obtain the test-case "coverage" in terms of the fraction of the planned test cases executed (see the x-axis in Figure 4.2).

4.2 Non-Operational Testing

If we assume that early software testing is primarily driven by the desire to cover and verify as many product operations, functionalities, and structures as possible, irrespective of how often they might be used in the field, we can develop "coverage"-driven reliability growth models that operate within the confines of a single testing phase.

The essentials of one such a model, described as a Rayleigh intensity model, are given in [Vou92]. The model has a unimodal intensity profile, and an S-shaped cumulative distribution function. In a more general case, the dynamics of the process translates into a Weibull failure detection model. Weibull-type model, using time as exposure, was considered by Wagoner [Wag73], but not in the context of non-operational profile testing. Also, the Shick-Wolverton model can be interpreted as a special case of the Weibull model class [Shi73, Mus87]. A number of other S-shape models exist [Yam83, Ohb84, Yam86, Toh89].

The unimodal failure-intensity profile, frequently observed during non-operational testing, is in sharp contrast with the monotonously decaying failure-intensity expected from the "classical" reliability models used with operational profile testing. Nevertheless, it is reasonable to expect that, before the testing is stopped, an overall decay in the failure-intensity needs to be observed. Of course, statistical variability of a single sample, that each individual testing effort represents, may mean that, in practice, a number of minor modes may be observed in the actual intensity profile.

A growth model can be used to fit and predict failure-intensity during non-operational testing. Figure 4.1 illustrates this. The exposure metric used is the execution time in minutes, and the failure-intensity is in terms of failures per hour. Shown are the observed and calculated (instantaneous) intensity, and the experimental and fitted average failure-intensity. We see that in both cases, and particularly in the latter one, the Weibull model appears to describe the empirical data well.

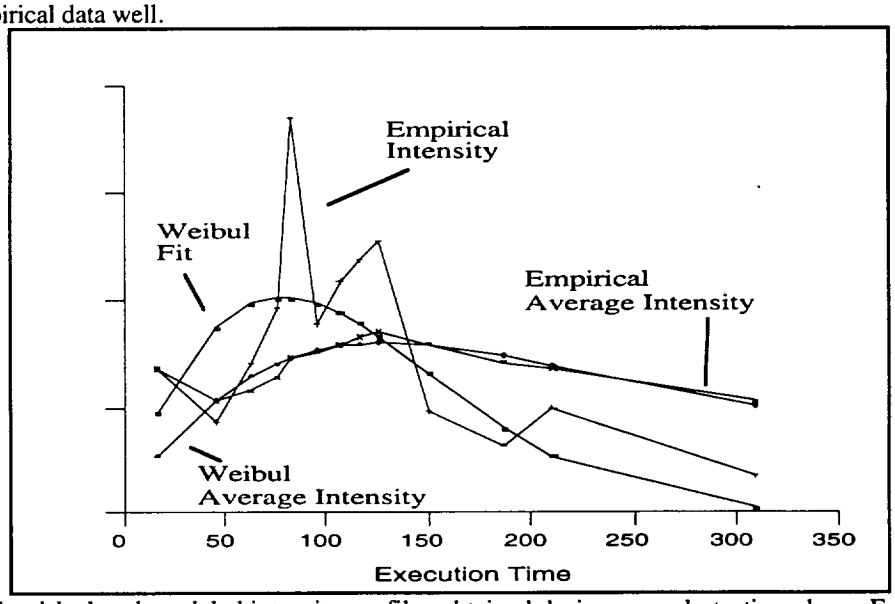

Figure 4.1 Empirical and modeled intensity profiles obtained during an early testing phase. Exposure is the cumulative test case execution time.

4.3. Process Control

The importance of reliability growth modeling is in the potential use of the **predicted** parameter, and derived variable, estimates in establishing end-of-phase quality conditions which, in turn, can be used in multi-phase models. To be useful in software process control the feedback, about the potential impact of the current testing (or lack of it) on the operational quality of software, has to be available as early as possible. For example, we would like to be able to answer questions like "What are the estimated failure intensity and residual fault counts for component X at the end of this testing phase?", "Are these values within the expected bounds for this phase?", "What is the impact on the operational reliability of the product?, "How much more testing is required?", etc.

We have seen that the risk models may use test-case intensity, effort, and usage to identify error-prone components. It is interesting to note that a change-level threshold may be an unstable condition, because improvements in the process are intended to destroy the correlation between the size of the change and the problem-proneness. If the parameters required by the risk model, for example the total expected number of failures or the corresponding test-case failure-intensity, can be estimated before a testing phase ends, then it may be possible to use

a multi-phase model to assess the impact of the current phase on the quality of software in some future life cycle phase, and direct the effort to components that may need extra testing.

The process involves prediction of end-of-phase quality conditions that describe a software state based on the information **before** a testing phase is complete. The approach requires modeling of the failure detection process occurring during the non-operational testing, and periodic estimation of the model parameters and derived conditions. The predicted conditions are then used by a risk model to assess the quality of software in some future phase.

Figure 4.2 shows an example of a Rayleigh model fit, made when about 60% of the planned test cases have been executed. Stable estimation bounds may require as much as 50-60% of the test plan to be completed[3].

Suppose that the metric of interest is average failure-intensity, and that the condition threshold is 0.3 failures per hour. Figure 4.3. then illustrates a possible process of stabilization of the intensity estimates. It shows the empirical average test-case failure-intensity as a line, and approximate estimate bounds as bars. The estimates are made using the data available up to the point of estimation. We see that as more data becomes available the bounds tend towards the observed average failure-intensity. Inspection of the average failure-intensity graph shows a decreasing trend in the intensity beyond the 80% point.

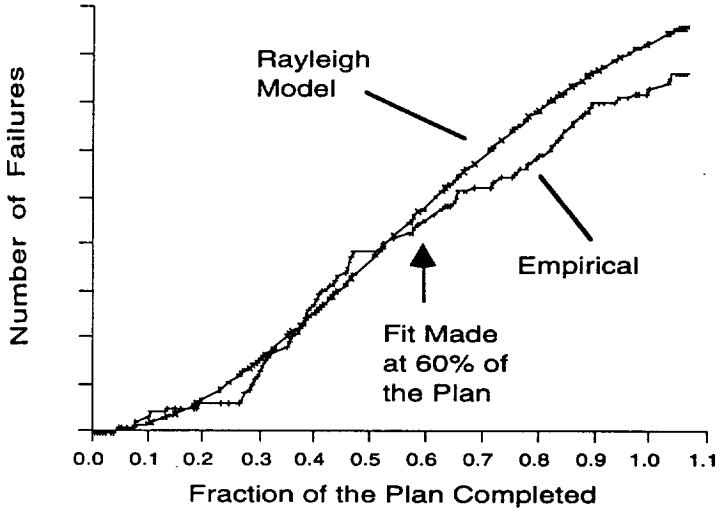

Figure 4.2 Rayleigh fit and projection at 60% of the test plan completion.

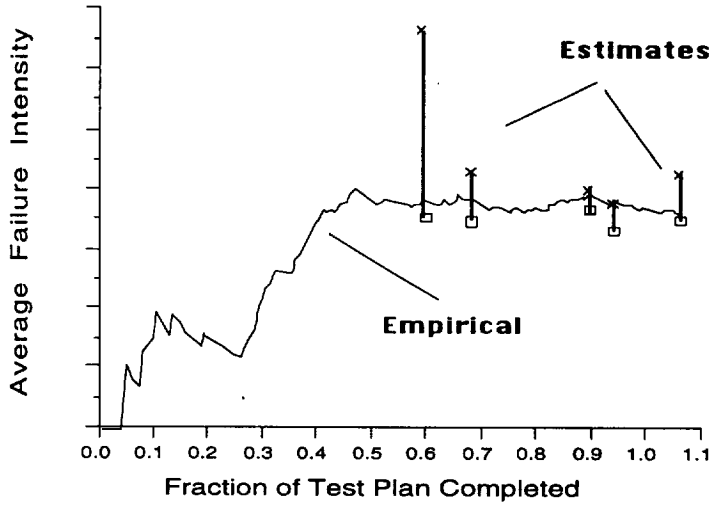

Figure 4.3 Actual and projected approximate bounds on Rayleigh average failure-intensity.

[3]Note that estimation of parameters may be achieved using least-squares or maximum likelihood. However, inference and computation of the confidence bounds may require more sophisticated techniques, such as time-series analysis..

The final step in the process is to feed the risk model assessment of the components back into the testing process.

Ideally, the reaction to this information would be quick, and correction would be applied already within that testing phase. However, in reality, introduction of an appropriate feedback loop into the software process, and the latency of the reaction, will depend on the accuracy of the feedback models, as well as on the software engineering capabilities of the organization.

For instance, it is unlikely that organizations below the third maturity level on the SEI Capability Maturity Model scale would have processes that could react to the feedback information in less than one software release cycle. Reliable latency of less than one phase, is probably not realistic for organizations below level 4 [Pau93]. This needs to taken into account when the level and the economics of SRE implementation is considered .

Acknowledgments

The first author is grateful to Dr. W. Jones for many invaluable discussions of the software reliability engineering issues.

References

[Boe89] B.W. Boehm, *Tutorial: Software Risk Management*, IEEE CS Press, 1989.

[Bri93] L.C. Briand, W.M. Thomas and C.J. Hetsmanski, "Modeling and Managing Risk Early in Software Development," Proc. 15th ICSE, pp 55-65, 1993.

[Bro92] S. Brocklehurst and B. Littlewood, "New Ways to Get Accurate Reliability Measures," IEEE Software, pp. 34-42, July 1992.

[Chr90] D.A. Christenson, S.T. Huang, and A.J. Lamperez, "Statistical Quality Control Applied to Code Inspections," IEEE J. on Selected Areas in Communications, Vol. 8 (2), pp. 196-200, 1990.

[Cra92] Cramp R., Vouk M.A., and Jones W., "On Operational Availability of a Large Software-Based Telecommunications System," Proc. Third Intl. Symposium on Software Reliability Engineering, IEEE CS, pp. 358-366, 1992.

[EhP93] W. Ehrlich, B. Prasanna, J. Stampfel, J. Wu, "Determining the Cost of A stop-Test Decision," IEEE Software, Vol. 10 (2), pp. 33-42, March 1993

[Ehr85] W. Ehrenberger, "Statistical Testing of Real Time Software," in *Verification and Validation of Real-Time Software*, ed. W.J. Quirk, Springer-Verlag, 1985.

[Jon91] W.J. Jones, "Reliability Models for Large Software Systems in Industry," *Proceedings First International Symposium on Software Reliability Engineering*, pp. 35-42.

[Kho90] T.M. Khoshgoftaar and J.C. Munson, "Predicting Software Development Errors Using Software Complexity Metrics," IEEE J. on Selected Areas in Communications, Vol. 8 (2), pp 253-261, 1990.

[Lyu92] Lyu and A.P. Nikora, "An Empirical Approach for Software Reliability Measurement by Linear Combination Models," IEEE Software, July 1992.

[Mun92] J.C. Munson and T.M. Khoshgoftaar, "The Detection of Fault-Prone Programs, " IEEE Trans. on Software Engineering, Vol 18(5), pp 423-433, 1992

[Mus87] J. Musa, A. Iannino and K. Okumoto, Software *Reliability (Measurement, Prediction, Application)*, McGraw-Hill 1987

[Mus93] J.D. Musa, "Operational profiles in Software-Reliability Engineering," IEEE Software, Vol. 10 (2), pp. 14-32, March 1993.

[Ohb84] M. Ohba, "Software Reliability Analysis Models," IBM J. of Res. and Development, Vol. 28 (4), pp. 428-443, 1984.

[Pau93] M.C. Paulk, B. Curtis, M. B. Chrissis, and C.V. Weber, "Capability Maturity Model, Version 1.1," IEEE Software, pp. 18-27, July 1993.

[Shi73] G.J. Shick and R.W. Wolverton, "Assessment of Software Reliability," Proc. Operations Research, Physical-Verlag, Wurzburg-Wien, pp. 395-422, 1973.

[Sin92] N.D. Singpurwalla and R. Soyer, "Nonhomogenous auto-regressive process for tracking (software) reliability growth, and their Bayesian analysis," J. of the Royal Statistical Society, B 54, 145-156, 1992.

[Toh89] Y. Tohma, R. Jacoby, Y. Murata, and M. Yamamoto, "Hyper-Geometric Distribution Model to Estimate the Number of Residual Software Faults," Proc. COMPSAC 89, IEEE CS Press, pp. 610-617, 1989.

[Vou92] Vouk M.A., "Using Reliability Models During Testing with Non-Operational Profiles," Proc. Second Workshop on Issues in Software Reliability Estimation, October 12-13, 1992, Bellcore, Livingston, N.J.

[Wag73] W.L. Wagoner, "The Final Report on a Software Reliability Measurement Study," Aerospace Corporation, Report TOR-0074(41112)-1, 1973.

[Yam83] S. Yamada, M. Ohba, and S. Osaki, "S-Shaped Reliability Growth Modeling for Software Error Detection," IEEE Tran. on Reliability, Vol. R-32 (5), pp. 475-478, 1983.

[Yam86] S. Yamada, H. Ohtera, and H. Narihisa, "A Testing-Effort Dependent Reliability Model for Computer Programs," The Trans. of the IECE of Japan, Vol. E 69 (11), pp 1217-1224, 1986.

Appendix II – Design and Review of Software Controlled Safety-Related Systems: The NCSU Experience With The Generic Problem Exercise*

Mladen A. Vouk and Amit Paradkar
Department of Computer Science, Box 8206
North Carolina State University, Raleigh, N.C. 27695
Tel: 919-515-7886, Fax: (1)-(919)-515-6497 or 7896
e-mail: vouk@adm.csc.ncsu.edu
e-mail: amit@bvcd.csc.ncsu.edu

Abstract

The Software Engineering Laboratory of the Department of Computer Science at North Carolina State University (Raleigh, NC) developed several prototypes of the GPE Boiler Control and Monitoring program. The prototype design favors safety over long mean-time_to_shutdown. The software process model adopted for the principal deliverable was a risk-sensitive variant of the (iterative) evolutionary prototyping model. We used a finite-state machine view of the problem and the object-oriented analysis (OOA) and documentation in combination with extensive specification, design and code reviews and state-based testing. We used the objects to form localized (decoupled) manageable finite-state machines and reduce the state coupling that may result in the state-explosion problem. We discuss some of the detected faults. Additional verification, testing and analysis work is in progress using cause-effect graphing and condition-based coverage analysis and testing. Results of these additional experiments are reported elsewhere. The results indicate that a standard to guide writing code for critical applications that is meant to be read may need to be considered. Our experience indicates that, for the problems that can be translated into solutions that do not exceed about 5000 lines of high-level language code, the optimal specification may, in fact, be the solution itself. This suggests a development paradigm for critical software applications in which the system would be developed and tested before trying to verify it using formal methods. This would remove the errors that can be found more efficiently through testing, and the tested program would then serve as the basis for development of one or more abstractions that could be verified against the specification and then re-tested to "close the gap" between the abstract system and the actual one.

1. Introduction

The Software Engineering Laboratory (SEL) of the Department of Computer Science at North Carolina State University (NCSU SEL, Raleigh, NC) participated in the Generic Problem Exercise (GPE) organized by the International Invitational Workshop on the Design and Review of Software Controlled Safety-Related Systems. The workshop organization was spearheaded by the Institute for Risk Research (IRR), University of Waterloo, Waterloo, Ontario, Canada. Following is the brief description of the development approach and the results obtained by the NCSU SEL.

---

* This work was supported in part by the NASA grant NAG-1-983, and the NSF grant CCR 8907807.

The principal purpose of the GPE was to explore the current issues related to the process of development and acceptance of software for safety-critical applications. The approach was to develop and evaluate multiple versions of an abstracted real-time control system. The specific problem was a boiler control system.

The history of the development effort, the participating NCSU personnel, and the developed versions are discussed in Section 2. Section 3 describes the process and the methods used to analyze and develop the GPE solution. Section 4 discusses some of the details related to the inputs to the development experience, including a critique of the original requirements specifications, and of the experimental environment and process. Section 5 discusses the outputs of the effort such as the design strategy (e.g., design for safety issues), product safety profile, and the auditability of the solution. The summary and the conclusions are given in Section 6.

2. Project

2.1 Goals

The solution of the GPE was undertaken with the following four main goals in mind

i) To design software that meets the provided customer specifications;
ii) To design, document, and write the required software for auditable safety;
iii) To explore analysis, design and testing issues related to the development of small to medium sized safety-critical software;
iv) To further explore issues related to multi-version software experiments.

The first two goals are dictated by the GPE experimental coordinators as evaluation criteria. The other two goals are of direct interest to NCSU SEL research directions, and they guided the development and verification approaches we selected. In a sense the goals i) and ii) are contradictory since the need for detailed specification and design analysis, particularly formal analysis, competes with the need to deliver, by a given date, an operational version of the GPE solution that can be tested using a simulator ("proof of the pudding").

The other two goals are within the scope of the NCSU SEL research plan. The following issues are being investigated in the context of the GPE in particular, but with an eye on the broader issues of multi-version experiments, and of the development of small to medium sized safety-related wide distribution commercial applications (e.g., software-based household appliances, medical devices):

a) state-based analysis, design and testing methods and tools,

b) BOR-SD based specification analysis, and BOR coverage testing (see section 3.2.2)
c) practical use of object-oriented analysis, design and coding
d) state-based testing of objects
e) usability of a risk-reduction software process models
f) software process modeling (definition of a software process risk object)

2.2 History

The first version of the GPE specifications was received in April 1992 (Revision 1, 20-Jan-92). NCSU SEL participation started in June 1992.

The first stage of the project was a thorough review of the first version of the specification document distributed by IRR and the development of the "local", NCSU, specifications (Software Requirements Specification document, SRS). The review process yielded a number of questions related to the specification and addressed to the GPE coordinators. Some of the encountered problems were identified as specification defects and GPE coordinators issued a specification update. An update was received in October 1992 (Amend 1 to Rev 1, 21-October-92).

In the second stage of the project, the updated specification was used to develop two operational prototypes of the boiler control program (BCP), several partial prototypes, and two prototypes of the boiler simulator for testing the control program over an RS232 interface on IBM PC compatible platforms running MS-DOS 5.0.

The third stage of the project involved the use of the information collected in the previous stages to develop release versions of the product prototype and to test them using designed test cases, random test cases, and two new simulators. One of the simulators was provided by the GPE coordinators, the other was developed by the NCSU SEL.

The NCSU GPE solution (version 2.0), and the software development documentation were submitted on May 20, 1993. However, the work continues on the analysis of the solution, and the development of an improved solution and additional tests (e.g., cause-effect and condition-based specification derived test development [Tai93a,b]). This continuing effort is expected to eventually result in the Version 3.0 prototype.

2.3 Personnel

The released versions of the NCSU BCP were developed by a three-person project team: A. Paradkar, I. Shields, and J. Waters. All three have industrial software development experience,

all three are NCSU graduate students, all three have graduate training in software engineering and software process risk management, and all have been selected because their areas of study and expertise would contribute to the goals of the project. Paradkar's area is software fault-tolerance [Vou90a]. As the chief architect of the "golden"-version used in the four-university NASA-sponsored multi-version experiments [Kel88], [Vou90b], [Eck91], [Vou93], he is the only of the three with some (academic) experience in the area of development of safety-critical software. Shields has practical real-life experience with the development of communication software based on finite-state machines. His research interests lie in the area of state-based testing particularly where it may apply to the testing of state-based objects. Waters is doing research in the area of software process modeling. The team that is involved in the post-release verification and testing of Version 2.0 prototype consists of A. Paradkar, M.A. Vouk (NCSU Computer Science Department faculty member), and K.C. Tai (NCSU Computer Science Department faculty member). Also participating in the research is a two person REU[4] team. The initial stages of the project also saw the participation of several auxiliary teams drawn from the graduate and undergraduate classes[5].

2.4 Versions

Table 1. NCSU prototypes

Prototype	Start/Finish Date	Size	Faults Reported After Release	Comment
A.x	Sep-92/Dec-92			Team A (3 NCSU full-time graduate students). Several partial exploratory prototypes some written in Smalltalk)
B	Sep-92/Dec-92	1790 lines of C	8	Team B (Four full time NCSU graduate students, part of CSC510 class project). Developed BCP and a simulator. Prototype B was tested by Team F.
C	Sep-92/Dec-92			Team C (Three MS students from an RTP industrial site taking CSC510 class, class project) Completed Only the Design.
D	Sep-92/Dec-92			Team D (Four MS students from a NC industrial site taking CSC510 class, class project). Completed Only the Design.
E	Pending			Team E (Four MS students from a Raleigh industrial site taking CSC510 class, class project).

4 REU = Research For Undergraduates supported in part through NSF grant number CCR 8907807. The two student REU team consists of: Sharon E. King and Evangeline K. Burgess.

5 The team that developed the only fully operational prototype that was delivered as part of the project in the graduate level software engineering class consisted of: Lauren L. Cullicut, William C. Royal, Susan L. Rundbaken, and Mark E. Snesrud.

F	Jan-93/May-93	about 2,500 lines of C		Team F (14 full time NCSU undergraduate students divided into tools, design and testing sub-teams worked on the testing, correction and extension of Prototype B as part of the CSC472 senior software engineering maintenance class project). They developed enhanced prototypes of the BCP and the simulator.
Version 1.0	January 1993 to May 12, 1993	about 4390 lines of C	4 (critical)	Team A (3 NCSU full-time graduate students). Designed and wrote both BCP and a simulator. Also used GPE provided simulator.
Version 2.0	May 12, 1993 to May 19, 1993	4495 lines of C	by 6/4/93: 1 (non-critical)	Team A (3 NCSU full-time graduate students). Additional testing and correction of BCP and a simulator. Also used GPE provided simulator.
Version 3.0	May 19, 1993 to ?			Team A+ (3 NCSU full-time graduate students) plus PI's. Extensive additional testing of BCP, including code and function coverage information.

Table 1 summarizes the prototype population developed at NCSU. The prototypes A trough F were developed in parallel and independently. The only common link was one of the Team A members (Paradkar) who was involved in the oversight and grading of the projects that resulted in prototypes B through F. This provided diverse and useful input to the Team A effort. Team F served as the independent evaluation team for prototype B. Prototype F was not used in the current study.

3. Process and Methods

3.1 Software Process Selection

The GPE was intended to simulate the development of a class of critical software-controlled systems. Some of the process drivers that were crucial to the selection of our process model were:

i) requirement for high system safety and robustness;
ii) qualifications of the available personnel, i.e., development teams that did not have previous experience with safety-critical software, all of them were students, only those in the principal team had formal training in risk management and finite state machines, most of the students did not have prior industrial work experiences;
iii) delivery schedule;
iv) lack of resident expertise in boiler engineering;

v) lack of initial understanding of the customer specifications;
vi) general frailty of the abstraction (poor communication links between the controller and the boiler, lack of sufficient redundancy in the boiler elements, lack of information on the failure rates of the components, lack of operational profile information, etc.);
vii) small size of the problem (initial code estimates ranged from 1500 to 2000 lines of code);
viii) lack of a readily available test-bed (simulator);

Because the system safety and robustness were of prime importance, but the system and the solution risks were not well understood, it was decided to use the process of an (iterative) evolutionary prototyping model which would be constrained by the time-schedule and the available personnel skills. The model did not attempt to simulate an actual industrial software development paradigm. Instead it implements a paradigm that has three phases, an exploratory phase (problem formulation and scope), a general problem solution phase, and an iterative solution refinement and validation phase. The first part includes considerable interaction between the engineering team and as many sources of information as possible in order to focus the problem and outline the most promising solution(s). The second phase is the development and implementation of the first detailed operational solution. The last phase involves iterative refinement of this solution to the limits dictated by the constraints (in this case quality of the customer specifications and the time-schedule), and validation of the solution against a real environment.

The central issue was a thorough analysis of the problem, exploratory probing of all possible effects of the different (possible) solutions, and gaining of a very thorough understanding of the problem and of the impacts of the solution. Because it is often the unorthodox combinations of conditions and inputs that cause safety problems, an in-depth, but also creative, approach to the investigation of the problem is called for. This, of course, invites use of formal methods for problem specification and analysis. However, the actual methodology is only an aid to the organization of the thoughts and the understanding and the recording of the problem and the solution.

The **Phase I** of our process was the **"risk reduction"** phase during which two working, and a number of partial prototypes, were specified, designed and implemented. The critical outputs from this phase were the following documents:

i) a semi-formal developer software specification document (English mixed with diagrammatic descriptions),
ii) a more formal architectural design document containing thorough, state-based, descriptions the problem and the overall solution strategy ,

iii) a draft user manual,
iv) verification and validation plan and deliverables (including a safety analysis), and
v) project plan and log (including risk management)

The Phase I analyses convinced us that the construction of a production-type version of the program was not feasible since there was not enough information (and resident expertise) about the basic boiler system and its operational environment. This decided us to reduce the final deliverable to a prototype, and to employ an evolutionary (incremental) approach to do that. The information collected during Phase I served as the input into Phase II.

During **Phase II** the functionality of the controller was developed in a series of small steps until the first version of a full operational prototype was constructed. Each step was limited to the implementation of an object of complexity and size commensurate with capabilities of a single programmer. That is, while the initial risk reduction phase involved a larger number of people, the second phase was essentially a one-person effort. There are three principal reasons for this: i) small problem size, ii) relatively high problem complexity, and iii) availability of experienced personnel. Once the specification and the design issues were determined, to a large extent through a series of verification and validation meetings, it was deemed more efficient, and safer, to have a single person with extensive in-depth knowledge of the problem (as well as previous experience on similar projects) work on the detailed design, implementation and unit testing. This phase yielded the first release prototype of the boiler control program.

The last phase, **Phase III**, is an **iterative refinement and validation**. It is still in progress. The phase delivers a series of prototypes each of which has been tested to a higher set of standards than its predecessor, and each one improves on the safety and reliability of the solution. Central to this phase is the gradual evolution of the BCP safety and reliability through repeated and increasingly more demanding verification of the implemented solutions through static verification (e.g., code reading, symbolic execution) as well as extensive dynamic testing. The latter requires a complete, accurate, and flexible boiler simulator and comprehensive test suites. One version of the simulator was supplied by the organizer (IRR) and one by NCSU. The use of the IRR simulator amounted to a back-to-back testing of the understanding of the specifications as implemented in the NCSU boiler control program, and the boiler behavior specifications as implemented in the simulator developed by the "customer". It involved a series of testing sessions and feedback between the NCSU and the NRC control site. The NCSU simulator was developed to overcome the simplicity and inadequacies of the customer supplied simulator. The generation of test cases is discussed in 3.2.

Special attention was paid to the auditability of all development steps, as well as all the design decisions. This is reflected through the design documentation, state-based design and coding, the detailed comments in the code, the test reports, and the general project log.

3.2 Environment, Methods, and Tools

The Boiler Control software was developed on Gateway 2000 machines (EISA, 33 MHz) running DOS 5.0 and Windows (when necessary). Most of the code was written in C (compiled using Borland C++ Version 3.0 and 3.1), although some of the prototypes were built using Smalltalk.

3.2.1 Specification

During part of the Phase I we used the SDL to enhance the development and analysis of the NCSU version of the specification. SDL is the finite state machine (FSM) based Specification and Description Language recommended by CCITT for unambiguous specification and description of the behavior of telecommunications systems [e.g. Bel91]. It was available through the SDT CASE tool set (running on a DECstation under Unix, and later on a PC). SDT is an integrated set of tools for development of real-time systems which aids the users in different phases of the development cycle and eventually allows automatic code generation in either C or Pascal [Tel90]. All SDT tools are built around SDL. Although SDL and SDT were very useful, we used the toolset only for initial analyses of the specifications and abandoned it in the Phase II of the project, in part because we had problems with the code generation, and in part because the learning curve for the language and for the tool was conflicting with the project schedule. We may return to this analysis in one of the future cycles of Phase III.

During the analysis and design stages we used object-oriented methods to form localized (decoupled) manageable finite-state machines and reduce the state coupling that may result in the state-explosion problem. We have found the object-oriented approach very valuable during the analysis and preliminary design phases, but overly cumbersome and inefficient in the implementation stages. We found more traditional implementation of the FSM easier to handle and debug.

A separate attempt to use Millner's Calculus of Communicating Systems and the Concurrency Workbench [Cle90] to verify all or parts of the system requirements design, and possibly implementation was delayed for lack of resources.

The NCSU specification and design is currently limited to the use of the English language and a combination of data and control flow graphs, state and event tables, and state and cause-effect graphs.

3.2.2 Risk and Safety Analysis

Failure risk analysis was based in part on the failure mode analysis supplied with the customer specification, and in part on the additional analyses of the communication links, communication protocol, characteristics of the boiler, and the analysis of the characteristics and impact of different solutions. The latter analyses are based on the state graphs and tables, cause-effect graph analysis and hierarchical condition-based specification and design analysis called BOR-SD analysis.

BOR-SD is a specification and design analysis approach that combines the cause-effect graphing with the condition analysis, at the level of software requirement and design specifications, to identify problems with the testability and completeness of the specification and the solution. The approach generates classes of testing scenarios. The BOR-SD approach derives from the condition testing strategy called **B**oolean **O**perato**R** (BOR) testing developed by Tai [Tai90, Tai93a]. The usual condition based testing focuses on the program predicates. Its advantage is that it limits test case generation to cases sensitive to Boolean and relational operator errors in the predicates, and that the number of such test cases is linear with the number of predicates. The specification and design oriented variant of the approach (BOR-SD stands for Boolean and Relational Operator based Specification and Design) first expresses the specification in terms of the cause-effect graphs (or the appropriate predicate calculus) and then applies the BOR principles to analyze a hierarchy of specification and design decision points [Tai93b]. This results in a set of test classes and test cases that can then be compared with the similar set derived from the implemented code. This focuses the attention on the differences, if any, **between** the specification and the implementation decision logic.

For example, BOR-SD is applied first to the specification to generate the testing scenario classes and the actual test cases. Then these test cases are used to evaluate the actually implemented code (e.g., actual or manual/inspection execution of the test cases). Since this technique compares requirement decision structure against the design and implementation decision structure it allows detection of missing logic and functionalities. For example, BOR-SD was instrumental in detection of an error of omission in the Version 2.0 prototype which was not detected by the state-based test suite.

3.2.3 Reviews

A number of reviews of the customer specification document, the NCSU SRS, and the preliminary design were conducted in the period June 1992 though December 1992. In most case the reviews were performed by all members of the team in group meetings. The meetings were held about once a week and lasted about one to two hours each. Design walkthroughs were held during January and February 1993. Again these were group efforts and lasted about 1-2 hours each once to twice a week. Code walkthroughs and inspections were individual efforts conducted about twice a week in the period March through May 1993.

3.2.4 Testing

The Boiler Control software Version 2.0 was tested using the test plan given in the Verification and Validation Plan document. Two distinct types of testing were used: unit testing and system testing. A total of about 456 test cases was generated for this version, based principally on the black-box and extremal-and-special value analysis of the system requirement, and the system states and associated transitions.

Dynamic testing was done using two test-beds. One in which the control program and the simulator share the same platform (computer), and the other where they reside on separate machines and communicate through the RS-232 serial link. The latter test-bed included two simulators. The NCSU developed simulator testing was supplemented with the Customer Supplied Boiler Simulator (CSBS). The CSBS gave us some insight into the customer's interpretation of the specifications, and despite its defects, also helped in revising the development team's perspective on some of the failure modes (especially simultaneous two component failures).

The testing was iterative. Groups of control program test cases would be run starting in the Normal or Degraded mode, after the control program was invoked and has successfully run through the SelfTest, SystemTest and Initialization modes. It was not possible to control the failure modes of steaming rate meter and Water level meter using the CSBS, and hence this functionality has not been tested using the customer simulator. The NCSU simulator was used for that part of the verification effort.

The currently ongoing testing effort is based on the BOR-SD generated test cases. Evaluation of the Version 2.0 based on this approach is not yet complete.

3.2.4.1 Test Suite Reproducibility

The test case descriptions in the Verification and Validation Plan and in the Test Log represent test case classes and usually do not have single numerical values specified. Instead what is specified is the starting boiler state and the ranges of the parameter values that cause desired transitions, including specific messages expected to be sent and received whenever required. However, because the test cases were designed to test for state transitions, it is always possible to reproduce the complete test suite from initial states without scripted test cases. The only requirement is to keep the parameter values within the prescribed partition ranges.

4. Requirements

The customer requirement specification is a well written document which greatly facilitated the experiment. Nevertheless, a number of specification problems have been identified.

There is a set of basic problems GPE shares with other GPE-type experiments conducted over the last 10 or so years [e.g., Kni86, Sco84,87, Kel88]. For example, the unavoidable (over-)simplification of the problem, the corresponding lack of the exact operational environmental and engineering information, the (un)availability of the problem-specific expertise at the development sites, etc. This tends to foster an atmosphere that "this pudding is not really being cooked for eating", at least not from an operational perspective, so there is a tendency to focus more on the general concepts and principles rather than provision of a production version of the software and an in-depth investigation of the limitations of the specific techniques that could be safely extrapolated to real-life systems. On the positive side, the simplicity of the boiler model and the control program abstraction provides a very good test-bed for proof of concept work and exploration of new ideas since it furnishes a common denominator for an affordable comparison of different methods.

We did not find any outright errors in the customer specification. But, we did identify a number of abstraction limitations, one major contradiction, and a number of ambiguities or omissions that could pose a safety hazard. One of the problems with the experiment was the long response time to specification-related queries. This prompted unilateral developer resolution of many requirement related questions that either do, or could pose, a safety hazard. The following discussion of these problems is not meant to be comprehensive, it only serves to illustrate the issues.

4.1 Abstraction Limitations

A more notable limitation of the boiler and control program abstraction is the lack of use of state-of-the art hardware solutions (e.g., assumed communication link was a 3-wire RS-232 with no DTR, no handshaking, no parity checking, no CRC, etc.). This in turn implies communication protocol problems and a less than robust data-link.

Another stumbling block is the lack of information on the reliability of the boiler hardware, and the lack of knowledge about the true state of the nature during the operation. This prevents construction of an internal reality-check model that may allow a certain measure of real consistency checking as opposed to acceptance testing[6]. For example, if a pump had supposedly failed and then had been repaired it would be better if the system (instrumentation or operator) were to supply, in addition to the message that the pump was now OK, the information on whether the reported failure was a false alarm or whether it was real. This would allow construction of a more sophisticated historical model which could be used to rank boiler components by trust, etc. Similarly, lack of information on the actual operational environment and operational usage profiles prevents construction of operational system tests and possible identification of special input states which the specification may fail to explicitly mention.

4.2 Ambiguities, Contradictions, and (C)omissions

The initial review of the first version of the specification document yielded a number of questions related to the specification and addressed to the GPE coordinators. Some of the encountered problems were identified as specification defects and GPE coordinators issued a specification update. An example of a possible ambiguity is the use of the THEN in the definition of the operating requirements (4.3.1). The intention was to describe a series of activities, but the placement of the word and the use of IF ... THEN constructs in the nearby text opened the possibility for mis-interpretation of some of the order of actions. Another clarified ambiguity arose from the English language description of the part of the communication protocol (4.8.2): "... make one transmission ..." really means "... make one continuous ... transmission". It should be noted that there was no unanimous agreement among the members of the Team A about the meaning of the queried text before it was clarified, which probably was a good litmus test for ambiguity.

---

6 An acceptance check evaluates states of nature as percieved by the control program based on the currently available information and a historical model based on the same informaion. A consistency check compares the internal view of the states of nature with the actual states of nature, this check can be current (laboratory testing) or delayed (operational testing).

More serious is the following contradiction between:

> Customer specification item B-3.1.1.3, sub-items PUMPINON, PUMPINOFF, WATFLOWON, WATFLOWOFF, WATERLEVEL, STEAMRATE: "This message will appear in each transmission. Lack of the message shall cause the program to go into the shutdown mode."

and

> Customer specification item C-1.6.2 (c): "The instrumentation system may fail to transmit anything in a given transmission period, that is there may be no messages between BOT and EOT markers, or no transmission at all. If there are **two** successive transmission failure from the instrumentation system this path shall be deemed to have failed."

The design-level solution involved allowance for a "second chance", a one transmission/reception cycle wait to confirm the failure of the link, but also institution of additional system status checks (through an internal flow model) to offset possible safety hazards that are involved in this.

An interesting specification (c)omission error is the one that led to a safe but annoying implementation bug. It was detected through the BOR-SD analysis.

> If on start-up the water level is below 40,000 lb. then the specification does not say what the action is in the case when at the same time all four pumps are stuck off. [The design-level decision was to shut the system down.] However, what arose from that is a possible implementation fault, namely: when on start-up the water level is below 40,000 lb. BCP is required to bring the level over 40,000 before the boiler can be started. But, let the pump A be **stuck OFF** when BCP needs it, then BCP chooses to start the next available pump, say X. When the water level exceeds 40,000 lb. the (redundant) requirement is to shut OFF pump X and then **turn ON** simultaneously all operational pumps (obviously A should not be included if it has not been repaired in the meantime). However, BCP Version 2.0 is missing the code that recognizes that A has already been marked as failed in the stuck OFF mode. Instead, it goes into an unintentional waiting loop which does not end until either the pump A has been declared operational or the operator interrupts the start-up.

There are a number of other places in the specifications that beg implementation of infinite waiting loops because they specify that the boiler control program is to wait for a specific message before proceeding. This can turn into an infinite wait and reduce availability of the system if the message is missing due to an external failure, for example in the instrumentation. A natural solution would be use of the time-out paradigm. This was not suggested by the specification document. The "waiting" problem is present for inputs from both the instrumentation system (e.g., BCP is expected to wait for a START message from the instrumentation system before it can proceed into the operational mode - NCSU design allows infinite wait) and the operator (e.g., on start-up operator is supposed to acknowledge display messages and explicitly request the system to continue after the display check. (NCSU design mitigates the problem with a time-out of 15 seconds).

An example of a potentially safety-critical specification omission to endorse the time-out paradigm is the following. To make use of the water level information received from the instrumentation system, BCP needs the water level calibration constant. However, according to the specification, the water level constant is not volunteered by the instrumentation system, the BCP MUST request the water level meter calibration constant from the instrumentation system before that information is given. This request does not have to occur on every transmission. But the safe way is to get the latest value at frequent intervals. Now, if the information is not forthcoming (e.g., failure in the instrumentation system) BCP could wait for ever (or work with the potentially wrong calibration constant). During that time a catastrophic failure can occur since correct water level cannot be computed without that constant. Shutdown based on the time-out paradigm would be a natural solution. Nothing was specified. The NCSU design uses a 10 second time-out (two tries at the message are allowed). Note that 5 second shutdown window is specified ONLY if it is known that the water-level is outside the allowed range.

5. Solution

When considering the solutions to the GPE the primary concern was the safety. The secondary one was the reliability, i.e. provision of the service without a shutdown as long as possible. A statistically high service reliability may mean that a decision to shutdown may be delayed at the expense of strict safety limits. Our design favors safety over long mean-time_to_shutdown.

5.1 Design for Safety

The principal safety features of the architecture we have adopted for our solution are the i) Polling, and ii) Flow Modeling.

The NCSU BCP **polls** the data-link for information and processes that information in a sequential (cyclic) fashion. This is deemed to be a safer approach than the interrupt driven multi-tasking approach. Only one BCP process is active at a time (of course the point is somewhat moot considering that BCP is running on top of the MS DOS) and interrupts are neither actively used by the NCSU BCP, nor are any system exception interrupts handled (that problem will be remedied in Version 3.0). It is interesting to note that we have identified failures of the customer-developed simulator (running under Windows) which are apparently caused by mouse interrupts.

To enhance specification based acceptance checking of the received data, and to facilitate isolation of the failures NCSU BCP implements an **boiler flow model**. The model is fed from the

available non-failed sources (components). It retains historical information on the boiler states and individual component parameter values. The information is combined into a self-consistent model of the boiler. If a source (component) fails the model operates on the last know correct information and the specification based assumptions on maximum and minimum allowable rates of changes, etc. If the model detects a contradiction with some other source an investigation is started. The model is at the center of the BCP ability to check for contradictions in the received system parameters and states. In situations where the failures prevent the model from isolating the problem shutdown is initiated.

Some examples of the design and implementation paradigms adopted to favor safety are:

- **No information is failure information.** If there is no information about the status of a component, the component is failed. For example, if a pump command is lost in transit the originating pump is tagged as failed. The design provides the second chance to verify the failure. In this case the principle in part compensates (stops error propagation) for the potential safety violation associated with the second chance contradiction decision .
- **Acceptance checking (contradiction testing).** Any kind of contradiction is tagged as a potential failure in the first cycle, checked in the second, and, if still there, a failure is declared. Acceptance checking is particularly useful with intractable, multiple component, failures that may not be explicitly (individually) identifiable with direct cross-check such as those against the individual component history. For example, when steam-rate and water-level meter readings do not contradict their individual histories, but contradict the combined internal history on same transmission.
- **Compound predicate decomposition.** Deliberate imposition of importance hierarchy in evaluation of compound OR predicates is encouraged already at the design level. For instance, the specification may list a number of causes, C_1 through C_x, that should, each on its own, result in the shutdown of the boiler. High-level programming languages tend to impose an evaluation hierarchy on such predicates in any case (or worse, like C or Pascal, the language standard may actually claim indifference or provide predicate "short-circuiting", etc.). The implementation of the predicate in the form that it may be formally specified in, e.g.

  $$\text{IF}(C_1 \textbf{ or } C_2 \textbf{ or } C_3 \textbf{ or } \ldots \textbf{ or } C_x) \ldots$$

  is a potential source of problems. The programmers may recognize this, but may implement a hierarchy of IF statements that may later prove problematic. Rather than risk that, the designers are encouraged to design a logically equivalent hierarchy in conjunction with all available safety criteria. For example,

  $$\text{IF}(C_1) \text{ Then SHUTDOWN}$$

  $$\text{IF}(C_2) \text{ Then SHUTDOWN}$$

  $$\bullet\bullet\bullet$$

  $$\text{IF}(C_x) \text{ Then SHUTDOWN}$$

  where C_1 would be the most critical condition, "water level out of bounds" (the criticality can be deduced from the specifications but it is not explicitly stated in the specifications).

This approach reduces the possibilities for making an error, as well as the testing complexity (e.g., the size of an adequate test suite is automatically reduced from 2^x to $x+1$).

- **Defensive coding.** Choice of the programming language is very important in safety-critical applications. We attempted building a prototype using Smalltalk, but gave this up once we ran into problems such as slowness of the generated code, garbage collection, etc. Given the number of pitfalls related to C, the use of that language in critical applications is a questionable choice at best. However, based on the proliferation of the C language, the speed of the code, and the availability of the compilers and tools, it is the language that is likely to be found in many critical applications (e.g., small to medium safety-critical medical applications). In the light of that, we have decided to use C but adopt a number of safe practices and see if that keeps us out of trouble. For instance, we never divide by a variable (all divisions are by pre-set constants), all loops handling arrays have fixed bounds (too keep the array indices in check), we do not make explicit pointer assignments (only during initialization), etc. This, of course, does not guarantee freedom from the C related traps (memory excursions and overwrites could come from other sources), but does provide some additional confidence.

5.2 Design for Reliability

In order to reduce the number of, possibly quite costly, false alarms and increase the mean time to shutdown we have implemented several algorithms which delay transition into the shutdown mode pending confirmation of failure. The overall philosophy is that reliability is subordinate to safety. The general form the this algorithm is: if the current cycle parameter reading is faulty or suspect, delay the decision to shutdown until next cycle in order to confirm that the inconsistency is really there. There are, of course, exceptions to this "second chance" rule. For example, when water level bounds are exceeded the shutdown is requested immediately. In some cases, the "second chance" approach is accompanied by additional safety actions to mitigate the possibility that the delay could lead to a catastrophe.

For example, the absence of some specific messages in each transmission (e.g., pump status) implies immediate shutdown. If this is the only failure, and the internal flow model indicates that a safety margin of at least 10 or more seconds exists, the "second chance" approach assumes that the data-link may be noisy and gives one more cycle to check for the presence of the message. If the second try does not succeed the shutdown is immediate.

Another example is the selective use of the run-time information that is NOT required to make immediate shutdown decisions. For instance, suppose that a pump is running and it appears to be serviceable both, according to the instrumentation data and the flow model. Then, let BCP on next cycle receive an unsolicited "pump OFF" from the instrumentation system. Since the switching off of the pump was not initiated by the BCP, and all other indicators are that the pump is serviceable,

BCP ignores the message until the flow model detects a contradiction, because it is the flow information, and not the pump information that is directly needed to make the shutdown decision.

To combat the possibility of a data-link failures BCP contains instructions for random seeding of the channel with test messages that need to be echoed by the boiler. This is far from a satisfactory solution, but was the only one available.

5.3 Auditability

To assure that the development process is auditable, we have carefully documented all project meetings, events, activities, etc. To assure that the BCP quality be auditable we have documented the software. The BCP software documentation includes the NCSU specification requirement and design, the verification and validation plan, the test log, and the user manual.

The solution is based on a finite-state machine description of the problem. The finite-state machine (FSM) concept was carefully preserved all the way into the implementation. Although this makes the actual programming more complex (or at least lengthy), this approach improves the reading and the understanding of the code since the implementation is a natural refinement of the design. Thus, the reader only needs to deal with the different levels of the FSM abstraction rather than a drastic change in the structure between the design and the implementation. In fact, we consider the code part of the detailed design specification and encourages direct reading. This approach may not be a good choice in general, but for a program that does not exceed about 5000 lines of high-level language code it is quite feasible. We have made extensive use of the BCP source code reading in the ongoing verification and testing based on BOR-SD.

5.4 Holes in the Design

In the last released prototype (Version 2.0/May 20, 1993) we have so far identified one potential non-critical bug, but no safety-critical faults with respect to the given GPE specification. We would expect that there are at least another 3 to 4 residual faults all of which could be safety-critical.

The potential sources of problems are many. Some are deliberate, such as the limited exception handling, some are not. Version 2.0 was designed to handle all declared single mode failures, but only a limited number of compound failures involving two or more components. The solution currently does not trap system exceptions such as the arithmetic overflow and underflow, illegal instruction exceptions, etc., but explicit design steps were taken to avoid occurrence of some of these exceptions. The adopted incremental approach to the growth of safety and reliability did not

require that a comprehensive test-suite be available for Version 2.0, only one that it be adequate from the standpoint of state-based testing in the sense that all single step state transitions were covered at least once. Additional verification, testing and analysis work using cause-effect graphing and condition-based coverage testing is in progress, and is expected to reveal additional defects in the solution. There are also potential safety-related problems inherent in the desire to balance the safety and reliability (e.g., the "second chance" approach). This is discussed in sections. 5.1 and 5.2.

Some other problems are beyond the control of this solver team. It is not difficult to design test cases that can punch holes in the customer specifications and in the Version 2.0 of the NCSU BCP. This includes the targeting of inherent limitations of the specified boiler (insufficient component redundancy in critical elements such as the steam rate meter, terrible communication links, etc.), the customer specification ambiguities, and the frailty of the customer supplied simulator. For example, the available measuring devices and their redundancy do not even allow detection, let alone mitigation, of compensating failures. An example of such a failure is the combination of the water level and steam rate meter failures that lets the water level meter failure manifest as a constant rate of difference per cycle in the reported level, and the simultaneous failure in the steam rate meter manifest as a constant difference.

6. Summary and Conclusions

As part of the GPE, several teams at NCSU developed multiple prototypes of the Boiler Control and Monitor Program to conform to specification provided by the organizers of the International workshop on the Design and Review of Software Controlled Safety-Related Systems.

The released prototypes were constructed to detect and handle all single-mode failures, as well as some two, three, and higher-order mode failures identified by the customer specification. The prototype design favors safety over long mean-time_to_shutdown. The software process model adopted for the principal deliverable was a risk-sensitive variant of the (iterative) evolutionary prototyping model. The finite-state machine (FSM) view of the problem and the object-oriented analysis (OOA) and documentation were combined with extensive specification, design and code reviews and state-based testing. We used the objects to form localized (decoupled) manageable finite-state machines and reduce the state coupling that may result in the state-explosion problem. In the last released prototype we have so far identified one potential non-critical bug, but no safety-critical faults with respect to the given GPE specification. However, this does not preclude the presence of undetected (residual) faults or robustness problems of which at least 3 to

4 could be safety-critical. Additional verification, testing and analysis work is in progress using cause-effect graphing and condition-based coverage analysis and testing. Results of these additional experiments are reported elsewhere [Tai93b], but are very encouraging.

We found the GPE experiment very interesting and valuable. It raised several research issues, and indicated that we may have reached the point of diminishing returns as far as multiversion experiments of the GPE type are concerned.

In the domain of the multiversion experimentation there is a set of basic problems GPE shares with other GPE-type experiments conducted over the last 10 or so years [e.g., Kni86, Sco84,87, Kel88]. For example, the unavoidable (over-)simplification of the problem, the corresponding lack of the exact operational environmental and engineering information, the (un)availability of the problem-specific expertise at the development sites, etc. This tends to foster an atmosphere that "this pudding is not really being cooked for eating", at least not from an operational perspective, so there is a tendency to focus more on the general concepts and principles rather than provision of a production version of the software and an in-depth investigation of the limitations of the specific techniques that could be safely extrapolated to real-life systems. This applies equally to the formal verification and the testing approaches. Future experiments of this kind would have to surmount this obstacle and offer firmer grounds for extrapolation by considering, perhaps smaller, problems in more detail and using personnel that has all the qualifications that a real-life team would need to develop a critical application. On the positive side, the GPE-type abstraction provides a very good test-bed for proof of concept work and exploration of new ideas since it furnishes a common denominator for an affordable comparison of different methods. In fact, the following two issues are probably

One potential research issue is related to the reading and inspection of specifications and the source code. For example, the source code of the released prototypes was written to be read and audited, and yet we found that some researchers had difficulty reading this code, primarily because the code writing, the documentation and the reading conventions were not synchronized. This suggests that a (perhaps application area sensitive) standard may need to be considered to guide the writing of the code for critical applications so that the developers, regulatory agencies, etc., would communicate in the same "formal" documentation language. The standard would have to be more specific than the usual good programming and commenting practices, but less restrictive than full formal specification languages. It would need to address the use, placement and format of the assertions (within and without comments), the re-use of standard solutions such as stack or queue implementations, the use of mnemonics, etc.

Another potential research issue is related to the development paradigm for small critical systems. Our experience, with this and other experiments of similar nature, indicates that for the problems that can be translated into solutions that do not exceed about 5000 lines of high-level language code, the optimal specification may, in fact, be the solution itself. This suggests a development paradigm for critical software applications in which a version of the system (a prototype) would be developed and tested before trying to verify it using formal methods. This would clarify the specifications and remove the errors that can be found more efficiently through testing. The tested program would then serve as the basis for the development of one or more abstractions that could be verified against the specifications, and then re-tested to "close the gap" between the abstract system descriptions and the actual one. The cycle may repeat several times. Work on this issue is in progress.

7. References

[Bel91] F. Belina, D. Hogrefe, and A. Sarma, *SDL with Applications from Protocol Specification*, Prentice-Hall 1991.

[Cle90] R. Cleavland, J. Parrow, and B. Steffen, "A Semantics-Based Tool for the Verification of Finite-State Systems, " Proc., 9th IFIP Symposium on Protocol Specification, Testing and Verification, June 1989, pp. 287-302, North-Holland, Amsterdam, 1990.

[Eck91] D.E. Eckhardt, A.K. Caglayan, J.P.J. Kelly, J.C. Knight, L.D. Lee, D.F. McAllister, and M.A. Vouk, "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability," IEEE Trans. Soft. Eng., Vol. 17(7), pp. 692-702, 1991, **Reprinted** in *Fault-Tolerant Software Systems: Techniques and Applications*, ed. Hoang Pham, IEEE Computer Society Press, pp. 72-82, 1992.

[Kel88] J. Kelly, D. Eckhardt, A. Caglayan, J. Knight, D. McAllister, M. Vouk, "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results", Proc. FTCS 18, pp. 9-14, June 1988.

[Kni86] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the assumption of Independence in Multi-version Programming", IEEE Trans. Soft. Eng., Vol. SE-12(1), 96-109, 1986.

[Sco84] R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Experimental Validation of Six Fault-Tolerant Software Reliability Models", IEEE FTCS 14,1984

[Sco87] R.K. Scott, J.W. Gault and D.F. McAllister, "Fault-Tolerant Software Reliability Modeling", IEEE Trans. Software Eng., Vol SE-13, 582-592, 1987.

[Tai90] K.C. Tai, "Theory of Condition-Based Software Testing", NCSU Computer Science Technical Report, TR-90-11 (September 91 revision)

[Tai93a] K.C. Tai, "Predicated-Based Test Generation for Computer Programs," Proc. 15th Intl. Conf. on Software Engineering, pp. 267-276, May 1993

[Tai93b] K.C. Tai, A. Paradkar, and M. Vouk, "Fault-Based Test Generation for Cause-Effects Graphs," submitted to CASCON '93, June 1993.

[Tel90] TELESOFT, SDT Case Tool, Telesoft, Malmoe, Sweden, 1990

[Vou90a] M.A. Vouk, A. Paradkar, and D.F. McAllister, "Modeling Execution Time of Multistage N-Version Fault-Tolerant Software," COMPSAC '90, pp. 505-511, 1990.

Reprinted in *Fault-Tolerant Software Systems: Techniques and Applications*, ed. Hoang Pham, IEEE Computer Society Press, pp. 55-61, 1992.

[Vou90b] Vouk, M.A., Caglayan, A., Eckhardt D.E., Kelly, J., Knight, J., McAllister, D., Walker, L., "Analysis of faults detected in a large-scale multiversion software development experiment," Proc. DASC '90, pp. 378-385, 1990.

[Vou93] Vouk M.A., McAllister D.F., Eckhardt, D.E., and Kim K., "An Empirical Evaluation of Some Software Fault-Tolerance Schemes in the Presence of Failure Correlation," **to appear** in the Journal of Computer and Software Engineering Special Issue on Reliable Software, 1993.

Appendix III – BGG/BRO

Abstract

The programs, "bgg" and "bro", are used to calculate static and dynamic code coverage metrics in PASCAL source code. Both programs were originally written on a VAX machine, but were ported to a DEC 5000 during 1992. The full manual set for the BGG/BRO toolset contains:

I. Introduction
II. Location of Code
III. Accounts
IV. BGG Operation
V. BRO Operation
VI. Compiling BRO
VII. VAX Machine Layout
VIII. DEC Machine Layout
IX. Bibliography

Most of the work was done in updating the BRO part of the toolset. The extended version of BRO allows the following test-set options

-bor : generate bro (or bor) test sets
-ibor : generate improved bro (bor) test sets
-bdd : generate bdd test sets
-ibdd : generate improved bdd test sets
-tt : generate the truth tables for each predicate

It also contains options to provide two alternatives for handling the NOT EQUAL operator. When NOT EQUAL is disabled it uses the min sets min_t = {<,>} and min_f = {=} for the relational operator "<>". Normally the NOT EQUAL is enabled, which uses the min sets min_t = {<>} and min_f = {=} for the relational operator "<>".

BGG / BRO User manual
Jim O'Connor
January 26th, 1993

Table of Contents

I. Introduction

The programs, "bgg" and "bro", are used to check for certain code coverage metrics in PASCAL source code. Both programs were originally written on a VAX machine, but were ported to a DEC 5000 during 1992. The BGG program ported smoothly, but problems were encountered with the BRO program.

II. LOCATION OF CODE

Two machines are of interest:

1) "druid" (formerly "csl36h")

   This machines contains working copies of both BGG and BRO. Details are in the VAX section.

2) "bvcd"

   This machine contains a working copy of bgg that is identical to the VAX copy, with the exception of a difference in the precision of the input and output data. The BRO copies on this machine are not fully functional due to machine dependent portability problems. The BRO version on this machines is, however, functional to various degrees in the new areas of N-PLUS testing that is described in Tai:1992.

   Further details on the two programs will follow.

III. ACCOUNTS

N/A

IV. BGG OPERATION

The following is a copy of the BGG man page which describes the various options available for this program.

****

BGG MAN PAGE

```
VAX                                                          bgg(se)


NAME
     bgg  - performs static and dynamic analysis (execution coverage) of
            Pascal code.

SYNTAX
     bgg file_name.p [up to two options]

DESCRIPTION
     The bgg command compiles and performs static and dynamic analysis of
     Pascal code using a number of metrics.

     This version of bgg is generated for analysis of Berkeley Pascal or its
     subset.

     Analysis can be performed using program control graph or data-flow graphs
     for individual variables. Analysis is performed for global, inter- and
     intra-procedural control and data flow. A summary is also provided for
     the whole program.

     Some of the active static metrics are: statement, line, & comment counts,
     cyclomatic number, branch, definition-use pair and path count.

     Some of the active dynamic coverage metrics are: statement, branch
     definition-use pair, p-uses, and c-uses coverage.

     To get more help on execution options type: bgg -h

     More complete documentation is available in the form of users manual and
     a paper describing the tool.

     It is possible to customize the bgg driver to access some analysis
     options which are not available in the default mode.

OPTIONS
     There are four processors that can be controlled: bgg-shell,
     bgg-bgc the graph generator, bgg-static the static analyzer, and
     bgg-dynamic the dynamic analyzer.

  ** bgg-shell options:

     bgg-shell runs bgc, bgg-static and bgg-dynamic in default modes unless
     otherwise is specified through options.

     default option:  graph generation only
                         * file_name.p file must be available
     other  options: -x  generate graph, static, and dynamic analysis
                         * file_name.p file must be available
                     -s  static analysis only
                         * all graph files must be available
                     -d  dynamic analysis only
                         * file_name.p.probe file must be available
                         * all graph files must be available
                     -a  static and dynamic analysis only
                         * file_name.p.probe file must be available
                         * all graph files must be available
                     -r  both static and dynamic analysis is
                         performed on reduced (data-flow) graphs
                     -h  help (this screen)
     warning: filenames "tables", "llgenout", "bgctables" are reserved names
              and will be deleted if they exist in the current directory
```

```
** bgg-bgc options:

   usage: bgc [options]
   required file:  tables, llgenout, test.p or option -f
   default:   all -pxxx options are on.
   options:   -v      print version and stop
              -prdef  turn off analysis of predefined functions
              -ppdef  turn off analysis of parameter pseudo-definitions
              -ppuse  turn off analysis of parameter pseudo-uses
              -pgdef  turn off analysis of global pseudo-definitions
              -pguse  turn off analysis of global pseudo-uses
              -pcdef  turn off analysis of constant pseudo-definitions
              -f      fname.p, where fname.p is source code
              -h      help (this screen)

** bgg-static options:

   usage: dustatic [options] < bgctables
   required file:  bgctables as standard input
   default:   control-flow analysis, iteration depth is one
   options:   -r   for analysis use reduced graph; default: full control graph
              -v   print program version only
              -i   xx
                   set depth of loop iteration to xx; default is 1
              -p   fname
                   fname contains list of procedures or functions
                   by their ordinal number, one a line, which are NOT
                   to be processed during static analysis;
                   default is to process all procedures/functions
              -h   help (this screen

** bgg-dynamic options:

   usage: dudynamic [options]
   required files: probe, bgctables
   default:   control-flow analysis
   options:   -r   for analysis use reduced graph
              -v   print program version only
              -h   help (this screen

BUGS
     This is a field testing release of bgg. Please remember that
     bgg is a research and teaching tool still under development.
     It contains some bugs we know about, and probably many we do not
     know about. So exercise care and check the results for consistency
     and sense.

     Please report all anomalies to

                          vouk@adm.csc.ncsu.edu

     bgg will only take complete programs which do not take input directly
     from the keyboard and output directly to the screen. All I/O has to be
     indirect (via files).

     bgg programs must have the (input,output) part.

NOTES
     Under VAX Ultrix bgg is very slow, so be patient. To start with, analyze
     only very small code. Code to be analyzed must be a complete program.

     On the DEC 5000 the following command MUST be executed prior to the
     execution of this program.
```

```
            "limit stacksize unlimited"

     This command is necessary because the default stacksize value is
     not adequate for BGG, which requires a large amount of stack space
     during operation.

*****************************************

To run a program through BGG:

In the case of either machine, you will see a directory named "bgg" in
the home account. Change to that directory and place your Pascal source
file in this directory. You can now execute the bgg program.

       Example:       "bgg foo.p -x"


WARNING: Please make sure that there are NO BLANK LINES at the end of
              you PASCAL program prior to running BGG on it. The BGG
              program will not function correctly if there are blank
              lines at the end of the source code file.

Note: All of the shell scripts for BGG are very VERBOSE. The file in-use,
       current work, as well as, location of the output results are all
       given by default. Currently you cannot "run silent" (no output).


V. BRO OPERATION

The BRO program exists in two different states on each of the different
       architectures. The VAX version is a completely operational unit
       that has no options. The DEC version of BRO which exists in the
       "~/bro_port" directory on bvcd WILL NOT RUN PROPERLY. Porting
       problems were encountered between the two machine architectures
       which cause one of the underlying programs, "bgc.instrument",
       malfunction during runtime. Time should be spent identifying the
       runtime bugs of this piece of the BRO code.


To run a program through BRO on the VAX:

       Example:

              The following will run the program "test.p" through BRO.

              1.     "cd to ~/bro"

              2.     copy "test.p" to this directory (or specify the
                                   complete path of the source code).

              3.     "bro test.p"

              The BRO script is quite verbose as it runs. The results will
              be saved to a file as show in the output of the BRO run.


To run a program through BRO on the DEC (bvcd):

       As mentioned above, the version of BRO located on bvcd in the "seg"
       account (bro_port directory) in non-functional due to port problems.
       There is, however, a partially operational copy located in
       "/projects/staats/bro/nplus2/bro3" directory. The programs of interest
       in this directory are "extract-ids" and "binary". These files represent
```

```
        the lastest BRO N-PLUS work done. "extract-ids" will generate the BRO,
        improved BRO, BDD, and improved BDD, as well as the truth tables for a
        program. The "binary" program is the "mutation tester" (see Tai:1992)

        The options for these programs are:

        extract-ids:

                -bor : generate bro (or bor) test sets

                -ibor : generate improved bro (bor) test sets

                -bdd :  generate bdd test sets

                -ibdd : generate improved bdd test sets

                -tt : generate the truth tables for each predicate


    binary:
                -bor : generate bro (or bor) test sets

                -ibor : generate improved bro (bor) test sets

                -bdd :  generate bdd test sets

                -ibdd : generate improved bdd test sets

                -equ  : Not equal disabled.  This will use the min
                        sets min_t = {<,>} and min_f = {=} for the
                        relational operator "<>". Normally the Not
                        equal is enabled by default, which uses the
                        min sets min_t = {<>} and min_f = {=} for
                        the relational operator "<>".

                -all  : Generate all test sets

Note: All of the shell scripts for BRO on the VAX machine are very VERBOSE.
        The file in-use, current work, as well as, location of the output
        results are all given by default. Currently you cannot "run silent"
        (no output).


VI. Compiling BRO

To compile the BRO program, follow these steps of each of the machines.

        VAX:
                "cd ~staats/bro/work/bgc.extract"
                "pc bgc.p -o bgc.extract"
                "llgen bgc.bnf"
                "cp ptableout llgenout.extract"
                (now copy the last file to your working directory)

                "cd ~/staats/bro/work/bgc.instrument"
                "pc bgc.p -o bgc.instrument"
                "llgen bgc.bnf"
                "cp ptableout llgenout.instrument"
                (now copy the last file to your working directory)

                "cd ~staats/pascal/new"
                "make"
                "cp extract-ids bro.static"
                "cc bro_dynamic.c -o bro_dynamic"
```

```
       DEC:
              "cd ~staats/staats_projects/bro/nlus2"
              "make"
              "cp extract-ids bro_static"
              "cc binary.c -o binary"


VII. VAX MACHINE (druid) LAYOUT

   Directory Structure:

 ~/Mail                        Directory of email exchanges between
                                     jao/staats/vouk/tai.

 ~/bgg                 Contains BGG program.

       bgg/bgg                 This is the MAIN PROGRAM for BGG. This will run
                     the specified file through BGG. Please see the
                     man page for further instructions.

                     Example:

                     "bgg foo.p -x"

                     This will run "foo.p" through BGG and place all
                     results (in the form foo.p.*) in the current
                     directory. The BGG program is VERY VERBOSE.

       bgg/BGG.MANPAGE
                     This is the BGG manual page, which is also
                     in the text above.

       bgg/run_all_actual
                     This shell script will run all of the L programs
                     with the actual data. The script automatically
                     stores the generated data for each L program as
                     it goes. See the script for further details.

       bgg/run_single_act
                     Given the name of an L program, this script will
                     run it through BGG using the actual data and save
                     the results in a directory (see script) automatically.

                     Example:  "run_single_act l17.3"


 ~/bro                 Contains BRO program.

       ~/bro/bro     This is the main BRO schell script which runs the
                     specified file through BRO and save the results in
                     the "~/bro/results" directory.

                     Example:

                     "bro test.p"

                     This example will run test.p, located here in the
                     current directory, through BRO and store the results
                     in the "~/results/test.p.single.(unique_number)"
                     directory.

       ~/bro/data1   Random data for L programs. Used by BGG and BRO on
```

```
                this machine.

~/bro/data2     Actual data for L programs. Used by BGG and BRO on
                this machine.

~/bro/results   All results from various BRO scripts are placed under
                this directory.

~/bro/run_L_acts
                This BRO script calls the "runact" script for each
                of the L programs -- (12.8, 13.2, 14.2, 117.3).
                Note: 17.4 never worked correctly with BRO.
                See "runact" description below for further details.

~/bro/run_L_rans
                This BRO script calls the "runran" script for each
                of the L programs -- (12.8, 13.2, 14.2, 117.3).
                Note: 17.4 never worked correctly with BRO.
                See "runran" description below for further details.

~/bro/run_L_to_dir
                This BRO script runs the specified L program through
                BRO and stores the results in the named directory.
                This essentially overrides the normal approach of
                placing the results of the BRO run in the
                "~/bro/results" directory.

                NOTE: This particular script runs the L program with
                the data file (called "rtest") that is in the ~/bro
                directory at the time of execution. It only runs the
                BRO program ONCE using this data file.

                Example:

                "run_L_to_dir 117.3 jao.work/set4"

                Note that the program "117.3.p" must exist in the
                directory "jao.work/set4" for this command to execute
                correctly.

~/bro/run_all_L_to_dir
                This BRO script calls the "~/bro/run_L_to_dir" script
                for each L program and specifies the
                "~/bro/jao.work/set4" directory for the results. This
                script is used to run the L program results to this
                directory, INSTEAD  of the usual "~/bro/results" dir
                used by the "~/bro/run_L_acts" script.


~/bro/run_testcase
                This BRO script will run BRO on the specified test
                program as it appears in the "~jao.work" directory.
                The syntax for this command is:

                "run_testcase <sub-dir> <test-set> <test-num>"

                Please see the README file under "~jao.work" for
                a better understanding of the above parameters.

                Example:
                        "run_testcase 1 1 1"

                        This will run BRO on the file "set1_1-1.p"
                        located in "~/jao.work/set1/set1_1" and
```

```
                        save the results in that directory.

      ~/bro/runact   This BRO script runs the specified L program through
                     BRO one time for each actual data test set. The results
                     are placed in the ~/results directory.

      ~/bro/runall_L       This BRO script runs all the L programs through BRO
                     with both the actual and random data by calling
                     "~/bro/run_L_act" and "~/bro/run_L_ran" (Please
                     see description above for these two scripts).

      ~/bro/runran   This BRO script runs the specified L program through
                     BRO one time for each random data test set. The results
                     are placed in the ~/results directory.


      ~/bro/jao.work       This directory structure contains all of the
                     various PASCAL test programs written to verify
                     that BRO handles PASCAL code and constructs. There
                     are README files in each directory in the structure
                     that are self-explainatory.


VIII. DEC MACHINE (bvcd) LAYOUT

   Directory Structure:

 ~/bgg               Contains BGG program.

      ~/bgg/bgg      This is the MAIN PROGRAM for BGG.

      ~/bgg/BGG.MANPAGE
                     This is the BGG manual page, which is also
                     in the text above.

      ~/bgg/L              This directory contains the current copies the
                     ported L programs, as well as the results from
                     the various scripts below.

      ~/bgg/results  Directory containing old results from the BGG
                     runs.

      ~/bgg/run_all_actual
                     This shell script will run all of the L programs
                     with the actual data. The script automatically
                     stores the generated data for each L program as
                     it goes. See the script for further details.

      ~/bgg/run_all_random
                     This shell script will run all of the L programs
                     with the random data. The script automatically
                     stores the generated data for each L program as
                     it goes. See the script for further details.

      ~/bgg/run_single_act
                     Given the name of an L program, this script will
                     run it through BGG using the actual data and save
                     the results in a directory (see script) automatically.

                     Example:  "run_single_act 117.3"

      ~/bgg/run_single_ran
                     Given the name of an L program, this script will
```

run it through BGG using the random data and save
the results in a directory (see script) automatically.

Example: "run_single_ran 19w"

~/bro_port — First attempt at BRO port to this machine. This code is NOT operational.

~/misc — This directory contains misc files used during the research work. The 3 files here are simply summaries of other data files, which were used to generate some graphs on a MAC for overheads used in a presentation by Drs. Vouk and Tai.

~/misc/L — This directory contains the L programs in ported form for the DEC.

~/misc/fts — This directory contains the data files and the data file generator for the fts84 program. The various "rprompts" files are used as data for the "L" programs.

~/motif — This points to the directory containing the MOTIF front-end written by Missy Whitfield for Dr. Vouk. This program is a GUI for the BGG program only, at this point in time. Please talk to Dr. Vouk for details on the operation of this GUI and its current status.

IX. Bibliography

Tai, K.C. "Predicate-Based Test Generation for Computer Programs", September 1992.

Appendix IV – Specification-Based Testing

Fault-Based Test Generation for Cause-Effect Graphs

K. C. Tai[¶] , Amit Paradkar[¶] , H. K. Su[§] , and Mladen A. Vouk[¶]

Abstract

The notion of cause-effect graphs (CEGs) has been used for the specification and test generation of a software system. In this paper, we present a fault-based approach to test generation for CEGs, called **BOR** (*b*oolean *o*perator) **testing**, which is based on the detection of boolean operator faults. We show how to generate a minimum BOR test set for a CEG and how to evaluate an existing test set for a CEG in order to determine whether additional tests are necessary for BOR testing. We have applied BOR testing to a CEG that specifies a real-time boiler control and monitoring system. The results of our empirical studies indicate that CEG-based BOR testing is very practical and effective.

1. Introduction

The notion of cause-effect graphs (CEGs) was developed for system specification and test generation [Elm73, Mye79]. A CEG consists of causes, effects, and graphical notations expressing logical relationships among causes and effects. A cause is an input condition, an effect is an output condition, and logical operators include AND ("^"), OR ("V"), and NOT ("~"). For example, below is a CEG with nodes N1 through N4 denoting causes, nodes N5 and N6 intermediate nodes, and nodes N7 and N8 effects.

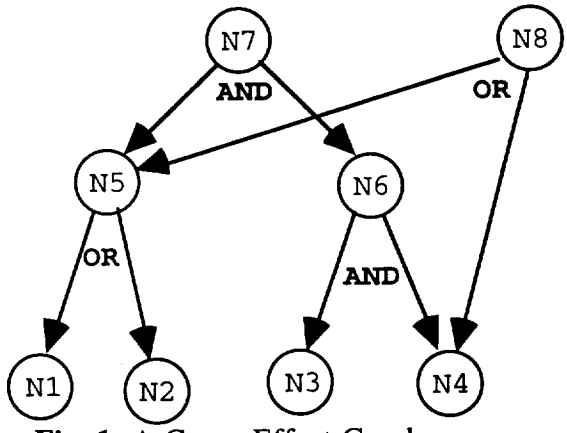

Fig. 1 A Cause-Effect Graph

[¶] The authors are with Computer Science Department, North Carolina State University, Raleigh, North Carolina 27659-8206. This work was supported in part by NSF grant CCR-8907807 and NASA grant NAG-1-983.

[§] The author is with IBM, Research Triangle Park, North Carolina.

The CEG for a software system can be analyzed to determine its completeness and consistency. Also, the CEG provides a basis for deriving specification-based test cases for the software system. One approach to test generation is to consider all possible combinations of causes. This approach, referred to as **exhaustive testing**, is impractical since the number of tests for a CEG is an exponential function of the number of causes of the CEG. A more practical test generation algorithm for CEGs was described in [Mye79], which is referred to as **algorithm CEG_Myers** and is shown in section 2.

In section 3, we describe the **BOR** (*b*oolean *o*perato*r*) **testing** strategy for boolean expressions and predicates. This testing strategy is based on the detection of boolean operator faults, including incorrect AND and OR operators and missing or extra NOT operators. In section 4, we show how to generate a minimum test set satisfying BOR testing for a CEG. In section 5, we discuss how to evaluate an existing test set for a CEG in order to determine whether additional tests are necessary for BOR testing. In section 6, we describe our empirical study of applying BOR testing to a CEG that specifies a real-time boiler control and monitoring system, which was used as the generic problem exercise for the International Workshop on the Design and Review of Software Controlled Safety-Related Systems. Finally, section 7 concludes this paper.

2. Algorithm CEG_Myers

Let G denote a CEG and let "t" and "f" denote "true" and "false", respectively. A test of G or any node in G is a sequence of "t"s and "f"s for the causes of G. The nodes in G are visited from effect nodes to cause nodes. Assumed that a node N with a specified output value is visited. First, one of the following rules (1)-(4) is applied to select a set of combinations of input values of N.

(1) If N is an OR node with output value "t", all combinations of inputs leading to an "t" output and having only one input being "t" are selected.

(2) If N is an OR node with output value "f", all combinations of inputs leading to an "f" output are selected.

(3) If N is an AND node with output value "t", all combinations of inputs leading to a "t" output are selected.

(4) If N is an AND node with output value "f", all combinations of inputs leading to an "f" output are selected. However,
   - for the combination of all inputs being "f", only one test is selected for N, and
   - for any combination with at least one input being "f", only one test is selected for each input being "t".

Then for each selected combination of input values of N,
- for each input value that is the output value of another node, say N', the same algorithm is applied to node N' with the specified output value, and
- the resulting test set for N is the set of all combinations of the test sets for the input values of N.

There is no analysis on the number of tests generated for G as a function of the numbers of causes and effects of G.

Now we show how to apply the above algorithm to generate tests for node N7 in the CEG in Fig. 1. A test for a node N in Fig. 1 is a list of "t"s and "f"s in the order of the cause nodes of N from left to right. For example, the test (t,f,t,f) for node N7 means that nodes N1, N2, N3, and N4 have values "t", "f", "t", and "f" respectively. As another example, the test (f,t) for node N6 means that nodes N3 and N4 have values "f" and "t" respectively. For node N7, we consider the two output values "t" and "f":

For N7 with output value "t", we apply rule (3) and have (t,t) as the only combination of output values of nodes N5 and N6. For N5 being true, we have (t,f) and (f,t). For N6 being true, we have (t,t). Thus, we have tests (t,f,t,t) and (f,t,t,t) for N7.

For N7 with output value "f", we apply rule (4) and have

(a) ("f" for node N5, "f" for node N6) with one test,

(b) ("t" for node N5, "f" for node N6) with one test for N5, and
(c) ("f" for node N5, "t" for node N6) with one test for N6.
For (a), we have only one choice, (f,f,f,f).
For (b), we choose (t,f) for N5. By applying rule (4) to N6 with output value "f", we have (t,f), (f,t) and (f,f). Thus, we have tests (t,f,t,f), (t,f,f,t) and (t,f,f,f) for N7.
For (c), we have only one choice, (f,f,t,t).

Thus, we have selected seven tests for node N7, with two of them producing "t" and five producing "f". Note that there are sixteen possible tests for node N7, with three of them producing "t" and thirteen producing "f".

3. The BOR Testing Strategy for Boolean Expressions and Predicates

The BOR (*b*oolean *o*perator) testing strategy for a boolean expression is to require a set of tests to guarantee the detection of boolean operator faults, including incorrect AND and OR operators and missing or extra NOT operators (with the assumption that the boolean expression does not contain faults of other types). A test set T for a boolean expression B is said to be a **BOR test set** for B if T satisfies the BOR testing strategy for B. In [Tai87] we showed that a BOR test set for a boolean expression is very effective for detecting all types of boolean expression faults, including boolean operator faults, incorrect boolean variables and parentheses, and their combinations.

One common approach to testing a program, referred to as **predicate testing**, is to require certain types of tests for each predicate (or condition) in the program. A number of predicate testing strategies have been proposed, including branch testing, domain testing, and others [Bei90]. However, these strategies are either ineffective or impractical for testing **compound predicates**, which are predicates with one or more AND/OR operators. In [Tai93] two testing strategies for compound predicates were described. The **BOR testing** strategy for a predicate requires a test set to guarantee the detection of boolean operator faults. The **BRO (*b*oolean and *r*elational *o*perator) testing** strategy for a predicate is to require a set of tests to guarantee the detection of boolean and relational operator faults, where a relational operator fault refers to an incorrect relational operator.

The generation of tests for BOR or BRO testing of a predicate is based on **constraints** for this predicate. For example, (t,f) and (<,=) are constraints for the predicate ((E1=E2) & (E3/=E4)), where E1 through E4 are arithmetic expressions. The coverage of (t,f) for this predicate requires a test making "E1=E2" to be true and "E3/=E4" false, and the coverage of (<,=) for this predicate requires a test making E1<E2 and E3=E4. A constraint for BOR testing of a predicate consists of "t" and "f", while a constraint for BRO testing of a predicate consists of "t", "f", ">", "<", and "=".

A set S of constraints for a predicate C is said to be a **BOR (BRO) constraint set** for C provided that if a test set T for C covers each constraint in S at least once, then T satisfies BOR (BRO) testing of C. For a predicate with n, n>0, AND/OR operators, a minimum BOR (BRO) constraint set can be easily constructed, which contains at most n+2 (2*n+3) constraints. For ((E1=E2) & (E3/=E4)), its minimum BOR constraint set is {(t,t), (f,t), (t,f)} and one of its minimum BRO constraint sets is {(=,>), (=,<), (>,>), (<,>), (=,=)}. Experimental results indicate that BOR and BRO testing are effective for the detection of various types of faults in a predicate and provide more specific guidance than branch testing for test generation.

4. Generation of A BOR Test Set for a Cause-Effect Graph

In a CEG, an effect node and its associated portion of the CEG denote a compound predicate in terms of causes. Since a CEG represents a set of compound predicates, the BOR testing strategy can be applied to generate tests. For the sake of simplicity, each cause node is viewed as a boolean variable and has "t" or "f" as its value. In [Tai93], algorithm BOR_GEN produces a minimum

BOR constraint set for a compound predicate. Below we describe an adapted version of algorithm BOR_GEN in order to produce a minimum BOR test set for a CEG.

The nodes in a CEG are visited from cause nodes to effect nodes. (Algorithm CEG_Myers does the opposite.) When a node is visited, a test set for this node and its associated CEG is constructed. After the visit of an effect node, a test for the CEG associated with the node is available. Before we show the test generation algorithm, we need to introduce a number of definitions.

Let N be a node in a CEG. The value of N on test X is denoted as N(X). Assume that S is a test set for N. S can be divided into two sets S_t and S_f, where S_t(N) = {X in S | N(X) = t} and S_f(N) = {X in S | N(X) = f}. Let u=(u_1, ..., u_m) and v=(v_1,...,v_n), where m,n >0, be two lists. The concatenation of u and v, denoted as (u,v), is (u_1,...,u_m,v_1,...,v_n). Let A and B be two sets. A$B denotes the union of A and B, A*B the product of A with B, and |A| the size of A. A%B, called the **onto** from A to B, is a minimal set of (u,v) such that u and v are in A and B, respectively, every element in A appears as u at least once, and every element in B appears as v at least once. Thus, |A%B| is the maximum of |A| and |B|. If both A and B have two or more elements, A%B has several possible values and returns any one of them. For example, assume that C={(a),(b)} and D={(c),(d)}. C%D has two possible values: {(a,c),(b,d)} and {(a,d),(b,c)}.

Algorithm BOR_GEN

For a cause node, its minimum BOR test set is {(t), (f)}. Assume that N1 and N2 are cause nodes. The minimum BOR test set for an AND node with N1 and N2 as inputs is {(t,t), (f,t), (t,f)}. (Note that (f,f) is not included.) The minimum BOR test set for an OR node with N1 and N2 as inputs {(f,t), (t,f), (f,f)}. (Note that (t,t) is not included.) The following three rules show how to generate a BOR test set for a node with one or both its inputs being intermediate nodes.

Assume that N1 and N2 are nodes in a CEG and that S1 and S2 are minimum BOR test sets for N1 and N2 respectively.

(1) A minimum BOR test set S for an AND node with N1 and N2 as inputs is constructed as follows:
S_t = S1_t % S2_t
S_f = (S1_f * {t2}) $ ({t1} * S2_f)
where t1 is in S1_t, t2 is in S2_t, and (t1,t2) is in S_t.

(2) A minimum BOR test set S' for an OR node with N1 and N2 as inputs is constructed as follows:
S'_f = S1_f % S2_f
S'_t = (S1_t * {f2}) $ ({f1} * S2_t)
where f1 is in S1_f, f2 is in S2_f, and (f1,f2) is in S'_f.

(3) A minimum BOR test set S" for a NOT node with N1 as the input is constructed as follows:
S"_t = S1_f
S"_f = S1_t

Now we show how to apply the above algorithm to node N7 and its associated CEG in Fig. 1, which denote to the compound predicate ((N1|N2) & (N3&N4)). For 1<=i<=7, let Si be a minimum BOR test set for node Ni.

For 1<=i<=4, Si_t = {(t)} and Si_f = {(f)}.
By applying rule (2) to node N5, we have
S5_t = {(t,f), (f,t)} and S5_f = {(f,f)}.
By applying rule (1) to node N6, we have

S6_t = {(t,t)} and S6_f = {(t,f), (f,t)}.
By applying rule (1) to node N7, we have
S7_t = {(t,f,t,t), (f,t,t,t)} and
S7_f = {(f,f,t,t), (t,f,t,f), (t,f,f,t)} or
{(f,f,t,t), (f,t,t,f), (f,t,f,t)}.

Algorithm BOR_GEN produces two minimum BOR test sets for node N7 in Fig. 1, which are referred to as N7_1 and N7_2. Each of N7_1 and N7_2 contains five tests. In section 2, we applied algorithm CEG_Myers to produce a test set for node N7 in Fig. 1, which contains seven tests and is referred to as N7_3. Below are N7_1, N7_2, and N7_3, each in one column, with each row showing the same test. N7_2 is a subset of N7_3.

N7_1	N7_2	N7_3
(t,f,t,t)	(t,f,t,t)	(t,f,t,t)
(f,t,t,t)	(f,t,t,t)	(f,t,t,t)
(f,f,t,t)	(f,f,t,t)	(f,f,t,t)
(f,t,t,f)		
(f,t,f,t)		
	(t,f,t,f)	(t,f,t,f)
	(t,f,f,t)	(t,f,f,t)
		(t,f,f,f)
		(f,f,f,f)

In a CEG, an AND or OR node may have three or more inputs. Rules (1) and (2) above can be easily extended for such nodes. Assume that an AND node N has nodes N1, N2, and N3 as inputs, that is, N is (N1 AND N2 AND N3), which is equivalent to ((N1 AND N2) AND N3). By applying rule (1) twice to node N, we have {(t,t,t), (f,t,t), (t,f,t), (t,t,f)} as its only minimum BOR test set. (Algorithm CEG_Myers requires these four tests as well as (f,f,t), (t,f,f), (f,t,f), (f,f,f) for the "t" and "f" outputs of node N.) Similarly, the only minimum BOR test set for (N1 OR N2 OR N3) is {(f,f,f), (t,f,f), (f,t,f), (f,f,t)}. (Algorithm CEG_Myers requires the same four tests for the "t" and "f" outputs of node N.)

An empirical study of comparing algorithms BOR_GEN and CEG_Myers was performed by using a set, referred to as SBE_4.2, of fifty-one non-equivalent boolean expressions [Su89]. Each boolean expression in SBE_4.2 contains three AND/OR operators and four distinct boolean variables. For each boolean expression in SBE_4.2, algorithm BOR_GEN was applied to generate a test set, say T, and then the number of other boolean expressions in SBE_4.2 that can be distinguished by T was determined. (Two boolean expressions are distinguished by T if at least one test of T produces different results on the two expressions.) Algorithm CEG_Myers was applied similarly. Below we present, for each of algorithms BOR_GEN and CEG_Myers, the average size of test sets for boolean expressions in SBE_4.2 and the average error detection rate by these test sets.

	Size	Error Detection Rate
BOR_GEN	4.9	99.73%
CEG_Myers	9.2	100.00%

The above results show that algorithm BOR_GEN is almost as effective as algorithm CEG_Myers for error detection and that the average number of tests produced by algorithm BOR_GEN is about one-half of that by algorithm CEG_Myers.

Definition. A cause-effect graph G is said to be a **singular cause-effect graph** (or singular CEG) if for every pair of cause node N and effect node N' in G, there exists at most one path between N to N'.

Consider the application of algorithm BOR_GEN to the non-singular graph shown in Fig. 2.

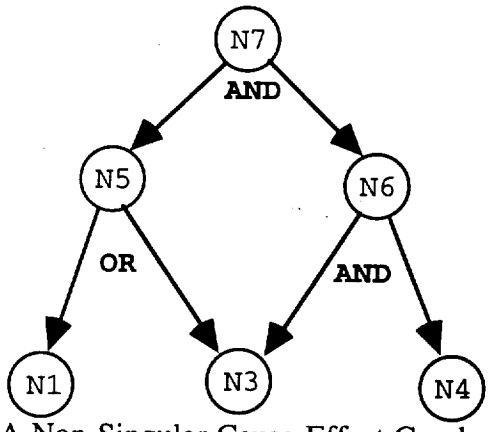

Fig. 2 A Non-Singular Cause-Effect Graph

As shown earlier for node N7 in Fig. 1, we have

S5_t = {(t,f), (f,t)} and S5_f = {(f,f)}, and
S6_t = {(t,t)} and S6_f = {(t,f), (f,t)}.

By applying rule (1) to node N7, we have two tests for S7_t, one is the merge of (f,t) for N5 and (t,t) for N6 and the other is the merge of (t,f) for N5 and (t,t) for N6. Since N3 is both the second input of N5 and the first input of N6, the first merge results in (f,t,t) for N7. The second merge, however, fails since N3 has conflicting values in the two tests to be merged. Because of the conflict, no test is resulted from the second merge. Hence, S7_t contains only (f,t,t). Note that algorithm BOR_GEN needs to be modified slightly in order to handle conflicts when tests are merged. Whenever a conflict occurs during the merge of two or more tests, no test is resulted from the merge. Thus, we have the following theorem.

Theorem 1: Algorithm BOR_GEN produces a minimum BOR test set for a singular cause-effect graph.

5. Measurement of BOR Coverage of a Cause-Effect Graph by a Test Set

Assume that a test set T for a CEG G already exists. We want to determine whether T is a BOR test set for G, and if not, we want to construct a minimum set of additional tests in order to satisfy the BOR testing strategy. One intuitive solution is to apply algorithm BOR_GEN to generate a minimum BOR test set, say T*, for G and then determine the **BOR coverage** of G by T as the coverage of T* by T. This solution, however, is not desirable. If G has two or more minimum BOR test sets, the choice of T* may significantly affect the coverage of T* by T. Furthermore, it possible that the use of a non-minimum BOR test set for G as T* gives T a higher BOR coverage. To illustrate this, consider the three BOR test sets N7_1, N7_2, and N7_3 for node N7 in Fig. 1. Both N7_1 and N7_2 have five tests and N7_3 has seven tests. For each of N7_1, N7_2, and N7_3 as T*, below we show the coverage of T* by each of the other two test sets.

T*	T	coverage
N7_1	N7_2 or N7_3	3/5
N7_2	N7_1	3/5
N7_2	N7_3	5/5
N7_3	N7_1	3/7
N7_3	N7_2	5/7

The first author of this paper has developed an algorithm† , which determines whether a test set T for a CEG G is a BOR test set, and if not, produces a minimum set T' of additional tests needed for satisfying the BOR testing strategy. Let m and n be the sizes of T and T', respectively. The **BOR coverage** of G by T is defined as n/(m+n). This algorithm also detects **redundant** tests in T, which can be eliminated without any impact on the BOR coverage of G by T.

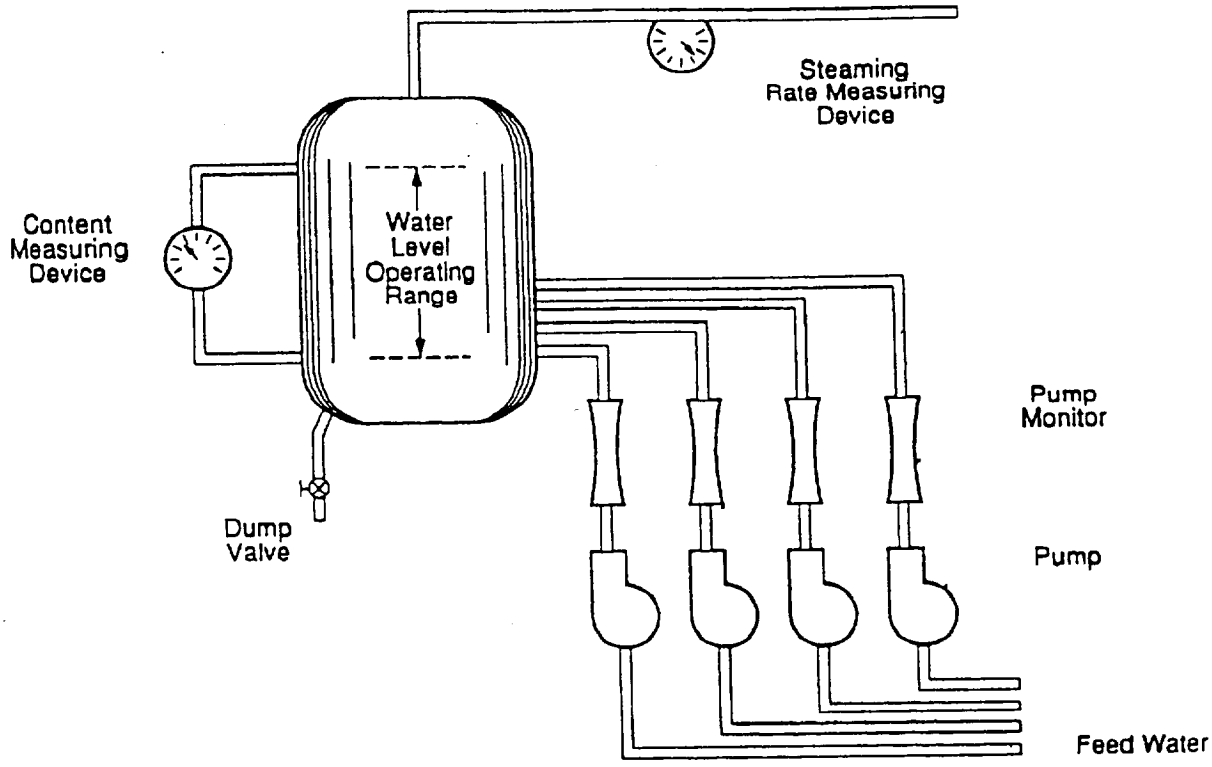

Fig. 3 Context Diagram for Boiler Control and Monitoring System

6. A Case Study: A Boiler Control and Monitoring System

In this section, we describe our empirical study of applying BOR testing to a CEG that specifies a real-time boiler control and monitoring system, which was used as the generic problem exercise for the International Workshop on the Design and Review of Software Controlled Safety-Related Systems. Fig. 3 shows the context diagram for the boiler control and monitor system. The system consists of a natural-gas fired water-tube boiler producing saturated steam. The steam flow may vary rapidly and irregularly between zero and maximum, following a varying external demand.

The water level in the boiler is regulated by the control of the inflow of feedwater. The water level must be kept between an upper and lower limits. If the water level is above the upper limit, water will be carried over into the steam flow and cause damage. If the water level is below the lower limit, boiler tubes will dry out and may overheat and burst. If the control of water level is lost, the boiler is shut down. The water level and the steam flow are measured by an instrumentation system, which report sensor values. The readings from sensors are transmitted over an intrinsically unreliable communication link to the control program. This control program is expected to perform the following tasks: (a) to regulate the water level by controlling the inflow of feedwater by appropriately turning pumps on or off at required instances, (b) to diagnose and isolate all the potential errors and issue a correction/repair request if any are discovered, (c) to display at all times

† This algorithm is being reviewed for patent application.

"best estimates" of various readings for the boiler operator, and (d) to accept any appropriate operator commands.

6.1 Derivation of a Cause-Effect Graph for the Boiler System

The specification provided for the boiler control and monitoring system was analyzed in order to derive a CEG. Since "boiler shutdown" is the most critical effect in the boiler system, we constructed a CEG only for this effect. This CEG is organized in five levels. Level 1 (the highest level) CEG for boiler shutdown is shown in Fig. 4. The annotations for nodes in level 1 CEG are given below.

E - boiler shutdown
C221 - externally initiated
C220 - internally initiated
C202 - operator initiated
C203 - instrumentation system initiated
C201 - bad startup
C200 - operational failure
C197 - confirmed keystroke entry
C198 - confirmed "shutnow" message
C196 - multiple pumps failure (more than one)
C195 - water level meter failure during startup
C194 - steam rate meter failure during startup
C193 - communication link failure
C192 - instrumentation system failure
C191 - C180 and C181
C190 - water level out of range
C180 - water level meter failure during operation
C181 - steam rate meter failure during operation

Nodes C180, C181, C190, and C192 through C198, which are cause nodes of level 1 CEG, are effect nodes of level 2 CEGs. Similarly, some of the cause nodes of level 2 CEGs are effect nodes of level 3 CEGs, and so on. Due to space limitation, CEGs of level 2 through 5 are not shown in this paper. Below is the list of cause nodes in the boiler's complete CEG.

C197 - confirmed keystroke entry
C198 - confirmed "shutnow" message
C120 - more than 1 corrupt message in 2 consecutive cycles
C121 - more than 1 test message data wrong in 2 consecutive cycles
C122 - more than 1 test message missing in 2 consecutive cycles
C123 - missing messages within STX and ETX characters for 2 consecutive cycles
C124 - no transmission for 2 consecutive cycles
C125 - missing pump information in a transmission
C126 - missing flow monitor information in a transmission
C127 - missing water level information in a transmission
C128 - missing steam rate information in a transmission
C129 - unexpected "start" message in a transmission
C130 - unexpected "boilstdby" message in a transmission
C131 - missing "boilstdby" message in a transmission
C132 - unexpected "boilevadj" message
C105 - steaming rate exceeds 0 lbs/hr
C103 - steaming rate exceeds 275,000 lbs/hr
C101 - steaming rate exceeds 550,000 lbs/hr
C93 - steaming rate below 550,000 lbs/hr
C91 - steaming rate below 275,000 lbs/hr
C76 - pump 4 indicator stuck off

C75 - pump 4 indicator stuck on
C74 - pump 4 stuck off
C73 - pump 4 stuck on
C72 - pump 3 indicator stuck off
C71 - pump 3 indicator stuck on
C70 - pump 3 stuck off
C69 - pump 3 stuck on
C68 - pump 2 indicator stuck off
C67 - pump 2 indicator stuck on
C66 - pump 2 stuck off
C65 - pump 2 stuck on
C64 - pump 1 indicator stuck off
C63 - pump 1 indicator stuck on
C62 - pump 1 stuck off
C61 - pump 1 stuck on
C27 - flow monitor 4 stuck off
C26 - flow monitor 4 stuck on
C25 - flow monitor 3 stuck off
C24 - flow monitor 3 stuck on
C23 - flow monitor 2 stuck off
C22 - flow monitor 2 stuck on
C21 - flow monitor 1 stuck off
C20 - flow monitor 1 stuck on

6.2 Measurement of BOR Coverage of the Boiler's Cause-Effect Graph by a Test Set

The boiler control and monitoring system was used in an earlier study [Vou93] at North Carolina State University. In the earlier study, the specification of the system was re-written using the finite-state machine model, and the test suites for the unit, integration and system testing of the boiler system were constructed. From the previous test suites, a total of 372 test cases were found to be related to the boiler shutdown effect. This set of 372 test cases, referred to as the **shutdown test set**, was used in our empirical studies of BOR testing.

In section 5, it was mentioned that an algorithm for measuring the BOR coverage of a CEG by a test set has been developed. We applied this algorithm to measure the BOR coverage of the boiler's CEG by the shutdown test set. Of the 372 shutdown tests, 59 tests (about 1/6 of total) were found to be redundant. Also, 24 more tests were needed for satisfying the BOR testing criterion. Most of the redundant tests deal with pump/flow monitor combinations. However, most of the additional tests needed for BOR testing also deal with pump/flow monitor combinations. The reason is that when the shutdown test set was constructed, the selection of tests for combinations of pump and flow monitors was done in ad hoc fashion.

As mentioned earlier, the shutdown test set was constructed according to the finite-state machine based specification of the boiler's system. It was accomplished by three persons with a total of approximately 100 person-hours. The boiler's CEG was constructed by one person in about 20 hours. (The person who constructed the CEG is one of the three persons who developed the shutdown test set.) If an automatic tool for algorithm BOR_GEN is available, then the only human effort is the construction of a CEG.

The use of CEGs for system specification offers a number of advantages. During or after the construction of a CEG for a system, ambiguities and inconsistencies in the specification of the system can be detected. In addition, the tests generated for the CEG, which are used to test the

implementation of the system, can also help the detection of errors in the specification or the CEG itself.

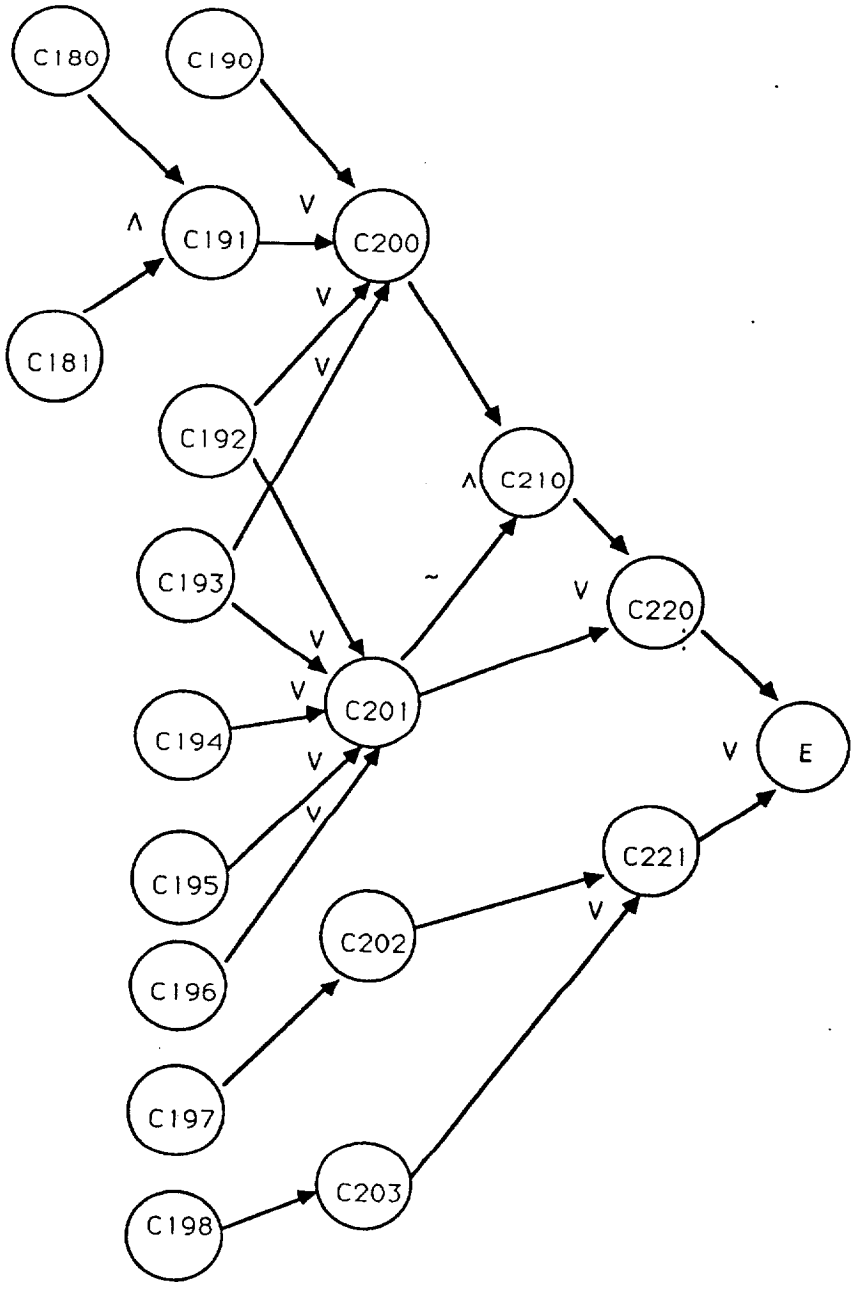

Fig. 4 Level 1 Cause-Effect Graph for Boiler Control and Monitoring System

6.3 Measurement of BOR Coverage of the Boiler's Implementation by a Test Set

As mentioned earlier, the BOR testing strategy for the coverage of predicates in a program was developed. So we investigated the BOR coverage of the boiler's implementation with the shutdown test set. The boiler's implementation contains about 4,500 lines of C code. Since no automatic tool was available for the measurement of BOR coverage of C programs, we chose only one module in the boiler's implementation. The selected module contains 360 C statements and 34 predicates, of

which 21 are simple predicates (i.e., no AND/OR operators) and the other compound predicates with one AND/OR operator. We manually transformed this module for the measurement of BOR coverage and generated 81 BOR constraints for the compound predicates in this module. (Two constraints are needed for a simple predicate and three constraints for a compound predicate with one AND/OR operator.) The shutdown test set was used to execute the selected module to collect the BOR coverage. Two of the 81 constraints were not covered by the shutdown test set. When the two uncovered constraints were further investigated, a bug in the selected module was discovered. The two uncovered constraints correspond to some of the additional tests needed for BOR testing of the boiler's CEG (see section 6.2).

For the boiler system, the causes in its CEG appear as predicates in its implementation and thus the tests generated for the CEG are inputs to the implementation. Such a test covers some BOR constraints in the boiler's implementation. However, a BOR constraint produced for a predicates in the boiler's implementation does not necessarily correspond to a test generated for the CEG. For example, assume that the boiler's CEG contains an OR node for shutdown, which contains causes N1, N2, and N3 as inputs. To satisfy BOR testing for this OR node, we need the test set {(t,f,f), (f,t,f), (f,f,t), (f,f,f)}, referred to as OR_3. If the implementation of this OR node is

```
if (N1 | N2 | N3) then { ...; shut down; }
...
```

then OR_3 is exactly the BOR constraint set for the predicate in the above if statement. Assume that the implementation of this OR node is

```
if N1 then { ...; shut down };
if N2 then { ...; shut down };
if N3 then { ...; shut down };
...
```

The BOR constrain set for each predicate in the above three if statements is {(t), (f)}. These BOR constraints do not have direct correspondence with the tests in OR_3. The reason is that the generation of BOR constraints for predicates does not consider combinations of predicates. However, the sequence of these if statements requires that when the second (third) if statement is tested for shutdown, the first (first and second) if statement(s) must have a false outcome. The test set OR_3 satisfies BOR testing for any possible sequence of the three if statements.

7. Conclusion

In this paper, we have described the BOR testing strategy and presented algorithm BOR_GEN, which produces a minimum test set for a CEG or predicate in order to satisfy BOR testing. We have also discussed the problem of evaluating an existing test set for a CEG in order to determine a minimum set of additional tests for BOR testing.

The empirical studies reported in section 6 show that the combination of the CEG notation and the BOR testing strategy provides a very powerful approach to the specification, analysis, and test generation of a software system. For the boiler control and monitoring system, we developed a multi-level CEG, which served as a basis for analysis and test generation. For the shutdown test set for the boiler's system, we measured its BOR coverage of the boiler's CEG, detected the existence of redundant tests, and determined the additional tests for BOR testing. We also measured the BOR coverage of the boiler's implementation by the shutdown test set. The investigation of additional tests needed for BOR testing (based on the boiler's CEG or implementation) discovered a bug in the boiler's implementation, which was not discovered by earlier extensive testing. A BOR test set for the boiler's CEG is very effective for detecting logical errors in the boiler's implementation since this test set, in effect, forces a comparison of the logical structures of the boiler's specification and implementation.

The measurement of BOR coverage of a CEG or predicate by an existing test set is very important for the following reason. To generate tests for a software system, the most economical way is to

randomly select test cases. However, empirical studies [Vou86] showed that the control/data flow coverage of a program by random tests quickly reaches a saturation point. Higher coverage of the program can be reached only by using functional tests (including boundary values, special values, etc.). Thus, a practical and effective testing strategy is to first apply random testing and measure its coverage (based on BOR coverage or other coverage metrics). Then additional tests are derived to improve test coverage.

References

[Bei90] B. Beizer, Software Testing Techniques, 2nd edition, Van Nostrand, 1990.

[Elm73] Elmendorf, W. R., "Cause-effect graphs in functional testing", TR-00.2487, IBM Systems Development Division, Poughkeepsie, NY, 1973.

[Mye79] Myers, G. J., The Art of Software Testing, Wiley, 1979.

[Su89] Su, H. K., "Test generation for boolean expressions and combinational logic circuits", Ph.D. dissertation, Dept. of Computer Science, North Carolina State University, 1989.

[Tai87] Tai, K. C., and Su, H. K., "Test generation for boolean expressions," Proc. COMPSAC (Computer Software and Applications) '87, 1987, 278-283.

[Tai93] Tai, K. C., "Predicate-based test generation for computer programs", Proc. Inter. Conf. on Software Engineering, May 1993, 267-276.

[Vou86] Vouk, M. A., Helsabeck, M. L., McAllister, D. F, and Tai, K. C., "On testing of functionally equivalent components of fault-tolerant software", Proc. COMPSAC (Computer Software and Applications) '86, Oct. 1986, 414-419.

[Vou93] Vouk, M. A., and Paradkar, A., "Design and Review of Software Controlled Safety-Related Systems: The NCSU Experience With the Generic Problem Exercise," Proc. Inter. Invitational Workshop on the Design and Review of Software Controlled Safety-Related Systems, Ottawa, June 1993.