Dependable Computing and Assessment of Dependability

Jean Arlat[1,2]
[1] CNRS; LAAS; 7, avenue du Colonel Roche;
[2] Université de Toulouse; UPS, INSA, INP, ISAE, UT1, UTM, LAAS;
F-31077 Toulouse Cedex 4, France

Abstract

This paper covers the main design and evaluation issues that are to be considered when developing dependable computer systems. In the first part it briefly addresses the fault tolerance techniques (encompassing error detection, error recovery and fault masking) that can be used to cope with accidental faults (physical disturbances, software bugs, etc.) and to some extent, malicious faults (e.g., attacks, intrusions). The second part covers the methods and technique— both analytical and experimental — that can be used to objectively assess the level of dependability achieved.
The trend in controlled experiments, from simple fault injection-based tests meant for evaluating a specific fault-tolerant computer architecture towards the development of benchmarks aimed at comparing the dependability features of several computer systems, is also briefly illustrated.

1 Introduction

This paper addresses the main design and assessment issues that are to be considered when developing dependable computer systems. Special emphasis is put on experimental evaluation of fault tolerance and on recent trends towards dependability benchmarking.
The first part briefly introduces the main concepts and current terminology related to *dependable computing*. The fault tolerance techniques (encompassing error detection, system recovery, and compensation) are briefly described. Also an analysis of the impact of fault tolerance on dependability is proposed.
The second part covers the methods and techniques — both analytical and experimental — for *dependability assessment*. The current evolution in research work on controlled experiments, from simple fault injection-based tests meant for evaluating a specific fault-tolerant computer architecture towards the development of benchmarks aimed at comparing the dependability features of several computer systems, is also briefly illustrated.

2 Dependable computing

2.1 Basic concepts

This section summarizes the main related terminology proposed by Jean-Claude Laprie and the IFIP WG 10.4 in [1], and updated in [2].
The dependability of a computer system characterizes its aptitude to deliver a service that can be *trusted* with *justified confidence*.
Depending on the application domain for which the system is intended, the accent can be put on various facets of the dependability; this means that dependability can be perceived from various, but complementary viewpoints that define its attributes:

- readiness for usage leads to the attribute of *availability*;
- continuity of service leads to *reliability*;
- non-occurrence of catastrophic consequences for the environment leads to *safety*;
- non-occurrence of unauthorized disclosure of information leads to *confidentiality*;
- non-occurrence of inadequate information alterations leads to *integrity*;
- ability to conduct repairs and introduce evolutions leads to *maintainability*.

The association with confidentiality of integrity and availability relative to the authorized actions leads to *security*.
The service may fail for two main reasons: i) it does not respect any more the functional specification, ii) the functional specification did not describe in an adequate way the intended function of the system. The latter meaning has led to an alternative more comprehensive definition that complements the original one. It is meant to provide a criterion to decide whether confidence can be granted to the service by referring to its aptitude to avoid service failures that are more frequent or more serious than acceptable. Failures of the service that are more frequent or more serious than acceptable reveal a dependability failure.
A *failure* of the service (or simply a failure, in short) is an event that occurs when the delivered service deviates from the correct service. A failure of the service is thus a transition from “correct service” to “incorrect service”, i.e., not fulfilling the function of the system. The delivery of an incorrect service is a “breakdown” of the service. The transition from incorrect service to correct service is the restoration of the service.
The deviation of the correct service can take several forms, which are the modes of failure; the consequences of the failures on the environment of the system are clas-

sified and ordered according to the severity of the failures. When the function of the system comprises a set of elementary functions, the failure of one or more services fulfilling these functions can leave the system in a degraded mode, which still offers a subset of services to the user. Several of these degraded modes can be identified, such as limited performance service, restricted service, emergency service, etc. In this case, a partial failure of the system is being considered.

The delivered service being a sequence of states, a failure of the service means that at least one state (external to the system) deviates from the correct service.

Such a deviation results from an *error*. The allocated or supposed cause of an error is a *fault*. The faults can be internal or external to the system. Usually, the former presence of a *vulnerability*, i.e., of an internal fault which makes it possible for an external fault to impair the behavior of the system, is necessary so that an external fault can lead to an error, and, possibly, to a failure.

Generally, a fault causes initially an error in the internal state of a component, the external state of the system not being immediately affected. It follows the definition of an error: part of the state of the system that is likely to provoke its failure, which occurs when the error impacts the service delivered to the user. It should be noted that number of errors do not affect the external state of the system, and thus do not cause a failure.

The dependability specification of a system describes what is necessary for the dependability attributes in terms of frequency and severity of the failures of the service for a given set of faults, or a given operational environment.

The development and operation of a dependable system requires the combined use of a set of methods that can be classified as:

- *fault prevention*: how to prevent the occurrence or the introduction of faults;
- *fault tolerance*: how to provide a service to fulfill the function of the system in spite faults;
- *fault removal*: how to reduce the presence (number, severity) of the faults;
- *fault forecasting*: how to estimate the presence, the future rate, and the possible consequences of the faults.

The principal concepts that were introduced so far can be summarized by the dependability tree shown in Figure 1.

As shown in the figure both fault prevention and fault tolerance are contributing to *dependability procurement* (how to deliver a trustworthy service with justified confidence?), while fault removal and fault forecasting are devoted to *dependability assessment*.

A recent generalization [3] has introduced the concept of *resilience* that extends the notion of dependability to account for system evolution and/or environment modifications by considering the *persistence* of service delivery that can justifiably be trusted, when facing such changes.

In the sequel of this section, we rather focus on the fault tolerance attribute. The issues related to dependability assessment (and especially, fault forecasting) are dealt with in Section 3.

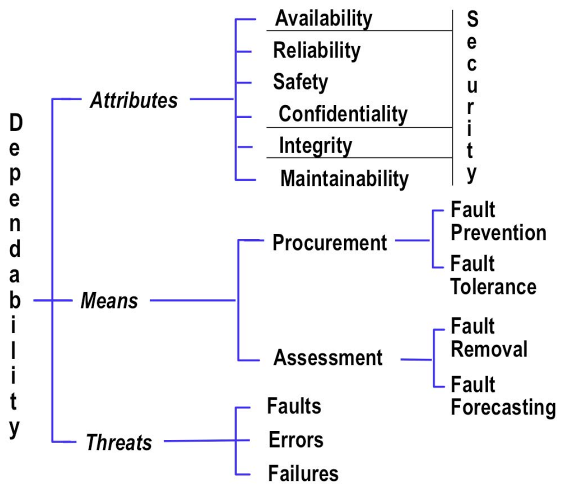


Figure 1: The dependability tree

2.2 Fault tolerance

Fault tolerance aims at avoiding the failure of the computer system and maintaining its operation by means of error detection and system recovery techniques that encompass: i) error processing (often enough when dealing with soft/transient faults) and ii) fault treatment (when coping with solid/permanent faults) in order to passivate the faulty component.

Usually, fault treatment is supplemented by corrective maintenance actions, in order to eliminate the passivated components.

In most cases, error detection is the primary action that initiates the fault tolerance strategy. Several techniques can be used: error detecting codes, dual execution and comparison, timing (watchdog) and execution checks, likelihood checks, etc. In that context, a very useful concept corresponds to the notion of *self-checking* component, that either delivers a correct service or explicitly raise an error signal upon failure.

Three main forms of error processing can be distinguished:

- *Backward recovery (rollback):* It consists in periodically saving the system state, so as to be able, upon detection of an error, to return the system to a previous (hopefully, error free) state.
- *Forward recovery*: As an alternative or complementary approach to rollback, it consists in searching for a new state acceptable for the system, from which it will be able to resume operation (possibly in a degraded mode).
- *Compensation*: It corresponds to the case when the erroneous state features enough redundancy to make it possible to mask the error; it can be applied upon detection of an error or systematically, independently of the presence or the absence of error. Typical examples include error correcting codes or Triple Modular Redundancy (TMR).

These techniques are comprehensive in the sense they apply to physical faults, design faults as well as to interaction faults — accidental or malicious (attacks). The classes of faults that can be tolerated by a given system depend on the assumptions considered for the fault tolerance mechanisms being implemented. This is conditioned by the independence of the redundancies with respect to the processes of creation and activation of the faults.

A classical fault tolerance approach is to carry out multiple treatments via several channels, sequentially or in parallel. When only physical faults are to be tolerated, the multiple channels can be identical, assuming that hardware components are likely to fail independently. This is however not adequate for coping with solid design faults or attacks; indeed, in that case, the multiple channels are required to implement the same function via distinct designs and implementations, thus resorting to the notion of functional diversification.

2.3 The actual impact of fault tolerance

We briefly investigate here how the fault tolerance techniques might influence the dependability, by analyzing their actual impact on the risks of fault, error and failure affecting a target system. In that respect, it is worth point out that the dependability of a system can be sketched as follows [4]:

$$\text{Dependability} \approx 1 - \Pr\{\text{fault}\} \times \Pr\{\text{error/fault}\} \times \Pr\{\text{failure/error}\}$$

where the Pr{.} denote relevant (absolute and conditional) probabilities.

The goal is to identify what would be the impact of developing a fault-tolerant solution with respect to a non fault-tolerant one in the light of these probabilities:

- **Pr{fault}**: If $\Pr_{NFT}\{\text{fault}\}$ denotes the probability of a fault in a non fault-tolerant system, the question is what can be inferred for a fault-tolerant version of that system? Actually, it is rather clear that due to the additional components (either hardware or software redundancies), it can be concluded that faults are more likely to occur, thus $\Pr_{FT}\{\text{fault}\} > \Pr_{NFT}\{\text{fault}\}$.
- **Pr{error/fault}**: In that case the issues at stake are less clear cut, however, many examples can be put forward that evidence that the implementation of a fault-tolerant system results in an increase of this probability. Just for illustration, let us consider the case of tests that are run to asses the “fault-proofness” of a system at start time or background procedures meant to reduce the risk of faults accumulating without being signaled (e.g., memory scrubbing). These tests and procedures clearly tend to increase the occurrences of an error when a fault is present.
- **Pr{failure/error}**: The picture at this stage is not very favorable as we have concluded that all previous relevant probabilities were increased in the case of a fault-tolerant system. Accordingly, it is not possible for a fault-tolerant system to actually exhibit an improved dependability, unless:

  $$\Pr_{FT}\{\text{failure/error}\} \ll \Pr_{NFT}\{\text{failure/error}\}$$

  Hopefully, this typically corresponds to the behavior observed for a fault-tolerant computer system in practice.

Two remarks:

1) This simple analysis exemplifies the limitation attached to architectures consisting in many replicated channels and missing efficient capabilities to detect, diagnose, isolate the failed channels and to reconfigure the system.
2) The conditional probability Pr{failure/error} is to be linked to the concept of “*coverage*” introduced in the late sixties (e.g., see [5]) to model the potential imperfect behavior of fault tolerance in reliability assessment studies as:

   $$c = \Pr\{\text{system recovers/system fails}\}$$

What corresponds to “system recovery” is dependent upon the target system; for example, in some situations recovery may only mean detection. Also, according to the current terminology a more adapted reference would rather be an “error” than a “failure”. Accordingly, it can be considered that $c \approx 1 - \Pr\{\text{failure/error}\}$.

Parameter c is thus essential as it has a prominent impact on the dependability level that can be achieved. Accordingly, some specific effort is needed to accurately estimate its actual (range of) value. These issues will be addressed in details in the next section.

3 Dependability assessment

3.1 The methods

Dependability assessment methods suitable for fault-tolerant computer architectures encompass three main approaches: verification (e.g., formal methods or testing), analytical evaluation (relying on models featuring stochastic processes) and controlled experiments (in particular, using fault injection techniques).

The diagram of Figure 2 details the link between these methods and the two attributes contributing to dependability assessment: fault prevention and fault forecasting (Figure 1).

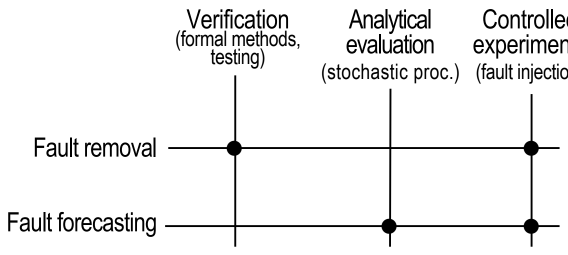


Figure 2: The main approaches for dependability assessment

While testing is still very much used, formal verification techniques (such as model checking) are increasingly applied in order to minimize the risk of residual design faults to be found in the target system. Still, such formal approaches are somewhat limited in scalability and thus have real difficulty in handling complex systems.

Model-based evaluation techniques are commonly used to evaluate and compare, from the dependability point of view, different architecture alternatives (encompassing various fault tolerance mechanisms) at the design stage.

Despite continuous progress in accounting for imperfect coverage of the fault tolerance mechanisms, the modeling approach has its own limits when one is trying to accurately describe the behavior of a computer system in presence of faults. This is in particular related to the level of abstraction that is attached to the models. Indeed, a model can be either inherently abstract (e.g., due to lack of detailed information) or kept intentionally abstract, so that the model is actually tractable.

This means that, in practice, other approaches are needed beyond analytical evaluation, in order to develop more precise/sophisticated models or account for actual system behaviors. These encompass simulation-based approaches and experiment-based approaches. In the latter case, in addition to field (failure) measurements and data analysis, a specific class of experiment-based approach is often being considered in the context of dependable computing, that is termed: *controlled experiments*. Here the concept of controlled experiments refers to fault injection experiments where faults are deliberately injected to objectively characterize the faulty behavior and/or assess dependability measures of a computer system.

In the sequel, we focus on the necessary connection between analytical evaluation and controlled experiments and illustrate the benefits that can be obtained towards the definition of *dependability benchmarks*.

3.2 Analytical evaluation

Analytical evaluation of dependability has been extensively investigated, leading to numerous proven methods and techniques that have been successfully developed and used.

3.2.1 Modeling techniques

Modeling relies on the analytical or graphical description of the system behavior. For a graphical description, dependability measures are assessed by allocating probabilities or stochastic processes to model parameters. Among the main types of models extensively used are reliability diagrams, fault trees and state graphs (Markov chains, stochastic Petri nets, etc.). These methods have long been recognized as a determining factor for rational decision making when considering different possible architectures or maintenance policies during the design of fault-tolerant systems (as detailed in studies on ESS, FTMP, STAR, SIFT systems), e.g., see [6] or in more recent studies, e.g., [7, 8].

Evaluation can be broken down into three main closely related phases associated with:

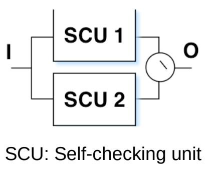
- the *choice of the measures* to be evaluated, which is generally part of system requirements;
- the *construction* of one (or several) model(s) that corresponds to the description of the behavior of the system being studied, using elementary stochastic processes and as a function of the measures considered;
- the *processing* of the model(s), that corresponds to the computation of the dependability measures.

Numerous software packages have been devised over the last thirty years to assist design engineers in their evaluation tasks (see [9-11]).

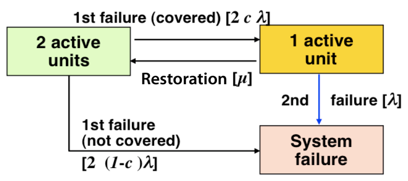
3.2.2 Modeling a fault-tolerant system

Let us consider a basic duplex architecture as described by the diagram of Figure 3-a. The architecture features two identical self-checking units (SCU) executing in parallel the same tasks on the same inputs (I). A switch is ensuring the delivery of the results of one (active) unit to the outputs (O).

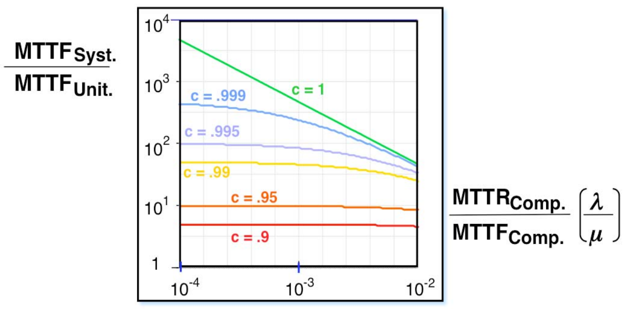
The relevant behavior is easily described by the simple state graph[1] shown on Figure 3-b. Upon failure of one unit (each unit is characterized by a failure rate λ). that unit is identified and passivated; accordingly, the service can be resumed by using the valid unit. Actually, this is assuming the perfect operation of the fault tolerance mechanisms (error detection, recovery, switch, etc.); this would correspond to a coverage (c) of 100%. Otherwise, a system failure is (conservatively) assumed (1 – c); for example, an error affecting the active unit is not detected and erroneous results are delivered to the output devices. Moreover, the system may also fail to deliver a proper service when both units are failed. Such a risk of exhaustion of the available redundant resources is compensated by the repair of the first failed unit (repair rate: μ). When repaired the unit is reinserted into the architecture.



a) Architectural diagram



b) Corresponding state graph



c) Analysis of the impact of the coverage factor

Figure 3: Modeling and evaluation of a duplex architecture

[1] The graph can be easily mapped to a Markov chain as depicted on the figure.

Figure 3-c illustrates the relative improvement concerning the Mean Time To Failure (MTTF) that can be achieved by such a duplex system with respect to the case of a single unit ($MTTF_{Unit} = 1/\lambda$). The horizontal axis is scaled according to the ratio of the Mean Time to Repair (MTTR) of one unit to its MTTF — actually, λ/μ. It is worth pointing out that in practice the respective orders of magnitude are very dissimilar: $1/\mu \approx$ hours, $1/\lambda \approx$ months. Accordingly, it follows that: $\lambda/\mu << 1$. The curves shown for different values of c, clearly indicate the strong impact of the coverage on the considered dependability measure. This confirms the simple analysis carried out in Section 2.3 and definitely motivates the need to accurately estimate such a parameter.

3.3 Controlled experiments

The types of controlled experiments that are being considered here are primarily meant to assess the behavior of computer systems in presence of faults. They rely on fault injection techniques that allow for a pragmatic testing to be conducted on an elaborated simulation-model of the target system or on the target system itself.

3.3.1 The fault injection attributes

A fault injection *test sequence* is made up of series of elementary experiments for which a faulty condition is applied to the target system. The test sequence is also characterized by an *input* domain and an *output* domain. The input domain corresponds to a set of injected *faults* ***F*** and a set ***A*** that specifies the data sustaining the *activity* of the target system and thus, the activation of the injected faults. The output domain corresponds to a set of *readouts* ***R*** that are measurements collected so as to characterize the target system behavior in the presence of faults. Finally, a set of *measures* ***M*** is obtained that is derived from the analysis and processing of the ***FAR*** sets. Indeed, each experiment in the test sequence specifies a particular point in the $\{F \times A \times R\}$ space. Altogether, the ***FARM*** sets constitute the major attributes that can be used to fully characterize a fault injection test sequence.

3.3.2 The fault injection techniques

Two types of criteria have to be considered for characterizing fault injection-based experiments aimed at validating fault-tolerant computer systems: the level of *abstraction* of the target system and the form of *application* of fault injection (see Table 1).

With respect to the level of *abstraction*, the target system can be:

- a *physical implementation* (possibly under the form of a prototype);
- a *simulation model* describing the structure and/or the behavior of the system under test.

The form of *application* of fault injection can be:

- *physical*, when faults are injected directly into the hardware components through physical or electrical alterations;
- *information-based*, by means of an alteration of digital variables or of memory contents.

Abstraction → ↓ Application	Simulation Model	Prototype or Real System
Logical and Information	Simulation-based	Software-implemented
Physical	Programmable Hardware	Hardware-implemented

Table 1: The fault injection techniques

Most of earlier studies associate these two criteria and can be simply characterized using the application form: thus, one can simply consider physical injection (hardware implemented) and fault simulation.

The search for solutions to mitigate the cost and limitations of simulation and to cope with the difficulty to devise physical injectors and carry out experiments, explains the increasing trend towards the developing of techniques for injecting at the information level on a physical target system. This is reflected by a departure from the goal of injecting actual faults by considering rather *error* injection [12]. Indeed, the consequences of faults are emulated by altering the execution of the software running on the target system. Such an approach is often referred to as software-implemented fault injection (SWIFI). Specific techniques include: i) x-oring at runtime data bits in registers or memory (*bit-flip*) or ii) modifying data or instructions in the program code before execution (*mutation*).

For obvious reasons, the conjunction between “Physical” application and “Simulation” has remained an empty option. Recently, this gap has been filled by the emergence of fault injection techniques that exploit the flexibility and reconfiguration capabilities offered by programmable hardware devices (such as FPGAs), both for implementing an executable model and for injecting faults, e.g., see [13].

3.4 From controlled experiments to dependability benchmarks

3.4.1 About dependability benchmarking

Performance benchmarks are widely used to evaluate system performance while dependability benchmarks are still at their very early stage. Indeed, several popular performance benchmarks exist for many specific domains that constitute invaluable tools to objectively assess and compare computer systems or components. The proposals for dependability benchmarks are more recent and also less widely recognized. However, one has to refer to the work carried out in the framework of the IFIP WG 10.4 SIG on Dependability Benchmarking[2] and within the IST project DBench[3], which have contributed to the issue of a com-

[2] www.dependability.org/wg10.4/SIGDeB

[3] www.laas.fr/DBench

prehensive book [14] gathering the main current advances and offers.

In practice, the basic foundations for dependability benchmarking heavily relies on the research activities developed to design, conduct and exploit controlled experiments featuring fault injection test sequences. Indeed, for what concerns the input domain, the fault dimension (the so-called *faultload* in that context) is essential to extend the classical *workload* that typically characterizes, together with the associated (performance) *measures*, a performance benchmark. Clearly, an analogy is to be made with respect to the ***F*** and ***A*** attributes mentioned earlier. In addition to performance measurements, e.g., performance degradation in presence of faults, the relevant measures attached to dependability benchmarking encompass a wide variety of facets including the assessment of system robustness, the characterization of failure modes and error signaling, the estimation of restart times, etc.

3.4.2 Features of dependability benchmarking

While fault injection is definitely central to dependability benchmarking, the related requirements and properties attached to the latter somewhat differ from those characterizing fault injection experiments meant for evaluating a specific fault-tolerant computer (FTC) system.

Table 2 depicts the main characteristics of dependability benchmarking

FTC system assessment	Dependability benchmarking
• One single target system	• Several target systems
• In-deep knowledge	• Limited knowledge only
• Testing of FTMs	• Global system behavior
• Fault and Activity sets	• Fault- and Work-load
• Sophisticated faults	• Reference (interface) faults
• •Measures: conditional dependability assessment	• Measures: dependability assessment (accounts for the fault occurrence process)
• One-of-a-kind process; heavy duty process, still fine	• Recurring process: lighter process, preferred
• Developer's view	• End-user/Integrator's view
• Results published, but experiment context often proprietary	• Results and procedure openly disclosed

Table 2: Characteristics of dependability benchmarking

While the assessment of a FTC system concentrates on a specific system, dependability benchmarking is actually meant to compare several architectures and systems.

Also, the levels of knowledge about the target system(s) significantly differ. Indeed, the target systems to be benchmarked can well be off-the-shelf computer systems or components (e.g., COTS operating systems).

Built-in FTMs are obvious targets to devise the fault injection experiments when one is assessing a FTC system. In the other case, detailed information about the implemented FTMs might not be available, so a more global analysis is usually carried out.

The fault sets (or faultload) differ from the point of view of their sophistication: indeed, while in one case the intimate knowledge about the target system allows for devising very elaborate faulty conditions, in the other case only easily *reproducible* and *portable* faultoads are to be considered. In practice, in order to guarantee a *fair* comparison among the systems to be compared, this means that faults affecting well-identified interfaces are to be preferred.

Concerning the Measures, while the former approach classically concentrates on the assessment of the efficiency of the fault tolerance mechanisms (FTMs) — thus mainly conditional dependability measures[4], dependability benchmarking aims at providing a more comprehensive set of measures that actually encompass the fault occurrence process.

Being usually a one-of-kind effort, the assessment of a FTC system can be a tedious (heavy duty) process; on the other hand, dependability benchmarking is rather a recurring process, accordingly it is expected that lighter procedures be applicable.

While the assessment of a specific FTC system typically refers to a developer's perspective, on the other side what is put forward is rather a end-user or an integrator perspective.

This has also an impact on the way the results are made available. For a benchmark to be useful, in addition to the measures being obtained, it is essential that the actual procedure be disclosed so that the experiments being performed can be actually reproduced.

Moreover, as for the case of the assessment of FTC system, to be meaningful, a benchmark should satisfy a set of properties such as representativeness and repeatability. However, some additional properties are also required: portability, cost-effectiveness, etc.

3.4.3 Towards dependability benchmarks

Dependability benchmarks are usually based on modeling and experimentation, with tight interactions between them. In particular, analysis and classification of failure data can support the identification of realistic faultloads for the experimentation.

On the other hand, measurements collected during experiments can be processed via analytical models to derive dependability measures (e.g., see [4]).

Some possible benchmarking scenarios can be sketched in reference to Figure 4, where layers identify the three steps *analysis*, *modeling* and *experimentation* and arrows A to E represent the corresponding activities and their interrelations [15].

For example a scenario consisting of modeling supported by experimentation would include the three steps and links A, B and E; in that case, the expected outputs are comprehensive measures obtained from processing the models.

[4] Typical examples include: the estimation of various coverage factors or error detection latencies, etc.

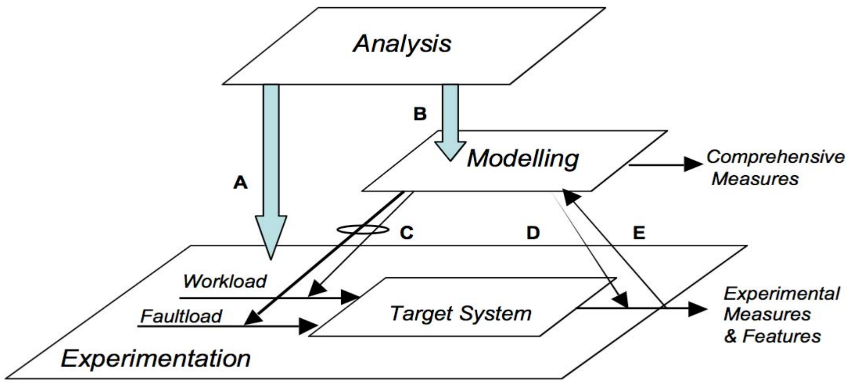


Figure 4: Dependability benchmarking steps and activities

3.4.4 The initial target: OS benchmarking

In recent years, several attempts have been made to propose and implement *benchmark prototypes* supporting the assessability of OS kernels.

Figure 5 depicts the software architecture of a computer system and provides a basis on which this thread of work can be illustrated. As shown on the figure, the kernel features three main interfaces with its environment. The first one (bottom) is basically concerned with hardware interactions, while the other two (top and right) are software related.

The "lightning" symbols in Figure 5 identify possible locations where faults can be applied. These locations and related faults are briefly described as follows:

1) The main interactions concerning the lower interface are made by raising hardware exceptions. Several studies (e.g., see [16, 17]) have been reported in which faults were injected by means of bit-flips into the memory of the system under benchmark or via special debugging interfaces [18].
2) The upper interface corresponds to the Application Programming Interface (API). The main interactions are made via kernel calls. A significant number of studies were reported that target the API to assess the robustness of OS (e.g., under the form of code mutations [19]), by means of bit-flips [20] or by altering the system calls [21, 22]**)**.
3) The third type of interactions are made via the interface between the drivers and the kernel. The proposal in [23] has concentrated on drivers code mutation. The work reported in [24] proposes a complementary alternative that explicitly targets the exchanges made between the device drivers and the kernel via their interface the "DPI (Driver Programming Interface)" [25].

4 Concluding Remarks

In complement to insights and dependability evaluations that can be obtained using analytical modeling and measurements, the conduct of *controlled experiments* based on fault injection are useful to better characterize the faulty behavior and dependability of a computer system.

Such controlled experiments are intended to yield three benefits:

- A better understanding of the effects of real faults and thus of the related behavior of the target system.
- An assessment of the efficacy of the fault tolerance mechanisms included into the target system and thus a feedback for their enhancement and correction (e.g., for removing designs faults in the fault tolerance mechanisms).

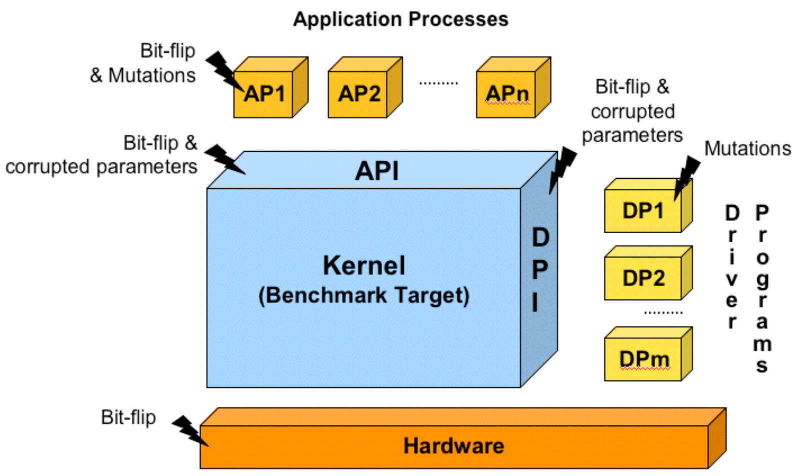


Figure 5: Interactions between an OS kernel and its environment and possible fault injection locations

- A forecasting of the faulty behavior of the target system, in particular encompassing a measurement of the efficiency (coverage) provided by the fault tolerance mechanisms.

In spite of such a solid basis, the development of practical benchmark instances is a complex task. Indeed, it is necessary to mitigate all the properties required for a benchmark (e.g., representativeness, repeatability, portability, non-intrusiveness, cost-effectiveness). Indeed, in practice, these properties are not independent. Some are correlated (e.g., portability and non-intrusiveness) while others are conflicting (representativeness and cost effectiveness). For example, a proper selection of the faultload relying on the software-implemented fault injection technique can contribute positively to fulfill the cost effectiveness property, but at the risk of degrading representativeness. Accordingly, much work on these directions is still needed.

Acknowledgement. The author would like to acknowledge the contributions of several PhD students and the fruitful interactions on the topics addressed in this paper with colleagues of the *Dependable Computing and Fault Tolerance* research group at LAAS-CNRS and of the IFIP WG 10.4. At LAAS, a special mention is directed to the memory of Jean-Claude Laprie, and thanks also go to Alain Costes, Yves Crouzet, David Powell, Jean-Charles Fabre, and Karama Kanoun.

References

[1] J.-C. Laprie (Ed.) Dependability: Basic Concepts and Terminology in English, French, German, Italian and Japanese, Vienna, Austria: Springer-Verlag, 1992.

[2] A. Avižienis, J.-C. Laprie, B. Randell and C. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing,” *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11-33, Jan.-March 2004.

[3] J.-C. Laprie, “From Dependability to Resilience,” in Proc.38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-2008) - Fast Abstracts, Anchorage, AK, USA, 2008.

[4] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie and D. Powell, “Fault Injection and Dependability Evaluation of Fault-Tolerant Systems,” *IEEE Trans. on Computers*, vol. 42, no. 8, pp. 913-923, August 1993.

[5] W. G. Bouricius, W. C. Carter and P. R. Schneider, “Reliability Modeling Techniques for Self-Repairing Computer Systems,” in *Proc. 24th. National Conference*, 1969, pp. 295-309, (ACM Press).

[6] A. Avižienis (Ed.), *Proceedings of the IEEE - Special Issue: Fault-Tolerant Digital Systems*, vol. 66 , no 10, pp. 1107-1268, 1978.

[7] D. Powell (Ed.) *Delta-4: A Generic Architecture for Dependable Distributed Computing*, Berlin, Germany: Springer Verlag, 1991.

[8] D. M. Nicol, W. H. Sanders and K. S. Trivedi, “Model-based Evaluation: From Dependability to Security,” *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 1, pp. 48-65, Jan.-March 2004.

[9] A. M. Johnson Jr and M. Malek, “Survey of Software Tools for Evaluating Reliability, Availability and Serviceability,” *ACM Computing Survey*, vol. 20, no. 4, pp. 227-269, December 1988.

[10] C. Hirel, R. Sahner, X. Zang and K. S. Trivedi, “Reliability and Performability Modeling Using SHARPE 2000,” in *Proc. 11th Int. Conf. on Computer Performance Evaluation: Modelling Techniques and Tools*, Schaumburg, IL, USA, 2000, pp. 345-349.

[11] D. D. Deavours and W. H. Sanders, “The Möbius Framework and its Implementation,” *IEEE Trans. on Software Engineering*, vol. 28, 956-969, 2002.

[12] J. Arlat and Y. Crouzet, “Faultload Representativeness for Dependability Benchmarking,” in Supplemental Volume of the Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN-2002) - Workshop on Dependability Benchmarking, Washington, DC, USA, 2002, pp. F.29-F.30. see also http://www.laas.fr/DBench.

[13] D. de Andrés, J. C. Ruiz, D. Gil and P. Gil, “Fault Emulation for Dependability Evaluation of VLSI Systems,” *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 4, pp. 422-431, April 2008.

[14] K. Kanoun and L. Spainhower (Eds.), *Dependability Benchmarking for Computer Systems*, IEEE CS Press and Wiley, 2008.

[15] K. Kanoun, H. Madeira and J. Arlat, “A Framework for Dependability Benchmarking,” in Supplemental Volume of the Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN-2002) - Workshop on Dependability Benchmarking, Washington, DC, USA, 2002, pp. F.7-F.8, see also http://www.laas.fr/DBench.

[16] 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, Feb. 2002.

[17] W. Gu, Z. Kalbarczyk, R. K. Iyer and Z. Yang, “Characterization of Linux Kernel Behavior under Errors,” in *Proc. Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN-2003)*, San Francisco, CA, USA, 2003, pp. 459-468.

[18] J.-C. Ruiz, P. Yuste, P. Gil and L. Lemus, “On Benchmarking the Dependability of Automotive Engine Control Applications,” in *Proc. Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN-2004)*, Florence, Italy, 2004, pp. 857-866.

[19] J. Durães and H. Madeira, “Emulation of Software Faults: A Field Data Study and a Practical Approach,” *IEEE Trans. on Software Engineering*, vol. 32, no. 11, pp. 849-867, November 2006.

[20] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun and T. Marteau, “Impact of Internal and External Software Faults on the Linux Kernel,” *IEICE Trans. on Information and Systems*, vol. E86-D, no. 12, pp. 2571-2578, December 2003.

[21] 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.

[22] K. Kanoun and Y. Crouzet, “Dependability Benchmarks for Operating Systems,” *International Journal of Performability Engineering*, vol. 2, no. 3, pp. 275-287, July 2006.

[23] J. Durães and H. Madeira, “Mutidimensional Characterization of the Impact of Faulty Drivers on the Operating Systems Behavior,” *IEICE Trans. on Information and Systems*, vol. E86-D, no. 12, pp. 2563-2570, December 2003.

[24] A. Albinet, J. Arlat and J.-C. Fabre, “Benchmarking the Impact of Faulty Drivers: Application to the Linux Kernel,” in *Dependability Benchmarking for Computer Systems* (K. Kanoun and L. Spainhower, Eds.), pp. 285-310, IEEE CS Press and Wiley, 2008.

[25] D. Edwards and L. Matassa, “An Approach to Injecting Faults into Hardened Software,” in *Proc. Ottawa Linux Symposium*, Ottawa, ON, Canada, 2002, pp. 146-175.