PANEL: RESEARCH AND DEVELOPMENT ISSUES IN SOFTWARE RELIABILITY ENGINEERING

Panel Chair:	*Michael Lyu*	(University of Iowa)
Panelists:	*Herbert Hecht*	(SoHaR Inc.)
	Hermann Kopetz	(Technical University of Vienna)
	Douglas Miller	(George Mason University)
	John Musa	(AT&T Bell Labs.)
	Mits Ohba	(IBM Corporation)
	David Siefert	(NCR)

Introduction

Michael R. Lyu, University of Iowa

Computers are bringing revolutionary changes to our life with their involvement in most human-made systems for sensing, communication, control, guidance and decision-making. As the functionality of computer operations becomes more essential and complicated in the modern society, the reliability of computer software becomes more important and critical.

Research activities in software reliability engineering have been vigorous in the past 20 years. Numerous statistical models have been proposed in the literature for the prediction and estimation of software reliability, and many research efforts and paradigms have been conducted for the design and engineering of reliable software. However, there seems to be a gap in between the achievements of software reliability research and the results from software reliability practice. We keep on hearing troublesome software projects, horrible software failures, and misconceptions in software reliability applications.

It is the purpose of this panel to bring together researchers and practitioners of this field to discuss the software reliability problems which will have tremendous impact to our daily life. The panel is expected to raise research and development issues under this concern, to address existing and potential problems, to resolve some misunderstandings and conflicts, and to reach a fundamental basis for the advancement of this field.

The panelists are invited to discuss those topics including, but not limited to, the following:

(1) What are the most urgent needs for software reliability practitioners?

(2) What kind of issues practitioners would like researchers to pursue?

(3) Did practitioners get satisfactory results from software reliability researchers?

(4) What are the most challenging software reliability issues researchers are facing today?

(5) Did researchers gain enough support to perform software reliability research?

(6) What kind of inputs or feedbacks researchers are seeking from practitioners?

(7) What practices should be developed and conducted based on the current research results?

(8) What is the gap in between software reliability modelers and measurers? How to abbreviate it?

(9) What kind of multi-institutional efforts have been, or should be conducted for acquiring software reliability standards, handbooks, benchmarks, database, tools, etc.?

The following sections consist the position statements written by each panelist under the panel title and the suggested topics.

Quantitative and Qualitative Concepts

Herbert Hecht, SoHaR

For Project Managers the reliability of the computing function as a whole is of primary concern, and for that purpose a combined quantitative hardware/software reliability expression is required. The responsibility for hardware and software functions is frequently separated immediately below the project management level, and therefore the project manager also needs separate models for allocating and controlling the achievement of adequate reliability. For these purposes broad statistical reliability metrics are suitable, particularly failures per unit time of computer usage or time unit loss of computer availability due to failures. Examples: failures per CPU-hr or outage-hrs per month.

The software manager is responsible for achieving the statistical reliability goals but in order to know where and how to improve the reliability more specific measurements are required. Quantitative approaches have so far been only of limited use in this domain. Audits, employment of software development and test tools, and test planning are largely guided by purely qualitative considerations. Therefore there exists at present no consistent methodology that permits the software manager to meet the quantitative requirements imposed by systems considerations with the tools at their disposal.

Two activities can bring about a connection between the quantitative and qualitative approaches, and can provide sorely needed advances toward achieving more reliable software. The first activity is the quantitative analysis of failures in terms of software development and test techniques that could have prevented them. The resulting data, particularly if they are weighted by severity of the failure, can provide the software manager with concrete information on the means of improving the reliability of his/her product.

The second step deals with the use of quantitative data as a test termination criterion. The present practice of ending test on the basis of schedule, budget, or (in the very best cases) attainment of a period of failure free operation, provides little useful feedback to the team that developed the software or for the test planning in other projects. Reliability growth measurement during formal test will permit termination on demonstration of a defined reliability level and will also provide insights into the effectiveness of different development and test methodologies.

I will present examples of these integrated practices.

Reliability of Real Time Systems

Hermann Kopetz, Technical University of Vienna

Since my background is in the area of fault-tolerant distributed real-time systems, my view is determined from this position.

In hard real-time systems, i.e., systems where a failure can have catastrophic consequences, a result must be correct, both in the domains of value and time. Since the behavior in the domain of time depends on the properties of the underlying hardware, an integrated software/hardware view has to be taken. The functional correctness of the software per se (i.e., correctness in the value domain) is not sufficient.

Many failures of real-time systems are related to synchronization and performance errors which manifest themselves as 'transient' system failures. In a failure statistics of a complex real-time system [Gebman 1988], it is recorded that less than 10% of the failures observed in the operation of the system can be reproduced within the sophisticated test environment. Similar results have been reported by other manufacturers of real-time systems. This implies that we do not fully understand the character and the interactions of the execution sequences which unfold over time in complex real-time systems and do not know how to build effective test procedures.

This problem has to be attacked from the perspective of design. We have to build real-time architectures that are easier to reason about. Most of the present day real-time systems are event triggered, i.e., as soon as an event occurs, the computer system takes a decision whether to process the task associated with this event immediately or the delay processing until sometimes later. These dynamic scheduling decisions can take a significant amount of processing time, which is then not available for the application software. Every different order of the events can give rise to a different scheduling decision and thus to a different execution sequence. The potential input space of event-triggered systems is enormous. It is difficult to reproduce an input scenario because the exact timing of input cases cannot be controlled easily. There are no methods known which can be applied to reason formally about the timing behavior (i.e. the performance) of complex real-time systems.

If we introduce a time-granularity in the system operation by looking at the events only at predefined points in the time domain (i.e., a time triggered architecture), the plurality of input cases can be substantially reduced. Furthermore, static scheduling strategies become feasi-

ble. The system structure will be more regular, i.e., more predictable and easier to understand and test. The price paid for this reduction in complexity is a reduced flexibility.

We feel that in the field of real-time systems every effort must be made to make the system clear and understandable. In our research on distributed real-time systems [Kopetz 1989] this has always been our primary goal. We have found that time-triggered real-time software is inherently easier to understand and test than event-triggered software. Further research efforts in this area seem to be well justified.

Statistical Issues in Software Reliability Engineering Research and Development

Douglas R. Miller, George Mason University

There are two major issues concerning software reliability: achievement and assurance. They are both very important. Obviously, software in critical applications must achieve high reliability in order for the system to function safely. But it is also necessary to have strong "a priori" assurance that the software is highly reliable before it can be put into use. For example, without reasonable assurance that high reliability has been achieved, flight critical avionics software in commercial aircraft should not be certified for public use.

So, the central focus of Software Reliability Engineering R&D is methodologies for achieving and assuring required levels of software reliability. The goal is reliable software. How do you do it? How do you know when you've done it? Furthermore, what are the most efficient ways to achieve and assure the reliability?

A central idea concerning reliability is "uncertainty." A given piece of software may or may not contain design flaws which will manifest themselves as system failures when the software is used at some time in the future. The point is that uncertainty is inherent to this phenomenon: we do not know if failures will happen and, if they do, when they will happen. To deal with this uncertainty, a scientific approach should be taken. The scientific approach involves experimentation, data collection, statistical modelling and analysis, and drawing inferences and conclusions which will support decisions about developing, testing and using software. The existence of probability seems inevitable here. It is necessary to quantify the uncertainty in terms of probabilities of various events occurring.

Based on information or data concerning software development, testing, previous failures, the usage environment, and any other observables, we would like to estimate (with confidence) the probability that a particular piece of software fails during a given time interval.

Reliability growth models attempt to estimate current reliability and predict future reliability growth for a given piece of software. These models base their estimates and predictions only on past failure times of the given piece of software. IBM's Clean Room used reliability growth models successfully. At the May 1990 Meeting of the IEEE Subcommittee on Software Reliability Engineering, successes were also reported by AT&T, HP and Cray Research. Unfortunately, the reliability growth modelling approach is limited in many ways: The models treat the software as a black box and are only valid for random batch (memoryless) testing or usage. The distribution of usage must be well know. The models do not make use of additional data or information which comes out during testing or usage. The approach does not give useful estimates for extremely high levels of reliability (e.g., avionics software and other safety-related systems).

There are many factors which contribute to the reliability of a piece of software. Case studies such as those sponsored by NASA Goddard's Software Engineering Laboratory explore the effect of various factors on software quality. Factors of interest include different development scenarios, different testing strategies, characteristics of programmers, and others. It can be shown that software quality correlates with various known factors, but calculating reliabilities from these factors seems difficult if not impossible. One very important category of information which should have significant value in predicting reliability of a piece of software is the programmer's personal subjective estimate of its reliability, especially after he has seen and done a post mortem on the first few bugs discovered.

Current practice is often based on engineering judgement. For example, commercial avionics software must be produced following guidelines presented in DO-178A, "Software Considerations in Airborne Systems and Equipment Certification," prepared by Special Committee 152 of the RTCA and currently under revision by Special Committee 167. If appropriate documentation supports compliance, the FAA certifies the software. The actual software is never examined as part of the certification. A major challenge facing the discipline of Software Reliability Engineering involves justifying this

type of approach (also contained in various Military Standards) in some objective, scientific sense.

To summarize: i)For certain classes of software projects, quantitative reliability estimation and prediction is possible (and is done) for individual programs. ii)Through general case studies it is possible to identify factors effecting reliability and thus a get qualitative sense of what constitutes good software development practice. iii)For many critical software systems requiring high reliability, the approach to reliability is very subjective.

It is clear that a quantitative, objective approach to software reliability should be applied to more software projects. This means going beyond the current practice of software reliability growth modelling. The key seems to be: It is necessary to use available data much more efficiently (and imaginatively). There are two categories of data sources: Additional data can be collected (and used) specific to any particular piece of software whose reliability is being assessed. More importantly, there is data from similar and related pieces of existing software; I don't think we know how to make effective use of this data.

The goal is better quantitative understanding (and exploitation of that knowledge) of many software phenomena: behavior of real-time control systems, intricacies of fault-tolerant systems, efficacy of testing, identification of usage distributions, etc. All this knowledge is related to classes of software. (It is necessary to understand more than single software systems individually, one at a time.) Software metrics must be a key feature in this general quantitative understanding, because the similarity between pieces of software must be measured in order to define classes of software.

To progress it is necessary to acquire data. An ideal (but expensive) source is controlled experimentation. For example, NASA Langley continues to sponsor experiments where replicated software is written. A better understanding of replicated batch-processing software has emerged from such experiments. Current experiments should improve understanding of replicated real-time control software. A second general source of data are real software projects. A prime example is the data collected and published by Musa; his data stimulated a flurry of activity in reliability growth modelling. Such experimentation and data collection is crucial. Experimenting and collecting useful data across general classes of software projects is a tremendous challenge.

The Software Reliability Gap: An Opportunity

John D. Musa, AT&T Bell Labs.

We are in the middle of both a problem and an opportunity. I like to call it the "software reliability gap" because the needs of software customers have outrun the current *practice* of software engineering. You can't tell whether they have outrun the technology, because there is much technology that hasn't been refined and applied.

The core of the problem is that intense international competition has made unidimensional needs obsolete. If we only needed to add reliability to software products, we would have many tools and methodologies to help us. The problem is that other customer requirements, such as level of cost and delivery date, would not be met. Customers have *multidimensional* needs that are interdependent and hence must be set and met more precisely than ever before. The precision required can only increase in the future.

Thus measurement is inevitable. Models are also inevitable; we need to know the factors that influence product attributes and how much each of them does, so that the software development process can be controlled to yield the desired objectives for the attributes. In short, competition is creating a technological vacuum or gap.

The principal quality attributes that customers cite as being significant are reliability, cost, and delivery date. Software reliability engineering is the last to develop of the three technologies supporting the measurement and modeling of these attributes. It is the keystone that makes quantitative software quality engineering possible. Since quantitative hardware quality engineering already exists, the development of software reliability engineering also makes quantitative *system* quality engineering possible.

Thus there is an enormous and rare opportunity to fill a widening gap, which makes this an exciting and challenging time.

What must software reliability engineering do to meet the challenge? In my opinion, several general things:

(1) We need to induce a variety of projects to try it. This is already happening, but greater variety would be useful. Care must be taken that it be applied correctly.

(2) The experience on these projects must be recorded, critiqued by others knowledgeable in

the field (to guard against misinformed applications), and published.

(3) Published experience should be organized and digested, so it can be more easily taught to practitioners and future practitioners.

(4) Problems that are blocking further progress and opportunities for new areas of application need to be identified, and they should be addressed by researchers.

These activities clearly offer major possibilities for practitioners, researchers, and educators. People who acquire and use software play an important role in clarifying the needs of the customer that are at the core of the driving forces acting on software reliability engineering.

Can I say anything more specific? I would like to close by entering brainstorming mode and throwing out some thoughts for you to discuss:

(1) We need research to tie software reliability more strongly to the earlier part of the development process. Part of this effort involves determining how fault density is affected by product and process variables.

(2) Little has been done to fulfill the promise of software reliability engineering for evaluating software engineering methodologies and tools. We need to help people do this.

(3) We need data on human and computer resource usage in test, so that resource usage parameters can be determined.

(4) The AIAA software reliability engineering guidelines effort, which includes development of a handbook, looks promising. Because of the diversity of contributors involved, it will be important to devote much effort to interaction between and integration of their views. We don't want a catalog.

(5) We need to strongly support our newsletter and our conference through personal participation in exchanging practical experience and research results. We need to keep the exchange flowing all year through our working committees.

(6) We need software tools (with as many generic elements as possible) to record as large a proportion of failures as possible automatically, particularly in the field but also in test. We need to integrate this system with manually-reported failure systems, but consider implementing the manual reporting online rather than on paper.

(7) The Software Engineering Institute has a methodology for assessing the quality level of software development processes. It does not currently directly include a software reliability engineering program among its assessment criteria. It should, and we should discuss with them how to add it.

I hope you will not only discuss these ideas here, but chew on them later as well. I hope you will add to this necessarily partial list of opportunities for action. I hope you will then seize some of them that appeal to you, and return as significant contributors next year or the year after.

Software Reliability Engineering from Japanese Perspective

Mits Ohba, IBM Corporation

"The wave comes from the East."

Both the computer technology and the quality control method were invented and matured in the US, and they were brought into Japan later. Japan has so far caught up quickly and become competitive in both areas. Especially, Japan is viewed as the leader in the area of quality control and quality management.

"Technology transfer begins when it is imported."

If we carefully review the processes by which Japan has caught up and gone further, we can find some similar patterns of technology development. The processes generally begin at the importing phase where technology is investigated and evaluated. Then there is the deployment phase, the migration phase, and finally, the Japanization phase.

"How does it go through?"

The deployment phase is the phase where the imported technology is widely used and the know-hows associate with it are accumulated. The migration phase is the phase where components of the technology are adjusted for the target environment(s). The Japanization phase is the phase where something additional and unique to Japan is added to the technology.

"How has Japanese software engineering evolved?"

Software engineering is a case in point. It was introduced into Japan in 1977, which was two years later

than the first IEEE Transaction on Software Engineering issued. Two years were spent on the importing phase followed by two years of deployment. The migration phase began in 1982 and lasted six years. The Japanization phase began in 1988. An example of the Japanization phase is what has become known as the "Software Factory" concept.

"Software reliability research is not an exception."

As a domain of research, software reliability engineering is not an exception to the Japanese process. The earlier work done in the US by Musa, Goel and Okumoto drew the attention of Japanese reliability researchers as their new field of study.

"What have Japanese researchers done in this field?"

To date they have: 1) evaluated the basic models proposed by the American researchers by applying them to real project data, 2) modified the models in order to fit the data, 3) developed new models by examining the implication of data and the assumptions of the basic models, and 4) addressed the new research issues of models to be resolved.

"Software factory did not need theories."

On the other hand, software reliability engineering as a practice has evolved differently. It was begun as a branch of software quality control practices in order to determine whether a product developed by a vendor was acceptable. The logistic curve model and the Gompertz curve model were widely used in the industry and became de facto standard models for software factories.

"Technology transfer is really the problem."

The implementation of the theory which has been developed by Japanese researchers is very slow. This is because the old models, with which the practitioners are familiar, are still sufficient for their needs. They will not change as long as the old practices work or until they recognize the advantages of the new theory. This is similar to the fact that people had believed the stars were rotating.

"How can we convince the people that the earth rotates?"

The most serious issue of software reliability engineering as a practice in Japan is the education of the people. It is similar to teach them that the earth rotates, not the stars. The models are not crystal balls. Prediction is made based on a set of assumptions. If the assumptions are not valid, a model based on them becomes a great nonsense. The Gompertz curve fits most of practical project data because of its flexibility. But, no one can explain what the model really means.

"Why do we believe that the earth is rotating?"

The most serious issue as a domain of research is to explain the relationship between test cases and reliability growth using reasonable models, which is also similar to explain the reason why the earth seems to be rotating. What software reliability growth tells is characterization of the state of software under evaluation. It does not tell how we can improve testing. Obviously, time is not the real factor for improving software reliability during the test phase.

"Can measurements and data be standardized?"

A serious issue for both practitioners and researchers is to establish standard ways of measuring software reliability in practice. The models are based on a set of assumptions. The models should be categorized based on 1) what they can predict (e.g., MTTF, number of errors), 2) what type of data they need (e.g., time between failures, number of failures between observations), 3) what assumptions they are based on, and 4) what type of software they can analyze.

Back To The Future

David Siefert, NCR

For the past 20 years, Software Engineering has provided us with the capability for producing highly reliable software. Software reliability is achieved, in part, through the applied discipline of standardized practices, methodologies, tools, and processes comprising the "science" of Software Engineering. Today, dependence on automation is greater than at any point in time in the world's history. Highly reliable products are expected and assumed! The very nature of the level of sophistication and complexity of modern systems are intended to be transparent to the end-user.

Applying Software Reliability Engineering Disciplines

Interestingly, the same practices, methodologies, etc. that lead to the development of reliable software are also the downfall! Why after all these years of "learning" is the world still not applying and improving Software Engineering disciplines etc.? Why do practitioners still develop and maintain software based upon the

approaches used 20 years ago (lack of applied discipline)? Why is it that researchers do not yet know exactly what is the minimum that should be done to develop reliable software? In support of consistently producing reliable software, why after 20 years is there still not a national database leading to the consistent project data collection, analysis, and ultimate determination of practices, tools, and therefore required disciplines? Shouldn't a Software Engineering "Bluebook" exist?

Software Reliability Engineering is addressed in the following two ways:

(1) Technical Aspects of Software Reliability

Technical software reliability consists of many items. Determining reliability goals is one activity. Reliability goals are typically referred to in "technical" terms. These technical terms are placed in product specifications. As it pertains to Software Reliability Engineering, these terms or goals are then tracked through product production to the achievement of the goals. The environment that the software was produced in, plays a significant impact on the results. These specified reliability goals often are determined through the application of software reliability models. An AIAA effort addressing Software Reliability is in the process of providing guidance to industry on which models to use and when. The computing industry has yet to standardize these specific models.

(2) End-User Software Reliability

The second form of Software Reliability Engineering is that of the end-user. The technical specifications which include the software reliability goals are expected to be mapped directly to the end-user's needs and expectations. Too often there is no known methodology to take qualitative and rather subjective unstructured feedback from the end-user and transform them into quantifiable and technically oriented input for use in determining software reliability. Without this methodology, there will remain to be software reliability difficulties. Meeting "specification" infers meeting the end-user's expectations. Meeting specification is certainly one essential form of measurement. Technical specifications are the result of analysis of the end-user's expectation - not the other way around. Too often the technical specification and the end-user's expectations are distinctly separate with no relationship between each other. This results in minimal confidence that the product will achieve it's expectations.

Environmental issues are also important. To understand software reliability, one must understand the environment software resides. The environment for software is systems! System components include other software and hardware. Reliability should be computed or budgeted in such a manner that reliability for each of the components of the computer environment can be determined, evaluated, measured, and tracked separately. Reliability should also address a "total" system or enterprise-wide solution. Typically, the end-user is affected by using or experiencing the "total" system. They typically have no ability to decipher the type of defect or anomaly that has occurred. It is not clear that they should. At any rate, Software Reliability Engineering needs to address the "total" system as well as the individual system components.

The Software Engineering community has reliability models that lead to establishing reliability goals. "High Confidence" goals (outputs) produced through the use of these models are dependent upon past history. This history should be retained in the form of a database. Interestingly, no new significant software estimation models have been revealed in the past 5 years. Without the use of such databases as input to and the "tuning" of such models, the community is no closer to estimating with high confidence levels the goals produced from the models as was able to be attained 5 years ago. The goals produced through the use of these models may not be any better than the "guess" of you or I.

Besides past history, the technically specified software reliability goals are established and dependent on some basic items of information:

- How is end-user's "needs" quantified?
- What is a software error, fault, and failure?
- What are the categories of software?
- How is Defect and Fault Density computed?
- What and how is line-of-code or Function Point, by language, determined?
- How is line-of-code or Function Point translated between languages?
- How is Defect Density affected by software production environmental issues?
- How is software to be tracked?

Recommendations in Improving Software Reliability

- For Practitioners:

(1) Practitioners must apply the disciplines considered to Software Engineering. Techniques, methods, tools, etc. as associated with planning, design, development, testing (including verification and validation), should be learned and rigidly applied.

(2) Each software production (or maintenance) organization should develop and maintain a Software Engineering Environment Process (SEEP). This process should consist of all disciplines, tools, etc. actually used in the production of the software - including the measurement systems, of which software reliability is a part.

(3) Practitioners should develop a database of past projects. The database should consist of such information as: the environment that produced the software, skill and types of personnel producing the software, Defect Densities, etc. This database is to be used as a basis for a Software Reliability Measurement Program (SRMP) and positioning for continuous improvement in Software Engineering.

(4) A software reliability measurement program (SRMP) should be put into place that consists of measures that address both the scope of the Software Engineering Environment Process and specific product related results. Measures should consist of indicator measures, e.g., Test Coverage and estimator measures - models to estimate reliability. The measurement program should consist of a methodology that addresses the use of the models beginning with the "how to" develop reliability goals and ending with an approach of a project post mortem. The previously mentioned database would maintain all data. The database would provide for causal root cause analysis and process improvement of the Software Engineering Environmental Process.

- For Computer Scientist Researchers:

(1) Researchers are to develop and maintain a national database (see above). The information contained in the database as previously noted should contain both product and environmental information. Researchers should evaluate the information in such a manner as to determine the best practices, methods, required skills etc. to continuously improve software reliability.

(2) Researchers should provide standards on such subjects as: language constructs, line-of-code definitions, Function Point, etc.

(3) Researchers should determine minimum impacts as to how to conclude with deriving "high confidence" software reliability goals, etc. Models are to be evaluated and maintained.

(4) Researchers should also determine education curricula for software engineering enabling the continuous achievement of high confidence reliable software.

(5) Researchers should determine how to quantify results from evaluating user's needs. These results are used as input into various different reliability tools, models, etc. as discussed earlier.

(6) Researchers should establish and maintain a "Blue Book for Software Engineering."

Concluding Comments

The world continues to embrace higher and higher levels of technology. Software is at the heart of the demand for complex features and functions which are packaged to make the complexity transparent to the end-user. High confidence software reliability is in jeopardy. Software Engineering processes that consist of disciplines, tools, methods, etc. are not being utilized consistently. The science of Software Engineering is not being practiced.

A need exists to focus on the basics; in the simplest form of understanding software and Software Engineering. Data needs to drive decisions. Attaining highly reliable software - consistently - positioned through processes for the purpose of improvement is essential. Researchers need to provide the "data driven" credibility in the baseline evaluations of software and software environments (and processes). Researchers need to see that the appropriate Software Engineering disciplines are applied - consistently and appropriately, evaluating the results, and improving the disciplines and processes.

The disciplines exist in the form of Software Engineering to produce reliability software! The discipline and formality required to achieve the results remain to be the challenge! The solution is: *"go BACK and apply the discipline TO get to THE FUTURE..."*