Workshop on Dependability Benchmarking

WORKSHOP #4

Co-Chairs

Philip Koopman
CMU, USA
<koopman@cmu.edu>

Henrique Madeira
Univ. of Coimbra, Portugal
<henrique@dei.uc.pt>

Classical features such as raw performance and functionality have long driven the computer industry to improve their products. But now, dependability and maintainability are seen as equally important. While there are relatively straightforward ways to evaluate and compare performance and functionality of different systems or components, the evaluation of dependability and maintainability features is much more difficult. Among the challenges that must be addressed are: incorporating the effects of software failures, characterizing the dependability of opaque off-the-shelf hardware and software components, including the effects of typical maintenance, operational, and configuration management procedures, and accommodating the fact that different application areas have different requirements for the various factors influencing dependability.

The goal of the Dependability Benchmarking Workshop is to provide a forum for the computer industry and academia to discuss problems associated with the evaluation and characterization of dependability and maintainability of components and computer systems. The identification of dependability benchmarking measures and the essential technologies for dependability benchmarking, including both experimental measuring and modeling technologies, are central aspects of this large discussion meant to garner ideas on practical and cost-effective ways to evaluate dependability and maintainability features.

This proceedings preprint contains short papers, work-in-progress reports, and position papers. These workshop contributions are being published in the:
Supplement of the 2002 International Conference on Dependable Systems and Networks.

Workshop on Dependability Benchmarking

Philip Koopman
ECE Department & ICES
Carnegie Mellon University
Pittsburgh, PA, USA
koopman@cmu.edu

Henrique Madeira
Information Engineering Deptartment
University of Coimbra
Coimbra, Portugal
henrique@dei.uc.pt

Classical features such as raw performance and functionality have long driven the computer industry to improve their products. But now, dependability and maintainability are seen as equally important. While there are relatively straightforward ways to evaluate and compare performance and functionality of different systems or components, the evaluation of dependability and maintainability features is much more difficult. Among the challenges that must be addressed are: incorporating the effects of software failures, characterizing the dependability of opaque off-the-shelf hardware and software components, including the effects of typical maintenance, operational, and configuration management procedures, and accommodating the fact that different application areas have different requirements for the various factors influencing dependability.

The goal of the Dependability Benchmarking Workshop is to provide a forum for the computer industry and academia to discuss problems associated with the evaluation and characterization of dependability and maintainability of components and computer systems. The identification of dependability benchmarking measures and the essential technologies for dependability benchmarking, including both experimental measuring and modeling technologies, are central aspects of this large discussion meant to garner ideas on practical and cost-effective ways to evaluate dependability and maintainability features.

This workshop is the outcome of the first two years of work of the IFIP 10.4 WG SIG on Dependability Benchmarking (SIGDeB) [SIGDeB02]. That SIG was formed in November 1999 under the IFIP 10.4 WG to promote the research, practice, and adoption of benchmarks for computer-related system dependability. Koopman & Madeira were the founding co-chairs.

The SIGDeB charter focusses on four areas in particular:

- Exchanging ideas about dependability benchmarking, including researchers and practitioners from universities, industry, and government agencies.
- Documenting the state of the art for dependability measurement and benchmarking.
- Creating lists of issues that must be resolved to advance dependability benchmarking to a mature science.
- Eventually, proposing a mechanism and agenda for a group to propose dependability benchmarks.

Of course those are long term objectives that will require far more than two years of work to accomplish. But several of the papers presented are the results of SIGDeB collaborations that have made progress. Beyond that, this workshop represents the first focussed exchange of ideas about dependability benchmarking in a public forum.

While the SIGDeB has been working, two other research programs have been created to address related areas: DBench and HDCP. Both DBench and HDCP have membership overlaps with SIGDeB, but are different in purpose.

DBench is a 3-year European research program to "define a conceptual framework and an experimental environment for benchmarking the dependability of COTS and COTS-based systems" [DBench02]. It emphasizes the areas of dependability measurement, identification of malfunctions/weaknesses, tuning components to improve dependability, and dependability comparisons. Most measurements use fault injection, and the majority of participants come from the fault tolerant computing community.

The HDCP (High Dependability Computing Program) is a long-term collaboration of US universities and NASA, with expected industry participation, to "ensure that the software we create meets the ever more challenging requirements of continuous operation, safety critical reliability, high integrity and high security" [HDCP02]. HDCP is formed largely of researchers from the software engineering community.

We are pleased that the papers being presented represent the SIGDeB, DBench, HDCP, and other researchers not formally affiliated with any of those groups.

References:

[DBench02] Dependability Benchmarking Project, *http://www.laas.fr/DBench/* accessed April 4, 2002.

[HDCP02] High Dependability Computing Program, *http://west.cmu.edu/research/hdcp.html* accessed June 11, 2002.

[SIGDeb02] SIG on Dependability Benchmarking *http://www.dependability.org/wg10.4/SIGDeB/* accessed April 4, 2002.

Workshop on Dependability Benchmarking

Tuesday, June 25, 2002

Co-chairs: Philip Koopman (Carnegie Mellon, USA), Henrique Madeira (Univ. Coimbra, Portugal)

Program Committee: Jean-Paul Blanquart (Astrium, France), Michel Cukier (Univ. Maryland at College Park, USA), Karama Kanoun (LAAS-CNRS, France), Johan Karlsson (Chalmers Univ. Sweden), Brendan Murphy (Microsoft Research, UK), Don Wilson (Compaq, USA)

Progress on Defining Standardized Classes for Comparing the Dependability of Computer Systems

Don Wilson
NonStop Enterprise Lab.
Hewlett-Packard Company
Aptos, CA, USA

Brendan Murphy
Microsoft Corporation
Cambridge
UK

Lisa Spainhower
International Business Machines
Poughkeepsie, NY
USA

Abstract

A number of the industrial partners of the IFIP WG 10.4 Dependability Benchmarking SIG (SIGDeB) have identified a set of standardized classes for characterizing the dependability of computer systems. The proposed classification system seeks to enable comparison of different computer systems in the dimensions of availability, data integrity, disaster recovery, and security. Different sets of criteria are proposed for computer systems that are used for different application types, e.g. transaction processing, process control, etc. This paper describes the classification system, and gives a progress report on the work to fill in the details of the classification criteria

1. Introduction

As computer systems become more and more continuously integrated into the daily activities of business, engineering, and scientific users, there is increased interest in being able to evaluate and compare the dependability of these systems. Considerable research has been done in an effort to establish benchmark tests for this purpose (e.g., see [1-5]), usually based on some form of fault-injection testing focused on single computers [6,11].

In spite of these efforts, and even considering that fault injection techniques are commonly used by developers to assess and tune their designs (e.g., see [7-10]), nothing has emerged which has gained even modest adoption in the industry for making comparisons among systems. Researchers acknowledge that:

- emulated faults will not represent the variety and scope of actual field faults [6, 14],
- fault injection cannot predict actual availability or MTBF [13],
- comparison of dissimilar architectures is extremely problematic [7, 13, 14].

The authors, working as members of the IFIP WG 10.4 Dependability Benchmarking SIG (SIGDeB) [15], are proposing a different method for making dependability comparisons. This method is to create a standardized classification system that could rate systems in each of the dimensions that affect dependability.

2. Dimensions of Dependability

Unlike performance benchmarks, which need to compare only the rate at which specific, pre-defined work gets done, dependability comparisons must consider many different aspects [12]. Is the system accessible when needed? Are the results correct? Is the data protected from physical hazards and unauthorized access?

To simplify the creation of comparison classes, it is useful to separate the various threats to dependability and treat them as different dimensions of the problem space. The authors have chosen Availability, Data Integrity, Disaster Recovery, and Security as the dimensions to be considered. When a system is rated according to the proposed classification scheme, it would receive an independent rating for each of these dimensions.

These dimensions are not truly independent; for example corrupted data could easily make an application unavailable. Thus, the problem space could certainly be divided up differently. However, these dimensions were chosen because they are readily understandable and are important concerns to users. It may also be argued that there are still other dimensions to the problem, such as physical safety. The structure of this proposal makes it simple to add further dimensions if they are deemed useful enough to the user community.

Unlike performance benchmarks, it was felt that dependability assessments should not restrict the system under test to be a single computer. Very few customers, requiring a highly dependable solution, would implement it on a single computer with a single data store. Any solution would likely be designed with a minimum of two interconnected computers, with some form of highly dependable storage configuration. As the system configuration is unrestricted, then any comparison of different systems should include the cost of each solution.

Transaction Processing	Typical applications are order processing, automated billing, credit authorization, automated retail, securities trading, reservations.
Message Handling	Typical applications are telephony, email, packet switching and routing, protocol conversion.
Process Control	Typical applications are manufacturing control, embedded device controllers, servo systems, network and system management.
Search and Retrieval	Typical applications are web-page serving, decision support, classical business reporting, broadcasting.
Analytical Calculation	Typical applications are simulation, modeling, and scientific data reduction.

Table 1: Application Types

3. Application Space

Users of different types of applications tend to have differing priorities when it comes to dependability.

A telephony application prizes availability and responsiveness very highly, but can afford minor data errors.

A stock trading application prizes data integrity above all else, requires high availability, must often adhere to strict securities regulations, but may have frequent off-hours where maintenance can be done.

A factory control system finds availability and accuracy essential, but is not concerned about operating during a power failure.

The authors have accordingly divided the application space into types that appear to have similar dependability requirements, as listed in Table 1.

The proposed classification scheme will define a different set of classes for each application type, on each dependability dimension. This structure is shown in Table 2.

The boundaries between classes are intended to be natural breakpoints in the spectrum of user-perceived availability requirements. The highest class is always intended to be essentially perfect behavior, whether or not it is achievable with current technology.

Application Type	Availability Classes	Data Integrity Classes	Disaster Recovery Classes	Security Classes
Transaction Processing	1. Perfection 2a. Retryable Workload 2b. Retryable / Planned Outages 3a. Delayable Workloads 3b. Delayable / Planned Outages 4. Enhanced 5. Unprotected	1. Perfection 2. Complete Detection 3. Enhanced 4. Unprotected	1. Perfection 2. Resume with Delay 3. DB Preservation 4. Unprotected	1. Perfection 2. Less 3. Lesser 4. Unprotected
Message Handling	Classes 1 to n	Classes 1 to n	Classes 1 to n	Classes 1 to n
Process Control	Classes 1 to n	Classes 1 to n	Classes 1 to n	Classes 1 to n
Search and Retrieval	Classes 1 to n	Classes 1 to n	Classes 1 to n	Classes 1 to n
Analytical Calculation	Classes 1 to n	Classes 1 to n	Classes 1 to n	Classes 1 to n

Table 2: Classification Structure

Class	Basic Requirements
1. Perfection	• The system must be able to correctly process every transaction submitted to it, within normal response times, all of the time. • All single failures, corrective actions, maintenance, and other potentially disruptive events are handled by the system without any noticeable effect on the users.
2a. Retryable Workload	• The system must be able to correctly process every transaction submitted to it, either within normal response times, or after a brief delay to recover from a disruptive event. • Disruptive events may not cause users to have to re-establish connection to the system nor to suspend submitting transactions. • No transactions may be lost nor may the database be left with inconsistent data due to incomplete transactions. • The system may request that incomplete transactions be re-submitted after recovering from an event, but the number of such transaction may not exceed 1 second worth of Normal Transaction Processing Capacity. • The Recovery Period for disruptive events may not exceed 10 minutes. Average capacity during the period may not be below 80% of Normal Transaction Processing Capacity.
2b. Retryable / Planned Outages	• Same as 2a, but the system may rely on taking an outage to perform certain planned operations, maintenance, repair, and upgrade tasks.
3a. Delayable Workloads	• The system must be able to respond to disruptive events and be ready to process transactions within 5 minutes. • The recovery period for disruptive events may not exceed 25 minutes. Average capacity during the recovery period may not be below 60% of Normal TP Capacity. • No transactions may be lost, nor may the database be left with inconsistent data due to incomplete transactions. • The users may be required to re-establish connection to the system and to identify and re-submit transactions that did not complete before the event. The number of such transactions may not exceed 1 second worth of Normal Transaction Processing Capacity.
3b. Delayable / Planned Outages	• Same as 3a, but the system may rely on taking an outage to perform certain planned operations, maintenance, repair, and upgrade tasks.
4. Enhanced	• The user can evaluate, for cost and effectiveness, the individual features that are intended to improve availability. • Disruptive events may result in a total outage of a system (requiring user intervention to perform reboot and recovery). • No transactions may be lost, nor may the database be left with inconsistent data due to incomplete transactions.
5. Unprotected	• None.

Table 3: Basic Requirements for Transaction Processing Availability Classes

4. Example: Availability for Transaction Processing

The authors have been developing the details of the proposal for Transaction Processing applications.

A discussion of the classes defined for Availability will illustrate how the classification system will function and suggest the work that still needs to be done for other application types and dimensions.

Systems are assigned to one of five *Classes* by meeting *Basic Requirements* over a set of *Factors* that affect availability. The Classes and Basic Requirements are shown in Table 3. The Factors are shown in Table 4.

To qualify for a specific rating, the system is evaluated against a list of criteria for each availability factor. An example of the evaluation criteria, for the Hardware Failure Factor, is shown in Table 5.

The criteria vary widely in scope, so each will need to be evaluated by its own appropriate method, e.g. a standard benchmark, a design audit, or analysis of field data. Since there are very many criteria to be satisfied, it is proposed that systems may still be evaluated for a given class, even if their designs do not meet all of the criteria, as long as all exceptions are disclosed.

One of the drawbacks of standardized performance benchmarks is the high cost of conducting and certifying a test run. Since a comprehensive dependability benchmark would be more complex, the implementation cost would be even more discouraging and the authors believe that vendors would not bear it.

Factor	Description
HW Failure	Intermittent or permanent HW fault, including design errors that manifest as component failure.
SW Failure	Improperly designed, built, or installed software that results in a detected failure that takes resources out of use.
Environmental Failure	A failure to provide power, cooling or other environmental requirement of the system.
HW Repair or Upgrade	Repair failed components, re-integrate repaired components, or install newer version (same form, fit, function).
SW Repair or Upgrade	Action to replace a faulty component, or install newer versions (same external interfaces); or to revert to previous versions.
Operating Configuration Change	Action to adjust system parameters for performance tuning, policy administration, access control, etc.
System Maintenance	Action required to maintain the integrity of the application, e.g. data backups, log dumps, resource monitoring.
Capacity Expansion (or Reduction)	Scaling the system for changes in volume of usage, e.g. additional HW, new SW, database reorganization.
System Management Skills	The level of skills, training, process control, and other human factors required to obtain the desired availability.
Denial of Service Attack	Attempt by un-authorized users to render the system inaccessible or unusable

Table 4: Factors that Affect Availability

The intention of this classification system is that the evaluation would be self-certified. Vendors or others wishing to classify a particular system would do the evaluation and publish the results, answering a set of standardized questions or performing tests that validate the evaluation criteria. This approach rests on the assumption that vendors would not risk their reputations, nor product liability claims, by making false statements against very specific standards.

5. Progress on Specifying the Classification System

To date, initial drafts have been written for Availability, Data Integrity, and Disaster Recovery for Transaction Processing applications. These drafts include classes, factors, minimum standards and evaluation criteria. Evaluation methods have not yet been proposed. A sub-committee of SIGDeB is currently doing a trial evaluation of a specific system to see if a standardized set of questions or evaluation tests can be developed.

Classes	Perfection	Retryable Workloads	Delayable Workloads	Reduced Impact of Failure
Minimum Standard	• No single HW failure may cause a properly configured system to violate the Basic Requirement.			
Required Disclosures	• What, if anything, is required in the application code or configuration to meet the standards. • Any cases where the system might lose its ability to recover from a HW failure, but not report this condition to the user (e.g., a backup resource fails, but the system does not detect the failure until it attempts to use the resource in a recovery action). • History of user-reported HW defects that violated minimum standards. • Any exceptions (to the minimum standards) in the system design, with quantified impacts.			• Cost / benefit of features to be evaluated
Comparative Measurements	• How long is the system susceptible to a second (unprotected) failure, i.e., while the first failure is detected and repaired? • Frequency of failures that cause a recovery process or remove a resource from the system. • Duration and impact of the recovery process.			• Frequency of failures that benefit from each feature

Table 5: Evaluation Criteria for Hardware Failure

References

[1] D. P. Siewiorek, J. J. Hudak, B.-H. Suh and Z. Segall, “Development of a Benchmark to Measure System Robustness”, in *Proc. 23rd Int. Symp. on Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, 1993, pp. 88-97 (IEEE CS Press).

[2] T. K. Tsai, R. K. Iyer and D. Jewitt, “An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems”, in *Proc. 26th Int. Symp. on Fault-Tolerant Computing (FTCS-26)*, Sendai, Japan, 1996, pp. 314-323 (IEEE CS Press).

[3] P. Koopman and J. DeVale, “Comparing the Robustness of POSIX Operating Systems”, in *Proc. 29th Int. Symp. on Fault-Tolerant Computing (FTCS-29)*, Madison, WI, USA, 1999, pp. 30-37 (IEEE CS Press).

[4] A. Brown and D. A. Patterson, “Towards Availability Benchmarks: A Cases Study of Software RAID Systems”, in *Proc. 2000 USENIX Annual Technical Conference*, San Diego, CA, USA, 2000 (USENIX Association).

[5] J. Arlat, J.-C. Fabre, M. Rodríguez and F. Salles, “Dependability of COTS Microkernel-Based Systems”, *IEEE Trans. on Computers*, vol. 51, no. 2, pp. 138-163, February 2002.

[6] J. V. Carreira, D. Costa and J. G. Silva, “Fault Injection Spot-checks Computer System Dependability”, *IEEE Spectrum*, vol. 36, pp. 50-55, August 1999.

[7] R. Chillarege and N. S. Bowen, “Understanding Large System Failures — A Fault Injection Experiment”, in *Proc. 19th Int. Symp. on Fault-Tolerant Computing (FTCS-19)*, Chicago, IL, USA, 1989, pp. 356-363 (IEEE CS Press).

[8] A. M. Amendola, L. Impagliazzo, P. Marmo and F. Poli, “Experimental Evaluation of Computer-Based Railway Control Systems”, in *Proc. 27th Int. Conf. on Fault-Tolerant Computing Systems (FTCS-27)*, Seattle, WA, USA, 1997, pp. 380-384 (IEEE CS Press).

[9] C. Constantinescu, “Validation of the Fault/Error Handling Mechanisms of the Teraflops Supercomputer”, in *Proc. 28th Int. Symp. on Fault-Tolerant Computing (FTCS-28)*, Munich, Germany, 1998, pp. 382-389 (IEEE CS Press).

[10] H. Madeira, R. Some, F. Moreira, D. Costa and D. Rennels, “Experimental Evaluation of a COTS System for Space Applications”, in *Proc. Int. Conference on Dependable Systems and Networks (DSN-2002)*, Washington, DC, USA, 2002 (IEEE CS Press).

[11] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie and D. Powell, “Fault Injection and Dependability Evaluation of Fault-Tolerant Systems”, *IEEE Transactions on Computers*, vol. 42, no. 8., pp. 919-923, August 1993.

[12] J.-C. Laprie, Ed., “Dependability: Basic Concepts and Terminology”, (Dependable Computing and Fault Tolerance, vol. 5, A. Avizienis, H. Kopetz and J.-C. Laprie, Eds.), *Vienna: SpringerVerlag*, 1992.

[13] M. Hsueh, T. Tsai and R. K. Iyer “Fault Injection Techniques and Tools”, *Computer*, vol. 30, no. 4, pp. 75-82, April 1997.

[14] J. Clark and D. Pradhan, “Fault Injection: A Method for Validating Computer-System Dependability”, *Computer*, vol. 28, no. 6, pp. 47-56, June 1995.

A Framework for Dependability Benchmarking

Karama Kanoun[*], Henrique Madeira[**] and Jean Arlat[*]
[*] LAAS-CNRS, 7, Avenue du Colonel Roche, 31077 Toulouse Cedex 4 — France
[**] DEI-FCTUC, University of Coimbra, 3030 Coimbra — Portugal
(kanoun@laas.fr, henrique@dei.uc.pt, arlat@laas.fr)

Abstract

This paper outlines a framework for defining dependability benchmarks of computer systems that is being investigated by the European project DBench[*]. The multiple dimensions of the problem are classified, then examples of benchmarking scenarios are presented. Finally, some research issues are discussed.

1. Introduction

The goal of benchmarking the dependability of computer systems is to provide generic ways for characterizing their behavior in the presence of faults. Benchmarking must provide a uniform, and repeatable way for dependability characterization. The key aspect that distinguishes benchmarking from existing evaluation and validation techniques is that a benchmark fundamentally should represent an agreement that is accepted by the computer industry and by the user community. This technical agreement should state the measures, the way these measures are obtained, and the domain in which these measures are valid and meaningful. A benchmark must be as representative as possible of a domain. The objective is to find a representation that captures the essential elements of the domain and provides practical ways to characterize the computer dependability to help system manufacturers and integrators improving their products and end-users in their purchase decisions.

The DBench project aims at defining a conceptual framework and an experimental environment for dependability benchmarking. This paper summarizes our first thoughts on the conceptual framework, as investigated in [1]. Section 2 identifies the various dimensions of dependability benchmarks. Section 3 presents some benchmarking scenarios. Section 4 introduces some research issues.

2. Benchmark Dimensions

The definition of a framework for dependability benchmarking requires first of all the identification and the clear understanding of all impacting dimensions. The latter have been grouped into three classes as shown in Figure 1.

- *Categorization dimensions* concern system and benchmark context description and thus allow organization of the dependability benchmark space into different categories.
- *Measure dimensions* identify dependability benchmarking measure(s) to be assessed depending on the choices made for the categorization dimensions.
- *Experimental dimensions* include all aspects related to experimentation on the target system to get the base data needed to obtain the selected measures.

Categorization dimensions

The *target system nature and application area* impact at the same time measures to be evaluated and measurements to be performed on the target system to obtain them.

The benchmark context includes:

- *Life-cycle phase* in which the benchmark is performed and the phase for which the results are intended..
- *Benchmark user:* person or entity who actually uses the benchmark results.
- *Benchmark scope*: results can be used either to characterize system dependability capabilities in a qualitative manner, to assess quantitatively these capabilities, to identify weak points or to compare alternative systems.
- *Result purpose*: External use involves standard results that fully comply with the benchmark specifications, for public distribution, while internal use is intended for system validation and tuning.
- *Benchmark performer*: Person or entity who performs the benchmark (e.g., manufacturer, integrator, third party, end-user). These entities have i) different visions of the target system, ii) distinct accessibility as well as observability levels for experimentation, and iii) different expectations from the measures.

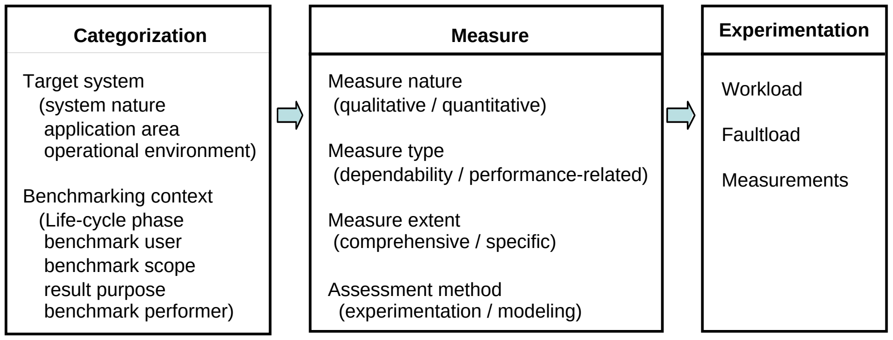


Figure 1 - Dependability benchmarking dimensions

[*] IST 2000-25425 http://www.laas.fr/DBench

Measure dimensions

The various usage perspectives impact the type (and the detail) of benchmark measures. Typically, end-users are interested in dependability measures defined with respect to the expected services (i.e., comprehensive measures, such as availability), while manufacturers and integrators could be more interested in specific measures related to particular features of the target system (e.g., error detection coverage). Comprehensive measures might be evaluated based on modeling.

Experimentation dimensions

The operating environment traditionally affects very much system dependability. The workload should represent a typical operational profile for the considered application area. The faultload consists of a set of faults intended to emulate faults that the system would experience in real-life situations. This is clearly dependent on the operating environment for the external faults, which in turn depends on the application area. Internal faults (e.g., software and some hardware faults) are mainly determined by the actual target system implementation. A dependability benchmark must include standards for conducting experiments and to ensure uniform conditions for measurement. These standards and rules must guide all the processes of producing dependability measures using a dependability benchmark.

3. Benchmark Scenarios

The set of successive steps for benchmarking dependability together with their interactions form a benchmarking scenario. Benchmarking starts by an analysis step for allocation of specific choices to all categorization and measure dimensions. The selection of the experimental dimensions is then achieved based according to theses choices.

Figure 2 gives a high level overview of the activities and their interrelations (represented by arrows A to E) for system dependability benchmarking. To illustrate how this general framework can be used in real situations, we have selected four examples of benchmark scenarios (S1 to S4).

S1: Benchmark based on experimentation only

S1 includes analysis and experimental steps, and link A. It is actually an extension of the well-established performance benchmark setting.

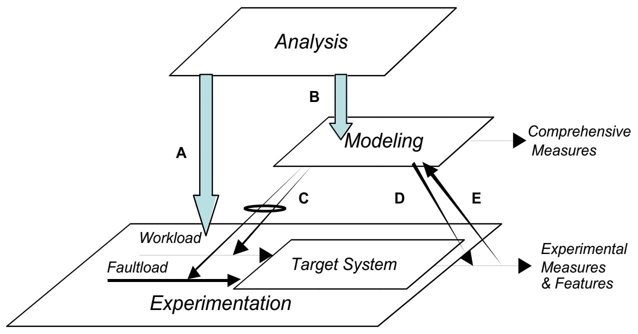


Figure 2 - Dependability benchmarking scenarios

S2: Experimentation supported by modeling

S2 includes the three steps and links A, B, C and D. The experimentation is guided, at least partially, by modeling.

S3: Modeling supported by experimentation

S3 includes the three steps and links A, B and E. Experimentation supports model validation and refinement. The expected outputs are comprehensive measures obtained from processing the model(s). However, the experimental measures and features, assessed for supporting modeling, may be made available from the experiments.

S4: Modeling and experimentation

S4 is a combination of S2 and S3 and includes all steps and all links. Its outputs are experimental measures and features, and comprehensive measures based on modeling.

4. Some Research Issues

Representativeness is a crucial concern, as benchmark results must characterize the addressed aspects of the target system in a realistic way. Regarding established performance benchmarks, the problem is reduced to the representativeness of performance measures and of the workload. For dependability, it is also necessary to define representative dependability measures and representative faultloads. Although the problem seems clearly more complex than for performance benchmarks, the pragmatic approach used in the established performance benchmarks offers a basis for identifying adequate solutions for dependability benchmarking representativeness.

It is worth mentioning that many technical problems still need to be resolved. The subsequent points summarize crucial research issues.

- The adoption of the workloads of established performance benchmarks is the starting point for the definition of workloads. However, some work still has to be done. For example one has to check whether the way the application spectrum as partitioned by the performance benchmarks is adequate for this new class of dependability benchmarks.
- The definition of representative faultloads encompasses specific problems that are currently being studied
- Definition of meaningful measures. In particular, special attention should be paid to confidence and accuracy of measurements and the possible impact of the measurements on the target system behavior (intrusiveness).

Finally, dependability benchmarks must meet certain properties to be valid and useful. In fact, benchmarks can be accepted only if results can be repeated and reproduced by another party.

Acknowledgement

All partners of the DBench project contributed to the work presented in this paper. The list of names is too long to be included in a two-page position paper.

Reference

[1] H. Madeira, K. Kanoun, J.Arlat, Y. Crouzet, A. Johanson, R Lindström, "Preliminary Dependability Benchmark Framework", DBench deliverable, September 2001. Available at http://www.laas.fr/dbench/delivrables.html

Including the Human Factor in Dependability Benchmarks

Aaron B. Brown, Leonard C. Chung, and David A. Patterson
Computer Science Division, University of California at Berkeley
387 Soda Hall #1776, Berkeley, CA, 94720-1776
{abrown,leonardc,pattrsn}@cs.berkeley.edu

Abstract

We describe the construction of a dependability benchmark that captures the impact of the human system operator on the tested system. Our benchmark follows the usual model of injecting faults and perturbations into the tested system; however, our perturbations are generated by the unscripted actions of actual human operators participating in the benchmark procedure in addition to more traditional fault injection. We introduce the issues that arise as we attempt to incorporate human behavior into a dependability benchmark and describe the possible solutions that we have arrived at through preliminary experimentation. Finally, we describe the implementation of our techniques in a dependability benchmark that we are currently developing for Internet and corporate e-mail server systems.

1. Introduction

Dependability benchmarks are a crucial factor in driving progress toward highly reliable, easily maintained computer systems [3] [9]. Well-designed benchmarks provide a yardstick for assessing the state of the art and provide the framework needed to evaluate and inspire progress in research and development. To achieve these goals, benchmarks must be accurate, realistic, and reproducible; in the case of dependability benchmarks, this means that they must evaluate systems against the same set of dependability-influencing factors seen in real-life environments.

One of the most significant of these factors is human behavior. A system's human operators exert a substantial influence on that system's dependability: they can increase dependability via their monitoring, diagnosis, and problem-solving abilities, but they can also decrease dependability by making operational errors during system maintenance. The human error factor is particularly important to dependability: anecdotal data from many sources has suggested that human error on the part of system operators accounts for roughly half of all outages in production server environments [2]. Recent quantitative studies of Internet server sites and of the US telephone network infrastructure numerically confirm the significance of human error as a primary contributor to system failures [4] [12].

Most existing work on dependability benchmarks has ignored the effects of human behavior, positive or negative; this is unfortunate, but perhaps not surprising given that human behavior is typically studied by psychologists or HCI specialists, not systems benchmarkers. In this paper, we present our first steps at bringing consideration of human behavior into the dependability benchmarking world, and describe our work-in-progress toward building a human-aware dependability benchmark. Although our methodology begins with a reasonably traditional dependability benchmarking framework, we expand on existing work by directly including human operators in the benchmarking process as a source of system perturbation.

Humans add significant complications to the benchmarking process, and much of our research focus is on how to include humans while keeping the benchmarks efficient and repeatable. A key insight is to measure the human dependability impact *indirectly*: our benchmarks measure the system, not the human, and we deduce the human dependability impact indirectly through its effects on the system. Other simplifying techniques that we will discuss include approaches for choosing and preparing human operators for our tests (Section 3), selecting human-dependent metrics that can be automatically collected (Section 2), developing an appropriate workload for the human operator (Section 2), and managing the inherent variability introduced by human operators (Section 3). We consider these approaches in the concrete example of an e-mail benchmark in Section 4.

2. Methodology

Traditional dependability benchmarks measure the impact of injected software and hardware faults on the performance and correctness of a test system being subjected to a realistic workload [3] [9]. For example, the system's performance might fall outside its window of normal behavior while it recovers from a hardware fault; the length of the recovery process and the magnitude of the performance drop are measures of the system's dependability in response to that fault. Typically, dependability benchmarks are run without human operator intervention in order to eliminate the possible variability that arises when human behavior is involved. But as dependability emerges from a synergy of system behavior and human response, ignoring either component or their interactions significantly limits the accuracy of the benchmark; both system and operator must be benchmarked together.

To accomplish this joint measurement of system and operator, we extend the traditional dependability benchmarking methodology by allowing the human operator(s) to interact with the system during the benchmark. The interaction takes two forms. First, the operator plays an active role in detecting and recovering from injected failures, just as they would in a real environment with real failures. Second, the operator is asked to carry out pre-selected maintenance tasks on the system (for example, backups/restores, software upgrades, system reconfiguration, and data migration), again to simulate real-world operator interaction with the system. We then measure the system's dependability as usual; unlike the traditional approach, the dependability result now reflects the net dependability impact of the human operator, be it positive or negative.

In essence, our approach is to create a standard system dependability benchmark with the human operator as a new source of perturbation, in addition to the standard perturbations injected in the form of software and hardware faults. We can classify the human perturbation based on how it arises. *Reactive perturbation* results from unscripted human action in response to an injected hardware or software fault, and reflects the operator's ability to detect and repair failures. If the operator is quick to respond and recovers the system efficiently, the perturbation will have a positive impact on dependability; if the operator makes mistakes, is slow to respond, or simply fails to respond at all, the impact will be negative. In contrast, *proactive perturbations* arise as the operator performs system maintenance tasks unrelated to failure occurrences. These too can have a negative or positive dependability impact depending on how well the operator performs the task, how many errors are made, and how the maintenance task itself affects the system.

An important change in the benchmark semantics arises when we include proactive perturbations. In traditional dependability benchmarks the system is expected to have a constant level of fault-free dependability; in contrast, with our methodology this baseline can change as the result of maintenance on the system (for example, an "upgrade" maintenance task could increase performance or redundancy). While this complicates the interpretation of benchmark results, it is more indicative of real-world dependability, where maintenance is common. It does require some care in cross-system benchmarking to ensure that similar maintenance is performed on all tested systems.

While the above description of our methodology implies that human operators must participate in the benchmark process, one might wonder if we could simulate the human perturbations and thus eliminate the human. Unfortunately, this reduces to an unsolved problem—if we were able to accurately simulate human operator behavior, we would not need human system operators in the first place! While the HCI community has developed techniques for modeling human behavior in usability tests [5], even in those approaches human involvement is required at least initially to build the model, and the resultant models are typically highly constrained and system-specific, making them inappropriate for use in a comparison benchmark.

Thus we are left with the approach of using live human operators in the benchmarks; this is the only way to truly capture the full unpredictable complexities of the human operator's behavior and the resulting impact on a system's dependability. To flesh out the approach, we must consider how to choose operators for the benchmarks, what maintenance tasks to give them, and what metrics we should use for the final dependability scores. We must also confront the challenges of dealing with human variability, performing valid cross-system comparisons with different operators, and structuring benchmark trials so that the number of human operators is minimized. We discuss workloads and metrics in the balance of this section, and return to the remaining challenges in Section 3.

2.1. Human operator workload

The human operator workload consists of two parts: a pre-specified set of maintenance tasks, and the interactions that arise naturally as the operator repairs injected faults. Since the reactive part of the workload is unscripted and depends on the particular operator's approach to the failures, we do not specify it in advance, and we will not consider it further here.

The pre-specified set of tasks that the operator carries out during the benchmark should be representative of the real-world maintenance carried out in production installations of the type of system under test. Note that we define "maintenance" rather broadly: any *operator* (non-end-user) interaction with the system that is not an immediate reaction to a failure is considered maintenance.

The ideal way to obtain a representative set of maintenance tasks is to carry out a "task analysis" study in which the experimenter shadows real operators as they run a production system similar to that being benchmarked [8]; recording how these operators spend their time provides a list of tasks ranked by importance or frequency. The drawback of task analyses is that they are time consuming and often impractical, especially when the type of system being benchmarked has never been deployed in production.

In such cases, a satisfactory set of maintenance tasks can be selected by an expert familiar with the target system's application domain, using published studies of what system administrators do as a guide [1] [6] [7]. An analysis of these studies suggests a set of general categories of maintenance tasks that apply to most systems:

Initial configuration: setting up new systems, including hardware, operating system, and application installations. Increasing the capacity of an existing system.

Reconfiguration: a broad category covering everything from small configuration tweaks to significant reconfigurations like hardware, OS, or application upgrades.

Monitoring: using monitoring tools or probes to detect failures, security incidents, and performance problems.

Diagnosis and repair: recovery from problems detected by monitoring tasks. Diagnostic procedures, root-cause analysis, and recovery techniques like hardware repairs, software reinstallation/configuration, security incident response, and performance tuning. Unlike "Reconfiguration" tasks, these are unplanned and unscheduled.

Preventative maintenance: non-urgent tasks that maintain a system's integrity, redundancy, and performance, or that adapt the system to changes in its workload.

For an example of how these task categories were specialized for a dependability benchmark for e-mail server systems, see Section 4.

2.2. Metrics

Traditional dependability benchmarks use performance and correctness measures to quantify dependability. Dependability scores are produced by examining how these measures deviate from their expected norms as the system is perturbed by injected faults. We can use this same approach to quantify dependability in our human-aware dependability benchmarks, since we are considering human operator involvement as just another perturbation source to the system. Dependability as measured by this approach reflects the net impact of the human operator: human error that affects performance or correctness will be manifested as reduced dependability, whereas human ingenuity in efficiently repairing problems or performing maintenance will manifest as improved dependability.

The major advantage of this approach is that it vastly simplifies the benchmark process compared to the alternative of trying to directly measure the human impact on dependability. Because there is no need to directly measure human error rate or the dependability impact of individual human actions, the collection of dependability results can be automated. Furthermore, it is easier to design benchmarks for cross-system comparison, as there is no need to match operator actions on one system to equivalent actions on another (often an impossible task). Of course, there is nothing preventing the benchmarker from collecting additional data on human error rates, error severity, or recovery time; such data can prove useful in evaluating a system's *maintainability*, although they are not needed for a dependability evaluation.

3. Reproducible benchmarks with humans

The inherent variability and unpredictability of human behavior makes it a challenge to achieve reproducibility when we include humans in our benchmarks. A crucial part of our human-aware benchmarking methodology is to manage the variance introduced by our human operators, both within a single benchmarking experiment and across benchmark runs on different systems or over time.

Variability in human operators comes from at least three sources. First, different prospective operators will have different backgrounds and base skill levels (compare, for example, an experienced sysadmin to a CS student). Second, operators may have different levels of experience with the system and the benchmark tasks. This is a particularly acute problem when benchmarks are carried out iteratively, as each iteration of the benchmark process increases the operator's experience with the system and can alter his or her behavior on subsequent iterations. Finally, there is a level of inherent variability in human behavior: two operators with identical experience and identical training given identical benchmark tasks may still behave differently.

3.1. Managing variability: single system runs

We propose a two-pronged approach for managing variability in one-off, single-system benchmarks. First, we appeal to statistical averaging, deriving the final dependability result from multiple iterations of the benchmark with different operators participating in each iteration. Second, we attempt to minimize the pre-averaging variability by selecting the participating operators from a set of people with approximately-equal levels of background and experience, and by providing training and support resources to further equalize their knowledge bases.

Results from our pilot studies suggest that between 5 and 20 operators (iterations) will be needed to gain a statistically-sufficient averaging effect; work from the UI community confirms these estimates and suggests that 4 or 5 operators maximizes the benefit/cost ratio [11].

3.1.1. Choosing operators

We can significantly reduce the variance between operator-participants by controlling for their background and skill levels. Because real operators vary greatly in their skills and experience, and because real installations have different demands for operator quality and dependability, we cannot establish a single set of selection criteria for all dependability benchmarks. Instead, we define several classes of operators, and allow the benchmarker to choose those which best match the target environment of the tested system. With this approach, results from one benchmark run should be comparable to results from other benchmarks using the same class of operators; benchmarks using different classes of operators might also be comparable if the operator level is used as a "handicap" on the results.

We observe at least three classes of benchmark operators (from highest to lowest qualification):

Expert: The operators have intimate knowledge of the target system, unsurpassed skills, and long-term experience with the system. These are operators who run large production installations of the target system for their day jobs, or are supplied by the system's vendor. Benchmarks involving these operators will report the best-case dependability for the target system, but may be realistic only for a very small fraction of the system's potential installed base.

Certified: The operators have passed a test that verifies a certain minimum familiarity and experience with the target system; ideally the certification is issued by the system vendor or an independent external agency. Benchmarks involving these operators should report dependability similar to what would be seen in an average corporate installation of the tested system.

Technical: The operators have technical training and a general level of technical skill involving computer systems and the application area of the target system, but do not have significant experience with the target system itself. These operators could be a company's general systems administration or IT staff, or computer science students in an academic setting. Benchmarks involving these operators will report dependability that is on average similar to that measured with certified operators, but there may be more inter-operator variance and more of a learning curve factor.

Should human-aware dependability benchmarks reach widespread commercial use (like the TPC database benchmarks [15]), they will probably use expert operators. Expert operators offer the lowest possible variance, are unlikely to make naïve mistakes that could make the system look undeservedly bad, yet still provides a useful indication of the system's dependability and maintainability. Published results from benchmarks like TPC often already involve a major commitment of money and personnel on the part of the vendor, so supplying expert operators should not be a significant barrier.

For non-commercial use of dependability benchmarking where experts are unavailable (as in academic or internal development work), using certified operators is ideal since certification best controls the variance between non-expert operators. As it may be difficult to recruit certified operators, it is likely that technical operators will be often be used in practice; we believe that accurate dependability measurements can still be obtained in this case by providing suitable resources and training as described below.

3.1.2. Training operators

We can reduce any remaining variance within a chosen class of operators by using standardized training to bring all operators to the same level of understanding of the test system. It has been our experience that this training must be done at a conceptual level to help the operator build a mental model of the system. The alternative, training on specific tasks that appear in the benchmark, leads to the unrealistic situation of operators follow checklists during the benchmark rather than relying on ingenuity, exploration, and problem-solving, as they would in real life.

Our initial experiments have suggested that an effective method for conceptual training is to first provide a high-level overview of the system's purpose and design, then have the operator carry out a simple maintenance task that requires exploration of the system's interfaces (for example, changing a configuration parameter that is buried deep in an unspecified configuration file or dialog box). If the initial task is well-designed, the operator will have built up enough of a mental model of the system and its interfaces to proceed with the benchmark. With this approach, very little formal training need be given, simplifying the deployment of the benchmark.

3.1.3. Resources for operators

Even with training, different operators may have different gaps in their knowledge that show up during the benchmark. To mitigate the resulting variance, operators should be provided with resources to fill these gaps. Two effective forms of resources are documentation and expert help.

Documentation provides a knowledge base upon which the operator can draw while performing the benchmark tasks. For maximum realism, we believe the operator should be provided with the unedited documentation shipped with the testbed system and be given access to the Internet and its various search engines. If at all possible, the documentation should be provided electronically so that its usage can be monitored automatically.

When documentation fails in real life, operators turn to experts for help. It is important to provide a similar, but standardized, option in the benchmarking process, both to increase realism and to provide an "out" should the operator get stuck or frustrated with a task. We propose to do this by making available a single "oracle" or expert during all runs of the benchmark. The expert must be intimately familiar with the system and the operator tasks; oftentimes the benchmarker can play this role.

A challenge is making the oracle available in such a way that it remains an appeal of last resort; if overused, the oracle becomes the target of the benchmark, not the operator. One approach used successfully in user studies is to make the oracle available via email [16], an approach that also reduces the demands on the oracle's time. Other possibilities include providing only a limited number of calls to the oracle, imposing an artificial time penalty for using the oracle, or implementing the oracle as an automated "I give up" button that simply completes the current task automatically or restores the system to a known state.

3.2. Managing the learning curve effect

One of the most challenging problems with using live human operators in dependability benchmarks is the *learning curve* effect: at the end of a benchmark iteration, the operator has learned something about the system, and will likely use that experience to perform better on subsequent iterations. This is particularly a problem in cross-system comparison benchmarks or iterative benchmarking of the same system, where the cost of using a fresh set of operators for each system/iteration would be excessive.

Compensating for the learning curve effect is a challenging problem that we are only beginning to address. A simple approach for comparison benchmarks is to randomize the order in which each operator uses the test systems; with a large enough pool of operators, the learning curve effects will be averaged across the systems. Alternately, the effect of the learning curve can be estimated and factored out by benchmarking each system repeatedly until its dependability results stabilize.

For iterative benchmarking of a single system (for example, during system development), other techniques are needed. The most promising approach is to create a system-specific model of human operator behavior, describing how long operators take to respond to problems, what kinds of responses are used, and what kinds of errors are made. Although general simulation of a human operator is intractable, it should be possible to achieve a system- and task-specific model using techniques developed in the HCI community. In particular, models might be created by observing live human operators in an initial benchmark iteration, or perhaps by using expert analysis in a cognitive walkthrough [5]. The benchmarker could use the model to simulate the operators' behaviors in later iterations, either manually or automatically. An open question is how long such a model would remain valid; after major changes to the system or its interfaces, it is likely that the model would need to be rebuilt.

4. An example: benchmarking e-mail

Our first target for evaluating our human-aware dependability benchmarking methodology is e-mail. E-mail has grown from its origins as a best-effort convenience to what is today a mission-critical enterprise service with stringent dependability needs. Surprisingly, no existing e-mail benchmark attempts to quantify dependability.

Our approach follows the general methodology described in Section 2. The benchmark applies a workload, injects perturbations, and collects metrics while the system is under the supervision of a human operator. The benchmark treats the e-mail service as a black-box for generality.

The workload of the benchmark consists of three components: performance, perturbation, and human workloads. The performance workload consists of a realistic simulation of e-mail traffic injected using standard protocols (SMTP and POP3). The simulated workload is based on the SPECmail2001 workload parameters [14] but is fully parameterizable to allow the user to explore system behavior under different load scenarios (for example, load spikes, which are an increasingly relevant dependability threat to Internet services). The perturbation workload has not yet been finalized, but we will likely start with two main types of fault injection: coarse-grained hardware and software faults. For example, we will inject storage system failures (corrupt data, failed disks, timeouts), network failures (corruption, transient connectivity loss, routing anomalies), and OS-level software faults (terminated processes, driver hangs, erroneous return values), among others.

Finally, the human workload consists of maintenance tasks chosen from the categories defined in Section 2.1 and arranged in three steps of increasing difficulty. The first step is a warm-up task consisting of a simple software reconfiguration such as changing the default domain of unqualified e-mail addresses; this step also serves as a "training" step, allowing the operator to become familiar with the system. The second step is a moderately difficult task such as installing and configuring a server-side e-mail virus filter. The third step is a challenging task such as moving a group of users from one server to another.

During each task, the benchmark measures the overall service dependability. Our dependability metrics consist of e-mail delivery delays and errors, the number of dropped/corrupted e-mails, and service performance in fault-free, induced-fault, recovery, and service overload scenarios.

Due to the difficulty of finding certified operators in an academic setting, we intend to use technical-level operators in our benchmark experiments as described in Section 3.1.1. We plan to automate the benchmark as much as possible, including the workload generator and instrumentation. Through these and other techniques, we hope to be able run operators through without a benchmarker present, except perhaps to serve as the on-call oracle.

5. Related work

Our perturbation-based benchmark methodology follows in the footsteps of existing work on dependability benchmarking and extends our earlier work on availability benchmarking, which measured the availability of RAID systems by perturbing them with simulated disk failures

[3]. Our methodology also fits into the dependability benchmarking framework defined by Madeira and Koopman [9], with our human-operator-induced perturbations making up the "upsetload" in their terminology. Where our methodology is unique is in its inclusion of the human operator as a perturbation source: we are not aware of any dependability benchmarks to date that include the human component in their dependability measurements.

Many of the techniques, issues, and proposed solutions in this paper are adaptations of traditional behavioral research techniques for human-computer interaction, such as those described in Landauer's excellent survey [8]. However, unlike the HCI approaches, it is our goal to measure the *system's* behavior rather than the human's—in our benchmarks, the human operator is not there to be directly observed or measured, but to provide realistic perturbation and stimulus to the system. In that sense our work is most similar to work in the security community on the effectiveness of security-related UIs, such as Whitten and Tygar's study of PGP [16]. While we can (and do) borrow advice on topics like selection of operators, task analysis, and experiment logistics from the HCI community, their standard experimental designs and metrics do not directly apply to our dependability benchmarking task.

Finally, our proposed e-mail benchmark differs from other widely-used e-mail benchmarks in that it measures dependability as well as performance. In particular, the two major email benchmarks in production use today (SPEC's SPECmail2001 [14] and Netscape's Mailstone [10]) focus only on performance, do not include facilities for injecting perturbations, and do not measure dependability beyond a simple count of dropped client connections. While some research e-mail systems have been evaluated under simple perturbation (*e.g.*, Porcupine [13]), none have included consideration of the human operator.

6. Conclusions and future directions

As dependability increasingly supplants performance as the essential metric for computer systems, dependability benchmarks are becoming essential tools for system designers and evaluators. Yet to date, dependability benchmarks have ignored the behavior of a computer system's human operators and administrators, a key piece of the dependability puzzle. In this paper we have presented a first attempt at addressing this deficiency: our human-centric benchmarking methodology should provide an effective means of incorporating the effects of human operator behavior into dependability measurements.

What we have presented here is just a first step, however. Our methodology will need to be proven and refined through extensive experimental verification; experimentation will also help explore the extent of cross-operator variability and the tradeoffs involved in issues such as selecting and training operators. We are pursuing this follow-on work in the context of our e-mail dependability benchmark. Other issues that remain to be explored include the development of techniques for pre-evaluating the skill level of participating operators, more advanced dependability metrics that are parameterized by the operator's skill level, and extensions that allow for a direct measure of a system's maintainability and scalability along with the indirect measurements extracted through the dependability metrics. These are all fruitful and important directions for future research, and we look forward to seeing them addressed by the community.

References

[1] Anderson, E. and D. A. Patterson. "A Retrospective on Twelve Years of LISA Proceedings." *Proc. 13th Systems Administration Conference (LISA XIII)*, Seattle, WA, 1999.

[2] Brown, A. and D. A. Patterson. "To Err is Human." *Proc. 1st Workshop on Evaluating and Architecting System dependabilitY (EASY '01)*, Göteborg, Sweden, July 2001.

[3] Brown, A. and D.A. Patterson. "Towards Availability Benchmarks: A Case Study of Software RAID Systems." *Proc. 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[4] Enriquez, P. "Failure Analysis of the PSTN." Unpublished talk available at http://roc.cs.berkeley.edu/retreats/spring_02/d1_slides/RocTalk.ppt, January 2002.

[5] Ivory, M. and M. Hearst. "The State of the Art in Automating Usability Evaluation." *ACM Computing Surveys*, 33(4):470–516, December 2001.

[6] Kolstad, R. "1992 LISA Time Expenditure Survey." *;login:, the USENIX Association Newsletter*, 1992.

[7] Kolstad, R. "Sysadmin Book of Knowledge." http://ace.delos.com/taxongate.

[8] Landauer, T. K. "Research Methods in Human-Computer Interaction." In *Handbook of Human-Computer Interaction, 2e*, M Helander et al. (ed), Elsevier, 1997, 203–227.

[9] Madeira, H. and P. Koopman. "Dependability Benchmarking: making choices in an n-dimensional problem space." *Proc. 1st Workshop on Evaluating and Architecting System dependabilitY (EASY '01),* Göteborg, Sweden, July 2001.

[10] Netscape, Inc. *Mailstone Utility.* http://docs.iplanet.com/docs/manuals/messaging/nms41/mailston/stone.htm.

[11] Nielsen, J., and Landauer, T. K. "A mathematical model of the finding of usability problems." *Proc. ACM INTERCHI '93*, Amsterdam, The Netherlands, April 1993, 206–213.

[12] Oppenheimer, D. and D. A. Patterson. "Architecture, operation, and dependability of large-scale Internet services." Submission to *IEEE Internet Computing*, February 2002.

[13] Saito, Y., B. Bershad, and H. Levy. "Manageability, Availability, and Performance in Porcupine: A Highly Scalable Internet Mail Service." *Proc. 17th Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, SC, 1999.

[14] Standard Performance Evaluation Corporation. *SPECmail2001*, http://www.spec.org/osg/mail2001/.

[15] Transaction Processing Performance Council Benchmarks. http://www.tpc.org.

[16] Whitten, A. and J. D. Tygar. "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0." *Proceedings of the 9th USENIX Security Symposium*, August 1999.

Benchmarking Semantic Availability of Dynamic Data Feeds

Orna Raz, Philip Koopman, Mary Shaw
Carnegie Mellon University
{orna.raz@cs, koopman@ece, mary.shaw@cs}.cmu.edu

Abstract

Many of the software systems we use for everyday purposes incorporate elements developed and maintained by third parties. These elements include not only code components and data bases but also dynamic data feeds from online data sources. Even though everyday software is not mission critical it must be dependable enough for its intended use. This is limited by the dependability of its constituting elements.

It is especially difficult to assess the dependability of dynamic data feeds because they exhibit not only "fail-silent" behavior but also semantic failures—delivery of unreasonable yet well structured results by a responsive data feed. Further, it is normal for the behavior of such data feeds to change. Unfortunately, the specifications of these data feeds are often too incomplete and sketchy to support failure detection.

We propose an approach for benchmarking the semantic availability of redundant data feeds. The fault model is defined as violations of inferred invariants about the usual behavior of a data feed.

1. Introduction

Everyday software is usually not mission critical, yet it must be dependable enough for its intended use. Assessing dependability requires a model of proper and improper behavior. However, specifications for everyday software are often incomplete and imprecise. When the software incorporates third party elements, such as code components, data bases, and dynamic data feeds from online data source, this situation is exacerbated. Assessing the dependability of dynamic data feeds is especially challenging, because a data feed remains under the control of its proprietor, who might change its format, semantics, or even remove it, as it is being used.

Examples of data sources include stock quotes, weather forecasts and airline ticket prices. A data feed captures a particular usage of a data source: for example, stock quotes for a specific company, the weather forecast for a specific city and airfare for specific origin and destination.

It is especially hard to automatically detect changes in the semantics of a data feed, since the data feed might superficially appear to be delivering the required service. This is the availability facet of dependability, under a *semantic fault model*: the data is delivered, it is syntactically correct, but it is inconsistent, out of range, incorrect, or otherwise unreasonable.

1.1. Semantic availability

Availability is defined as "readiness for correct service", "a measure of the delivery of correct service with respect to the alternation of correct and incorrect service" [1]. We, therefore, define *semantic availability* of a data feed to be its readiness for usage, indicated by whether the data feed delivers reasonable results. We assume the data feed is responsive (no connectivity failures) and delivers parsable results (no syntax/form failures). The availability of the data feed directly affects the availability of the system using it. To measure and assess this availability, the delivery of semantically correct service needs to be estimated, with respect to the alternation of semantically correct and semantically incorrect service. Detecting semantic failures would enable us to estimate the semantic correctness of a service.

Fault tolerance approaches to detection often use state-space methods [4]. This requires specifications of states and transition probabilities between states (left hand side of Figure 1). Masking, which does not require detection, requires specifications of outputs and their selection.

However, a particular problem in the domain of dynamic data feeds is that their specifications are sketchy and incomplete. Unfortunately, the analysis simplicity of the state-space model is not applicable in our setting. In [2] we noted state-space models are difficult to work with when the specifications are inaccurate and suggested an alternative gradient view. The gradient view, depicted on the right hand side of Figure 1, emphasizes the direction of the transitions rather than the precise distinction among states: transitions may degrade or improve performance, though the distinction between working and broken may be fuzzy.

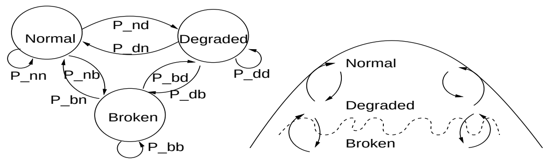


Figure 1. Degradation and failure described by a state-transition diagram and by a gradient view

In [3] we introduced an approach for detecting semantic anomalies in dynamic data feeds, following the gradient view. Rather than demanding better specifications, we infer invariants about the behavior of a data feed using and adapting existing statistical and machine learning techniques. We then use these invariants as proxies for missing specifications. Initial feasibility results indicate these invariants suffice for good-enough detection of semantic anomalies (in the context of stock market tickers).

We believe our approach of inferring proxies for missing specifications, in the form of invariants, can be used to create benchmarks for the semantic availability of redundant data feeds.

2. Benchmarking redundant data feeds

We propose to define benchmarks for evaluating and comparing redundant data feeds (data feeds that provide similar service) based on invariants about the behavior of the data feeds. The measures we propose for the benchmark are the number and nature of violations of invariants (anomalies).

Fault injection approaches often use bit flips to emulate failures and assume a fail-fast, fail-silent behavior. However, for semantic failures in data feeds, it is not clear what a bit-flip fault model would measure. It may test a subset of correctness failures, but we believe there is a need for a stronger fault model. Instead of bit flips we propose violation of invariants as a fault model.

Unfortunately, not only are invariants about the behavior of a data feed rarely provided but also the behavior of the data feed may change. We suggest determining, periodically, a standard set of invariants to be used as a benchmark. These invariants may be not only stationary, but often *adaptive*: the invariants may change as the behavior of the data feed changes.

Our approach of inferring proxies for missing specifications could be used to automate parts of both creating the standard set of invariants and producing benchmark measurements, as follows: periodically,

1. use our invariant inference framework and tools to synthesize a list of candidate adaptive invariants, then

2. have a certification authority, composed of domain experts, select the standard set of invariants from the list (selection through a social process). Constantly:

3. use the standard set of invariants for anomaly detection in the redundant data feeds under test.

Anomalies are detected by evaluating each invariant in the standard set over fresh observations of each of the redundant data feeds and reporting an anomaly when an invariant evaluates to false. We assume it is possible to synchronize the redundant data feeds.

Various comparison metrics of redundant data feeds are possible. For example: (1) the number of detected anomalies and (2) the nature of the anomalies; a larger weight should probably be given to anomalies that are more severe. These metrics could be combined with metrics that measure connectivity and syntax/form availability, for a more complete picture regarding the availability of a data feed.

3. Summary and challenges

We propose: (1) a reference model for how to benchmark the semantic availability of redundant data feeds and (2) tools to aid certification authorities in creating a standard set of invariants. Our premise is that choosing from a list of inferred invariants is easier than creating this list, so having a machine synthesize the list is helpful.

The process of deciding what to benchmark and how to do so is inherently subjective. The task of a certification authority should be made feasible. We believe a fruitful direction is to limit the human intervention and level of expertise required. A possible direction is to require experts to approve only templates of invariants.

Our invariant inference engine cannot guarantee to detect all anomalies. This is true for any technique that does not demand complete specifications. Further, data feeds are dynamic: they are often expected to change. The benchmarks we suggest are, therefore, adaptive. An open issue is when the benefits of adapting to data feed changes are greater than the risks of having a drifting benchmark.

4. Acknowledgments

This research is supported by the National Science Foundation under Grant CCR-0086003 and by the Software Industry Center at Carnegie Mellon University.

References

[1] A. Avizienis et al. Fundamental concepts of dependability. Technical report, UCLA CSD Report no. 010028, 2001.

[2] O. Raz et al. An approach to preserving sufficient correctness in open resource coalitions. In *IWSSD-10*, 2000.

[3] O. Raz et al. Semantic anomaly detection in online data sources. In *ICSE'02*, 2002.

[4] A. Villemeur. *Reliability, Availability, Maintainability, and Safety Assessment*. Jon Wiley & Sons, 1992.

Using Bayesian Theory for Estimating Dependability Benchmark Measures

Michel Cukier and Carol S. Smidts

Center for Reliability Engineering
Department of Materials and Nuclear Engineering
University of Maryland at College Park
{mcukier, csmidts}@eng.umd.edu

1. Introduction

Assessing the quality of service of a computer system is a difficult task. A lot of work has been conducted on evaluating quality of service attributes like performance, robustness, and dependability. Two approaches used for evaluating performance and robustness are modeling and benchmarking. For evaluating dependability, modeling can be used either alone or combined with fault injection [Sie92, Kan91]. However, less work has been conducted on building dependability benchmarks. A dependability benchmark can be defined as "a way to evaluate the behavior of components and computer systems in the presence of upsets, allowing the quantification of dependability attributes or the characterization of the systems into well defined dependability classes" [Mad01].

This paper focuses on the quantification part of the definition. The goal of this paper is to propose the use of Bayesian estimation methods for quantifying dependability attributes. We first will give a brief overview of two estimation Schools in Section 2. We will then illustrate our proposal by focusing on a key parameter for fault-tolerant systems, the coverage factor. We will introduce the coverage factor in Section 3 and present coverage factor estimations in Section 4.

2. How does the Bayesian theory differ from the frequentist theory?

The frequentist School and the Bayesian School are two important estimation branches in statistics. We now briefly compare the applicability of the two theories. After having conducted some experiments, there are different ways for processing the obtained results in order to get an estimation.

An estimation obtained using the frequentist theory is based only on the results collected during the experiments. The distribution associated with the experiment is often introduced in order to obtain more accurate estimations.

When applying the Bayesian theory, an estimation is based on the results collected during the experiments and a prior knowledge of the estimation. This prior knowledge could be based on previous experimental results or on expert knowledge. As for the frequentist case, a distribution is often associated with the experiment. In the Bayesian case, another distribution is introduced to include the prior knowledge, called a *prior distribution*. The combination of these two distributions leads to the *posterior distribution*. The posterior distribution is then used to calculate the estimation.

This combination of two sources of information often has the advantage that, if the experimental results confirm the prior knowledge, a smaller number of experimental results compared to the frequentist approach will be needed to obtain the same estimation.

One of the purposes of a benchmark is to obtain an estimation without having to gather too many experimental results. However, in that case, the estimation might then not be very accurate. The use of Bayesian methods might increase the accuracy since, besides experimental results, a prior knowledge is also used to calculate an estimation. The following sections will present some examples showing the advantages of the Bayesian method for an important parameter of fault-tolerant systems: the coverage factor.

3. Coverage factor

Let us first formally define the coverage factor. The reaction of a fault-tolerant system will depend on two different inputs: the inserted *upsets* and the submitted *workload*. The upsets are the inputs specific to the fault tolerance mechanisms. The workload represents the environment. The overall input space of a fault-tolerant system is then the Cartesian product $G = U \times W$, where U is the upset space and W is the workload space. Let us define H as a variable characterizing the handling of a particular upset. The coverage (factor) of a fault-tolerant system can then be defined as:

$$c = \Pr\{H = 1 | g \in G\}$$

i.e., the conditional probability of correct upset handling, given the occurrence of an upset/workload pair g.

4. Frequentist and Bayesian estimations

From now, we assume a representative sample, i.e., where the selection probability is equal to the relative probability of the occurrence of a given upset/workload pair.

Since fault-tolerant systems will, most of the time, correctly handle an upset, we will focus on the non-coverage as a measure ($\bar{c} = 1 - c$), and more specifically on the upper confidence limit of the non-coverage. Since the number of upsets not correctly handled follows a binomial distribution with parameters n (number of inserted upsets) and $\bar{c}$ (non-coverage), the 100γ upper confidence limit is given by:

- Frequentist theory:

$$\bar{c}_\gamma = \frac{(x+1)F_{2(x+1),2(n-x),\gamma}}{(n-x)+(x+1)F_{2(x+1),2(n-x),\gamma}}$$

- Bayesian theory:

$$\bar{c}_\gamma = \frac{(x+k)F_{2(x+k),2(n-x+l),\gamma}}{(n-x+l)+(x+k)F_{2(x+k),2(n-x+l),\gamma}}$$

where n is the number of inserted upsets, x is the number of upsets incorrectly handled, γ is the confidence level, and $F_{\upsilon 1,\upsilon 2,\gamma}$ is 100γ percentile points of an F distribution with $\upsilon 1, \upsilon 2$ degrees of freedom [Joh69, p.59]. Moreover, in the Bayesian case, we have selected a prior distribution following a beta distribution with parameters l and k: beta(k, l). The reasons for this choice are: first, that one special type of a beta distribution is the uniform distribution (when k=l=1), which effectively expresses the fact that there is *no* prior knowledge, and second, that since the experiment follows a binomial distribution, the posterior distribution will also be a binomial distribution with parameters x+k and n-x+l: beta(x+k, n-x+l).

Let us now compare the estimations obtained with the frequentist approach with the ones from the Bayesian. We will consider three prior distributions: the uniform distribution (k=l=1) where no assumption is made on the fault-tolerant system behavior, a distribution with a slight confidence that the fault tolerance system will behave correctly (k=1, l=3), and a distribution with a very high confidence that the system will correctly handle upsets (k=1, l=100). The density function ($f_Z(z)$) for these three prior distributions with a beta(1, 2) for completeness are shown on Figure 1.

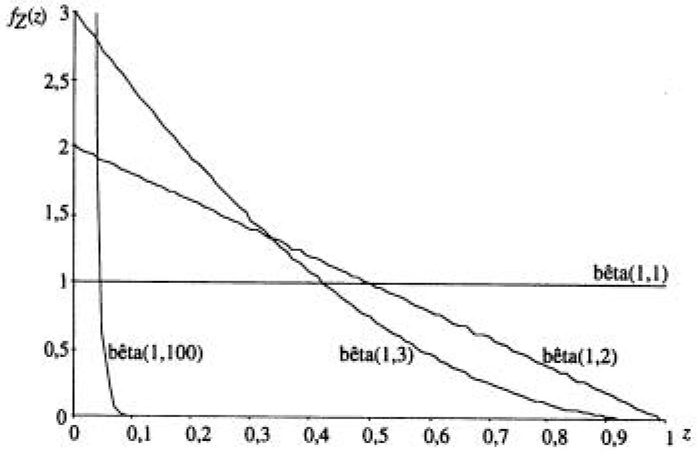


Figure 1 Prior distribution density functions

Since fault-tolerant systems will correctly handle most of the upsets, we will compare the frequentist and Bayesian estimations when 0 or 1 upset is not correctly handled. We compare these estimations when 10, 100 and 1000 upsets are inserted. The obtained estimations are shown in Table 1.

From the table, we observe that the confidence limit is smaller when using the Bayesian approach, leading to a smaller confidence interval and thus a more accurate estimation. When applying a prior distribution with high confidence in the correct upset handling, the obtained estimations become more accurate. The reason is that the small number of incorrectly handled upsets (0 or1) is in accordance with the prior knowledge that almost no upset will be incorrectly handled. An important observation is that for a small number of inserted upsets, an order of magnitude might be gained in the estimation accuracy when using the Bayesian approach and a strong confidence of upsets being correctly handled by the system.

Up-sets	Incor.	Frequentist	Bayesian		
N	X		k=1, l=1	k=1, l=3	k=1, l=100
10	0	0.369	0.342	0.298	0.0410
	1	0.504	0.470	0.413	0.0588
100	0	0.0450	0.0446	0.0437	0.0227
	1	0.0645	0.0639	0.0627	0.0327
1000	0	0.00459	0.00459	0.00458	0.00418
	1	0.00662	0.00661	0.00660	0.00602

Table 1 Comparison of frequentist and Bayesian estimations

5. Conclusion

We have shown in this paper that Bayesian theory might be worth considering when quantifying dependability attributes. We have illustrated the relevance of the Bayesian theory by presenting a simple example: the estimation of the coverage factor of a fault-tolerant system.

Acknowledgments

The authors would like to thank David Powell and Jean Arlat for their comments and for providing the example used in this paper.

References

[Cuk99] M. Cukier, D. Powell, J. Arlat, *Coverage Estimation Methods for Stratified Fault-Injection,* in IEEE Trans. On Computers, vol. 48, no. 7, pp.707-723, July 1999.

[Joh69] N.L. Johnson, S. Kotz, *Distributions in Statistics – Discrete Distributions*, New York, John Wiley & Sons, 1969.

[Kan91] K. Kanoun, J. Arlat, L. Burrill, Y. Crouzet, S. Graf, E. Martins, A. MacInnes, D. Powell, J.-L. Richier, J. Voiron, "Validation", in *Delta-4: A Generic Architecture for Dependable Distributed Computing*, (D. Powell, Ed.), pp. 371-406, Berlin, Germany, Springer-Verlag, 1991.

[Mad01] H. Madeira, P. Koopman, *Dependability Benchmarking: making choices in an n-dimensional problem space*, in First Workshop on Evaluating and Architecting System Dependability (EASY), joint organized with IEEE/ACM 28th International Symposium on Computer Architecture (ISCA) and the IEEE International Conference on Dependable Systems and Networks (DSN), Gothemburg, Sweden, July 2001.

[Sie92] D.P. Siewiorek, R.S. Swarz, *Reliable Computer Systems – Design and Evaluation*, Bedford, MA, USA, Digital Press, 1992.

Empirical Evaluation of Techniques and Methods Used for Achieving and Assessing Software High Dependability

Ioana Rus
Fraunhofer Center for Empirical Software Engineering Maryland
irus@fc-md.umd.edu

Victor Basili
Marvin Zelkowitz
University of Maryland and Fraunhofer Center for Empirical Software Engineering Maryland
Basili,mvz@cs.umd.edu

Barry Boehm
University of Southern California boehm@usc.edu

For achieving high dependability of software intensive systems, not only product dependability benchmarking is needed but also benchmarking of technologies and processes for achieving and assessing software dependability. Dependability engineering, and more specifically technology management and assessment of effectiveness and efficiency of different technology interventions, is the objective of the work we introduce here. This work is performed as part of the *High Dependability Computing Project* (HDCP)[1] that is an incremental, five-year, cooperative agreement, part of a broad strategy for dependable computing, that links NASA, corporate partners and universities and research centers such as Carnegie Mellon, University of Maryland, Fraunhofer Center Maryland, University of Southern California, Massachusetts Institute of Technology, University of Washington and University of Wisconsin. For now the focus is on NASA projects, but the results will be captured and organized in an experience base, so that they could be disseminated and applied to other organizations. For example, the first step would be to extend the results to organizations that are members of the *High Dependability Computing Consortium* (HCC)[2] and the *Sustainable Computing Consortium* (SCC)[3].

As part of our activities we are looking at a series of steps to evaluate such interventions. Developing high dependability software requires specifying the dependability requirements, using development techniques and methods (that we will call "technologies") to build-in high-dependability as the product is developed, and also technologies to verify that the required dependability has been achieved. Our research focuses on evaluating the effectiveness of these technologies with respect to achieving and assessing the desired dependability, and also the cost of using these techniques. For this purpose we are employing diverse empirical evaluation methods such as case studies, pilot projects, project monitoring, assertion, field study, literature search, lessons learned, static analysis, replicated experiment, synthetic experiment, dynamic analysis, product and process simulation.

The technologies might be evaluated with respect to dependability if applied in isolation, as well as if they are combined in various ways (since different technologies are used in different development phases and also different technologies might address different attributes of dependability). Technology comparison might also be required, therefore the need for a common set of measures that can be applied to the results of all technologies (or at least to the ones comparable to each another, i.e., addressing the same attribute). Some technologies might work for specific contexts (e.g. application domain, type of system - concurrent processes, distributed systems, real-time systems, db transactions, operational environment) but not for all situations, so these circumstances must also be studied and identified.

In order to perform technologies evaluations we need to determine the variables that we will observe, measure, and analyze. Therefore we need to have a model of dependability (sub-attributes and measures), for the delivered software. In addition we also need indicators that can be measured during development and help predicting the dependability of the operational system.

Given that dependability is a behavioral property of a system, depending on the environment and the way the system is operated, we see the following questions to be addressed for determining useful measures for our technology evaluation:

- What are the measures for the dependability of a system and how does that translate to software?

[1] http://amesnews.arc.nasa.gov/releases/2002/02_03AR.html
[2] http://www.hdcc.cs.cmu.edu/
[3] http://www.sustainablecomputing.org/

- What does *high dependability* mean (if dependability is a combination of other attributes such as reliability, security, availability, robustness, then what are the values of these attributes and how are they combined to result in high dependability)?
- What are the indicators in intermediate phases of development that allow prediction of the dependability of the deployed system?

If we consider the perspective of a maturing dependability technology we can view each high-dependability technology as passing through a series of evaluation milestones, each stressing the technology and demonstrating its context of effectiveness. Technology researchers will specify the goals for their technologies relative both to *needs*—as specified by users or identified by empirical investigation—and to the *models* for high-dependability. These goals will be established as criteria for studying the technology and identifying the characteristics of the milestone in which the technology is applied. In the assessment process, we identified four steps and corresponding milestones described below. Having a well-defined model and measures of dependability is an indispensable requirement for each of the four test-bed levels mentioned here.

Milestone 1. Internal set: Typically, the technology researcher (creator) has applied the technology to some internally developed set of examples. This set will act as a first milestone for that technology. The technology will be applied to that set of examples defining the milestone by an independent source to make sure the documentation and robustness is sufficient to allow for independent application of the new technology. Thus, before moving the initial examples to the basic common milestone, the technology must have been applied on a technologist-developed test set and that test set should be characterized and used to generate a technology specification and set of criteria for dependability specific to that technology. That initial test set of examples should be contributed to the basic common set, which will be stored in the experience base.

Milestone 2. Basic common set: We can build a basic set of common examples that we can use for applying each of the technologies. The goal is to create a larger universe of problems on which to stress and analyze technologies, both individually and in groups. As stated above, one source of such examples is the internal test sets of the individual developers. However, based on the models, the analysis of the individual test sets, and the analysis of industry problem areas, new examples can be added to this set. This set will allow various technologies to be compared and their strengths and limits assessed empirically. And, of course, it provides a larger domain of potential application for each of the technologies by enlarging the universe of examples. Experiments will be defined for this milestone based on the technology to be tested and the goals established for that technology relative to the milestone.

From industry's point of view, this level offers some insight into what combinations of technologies might be most effective under what conditions and for which problems. From the technology researcher's point of view it provides feedback on how a technique might be expanded and evolved. For the empirical researcher, this milestone will provide new insights into models and goals.

Milestone 3. HDCP domain-specific off-line set: This milestone consists of a domain-specific set of examples, from areas of greatest high-dependability. Ideally, examples in this set will have failure data from real experience associated with them. A committee consisting of NASA personnel as well as HDCP decision makers and technology researchers, again supported by empiricists, will make the choices. This milestone will provide better models of dependability more directly pointed at NASA and HDCC requirements. We will define a different class of experiments for this milestone, involving application domain experts.

Once again insights will be gained on how the various technologies can be integrated and under which circumstances each should be applied, based on decisions such as understanding of the anticipated failures for the problem, the expertise of the appliers of the technology, the effectiveness of the technology for certain classes of faults, and the cost of applying the technique. Success at this milestone should imply that the technology deserves more careful packaging for wider application—high-quality documentation, training materials, tool support, and the like.

Milestone 4. Live examples: This milestone definition is specific to part or all of a system currently under development. Although the techniques have passed through each of the prior milestones, there is clearly a need for risk mitigation. Continual observation by the empiricists is needed and alternate actions are predefined to make change possible when necessary. Experiments may consist of the technology being applied on only part of the system, so a comparison can be drawn with other parts, or it might be a case study of the entire project.

Based on the results of these studies, the technology can be fully packaged for use and placed on the NASA technology shelf as a transferred technology, or it may require a second or third live example for further study of its effectiveness. Ideally, examples in this set will have failure data sets from real experience associated with them.

The Set-Check-Use Methodology for Detecting Error Propagation Failures in I/O Routines

Michael W. Bigrigg
Institute for Complex Engineered Systems
Carnegie Mellon University
Pittsburgh PA 15217
bigrigg@ices.cmu.edu

Jacob J. Vos
Institute for Complex Engineered Systems
Carnegie Mellon University
Pittsburgh PA 15217
jvos@ece.cmu.edu

Abstract

A methodology is presented that will detect robustness failures in source code where I/O errors could occur and where there is no mechanism in place to handle the error. The details of the methodology are described showing how traditional compiler data flow analysis can be augmented to find structurally, within the application, code that can be used to perform error checking. In addition we describe how this code can be used to ensure the correctness of the I/O error checking.

1. Introduction

File systems routinely make extraordinary attempts on behalf of the application to provide data whenever possible to the user. Yet, problems such as network congestion or outages and heavily loaded systems can lead to failure-like situations, making it impossible for the file system to complete the entire requested operation. These situations are usually only transient and still enable the file system to provide a partial result. An example of a true failure condition for a file system is a request for data where no data is available, i.e., a read past the end of a file. This is different from a situation in which data is available, but is currently not accessible, such as when using a disconnected mobile device. When applications not intended for an unreliable environment are ported from a desktop environment to a wireless system, the application programmer must account for all such unpredictable behavior. As programmers, we often overlook error checking when we are overwhelmed with the task of identifying all possible error situations, or neglect checking in the belief that some errors are inconceivable.

Local file system interfaces are typically identical to those of a distributed file system, though the potential for failure at each is greatly different. In a local file system, failures are catastrophic. If the hard drive or other local storage device fails, it often signals the end of the device's usable lifetime. Failures in distributed file systems are more common and it is possible to recover from them. They are usually the result of unreachable remote storage devices or device overloading due to network partitioning, poor load balancing, or denial of service attacks. Users have fundamental, but not often expressed assumptions about the reliability of the system an application is built for. Yet unhandled error conditions lead to potential software failures when the underlying system cannot satisfy our requests and the application was built assuming that it can.

We present a methodology based upon program static analysis to track the propagation of error reporting in order to determine the assumptions used when the software was created.

2. Software Fault Detection Related Work

There are many approaches to using program analysis for the detection of software faults. These systems are typically aimed at providing information to the programmer in order that the program source code may be modified to eliminate software faults. In identifying code errors, there are strict guidelines regarding right and wrong within an application, i.e. dealing with the disabling and re-enabling interrupts, or the assumptions about integer size. Our method does not establish the correctness of code, instead establishing the existence of code that will ensure correctness.

Errors in an I/O system can only be identified at runtime and only after checking the status of the I/O call. One type of fault analysis techniques will run the entire program or a subset of the program to observe its behavior. As well, controlled errors can be introduced to examine how the software behaves by passing external faults into the application that cause it to fail. Such approaches include fault injection through random memory corruption or corruption of the storage system (FlakyIO) [1], passing values typically known to cause exceptions into an individual software module through its software interface (Ballista) [5], and the creation and use of a comprehensive test suite. In particular, this type of approach makes it possible to identify the type of input or condition that has led to the fault, but does not identify any remedial action that should be taken by the application.

Compile-time analysis attempts to identify program features that would cause a program to behave improperly. The analysis focuses on a particular characteristic that is typically the base cause of faults such as portability problems (i.e., when moving an application from one machine architecture to another (lint) [4]). We, however, are attempting to find portability problems, not between architectures, but between systems that have different reliability guarantees on their file system. Other approaches, such as LcLint [3] and mc [2] use programmer-defined rules that specify acceptable behavior to drive the analysis. These two systems are the most closely related projects to ours in method, but their purpose is to capture the assumptions about a program in order to establish correctness, while our focus is to uncover the original assumptions made about a program.

We present a methodology that will detect robustness failures in source code where I/O errors could occur and where there is no mechanism in place to handle the error. Many programmers fail to incorporate error checking in specific classes of I/O operations and rely on certain assumptions such as "file output is always guaranteed" to ensure correct application operation. It is this absence of error checking that we intend to detect with our methodology. Our approach to uncovering these situations combines an augmented data flow analysis with the semantics of the I/O error reporting.

We describe the details of the methodology showing how traditional compiler data flow analysis can be used to find structurally, within the application, code that can be used to perform error checking. In addition we describe how this code can be used to ensure the correctness of the error checking.

3. Error Reporting in C I/O Routines

Errors are reported in C I/O routines using out-of-range values. The return values of these routines are either a useful result (upon successful completion of the call) or an indication of the error that occurred (upon an unsuccessful completion). For instance, the successful return of the `fopen` call is a handle to a file. The range of values for a file handle is an unsigned integer greater than zero. A zero, also referred to as NULL, is then used to report that the file system was unable to open the file. The return of a `fread` call uses out-of-range values to transmit not only an error condition, but also specifies an end of file condition as well. The return identifies the number of bytes that have actually been read. The fread call, like all data buffer operations, will read up to but no more than the number of bytes that have been requested. A return of zero does not signify an error condition, just that no data is currently accessible such as at the end of a file. It is a negative return value that signifies an error condition. Since a single value can potentially be both an error condition and also a valid result, it is not until tested that we know. Just like Schroedinger's cat, we cannot tell what the value is until it is examined. When writing a program, we have to assume that both outcomes are likely and cannot assume one or the other.

The values that specify an error condition are based on the I/O routine itself. An examination of the C standard I/O library [8] shows the behavior of I/O function calls upon an error condition:

- Functions that return pointers use a NULL to designate an error condition: `fmpfile`, `fopen`, `freopen`, `fgets`.
- Functions that use EOF as an error condition: `fclose`, `fgetc`, `getchar`, `putchar`, `puts`, `ungetc`.
- Functions that use a non-zero for an error condition: `remove`, `rename`.
- Functions that use a negative number for an error condition: `fputs`, `fgetpos`, `fseek`, `fsetpos`, `fprintf`, `fscanf`, `print`, `sprintf`, `sscanf`, `vfprintf`, `vsprintf`, `fputc`, `fputs`, `gets`, `putc`, `fread`, `fwrite`.
- Functions that use a -1 for an error condition: `ftell`.

Only the data buffer operations (`fprintf`, `fscanf`, `print`, `sprintf`, `sscanf`, `vfprintf`, `vsprintf`, `fputc`, `fputs`, `gets`, `putc`, `fread`, and `fwrite`) overload the return with three potential values.

In addition, we must identify the result value. The result is the value achieved upon successful completion of the call and may be passed through a return or through an argument. The buffer operations have not only the result in the return but also an argument (a buffer), which is also a result. Not only is it important to distinguish the error from the result in the return, but also it is important to acknowledge the error before using the buffer contents. Therefore, error checking must occur before the use of any result values.

4. Identification of Error Checking

Correct error checking associated with an I/O routine must occur between the *set* (called a definition or simply *def*) of the potential error value and *use* of a result value or values along all possible paths of execution. For instance, the C code example in Figure 1 could lead to a program crash, while the code example in Figure 2 uses program logic to safeguard against a possible error condition.

```
fin = fopen("foo","r");
fread (buf, sizeof(int), 10, fin);
```

Figure 1. Code that may lead to a failure

```
fin = fopen("foo","r");
if (fin != NULL) {
    fread(fin, sizeof(int), 10, buf);
}
```

Figure 2. Program logic guards against a possible unsuccessful result

We augment traditional data flow analysis to identify missing error checking. Data flow analysis is a traditional technique used by compilers during the optimization phase as a tool to guarantee the correctness of program transformations. Value chains, called *def-use chains*, are identified between the definition of a value and the places the value is used. Data analysis is performed on values and not on variables. Figure 3 shows how a value chain is formed, dependent on the instance of a value in a variable, rather than on the name of the variable.

```
a =3; /* def of a1 */
b =a +5; /* use of a1 , def of b1 */
a =8; /* def of a2 */
```

Figure 3. Formation of a value chain

We augment the def-use chains to additionally include the *check* of a value. We define a *check* as a *use* of a value that additionally falls within the expression of a conditional statement. There is already a large body of work on the mechanisms for computing def-use chains [7]. The conditional is a guard against incorrect usage of the result value. The conditional expression that acts as an error guard may be part of any conditional structure including *if* and *if-else* statements as well as *while* and *repeat* loops as shown in Figure 4.

```
n = fread (fin, sizeof(int), 1, buf);
while (n > 0) {
    k += buf[0];
    n = fread (buf, sizeof(int), 1, fin);
}
```

Figure 4. Formation of a value chain

While *set-check-use* is a straightforward approach, there are a few issues to incorporate into our methodology. Error values and the result values are not bound to a specific variable as shown in Figure 5. These values can be assigned to other variables or even modified. In these cases, we need to track the values to make sure that the *use* of the result values does not occur before the *check* of the error values.

```
a = fopen ("foo.txt","r");
b = a;
if (b != NULL) {
    n = fread(buf, sizeof(int), 1, a);
}
```

Figure 5. Analysis based on values not variables.lue chain

It is also important to note that the use of the result value need not exist only within the body of the conditional, and that the conditional may be used to reset the result variable as shown in Figure 6. Again this involves tracking the values through all execution paths. Once the tracked value is overridden with another value, the tracking of the previous value stops along that path of execution.

```
a = fopen ("foo.txt", "r");
if (a == NULL) {
    a = stdin;
}
```

Figure 6. Resetting a value for protection

We know that there is a check, but that does not mean that the expression will accurately identify an error situation.

Finally, in order to determine if the check is valid we must examine the conditional expression. This will be explained in the remainder of this section.

Another aspect to detecting missing robustness checks is the use of error information from the language, as outlined in the previous section, to guide the *set-check-use* approach by determining which value identifies the error. The error propagation information provides a heuristic approach similar to error classification schemes [6]. An example is given to show how semantic information would drive the *def-check-use* analysis. In the case of file opening as shown in Figure 7, the def, check, and use locations use the same value for the analysis. Between the *def* and *use* of *a*, there should be a *check* of *a*.

```
a = fopen("foo","r"); /* def of a */
if (a != NULL) /* check of a */
  fread (buf, sizeof(int), 10, fin); /*use of a */
```

Figure 7. *Def-Check-Use* of the same value

In an `fopen` call, the return holds the error value. A NULL return value designates an error condition. The *def*, *check*, and *use* is to use the same value, *a*, which is the value returned from the `fopen` call. We acknowledge that it may not be possible to statistically determine the validity of the check.

5. Analysis of a Simple Program

The *wc* program is part of the GNU textutils collection of programs. Its purpose is to count the number of lines, words, and characters in a file or files identified on the command line. The v2.0 program consists of 371 text lines with 118 lines of code. It can be identified to have been in use for the past 16 years (from 1985 to 2001). It was written in C and consists of four functions. There are no instances of control flow issues where error checking only exists in a subset of execution paths. A hand analysis of the main source program was performed using the methodology presented to detect failure and to check for error conditions. The results are summarized in Table 1.

Table 1. Hand Analysis of the *wc* program

Routine	**Total**	**Checked**	**Unchecked**
fprintf	1	0	1
Printf	7	0	7
Puts	1	0	2
putchar	1	0	2
fstat	1	1	0
lseek	2	2	0
read	3	3	0
open	1	1	0
close	2	2	0
setlocale	1	0	1
SUMMARY	**20**	**9**	**11**

A simple calculation of the number of checks that should be performed against the number of checks actually produced results in a 45% reliability rating. The usefulness of this rating is not promoted as it does not reflect the frequency of each call, but can be included as a guide to understand the program behavior.

The major assumption that was made by the *wc* program is that all output is guaranteed to succeed.

6. Future Work

The Set-Check-Use Methodology (SCUM) work is part of the PARIS project at CMU, which is attempting to use program analysis techniques to analyze the reliability of programs. We can see the strength of this methodology for determining the reliability of an I/O program, and are in the process of implementing it in a tool that will report on the presence and absence of error checking in programs to construct a rating of reliability. At the same time, we are attempting to classify the missing error checks, and so become able to specify the types of assumptions that are made about the operating environment for I/O applications automatically.

7. Acknowledgements

This work supported by the Pennsylvania Infrastructure Technology Alliance and also as part of the PASIS project supported by DARPA/ISO's Intrusion Tolerant Systems program (Air Force contract number F30602-99-2-0539-AFRL). We would also like to thank Joan Digney for her help in preparing this paper.

8. References

[1] Michael W. Bigrigg and Joseph Slember. Testing the Portability of Desktop Applications to a Networked Embedded System. *Workshop on Reliable Embedded Systems*, Oct. 2001.

[2] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. *Proceedings of the 2000 OSDI Conference*, Oct. 2000.

[3] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. SIGSOFT *Symposium on the Foundations of Software Engineering*, Dec. 1994.

[4] S.C. Johnson. lint, a C Program Checker, *Computer Science Technical Report*, Number 65, 1978.

[5] Philip Koopman. Toward a Scalable Method for Quantifying Aspects of Fault Tolerance, Software Assurance, and Computer Security. *Computer Security, Dependability, and Assurance: From Needs to Solutions (CSDA'98)*, Nov. 1998.

[6] Roy A. Maxion and Robert T. Olszewski. Improving Software Robustness with Dependability Cases. *28th International Symposium on Fault Tolerant Computing*, June 1998.

[7] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Pub. 1997.

[8] P.J. Plauger. *The Standard C Library*. Prentice Hall, 1992.

Defect and Fault Seeding In Dependability Benchmarking

Barry Boehm, Daniel Port
University of Southern California
{boehm, dport}@sunset.usc.edu

Abstract

Defect and fault seeding is often considered for gathering empirical estimates within reliability models. Traditional defect seeding is fraught with difficult to resolve validity concerns when attempting to estimate true defect and fault populations. With dependability benchmarking we are less concerned about true defect and fault estimates, rather we wish to compare the relative effectiveness of dependability approaches. We propose that in this context the traditional concerns regarding defect and fault seeding techniques may not be as difficult to address and that potential new approaches may be useful as a means of benchmarking approaches to dependability.

1. Introduction

Within the context of traditional defect seeding suppose you use a dependability benchmarking capability to compare the performance of two tools on a system under test (SUT) – or its equivalent in code, design, or specification analysis – and both tools find the same 3 defects. This gives you a good comparative analysis of the tools, but leaves you wondering whether these three defects are 100% of the three remaining defects in the SUT, 10% of the 30 remaining defects, or something else.

Defect seeding approaches attempt to estimate the size of the defect population and the absolute effectiveness of defect detection techniques by deliberately introducing defects into a system. The general approach is:

- Insert N defects in the system under test (SUT).
- Run the tests, find M seeded defects, K unseeded defects.
- Estimate remaining defects as:

$$R = K * ((N-M)/M), \text{ from } K/(K+R) = M/N.$$

Thus, if you seed the SUT with 10 defects and each tool also finds 6 of the 10 seeded defects, you can estimate that the 3 defects found by the tools are 60% of 5 previously-undetected defects in the SUT, and that two remain.

The defect seeding concept has been around since at least the early 1970's, but has fallen from practice because of difficulties in seeding defects in ways which satisfy the underlying assumptions of the estimation formula. These include:

1. The seeded defects are representative of existing defects. Seeding is mostly done by developers, whose blind spots miss many sources of defects.

2. The test profile is representative of the operational profile. Again, the developers' knowledge of actual usage patterns is generally highly imperfect.

3. The SUT is developed without knowledge of the seeding profile. If the seeded defects become well-known, there are risks of consciously or unconsciously tailoring the tool to look good on the seeded defect sample.

4. The source code available for defect seeding. As systems become increasingly COTS-based, this difficulty increases.

Individually and in concert, therefore, these assumptions are often invalid to some extent, leading to inaccurate estimates. However, there are not many strong alternative approaches available, and we feel it is worth exploring new approaches to defect seeding, which significantly strengthen the ability to satisfy these assumptions.

2. Potential New Approaches

We present several approaches that may help address some of the complications of the traditional approach to defect seeding (see [1] for details on several of these).

1. Change Histories. If the SUT is (say) version 3.4 of a given system, one can use fixes from earlier versions as sources of seeded defects. These have the representativeness advantage of having been real defects, but have the shortfall of having been the most detectable defects using current techniques. Also, the version changes may be complex combinations of defect fixes and

general upgrades, which makes preparing an appropriately-seeded SUT more difficult.

2. N-Version Programming. One can generate further representative defects by giving the specs to different programmers and generating a family of SUT versions. Comparative analysis of the defects found in the SUT versions can also generate estimates of the likely number of residual defects. Studies of N-version programming have shown that it is an imperfect source of independent implementations, and it can also be expensive, but it appears to be worth exploration. Program mutations are a similar source of defect-seeding alternatives.

3. Randomized Defect Seeding. One way to reduce the risks of gaming the seeding profile is to select random seeded defects from a large and/or parameterized sample. This also opens up the possibility of using multiple-run population estimators such as jackknife and bootstrap methods.

4. Use of Defect Distribution Statistics. One can also combine randomized defect seeding with defect distribution statistics to address the defect representativeness issue. Orthogonal Defect Classification statistics are a good example.

5. Connectors and Wrappers. One can deal with COTS defect and fault seeding to some extent by using connectors or wrappers to simulate potential real faults. Examples are data corruption faults via wrapper modifications of the output stream, or uses of connectors to generate communication failures (timing, handshaking, noise, etc.)

3. Issues for Discussion

Some issues worth exploring at the Workshop include:

- Mapping of preferred defect seeding approaches to dependability attributes. These include: Robustness (reliability, availability, survivability), Protection (security, safety), Quality of Service (accuracy, fidelity, performance assurance), and Integrity (correctness, verifiability).

- Alternative concepts and approaches. These could include mutation testing as defect seeding; model-driven approaches; information theoretic approaches; or game theoretic approaches.

- Special application challenges, such as scalability, test oracles (e.g., for agent-based systems of systems), and Heisenbug effects (additional defects induced by seeded defects, such as timing and synchronization defects).

4. References

[1] Voas, J., McGraw, G., *Software Fault Injection*, Wiley, 1998.

System Recovery Benchmarking

Ji Zhu, James Mauro, Ira Pramanick
Sun Microsystems, Inc.
{ji.zhu, james.mauro, ira.pramanick}@sun.com

Abstract

This paper presents the rationale and usefulness of developing a system recovery benchmark. The speed with which a system can return to service following an outage is a critical factor in overall system availability. General purpose computer systems, such as UNIX based systems, tend to execute the same sequence or series of steps during outage recovery and system startup. Our experience has shown that these steps are repeatable and measurable, and can thus be benchmarked, much like performance benchmarks (e.g. TPC, SPEC). A defined set of measurements, coupled with a specification for representing the results and system variables, provides the foundation for system recovery benchmarking.

1. Introduction

In [7], a hierarchical framework, named R^3, is established to benchmark availability. R^3 represents the three system attributes that are key to availability identified in the framework. These attributes are Rate (rate of fault and maintenance events), Robustness (a system's ability to handle fault, maintenance, and system-external events, and the resulting degree to which it remains available in the face of these events), and Recovery (the speed with which a system can return to operation following an outage). The R3 framework provides for a benchmark that incorporates all defined attributes, thus yielding a downtime-per-year metric. Alternatively, benchmarks can be defined that focus on some subset of the framework attributes and their sub-metrics. This is very similar to the performance benchmarking space, where there are benchmarks that measure the system as a whole, as well as benchmarks that measure a specific subsystem.

In this paper, we discuss our rationale for creating a benchmark specification designed to measure the recovery attribute of general purpose computer systems. Our objective is to convey two basic assertions: recovery time can be benchmarked; and a recovery time benchmark is useful and meaningful. Previous research in availability benchmarking has been focused on benchmarking system robustness attribute [1, 2, 3, 6]. There has been a lack of research in benchmarking recovery aspect of system availability.

2. Background and Motivation

Historically, there have been several design approaches to building computer systems that can meet strict business requirements for availability. Fault tolerant systems implement lock-step execution with results comparison across redundant hardware components, providing the ability to detect and recover from faults without a service disruption. Such designs have carved out a niche market in the industry, but have not seen broad adoption due to cost (overall price/performance) and scalability considerations.

General purpose computer platforms, such as Unix-based servers, offer designs that allow system to recover quickly from an outage. In the context of this paper, quick recovery refers to a computer platform returning to service in an automated fashion.

On a standalone server, quick recovery is facilitated through system firmware and component blacklisting capabilities. When a fault event occurs, the system panics and reboots. During the reboot process, hardware diagnostic software configures around the failed component, allowing the system to return to service quickly and without human intervention. Clustered systems provide redundancy by clustering two or more systems together with a software framework that does cluster management, failure detection and automated recovery. When a cluster node fails, the cluster software initiates a failover of the services that were being provided by the failed node, to other nodes in the cluster, thereby minimizing service disruption.

There are several major benefits of choosing quick recovery over fault tolerance. General purpose computers provide better price/performance, better scalability, and a much larger selection of commercially available software. From an availability perspective, general purpose computers have been proven adequate even in environments demanding very high levels of availability. General purpose computers have been installed across a broad range of industry segments to run mission critical applications for years.

3. Can We Benchmark Quick Recovery Time?

General purpose computer systems tend to execute the same sequence or series of steps during outage recovery and system startup. On a standalone server, quick recovery is mostly in the form of a kernel panic call and a core dump, followed by a system reboot, which usually includes firmware-based hardware diagnostics prior to the operating system boot. On clustered systems, quick recovery is in the form of a reconfiguration of cluster framework and a restart of services on the surviving nodes. Our experience has shown that recovery times are repeatable and measurable, and can thus be benchmarked, much like performance benchmarks (e.g. SPEC[4], TPC[5]).

4. Is A Quick Recovery Time Benchmark Meaningful?

For general purpose computers, outages that can quickly be recovered from, account for most of the system outages. On mid-range and high-end Unix servers, systems can quickly recover from faults of most its components (processors, memory, cache, interconnect, controller, etc.). Together these components account for up to 80% of total system hardware failure rate. Software faults can also be worked around by rebooting the system. For clustered systems, failover is the dominant mode of recovering from a hardware or a software fault.

Most outages on mid-range and high-end general purpose computer servers fall into the quick recovery category. This makes a system quick recovery benchmark useful for evaluating a broad range of systems, as it provides a meaningful representation of system outage duration.

5. Issues

A full discussion of the issues and their solutions in a quick recovery benchmark is outside the scope of this position paper. We are working on them and will report our progress in the future. The following is a list of a few important issues we have identified in the quick recovery benchmark for a standalone server.

- System Size - The number of processors, amount of physical memory, number of IO channels and the number of actual storage devices (disks) directly impact system recovery time.
- Service Processor - Systems with service processors add a level of complexity to a full restart cycle if the service processor must be restarted as well.
- Domains - Single physical systems able to run multiple operating system kernel instances have multiple recovery scenarios; those effecting the domain, and those effecting the entire system.
- Firmware Setting - Most systems offer parameters that determine the level of diagnostic testing the system will do at startup/restart. Going from least intensive to most intensive settings can dramatically alter restart time.
- File Systems - File system integrity checking is a critical phase of system startup/restart. The number and size of file systems, as well as options such as logging, can have a significant impact of startup/restart time.

We are working on defining outstanding issues for clustered systems.

6. Conclusions

It is our contention that quick recovery time measurement is a meaningful availability benchmark since it represents one important aspect of system availability - the outage duration when a recoverable fault occurs. We believe that it is an attainable goal to develop a benchmark on quick recovery time much like performance benchmarks. We are currently working on system quick recovery benchmark specifications for a standalone server and a clustered system.

7. References

[1] P. Koopman et al, "Comparing Operating Systems Using Robust Benchmarks", *Proceedings of the 16th Symposium on Reliable Distributed Systems*, pp. 72-79, Oct. 1997.

[2] H. Madeira and P. Koopman, “Dependability Benchmarking: making choices in an n-dimensional problem space.” *Proceedings of the first workshop on Evaluating and Architecting System Dependability*, July, 2001.

[3] D. P. Siewiorek et al, "Development of a Benchmark to Measure System Robustness", *Proceedings of the 1993 International Symposium on Fault-Tolerant Computing*, pp. 88-97, June 1993.

[4] Standard Performance Evaluation Corporation, http: //www.spec.org/, *Current.*

[5] Transaction Processing Council, *http://www.tpc.org/,* Current.

[6]. T. Tsai et al, "An Approach Towards Benchmarking of Fault Tolerant Commercial Systems", *Proceedings of the 1996 Symposium on Fault-Tolerant Computing*, pp. 314-323, June 1996.

[7] J. Zhu et al, "R-Cubed: Rate, Robustness and Recovery - An Availability Benchmark Framework". To appear in *Technical Report Series*, Sun Microsystems Inc.

Faultload Representativeness for Dependability Benchmarking

Jean Arlat and Yves Crouzet
LAAS-CNRS, 7 Avenue du Colonel Roche, 31077 Toulouse Cedex 4, France

1. Introduction

In spite of several pioneering efforts (e.g., see [1-4]), and the related initiatives currently being developed — e.g., the IFIP WG 10.4 SIGDeB[1] and the European IST project DBench [5], there is still a significant gap between i) the level of recognition attached to robustness benchmarks and fault injection-based dependability benchmarking, and ii) the wide offer and broad agreement that characterize performance benchmarks (e.g., see [6]). Much effort is needed before the same standing can eventually be achieved.

In practice, basic attributes such as workload, faultload, measurements and measures precisely characterize a dependability benchmark. Clearly, the determination of a *representative faultload* is one of the key issues for specifying a dependability benchmark. In particular, one important question is to figure out whether a focused set of techniques (ideally, a single one) could be identified as sufficient to generate a faultload for many classes of faults, or whether a distinct technique is needed for each class.

2. Fault Injection-based Validation

For the past 30 years, many efforts were reported on the use of fault injection for contributing to the validation of fault-tolerant systems, sometimes in cooperation with other dependability validation techniques (e.g., formal verification or analytical modeling).

Numerous injection techniques were proposed [7], ranging from i) simulation-based techniques at various levels of representation of the target system (physical, logical, RTL, PMS, etc.), ii) hardware techniques (e.g., pin-level injection, heavy-ion radiation, EMI, power supply alteration, etc.), and iii) software-implemented techniques that support the bit-flip model in memory elements. Many tools were developed to facilitate the conduct of experiments based on these various techniques.

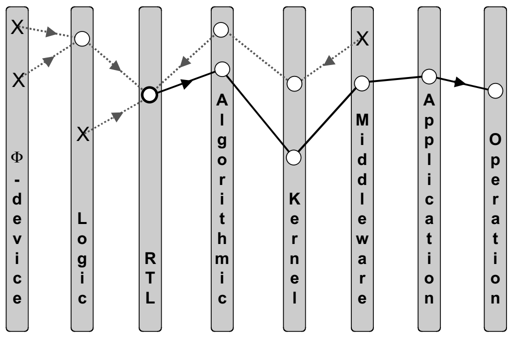
Building on these advances, fault injection made progressively its way to industry, where it is actually part of the development process of many manufacturers, integrators or stakeholders of dependable computer systems. This confirms the pertinence of the approach. Nevertheless, the investigations carried out concerning the comparison of the impact of i) some specific injection technique with respect to real faults (e.g., see [8, 9]) and ii) several injection techniques (e.g., see [10, 11]), have shown mixed results. Some techniques were found to be quite equivalent, while others were rather complementary. Thus, it is necessary to provide a general framework to better analyze the variability of these results and help develop a comprehensive research effort to better address the question raised at the end of Section 1.

[1] Special Interest Group on Dependability Benchmarking — See: http://www.dependability.org/wg10.4/SIGDeB.

3. Faultload Representativeness

In the case of dependability benchmarking, the main question is to identify the technology that is both *necessary* and *sufficient* to generate the faultload to be included into a dependability benchmark. Several important issues have to be accounted for in this effort:

1) As shown by Figure 1, several relevant levels of a computer system can be identified where faults can occur and errors can be observed (e.g., physical-device, logic, RTL, algorithmic, kernel, middleware, application, operation). Concerning faults, these levels may correspond to levels where real faults are considered and (artificial) faults can be injected. Concerning errors, the fault tolerance mechanisms (especially, the error detection mechanisms) provide convenient built-in monitors.
2) For characterizing the behavior of a computer system in presence of faults, it is not necessary *a priori* that the injected faults be "close" to the target faults (reference), it is sufficient that they induce similar behaviors. Indeed, similar errors can be induced by different types of faults (e.g., a bit-flip in a register or memory cell can be provoked by an heavy-ion or as the result of a glitch provoked by a software fault). What is important is not to establish an equivalence in the fault domain, but rather in the error domain.
3) What matters is that the respective error propagation paths *converge* before the level where the behaviors are observed. Two important parameters can be defined on these various levels (Figure 2):
   - the *distance* $\boldsymbol{d_r}$ that separates the level where faults are injected from the ***reference*** fault level(s);
   - the *distance* $\boldsymbol{d_O}$ that separates the level where the faults are injected from the levels their effects are ***observed***.



X: reference fault locations — O: Observation locations
Figure 1: Target system levels and fault pathology

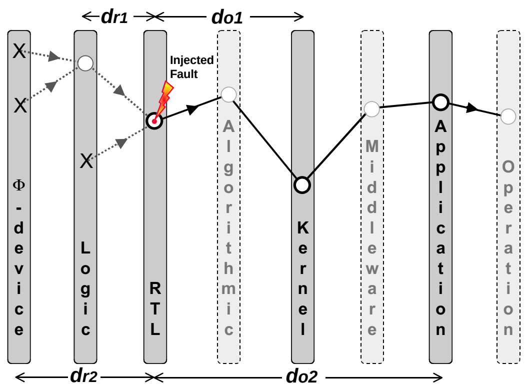


Figure 2: Reference fault and observation distances

The shorter $\boldsymbol{d_r}$ and the longer $\boldsymbol{d_o}$, the more it is likely that the injected faults will exhibit behaviors similar to those provoked by the targeted reference faults.

4) In practice, it may be the case that the presence of a specific fault tolerance mechanism (FTM) on one target system (and not on the other one(s)) will alter the error propagation paths. This has a significant impact on the scope of (real) faults actually covered by the injected faults, whenever the FTM is implemented at level located between the level of the targeted faults and the level where the faults are injected and thus intercept the error propagation paths. Indeed, assuming a perfect (100%) coverage for the FTM, then representativeness (with respect to the targeted faults) of the benchmark using the faultload characterized by the injected faults would then be zero. This could be simply accounted for by introducing another distance parameter: the *distance* $\boldsymbol{d_m}$ separating the level where the faults are injected from the level where the fault tolerance ***mechanism*** is acting.

5) From a dependability benchmarking point of view, it might not always be possible nor cost-effective to have access to the actual structure of the target system to identify *a priori* a faultload complying with the representativeness property. Accordingly, an alternative could be to favor a standard fault injection technique that is less than perfect, but that is easy to implement and that induces a large set of errors, and then to establish a dialogue with the target system provider in order to derive a fair interpretation or post processing of the benchmark measurements.

4. Concluding Remarks

The paper proposed a framework for addressing and illustrating explicitly — with respect to the various *levels* were faults may affect a computer system — the problems attached to the definition of a *faultload* to be used for dependability benchmarking, Thus, the presentation has focused on the related *distances* that are useful to precisely characterize the links between the target faults, the injected fault and the measurements carried out.

For sake of brevity, other related issues (e.g., *"distance"* between targeted and injected faults with respect to *frequency/distribution* of occurrence) were not considered. Nevertheless, we advocate that a possible extension to this framework aimed at accommodating such a concern can be derived from the results presented in [12].

Finally, it is worth pointing out that, even in such a preliminary form, this framework was found helpful within the DBench project [5] for defining and coordinating the experiments aimed at assessing the *representativeness* and *equivalence* of various fault injection techniques.

Acknowledgement. This work is partially supported by the DBench project (IST 2000-25425). Our thanks go to Jean-Charles Fabre, Karama Kanoun and Jean-Claude Laprie at LAAS, as well as Henrique Madeira (Coimbra Univ., Portugal) and Philip Koopman (CMU, USA), for the insightful discussions on the issues addressed herein.

References

[1] T. K. Tsai, R. K. Iyer, D. Jewitt, "An Approach Towards Benchmarking of Fault-Tolerant Commercial Systems", *Proc. FTCS-26,* Sendai, Japan, 1996, pp. 314-323 (IEEE CS Press).

[2] A. Mukherjee, D. P. Siewiorek, "Measuring Software Dependability by Robustness Benchmarking", *IEEE Trans. on Software Engineering*, vol. 23, no. 6, pp. 366-323, 1997.

[3] P. Koopman, J. DeVale, "Comparing the Robustness of POSIX Operating Systems", *Proc. FTCS-29,* Madison, WI, USA, 1999, pp. 30-37 (IEEE CS Press).

[4] J. Arlat, J.-C. Fabre, M. Rodríguez, F. Salles, "Dependability of COTS Microkernel-Based Systems", *IEEE Trans. on Computers*, vol. 51, no. 2, pp. 138-163, 2002.

[5] K. Kanoun, J. Arlat, D. J. G. Costa, M. Dal Cin, P. Gil, J.-C. Laprie, H. Madeira and N. Suri, "DBench – Dependability Benchmarking", *Supplement of Proc. DSN-2001,* Göteborg, Sweden, 2001, pp. D.12-D.15. (http://www.laas.fr/DBench).

[6] J. Gray (Ed.), *The Benchmark Handbook for Database and Transaction Processing Systems, S*an Francisco, CA, USA: Morgan Kaufmann Publishers, 1993.

[7] J. V. Carreira, D. Costa, J. G. Silva, "Fault Injection Spot-checks Computer System Dependability", *IEEE Spectrum*, vol. 36, pp. 50-55, August 1999.

[8] M. Daran, P. Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations", *Proc. ISSTA'96,* San Diego, CA, USA, 1996, pp. 158-171 (ACM Press).

[9] H. Madeira, D. Costa, M. Vieira, "On the Emulation of Software Faults by Software Fault Injection", *Proc. DSN-2000,* New York, NY, USA, 2000, pp. 417-426 (IEEE CS Press).

[10] D. T. Stott, G. Ries, M.-C. Hsueh, R. K. Iyer, "Dependability Analysis of a High-Speed Network Using Software-Implemented Fault Injection and Simulated Fault Injection", *IEEE Trans. on Computers*, vol. 47, no. 1, pp. 108-119, 1998.

[11] P. Folkesson, S. Svensson, J. Karlsson, "A Comparison of Simulation Based and Scan Chain Implemented Fault Injection", in *Proc. FTCS-28,* Munich, Germany, 1998, pp. 284-293 (IEEE CS Press).

[12] D. Powell, E. Martins, J. Arlat, Y. Crouzet, "Estimators for Fault Tolerance Coverage Evaluation", *IEEE Trans. on Computers*, vol. 44, no. 2, pp. 261-274, 1995.

What's Wrong With Fault Injection As A Benchmarking Tool?

Philip Koopman
ECE Department & ICES
Carnegie Mellon University
Pittsburgh, PA, USA
koopman@cmu.edu

Abstract

This paper attempts to solidify the technical issues involved in the long-standing debate about the representativeness of fault injection as a tool for measuring the dependability of general-purpose software systems. While direct fault injection seems appropriate for evaluating fault tolerant computers, most current software systems are not designed in a way that makes injection of faults directly into a module under test relevant for dependability benchmarking. Approaches that seem more likely to create representative faults are ones that induce exceptional operating conditions external to a module under test in terms of exceptional system state, exceptional parameters/return values at an API, failed system components, or exceptional human interface inputs.

1. Introduction

Fault injection has long been used as a way to evaluate the robustness of computer systems. Traditional fault injection techniques involve perturbing or corrupting a computer's memory or operating elements to create a fault. Mechanisms to inject faults include heavy-ion radiation, probes that inject electrical signals into hardware circuits, mutation of programs, and software probes that corrupt system state as programs are being executed. These techniques are widely considered useful in evaluating a system's response to a hardware fault such as a radiation-induced bit value inversion or a "stuck-at" fault. They also can be effective at exercising various fault handling mechanisms, both software and hardware.

While previous generation system dependability was often dominated by hardware faults, in the vast majority of modern computer systems it is widely believed that software problems are a more frequent cause of system-level failures. Thus, attempts to measure and improve overall system dependability are in need of a way to measure software dependability. Over time, fault injection has come to be used as a controversial technique for assessing software fault tolerance and system-level dependability.

It is arguably the case that the question of representativeness is the single biggest open issue in fault injection today. Fault injection, especially software-based fault injection, is a reasonably convenient and effective way to evaluate systems. However, it is not always clear what fault injection results mean for fielded systems. Proposing the use of fault injection to benchmark system dependability typically triggers vigorous debate, often with polarized viewpoints.

This paper seeks to identify a possible source of the large gap between opposing views in the debate on fault injection representativeness as well as place into perspective some of the strengths and weaknesses of fault injection approaches. While this paper presents opinions rather than scientific facts, it is based on experiences from extensive interactions with industry and the technical community as part of the Ballista robustness testing project as well as two years as co-chair of the IFIP 10.4 WG SIG on Dependability Benchmarking.

This paper has been written to start a discussion rather than end one. None of the issues discussed below are really quite as black and white as they are portrayed to be. In particular, any tool that affords a different view of a system for analysis brings with it some benefits. However, the disparity between benefits claimed by some fault injection proponents versus the lack of value perceived by some fault injection opponents suggests that a clear and precise articulation of issues would help focus the debate. It is our hope that this paper will put future discussions on this topic on a more solid technical foundation.

2. Fault injection as a fault tolerant computing evaluation tool

One way to understand fault injection is to hypothesize a system that would be sure to detect and recover from all possible fault injection experiments (*i.e.*, one that would be perfect if evaluated using fault injection, although of course in practice such perfection is not attainable). We assume the classical situation in which faults are only injected within the boundaries of a software module under test (other situations are discussed Section 5). Thus, a software fault injection experiment consists of injecting a fault into the data or program of a module and evaluating whether the result matches that of a so-called golden run of an uncorrupted copy of that same module. Compared to traditional software testing techniques, this method of evaluation has the distinct virtues of being relatively inexpensive to develop and execute, as well as requiring minimal information about the software being tested.

A software module which would be completely invulnerable to such injected faults would have to incorporate exception handlers and acceptance tests. Exception handlers would catch and convert any possible exceptions generated by an injected fault into a clean error return operation followed by a subsequent recovery operation. Because arbitrary changes can be made to a module under test by fault injection, exceptions generated can potentially include those from illegal instructions, memory protection violations, and other activations of machine checks. However, not all program state corruptions result in exceptions. Thus, acceptance tests would also be required in order to catch and recover from any possible incorrect (but non-exceptional) result due to program or data mutation.

The above description of an idealized system should help explain why fault injection is historically popular in the fault tolerant computing community. A system that can withstand an arbitrary fault can be said to be dependable without concern for the specifics of the situation it is being used in. Systems that employ fault tolerant techniques often have hardware support that tends to detect or correct injected faults, as well as software that is designed for recoverability after failure and for self-checking of computational results. Note that the requirement for an acceptance test that can detect an arbitrary fault can be difficult to meet, and often leads to multi-version software design and other expensive implementation techniques that can be justified only for the most critical systems.

Perhaps surprisingly, the widely used technique of input value validity checking is not sufficient to assured dependability against arbitrary fault injections. In the common case, an injected fault can be expected to create an exception or incorrect value after a module performs validity checking on input values. Thus, software would have to be able to handle arbitrary corruptions of data values even if they had previously passed input validity tests. Adding input value tests can help strengthen a weak system by adding indirect acceptance tests so that one module's output gets tested as it becomes the next module's input. But, it does not seem that a fully robust system can rely upon them exclusively.

Put another way, all exceptions that can be generated by hardware in any circumstance must be handled, even ones that appear to be impossible based on the software design. For example, a null pointer check might be changed by an injected fault into an instruction that corrupts a pointer instead of ensuring it is valid – there is no assurance that input validity checks will suffice.

The issue of what happens when exception handlers or acceptance tests are themselves corrupted is of course relevant. One can argue that the chance of a value being changed after its validity check is larger than the chance that an arbitrary fault will result in both a corrupted value and the propagation that value past an acceptance test. But regardless of that argument, it is clear that validity checks alone do not suffice to ensure dependability for an arbitrary fault model.

Although it is still an open question whether direct injection of arbitrary faults leads to representative results with regard to software dependability evaluation, systems built to withstand arbitrary fault injections successfully can clearly endure very difficult operating conditions. And in fact these are the kind of systems that the fault tolerant computing community has long dealt with. While it is impossible to be perfectly robust under every conceivable set of concurrently injected faults, systems that perform well during a fault injection campaign truly deserve to be called robust, but are likely to achieve this result at significant expense.

The controversial question of concern is: should we measure all systems by their response to arbitrarily injected faults? Or, for that matter, is it appropriate to compare any systems by such a standard?

3. Everyday, but mission-critical, software

In most computing, even many mission-critical computing applications, the expense of creating and operating a completely fault tolerant computing system can be too high. Thus, optimizations are employed to reduce development and run-time costs. While programs employing these optimizations can at times be less dependable than more traditional fault-tolerant software systems, there is no essential reason why this need be the case for the types of faults actually encountered in everyday operation.

First and foremost, general purpose computing assumes that hardware faults are rare, are detected by hardware mechanisms, and need not be dealt with by application software. This is by and large a reasonable set of assumptions. In current systems, system crashes and problems created by software failures seem to outnumber hardware failures by a large margin. Furthermore, well known and affordable hardware techniques such as using error detection coding on memory and buses are available to reduce the chance of undetected hardware faults and, if desired, provide automatic recovery from likely hardware faults. (Whether customers decide to actually pay for hardware reliability features is in fact an issue, but arguably one that won't be resolved by creating even more hardware or software reliability mechanisms that have non-zero cost to implement or execute.) Mainframe computers have provided very reliable hardware for many years – thus the capability to provide effectively failure-free hardware environments is more a matter of economy than practicability.

Given that application software can consider hardware to be failure-free for practical purposes, significant and important optimizations can be employed to simplify software and speed up execution. The general idea behind these optimizations is to avoid repeating validity checks and avoid providing unnecessary exception handlers.

Given failure-free hardware, validity checks need only be computed once for any particular value. As a simple example, if a memory pointer must be validated before it is dereferenced, it suffices to check pointer validity only once upon entry to a module (and after pointer updates) rather than every time that pointer is dereferenced. Many such checks can be optimized using the general technique of common subexpression elimination. (Compilers might well perform this optimization even if checks are explicitly included before each pointer use in source code.) Furthermore, if a set of related modules is designed to be used together, such checks can be optimized across module boundaries and only invoked at external entry points into the set of modules. Similarly, if exception handlers are preferred to data validity checks, these need only be provided at external software interfaces.

Of course it is well known that data validity checks used in practice are typically imperfect and could easily be improved in many systems. However, the important point for present purposes is that this is the preferred approach to attaining robustness in such systems, and that most such systems typically have at least some validity checks of this nature in place. For example, in mission-critical software systems, it is common practice to add additional "wrappers" to perform validity checks at strategic places in the system, such as before every call to the operating system.

An additional situation that can be considered an optimization is that acceptance tests used at the end of module execution are often omitted or reduced to rudimentary form. Instead, what is supposed to happen is that correctness checks are moved from run time to development time, and take the form of software testing suites and compiler analysis to ensure software conformance to specifications. Given the assumption of no hardware failures, this is a reasonable way to reduce fielded software size and execution time while ensuring correct operation. It is of course recognized that test suites are generally imperfect; however this does not change the fact that this is the prevalent approach in ordinary software systems. Even in systems with rigorously defined exception-handling interfaces, it is commonly the case that only exceptions that are expected to be generated are supported, documented, and tested. Exceptions that are impossible to generate, based on analysis of source code and a failure-free hardware assumption, are usually not handled gracefully since doing so would be seen as a waste of development resources within a finite budget of time and money.

Now let us say that a software module developed according to the above implementation philosophy is subjected to fault injection as a way to exercise the system in an attempt to find software dependability problems. The important question for this case is how to interpret a failure that is created by a fault injected into the code or data spaces of a module under test. There are two main cases of interest to consider based on the fault injection outcome.

(1) The injected fault produces a valid, but incorrect, intermediate result that propagates to the output. If there are no correctness defects in the module under test, this can only correspond to incorrect data being presented at the external inputs of the module. That situation does not correspond to a defect of any kind in the module under test because the program that was actually written does not produce the same output as the fault-injected output for that particular system state and set of input conditions, and is therefore not representative of expected failures. Saying that an ordinary software module is defective because it gives a different answer when processing internally corrupted data (or executing arbitrarily modified instructions) than when processing uncorrupted data is, in the general case, unreasonable.

A different argument that is sometimes made is that such a situation represents a *hypothesized* fault (similar in intent to the concept of "bebugging" via source code mutation in the software testing community). In this case such a fault can only be argued to be useful in evaluating the effectiveness of the test suite. But even then interpretation of results must be made carefully if tests have been designed in a "white box" manner, taking into account the structure of the module being tested. It might not be reasonable to criticize a test suite for failing to find a defect in a program that has been mutated to have a different structure or functionality than the one the tests were designed for. In other words, the concept of optimization can and does extend to designing tests that are only relevant for the software that has been specified in light of reasonable and common defects. Furthermore, this use of fault injection requires availability of a test suite, which is seldom the situation encountered when fault injection is used as a dependability metric.

(2) The injected faults produce an inappropriate exception or crash. In this case determining the correctness of module functionality is simplified by applying a default specification that the module should not produce an undocumented (or unrecoverable) exception and should not crash or hang. These sorts of module failures can be legitimately called robustness failures, which reduce software dependability, but only if they could plausibly happen during the execution of a real application program.

Consider a pointer validity check to illustrate the issue of representativeness in this situation. A program might check a pointer passed as an input parameter at the beginning of a module, and then dereference it many times during execution without altering its value. Given that hardware faults are not under consideration, that program can never suffer a memory protection violation by dereferencing that pointer. However, a fault injection campaign might well create an invalid pointer and a subsequent exception or crash. It is difficult to claim that this result measures software dependability, because it could never happen during production use. Even more importantly, it is unreasonable to claim that it is representative of a hypothesized software robustness defect, because in fact this is a case in which the software was specifically designed to be robust to the very type of fault that was injected!

To further illustrate the problem with representativeness in this case, consider a situation in which a null pointer check has been omitted from a piece of off-the-shelf software available only in executable form. The traditional solution to cure robustness failures of this type would be to add an external wrapper to the module that checks for pointer validity. Adding that wrapper makes the software robust to null pointer inputs. But, this wrapped module would still suffer an exception in response to a fault injection that creates a null pointer after the wrapper performs its check. Thus not only would robust software appear to have robustness failures under a fault injection experiment, but hardened software would similarly appear to be non-robust.

Software that performs acceptance tests might tend to appear more robust under fault injection than software that just performs input validity checks. This is because any injected faults that are activated during the execution of a module would, theoretically, be detected and handled by an acceptance test. However, optimizations would still introduce vulnerabilities to fault injection, such as not implementing exception handlers for exceptions that are impossible in fault-free hardware operation. For an invalid pointer example, there is no reasonable basis for incurring the cost of memory address exception handlers if all pointers are known to be valid based on previous validity checks.

Thus, direct fault injection experiments are not suitable for general-purpose software robustness benchmarking. They are unlikely to show an improvement in robustness when proven techniques are used to improve software robustness via filtering out invalid or exceptional input values. While it could then be claimed that fault injection would alternately test whether all possible exceptions were caught by a module, this approach has a similar problem in that fault injection is likely to generate exceptions that can't possibly happen in real program runs. Furthermore, even adding exception handlers will not necessarily catch injected faults that generate unexpected exceptions or, worse, disable or subvert the exception handler itself.

4. The importance of realistic fault activation

The key problem with using direct fault injection on a software module is that doing so does not take into account whether the injected fault can be activated by inputs during real execution, and similarly whether it can slip by any available exception handler. Mainstream software development, even for critical systems, traditionally achieves robust operation in large part by restricting the ability of exceptional values to activate potential fault sites via a set of input validity checks. It then seeks to handle only those exceptions that can actually be encountered (*i.e.*, ones for which input checks are not implemented or cannot provide complete coverage) to minimize software complexity.

Of course most software is certainly not perfect. Not all values are checked before being used. Not all exceptions that can be generated are handled gracefully. However, it appears that fault injection has trouble distinguishing whether a general purpose software module (as opposed to a specially created fault tolerant software module) would in fact be operationally robust.

To use an analogy to the problem under consideration, consider the task of making a camping tent absolutely dark, with no light entering whatsoever, perhaps for the purpose of photographic film processing. A single layer of black plastic sheeting might suffice for this task, but might also be imperfect or develop holes while in use. A fault tolerant computing analogy might well involve using multiple layers of plastic sheeting to reduce the probability of aligned multi-layer holes. Fault injection would then involve putting a few small holes in individual layers to ensure that light still did not penetrate, simulating the process of holes appearing due to wear and tear during use.

By the same analogy, a fault injection technique applied to a cost-sensitive situation would be to prick holes in a single layer plastic tent and assert that light indeed penetrated into the tent when a hole was made (assume that after each such hole is evaluated it is automatically patched). Furthermore, fault injection might prick and illuminate holes in portions of the tent which were not exposed to light in the first place such as plastic buried under dirt at the edges of the tent. (A more extreme version of fault injection would be to shine a flashlight inside the tent and assert that light had entered; but an expected response in that case would be to eject the person holding the flashlight!)

While pricking holes in a single-layered plastic tent indeed serves to illustrate the vulnerability of using a single layer of plastic, one should not be surprised that owners of such a tent don't see much value in a fault injection exercise. A more effective approach for their situation would be to create ways to identify existing holes so they could be patched, perhaps by entering the tent and patching places where light leaked in, analogous to installing software wrappers to improve dependability. Thus, the core problems of a fault-injection based approach in this analogy of an *optimized, cost-sensitive system* are (1) identifying defects that aren't in the system *as it is actually implemented*, and (2) creating false alarms to such a degree that effort is arguably better spent on other activities. Saying that optimizing systems for cost can lead to less robust systems is of course a valid point, but does not change the fact that most purchasers demand optimized systems; insisting that no such systems be implemented is simply unrealistic.

The key problem is one of whether activation of a given potential fault could actually occur in a real system under real operating conditions, or if it can occur, whether it takes place under a condition that the designers of the system feel justified in ignoring. Direct fault injection creates situations that might be impossible to achieve in real execution, by for example creating an exceptional value just after a validity check. Without a measure of whether activation of an injected fault is possible in a real system, it is difficult to use the results of direct fault injection for everyday software. For this reason, direct fault injection is ultimately no substitute for a good development and test process. It can, however, help in designing a module to tolerate design faults contained in *other* modules and its execution environment as discussed in the next section.

5. Representative uses of fault injection

Given the above discussion, it would be a mistake to conclude fault injection is of no use in evaluating the dependability of mainstream software. The real issue is using fault injection in a manner appropriate to the relevant assumptions and development model for any particular system under consideration. For systems designed to tolerate generalized failures as described in Section 2, fault injection seems like a useful tool to use in overall system evaluation. But lack of tolerance for directly injected faults does not demonstrate software undependability; it merely fails to make a case for software being dependable in the face of arbitrary runtime faults.

Some of the ways in which fault injection can be helpful are listed below. Note that these techniques avoid injecting faults directly into the module under test, but rather inject faults into external interfaces or the surrounding environment. Thus all such approaches have the virtue of being non-invasive with respect to the actual execution of the module under test and thus present a fault model that does not assume a particular approach is used within a module for providing dependable operation.

- **Callee interface fault injection.** Faults can be injected into the inputs of a module to check for out-of-specification responses. The presumption here is that some external software defect exists that feeds an exceptional value (one that is invalid rather than merely incorrect) to the module under test. This approach is only representative to the degree that injected faults could really happen in operation. It tends to be most useful for very widely used application programming interfaces (APIs) for which all uses cannot possibly be foreseen, as well as for faults injected that are common results of programming errors (*e.g.*, submitting a null value as a pointer parameter to a procedure call). A variation on this theme is testing for security vulnerabilities such as buffer overflow vulnerabilities.
- **Caller interface fault injection.** Faults can be injected in values returned to the module under test from other software that it calls via external calling interfaces. Representativeness in this situation can be tricky to determine if the module is only destined to run on one version of supporting software if that software cannot actually generate the faults used for testing. However, the utility of this approach increases if the module under test must be portable or the underlying software can be expected to be upgraded or revised over time.
- **Forced resource exhaustion and environmental exceptions.** Exceptional conditions such as having no allocable memory or a full hard disk can stress a software module under test in terms of its ability to process exceptions. This can be seen as a special case of fault injection into the external caller interface as just described, but is in general an important case to cover.
- **Failure of cooperating processes**, tasks, and computers in a distributed system.
- **Configuration, maintenance**, and other failures in the supporting infrastructure and execution environment, including failed hardware components such as disk drives, missing or incorrectly configured software supporting the execution of a module under test, and even installation of incompatible versions of software.
- **User interface failures**, in which users create exceptional inputs or operational situations.

The above approaches are all relevant to some degree to both fault tolerant software systems as well as general-purpose, optimized software systems because they deal with factors outside the control of designers performing software module optimization.

6. Conclusions

Direct injection of faults into a software module can provide evidence of software dependability. Certainly any piece of software that can tolerate a large fraction of possible arbitrary changes to its operation and still provide correct answers is dependable, and is likely to provide extensive exception handling capabilities to provide that dependability. However, this does not make such fault injection suitable as a benchmarking technique in general, because much software is not designed to withstand fault injection, but rather is optimized assuming correct hardware operation.

Using direct fault injection as a dependability benchmark commits the logical fallacy of "denying the antecedent" (this fallacy is an argument of the form: "If A then B; Not A, thus not B"). In this instance the specific logical fallacy would be: "if software performs well under fault injection then it is dependable; a particular piece of software performs poorly on a fault injection test, thus it is not dependable." Because much software is optimized in ways that preserve dependability for the expected case of failure-free hardware, one *cannot* conclude that software that does poorly at direct fault injection is undependable. The most that can be inferred is that the results provide suggestive evidence that software does not use exception handling as its primary means of providing dependability (but even that statement assumes that injected faults did not affect exception handling itself).

It is possible that the controversy over fault injection continues because of disparate world-views of how robust software should be created. For systems that must achieve comprehensive fault tolerant operation, including withstanding hardware faults, injecting faults directly into a module under test can yield insight into operation in the face of equipment faults. Traditional fault tolerant software, and in particular software that has comprehensive acceptance test support and exception handling, might be evaluated using fault injection techniques. The ability to demonstrate how such injected faults correspond to measurements of software defects rather than hardware defects is still a research topic. But, the arbitrary nature of injected faults that are permitted to activate at the software level rather than being compensated for by hardware mechanisms can be argued to provide a substantial exercise of the software fault tolerance designed into a system.

General-purpose software tends to be optimized to eliminate redundant validity checks and superfluous exception handlers under the assumption that hardware faults are both rare and detected without the involvement of application software. This means that a fault that is injected might well result in a failure that is not representative of the particular software module under test (*i.e.*, the injected fault might create an exceptional value at a point in a computation where such a value would have already been caught by validity checks). Direct fault injection experiments on general-purpose software tend to underestimate the degree of operational robustness available, and in particular are poorly suited to measuring the improvement of robustness afforded by adding input value checks to software interfaces via code modification or addition of robustness wrappers. Thus, fault injection results in this case distinguish *which approach* was used to ensure software dependability (validity checks versus exception handling) rather than solely *whether* the software is in fact dependable.

For systems in which hardware is presumed to have enough built-in capabilities to offer fault-free operation for practical purposes, direct fault injection seems to offer little evaluative benefit in the absence of an accurate fault activation analysis. But, fault injection into the surrounding environment seems attractive as a way to characterize exceptional condition response. It is important in every such experiment to clearly state several things for the results to be meaningful: the fault hypothesis being evaluated, the correspondence between the faults being injected versus that fault hypothesis, and the assumptions being made about the structure of software being evaluated. And finally, it is important to avoid confusing the ability of fault injection to demonstrate robustness with the fact that lack of such a demonstration does not prove that software is non-robust within reasonable operating environments.

7. Acknowledgments

The reader will note a lack of scholarly references in this paper. The important parts of the relevant discussions have taken place in meetings and e-mails over a period of years, and thus are impractical to cite or even attribute. Citing only scholarly publications as sources of various current and historical ideas would overlook that many of these ideas circulated long before they were published or aren't formally published, and would likely provoke a variety of divisive responses from authors who object to our interpretation of their work. Thus we simply assume that the reader is familiar with fault injection as a general technique, and we acknowledge that none of the technical ideas in this paper are new; we simply summarize experiences and possibly present them in a new light.

Discussions with the members of the IFIP WG 10.4 SIG on Dependability Benchmarking and in particular with Henrique Madeira have been invaluable in forming the thoughts presented in this paper. This paper is, however, solely the opinion of the author. This material is based upon work supported in part by DARPA and US Army Research Office under Award No. C-DAAD19 01-1-0646, and the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the view of the sponsoring agencies.