System Dependability: Characterization and Benchmarking

Yves Crouzet, Karama Kanoun

System Dependability — Characterization and Benchmarking

Yves Crouzet[1,2] and Karama Kanoun[1,2]

1: LAAS-CNRS, 7 Avenue du Colonel Roche, F-31077 Toulouse Cedex 4, France

2: Université de Toulouse; UPS, INSA, INP, ISAE; UT1, UTM, LAAS; F-31077 Toulouse Cedex 4, France

{Yves.Couzet, Karama.Kanoun}@laas.fr

Abstract

This chapter presents briefly basic concepts, measures and approaches for dependability characterization and benchmarking, and examples of benchmarks, based on dependability modeling and measurements. We put emphasis on Commercial Off-the-Shelf components (COTS) on COTS-based systems. To illustrate the various concepts, techniques and results of dependability benchmarking, we present two examples of benchmarks: one addressing the system and service level of a COTS-based fault tolerant system, and one dedicated to COTS software components. The first benchmark shows how dependability modeling can be used to benchmark alternative architectural solutions of instrumentation and control systems of nuclear power plants. The benchmarked measure corresponds to system availability. The second benchmark shows how controlled experiments can be used to benchmark operating systems taking Windows and Linux as examples. The benchmark measures are robustness, reaction time and restart time.

Key words

Dependability benchmarking, dependability assessment, assessment based on analytical modeling, assessment based on measurement, COTS

1. Introduction

System performance is no longer the only factor that keeps customers satisfied. Dependability is increasingly playing a determinant role, as well, not only of life-critical systems but also for money-critical systems. Dependability assessment should be addressed from the early design and development phases of such systems. As a matter of fact, performance benchmarks greatly participated in improving system performance, as it is not conceivable nowadays to provide a system without indications on its performance attributes allowing users to appreciate the capacity of their computer systems properly. Moreover, some system vendors design and implement computer systems in such away that they satisfy specific performance benchmark by construction.

Considering dependability, even tough some system vendors provide information related to its dependability, this practice is seldom, and even when such information does exist it is provided in various forms according to the venders, making it very hard to compare alternative systems from the dependability point of view.

Benchmarking the dependability of a system consists in evaluating dependability or performance-related measures in the presence of faults, in a well-structured and standardized way. A dependability benchmark is intended to objectively characterize the system behavior in the presence of faults. Actually, despite the broad range of promising work on dependability benchmarking, we are far from having reached the consensus needed to get to the current situation of performance benchmarks. Dependability benchmarking is an emerging domain, and a great variety of benchmarks has been defined, e.g., robustness benchmarks, fault tolerance benchmarks, recovery benchmarks. Most of these dependability benchmarks have been defined in a non-coordinated way. No standardization results have been published to help in this process. Nevertheless, the work achieved so far is necessary to pave the way for standard dependability benchmarks, even tough all actors agree on the fact that implementing dependability benchmarks is essential for competition.

One of the difficulties steams from the fact the spectrum of measures is wider for dependability benchmarks than for performance benchmarks. It is thus very important to agree both on the dependability benchmark measures of interest, and on the way they can be obtained in a non-ambiguous manner. A way of

reducing the spectrum is to address specific categories of systems, which help understanding and solving progressively the problems. Based on the fact that the current trend is to build systems composed of Commercial Off-the-Shelf components (COTS), a natural start is to address COTS and COTS-based systems. In this chapter we will concentrate on COTS and COTS-based fault tolerant systems for critical systems.

The paradox with using COTS components is that, on the one hand, their large-scale usage increases the confidence that one may have in their general dependability but, on the other hand, this same "large-scale usage" argument may not constitute a sufficient dependability case for using COTS in critical applications. Thus, one is faced with providing further characterization and benchmarking of components over whose design one has had no control.

The chapter is composed of 7 sections. In Section 2 we will present dependability measures that can be used for benchmarking fault tolerant system and components. As system benchmarking makes use of dependability assessment approaches based on modeling and on measurement, the basic assessment approaches will be presented in Section 3. Section 4 will address the benchmarking concepts and give examples of existing dependability and performance benchmarks. Sections 5 and 6 present two case studies related respectively to an example of COTS-based systems for Instrumentation and Control system of nuclear power plants, and to COTS, taking as an example the case of operating systems. Finally, Section 7 concludes the chapter.

2. Dependability Measures

The dependability characteristics of a system or a component can be expressed either qualitatively, in terms of attributes and features describing the system capacities and properties, or in terms of quantitative measures. As the occurrence or activation of faults in a system may lead to performance degradation without leading to system failure, dependability and performance are strongly related. Thus, the evaluation of system performance under faulty conditions, in addition to dependability measures, will allow characterizing completely the system behavior from the dependability point of view.

We distinguish two kinds of dependability measures: comprehensive and specific measures. Comprehensive measures characterize the system globally at the service delivery level while specific

measures characterize particular aspects of a system or a component (e. g., the fault tolerance mechanisms or the system behavior in presence of faults). In the following we first provide examples of comprehensive measures, then examine specific measures and finally we address performance related measures.

2.1 Comprehensive Dependability Measures

Dependability is an integrative concept that encompasses the following basic attributes [76, 11] (see also Chapter 1??):

- Availability: readiness for correct service.
- Reliability: continuity of correct service.
- Safety: absence of catastrophic consequences on the user(s) and the environment.
- Confidentiality: absence of unauthorized disclosure of information.
- Integrity: absence of improper system state alterations.
- Maintainability: ability to undergo repairs and modifications.

Several other dependability attributes have been defined that are either combinations or specializations of the six basic attributes listed above. Security is the concurrent existence of a) availability for authorized users only, b) confidentiality, and c) integrity with 'improper' taken as meaning 'unauthorized'. Dependability with respect to erroneous inputs is referred to as robustness.

The attributes of dependability may be emphasized to a greater or lesser extent depending on the application: availability is always required, although to a varying degree, whereas reliability, safety and confidentiality may or may not be required. The extent to which a system possesses the attributes of dependability should be interpreted in a relative, probabilistic sense, and not in an absolute, deterministic sense: due to the unavoidable presence or occurrence of faults, systems are never totally available, reliable, safe, or secure.

The evaluation of these attributes leads to view them as measures of dependability. The associated measures are referred to as **comprehensive dependability measures** as i) they characterize the service delivered by the system, ii) they take into account all events impacting its behavior and their consequences

and iii) they address the system in a global manner, even though the notion of system and component is recursive and a system may be a component of another system.

Measures associated with the above attributes have been defined in [76, 11] as follows:

- **Reliability** measures the continuous delivery of correct service or, equivalently, the time to failure.
- **Availability** measures the delivery of correct service with respect to the alternation of correct and incorrect service.
- **Maintainability** measures the time to service restoration since the last failure occurrence, or equivalently, measures the continuous delivery of incorrect service.
- **Safety**: when the state of correct service and the states of incorrect service due to non-catastrophic failure are grouped into a safe state (in the sense of being free from catastrophic damage, not from danger). Safety measures the continuous safeness, or equivalently, the time to catastrophic failure. As a measure, safety is thus reliability with respect to catastrophic failures.

The joint evaluation of performance and dependability leads to the notion of performability.

2.2 Specific Dependability Features and Measures

For some systems, the dependability can be expressed in terms of properties or features the system must satisfy in the presence of faults. For example:

- One feature could be for example "the system should be *fail-controlled"*, meaning that the system should fail only in specific and controlled modes of failure, such as i) fail-halt or fail-silent modes, when, to an acceptable extent, all failures lead to halt the system or to make it silent; or ii) fail-safe mode, when failures are all minor ones, to an acceptable extent.
- Another feature could be "the system should be *Fail-safe/Fail-silent"*, meaning the system should be fail-safe after the first failure and fail-silent after the second failure

One has to assess to which extend these features, expressed in qualitative terms, are statistically satisfied.

Additionally, it may be interesting to assess specific aspects of system behavior without necessarily taking into account all the processes impacting its global behavior and without addressing the service delivery level. This concerns essentially features related to i) system error detection and fault tolerance

capabilities, ii) maintenance facilities, iii) system evolution capacities. These features are of prime interest when building COTS-based systems where such information should be made available for system integrators, otherwise the latter cannot rely on the COTS components to build the system.

Without being exhaustive, we illustrate below the kind of features that are worth to be investigated in the context of dependability benchmarking.

Examples of features related to fault tolerance capabilities:

- Detection and recovery of permanent hardware and/or software faults
- Detection and recovery of transient hardware and/or software faults
- Detection and recovery of successive faults
- Error containment (avoidance of error propagation)
- On-line fault diagnosis
- Protection against operational errors (accidental / intentional)
- Failure modes
- Recovery after power failure

Examples of features related to maintenance and evolution:

- On-line repair
- On-line backup
- Detection of inconsistent upgrade

It is worth to mention that features such as extendibility, scalability and modularity may be considered as essential for a system, even though they are not directly related to dependability, but they may impact system dependability.

The list of features above is provided in a generic manner, and each feature has to be specified precisely to characterize the dependability of a system. In particular, one has to specify the exact nature and location of errors that can be detected, contained (whose effects can be confined) or tolerated. For example, a system may be tolerant to hardware faults without being tolerant to software faults.

In order to have an accurate knowledge about the system behavior, features should be completed by quantitative information. In particular, one has to know to which extent these features are fulfilled. This leads to associate to each feature one or more **specific dependability measures** to be quantified to describe its behavior accurately.

Usually features are assessed through their efficiency. The latter has two complementary dimensions: i) a time dimension corresponding to the duration of the considered action (error detection, recovery or containment, fault diagnosis and system repair) and ii) a conditional probability of success of an action, provided it has been initiated (referred to as coverage factor or coverage). For example fault diagnosis coverage is defined as the probability that a fault is correctly diagnosed given the fact that an error is detected. However, for some systems, action duration or action coverage may have more impact and emphasis may thus be put only on the most influential dimension of the efficiency.

2.3 Performance Related Measures

Classical performance measures include measures such as system response time and system throughput. In the context of dependability benchmarking, performance evaluation addresses the characterization of system behavior in the presence of faults or with respect to the additional fault tolerance mechanisms. For example, some fault tolerance mechanisms may have a very high coverage factor with a large time overhead in normal operation. It is interesting to evaluate such time overhead. Concerning the system behavior in the presence of faults, following fault occurrence or fault activation, either the system fails or a correct response is provided (correct value, delivered on time). Indeed, a correct value delivered too late with respect to the system specification is to be considered as a failure, mainly for hard real-time systems.

In the presence of errors, a system may still provide a correct response with a degraded performance. Hence, the response time and the throughput (which are at the origin pure performance measures) become dependability and performance related measures characterizing the **system performance in the presence of faults**.

2.4 Comments on Features and Measures for Dependability Assessment

We have presented various dependability attributes and features and their associated measures as well as performance related measures that allow characterization of the dependability of a system or a component. We have distinguished two kinds of dependability measures: comprehensive and specific measures. Comprehensive measures characterize the system globally at the service delivery level, taking into account all events impacting its behavior and their consequences on the application or service delivery. Specific measures characterize particular aspects of a system or a component related for example to system behavior in the presence of faults and fault tolerance capabilities. Each measure characterizes one side of the multi-faceted problem. The variety and number of comprehensive and specific measures show the complexity of dependability characterization.

Approaches for assessing the various measures will be presented in the next section.

3. Basic Dependability Assessment Approaches

During the system design and integration phases, dependability analysis and assessment, along with other performance and functional analyses, supports the selection of the most suitable system architecture with respect to the dependability requirements of the application. Section 2 presented measures of dependability to be assessed to characterize fault tolerant systems and their components. This section presents assessment approaches that can be used to assess these measures.

Assessment is usually carried out by means of three complementary approaches, based i) on analytical modeling approaches, or ii) on field data, resulting from the observation of real-life systems (i. e., field measurement), or iii) on controlled, off-line, experiments (i. e., experimental measurement). Each kind of approach has its advantages and limitations. Obviously, when ever possible, results based on field data are most significant than those based on experimentation and modeling. Unfortunately, such analyses are not always available because they require the collection of very specific data items, most of the time related to (hopefully) very rare events, hence requiring a long period of data collection to lead to statistically significant results.

Analytical modeling approaches are commonly used to support the selection of candidate systems for which the most significant dependability attributes, among those presented in the previous section, are

compared. Modeling requires the knowledge of i) the system functions, ii) the system architecture used to fulfill these functions (in terms of components and interactions between the components), as well as in terms of fault tolerance and recovery mechanisms embedded in the system to increase its dependability, and iii) the maintenance strategy adopted during system operation in case of failures of multiple components. This knowledge is then used to model the system behavior as resulting from failure occurrence, error detection, error propagation or confinement, system recovery or reconfiguration, failure modes, maintenance facilities, etc. At the model level, these events and activities are expressed by means of event rates and conditional probabilities of success or failure, referred to as model parameters. Numerical values of the model parameters are needed to process the model. They are usually evaluated derived from measurement.

Field measurement is based on data collected on the system and its environment during its development, validation and operational life. Data collected concern failure and maintenance processes: time of failure occurrence, nature of failures and impact on system services, faulty component, recovery time, repair duration, restart time, etc. This information allows evaluation of measures such as the mean time between failures or the system failure rate, the failure rates according to some specific (critical) failure modes, the system components' failure rates, and system availability.

Experimental measurements, typically performed off-line, on the system or on a prototype might complement very well field measurement, mainly for parameters describing the system behavior in the presence of faults. As fault occurrences constitute rare events, injecting faults that mimic or simulate real faults in the system or in its environment allows the assessment of the fault tolerance mechanisms, and, whenever possible, the improvement of these mechanisms.

In the context of COTS-based systems, the combined use of the three approaches is recommended, and generally provides very useful results for system providers, system integrators, and system users. Since COTS are generally widely used in several architectures, data collection either by system providers or users is achievable and can provide accurate and statistically meaningful results even over relatively short periods of time (compensated by a large number of systems). This is a very useful means for assessing COTS component's and COTS-based systems failures rates and failure modes. During the integration of the COTS-based system, controlled experiments are carried out to complement this information, related essentially to

fault tolerance mechanisms. Based on these two sources of information, dependability models are built and processed to derive dependability measures at the system and service level.

The rest of this section is dedicated to analytical modeling for assessing comprehensive dependability measures, and to controlled experimentation by means of fault injection techniques for assessing specific measures.

3.1. Assessment based on Analytical Modeling

With respect to dependability modeling, the main difficulty results from the various dependencies between the components. These dependencies may result from functional or structural interactions between the components or from interactions due to system-level fault tolerance mechanisms, reconfiguration and maintenance strategies. State-space models, in particular Markov chains, are commonly used for dependability modeling of computing systems. They are able to capture various functional and stochastic dependencies among components and allow evaluation of various measures related to dependability and performance (i.e., performability measures) based on the same model, when a reward structure is associated to them (see Trivedi Chapter).

To master model construction and processing, the system is decomposed into subsystems and their sub-models are built individually. Depending on the complexity of the system and the kind of dependencies to be modeled, two classes of approaches have been developed to build and solve the system model: compositional approaches or decomposition/aggregation approaches [60]. In the compositional approaches the system model results from a modular composition of the sub-models that is then solved as a whole. Most of the published work related to modular model construction concentrates on rules to be used to construct and interconnect the sub-models. In the decomposition/aggregation approaches, the sub-models are solved individually, and the obtained individual measures are aggregated to compute the dependability measures of the overall system. The first class of approaches is very convenient for systems having a high degree of dependency between its components while the second class of approaches provides an easy way to describe the behavior of systems that can be decomposed into sets of sub-systems with a low degree of dependency, or without dependencies to make the computation tractable.

In the following, we present briefly examples of modeling approaches pertaining to the two classes. We put more emphasis on the former as, in Section 5, the case study will be based on a compositional approach, and more specifically on the approaches defined in [64] and [18].

3.1.1. Compositional Modeling Approaches

The most popular technique used to build dependability and performability state space models in a modular way are based on Petri nets, and more precisely Generalized Stochastic Petri Nets (GSPNs). Composition of Petri Net (PN) consists in constructing PN models from a set of building blocks by applying suitable operators of places and/or transition superposition. Initially investigated for no timed models, composition approaches have been then explored for stochastic extensions of Petri nets. For example, [24] explored composition in the context of Stochastic Petri Nets (SPNs) and [39] proposed a systematic compositional approach to the construction of parallel hardware-software models. The GSPN composition rules are based on the concept of matching labels associated with transitions and places of a GSPN, and the superposition of transitions (places) with matching labels, each one belonging to a different GSPN. Also, composition operators have been defined to facilitate model composition. For example, in the context of Stochastic Activity Networks (SAN) that constitute an extension to GSPNs [81], two composition operators are defined (*Join* and *Replicate*) to compose system models based on SANs.

When the modeled systems exhibit various dependencies that need to be explicitly described in the dependability models. Various modeling approaches have been proposed to facilitate the construction of large dependability models taking into account such dependencies. For example, the block modeling approach defined in [64] provides a generic framework for the dependability modeling of hardware and software fault-tolerant systems based on GSPNs. The proposed approach is modular: generic GSPN submodels called block nets are defined to describe the behavior of the system components and of the interactions between them. The system model is obtained by composition of these GSPNs. Composition rules are defined and formalized through the identification of the interfaces between the component and interaction block nets. In addition to modularity, the formalism brings flexibility and re-usability thereby allowing for easy sensitivity analyses with respect to the assumptions that could be made about the behavior

of the components and the resulting interactions. The main advantage of this approach lies in its efficiency for modeling several alternatives for the same system as illustrated for example in [67] and Section 5.

The efficiency of the block modeling approach can be further improved by using an incremental and iterative approach for the construction and validation of the models as suggested in [47]. At the initial iteration, the behavior of the system is described taking into account the failures and recovery actions of only one selected component, assuming that the others are in an operational nominal state. Dependencies between components are taken into account progressively at the following iterations. At each iteration, a new component is added and the GSPN model is updated by taking into account the impact of the additional assumptions on the behavior of the components that have been already included in the model. Similarly to the block modeling approach, sub-models are defined for describing the components behaviors and specific rules and guidelines are defined for interconnecting the submodels. This approach has been successfully used to model the dependability of the French air traffic control computing system [48].

An iterative dependability modeling approach has been also proposed in [18] where the construction and validation of the GSPN dependability model is carried out progressively following the system development refinement process, to facilitate the integration of dependability modeling activities in the system engineering process. Three main steps are distinguished. The first step is dedicated to the construction of a functional-level model describing the system functions, their states and their interdependencies. In the second step, the functional level model is transformed into a high-level dependability model based on the knowledge of the system's structure. A model is generated for each pre-selected candidate architecture. The third step is dedicated to the refinement of the high-level dependability model into a detailed dependability model for each selected architecture. Formal rules are defined to make the successive model transformations and refinements as systematic as possible taking into account three complementary aspects: i) component decomposition, ii) state/event fine-tuning, and iii) stochastic distribution adjustments. This approach allows the integration of various dependencies at the right level of abstraction: functional dependency, structural dependency and those induced by non-exponential distributions. A case study is described in [19].

Actually, the approach presented in [18] can be seen as a special case of the more general class of techniques based on layered and multi-level modeling methods, where the modeled system is structured into

different levels corresponding to different abstraction layers, with a model associated to each level. Various techniques based on this idea have been developed, see e.g., [39, 59, 16, 17, 85, 21, 77, 84, 33].

3.1.2. Decomposition and Aggregation Approaches

The decomposition and aggregation techniques depend on the type of measures to be evaluated and on the modeling formalism. Generally approximate solutions are provided for the composition of the results derived from the submodels. A decomposition and aggregation theory for steady state analysis of general continuous time Markov Chains has been proposed in [31]. The quality of the approximation is related to the degree of coupling between the blocks into which the Markov chain matrix is decomposed. In [20] the authors present an extension of this technique specifically addressed to the transient analysis of large stiff Markov chains, where stiffness is caused by the simultaneous presence of "fast" and "slow" rates in the transition rate matrix.

Time-scale based decomposition approaches have been applied to Non-Markovian stochastic systems in [52], and to GSPN models of systems containing activities whose durations differ by several orders of magnitude in [3]. For example, in [3] the GSPN model is decomposed into a hierarchical sequence of aggregated sub-nets each of which is characterized by a certain time scale. Then these smaller sub-nets are solved in isolation, and their solutions are combined to get the solution of the whole system. In [29], the overall model consists of a set of submodels whose interactions are described by an import graph.

The decomposition approach in [34] is based on a new set of connection formalisms that reduce the state-space size and solution time by identifying submodels that are not affected by the rest of the model, and solving them separately. The result from each solved submodel is then used in the solution of the rest of the model.

3.2. Assessment Based on Controlled Experiments

In the context of dependable and fault tolerant systems, controlled experiments are mainly used to analyze qualitatively or to assess the behavior of the system in the presence of faults. The aim is either to identify possible weaknesses in the system to be dealt with at the design level to improve system dependability, or to assess specific dependability measures to provide numerical values for performability models to allow

accurate estimation of comprehensive measures.

In this context, controlled experiments are mainly based on fault injection techniques. The observation of the system behavior after fault occurrence allows assessment of conditional probabilities such as probability of error detection, probability of error containment or probability of error recovery.

Several surveys have been devoted to fault injection techniques that are more or less detailed (see e. g., [54, 14, 96] and Chapter 7 of [82]). In the rest of this Section we will briefly outline some techniques and tools for i) injecting hardware faults, for ii) injecting or emulating software faults, and for iii) message-based fault injection.

3.2.1. Techniques for injecting hardware faults

In this section, we describe four techniques for injecting or emulating hardware faults. The two first techniques are related to fault injection into real hardware systems, based respectively on hardware-implemented fault injection and on software-implemented fault injection. The two other techniques are related respectively to simulation-based fault injection and to hardware emulation-based fault injection

Hardware-implemented fault injection includes five techniques: pin-level fault injection, electromagnetic interferences, power supply disturbances, radiation-based fault injection and test port-based fault injection.

- In pin-level fault injection, faults are injected via probes connected to electrical contacts of integrated circuits or discrete hardware components. Many experiments and studies using pin-level fault injection were carried out during the 1980's and early 1990's, and several tools were developed, for example, MESSALINE [7] and RIFLE [78]. A key feature of these tools was that they supported fully automated fault injection campaigns.
- Electromagnetic interferences are common disturbances in automotive vehicles, trains, airplanes, or industrial plants. Such techniques are widely used to stress digital equipment thanks to the use of a commercial burst generator in conformance with the IEC 801-4 standard (CEI/IEC).
- Power supply disturbances are rarely used for fault injection because of low repeatability. They have been used mainly as a complement to other fault injection techniques in the assessment of error detection mechanisms for small microprocessors [72].

- Radiation-based fault injection is a technique for analyzing the susceptibility of integrated circuits to soft errors, i.e., bit-flips caused when highly ionizing particles hits sensitive regions within in a circuit. In space, soft errors are caused by cosmic rays, i.e., highly energetic heavy-ion particles. The sensitivity of integrated circuits to heavy-ion radiation can be exploited for assessing the efficiency of fault tolerance mechanisms [51] and [72]. In [9] the three techniques described above (pin-level fault injection, electromagnetic interferences, heavy-ion radiation) have been used for the validation of the MARS architecture.
- Test port-based fault injection techniques use built-in debugging and testing features, which can be accessed through special Input/Output-ports, included in modern equipments (microprocessors, boards), known as test access ports. Test ports are defined by standards such as Nexus (IEEE-ISTO 5001 standard [57]) for real-time debugging, or JTAG (IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture [56]). Freescale provides a proprietary solution for debugging known as Background Debug Mode facility. The type of faults that can be injected via a test port depends on the debugging and testing features supported by the target microprocessor. Faults can be injected in all registers of the microprocessor. Using Background Debug Mode facilities and Nexus, faults can be injected in the main memory. Tools that support test port-based fault injection include GOOFI [1] supporting both JTAG-based and Nexus-based fault injection and INERTE [93] supporting Nexus-based fault injection. An environment for fault injection based on Background Debug Mode is presented in [86].

Software-implemented fault injection techniques allow injection of faults through the software executed on the target system. There are basically two approaches to emulate hardware faults by software: run-time injection and pre run-time injection. In run-time injection, faults are injected while the target system executes a workload. This requires a mechanism that stops the execution of the workload, invokes a fault injection routine, and restarts the workload. In pre run-time injection, faults are introduced by manipulating either the source code or the binary image of the workload before it is loaded into memory. Numerous tools are capable of emulating hardware faults through software (see e. g., FIAT [13], FERRARI [63], FINE [71], DEFINE which was an extension of FINE [70], FTAPE [91], DOCTOR [53], Xception [25] and MAFALDA [8]).

Simulation-based fault injection can be performed at different levels of abstraction, the device level, logical level, functional block level, instruction set architecture level, and system level.

FOCUS [27] is an example of a simulation environment that combines device-level and gate-level simulation for fault sensitivity analysis of circuits with respect to soft errors.

At the logic level and the functional block level, circuits are usually described in a hardware description language (HDL or VHDL). Several tools have been developed that support automated fault injection experiments with HDL models, e.g., MEFISTO [58] and the tool described in [36]. There are different methods for implementing fault injection, by modifying the VHDL code [10] or modifying the HDL simulator, or commanding the simulator through scripts.

DEPEND [49] is a tool for simulation-based fault injection at the functional level: a simulation model in DEPEND consists of number of interconnected modules, or components, such as CPUs, communication channels, disks, software systems, and memory.

Hardware emulation-based fault injection is based on the use of large Field Programmable Gate Arrays (FPGAs) circuits. This technique provides means for conducting model-based fault injection for analyzing the impact of faults in hardware circuits [75]. It has all the advantages of simulation-based fault injection such as high controllability and high repeatability, but requires less time for conducting a fault injection campaign compared to software simulation.

Fault injection can be performed in hardware emulation models through compile time reconfiguration and run-time reconfiguration. An approach for compile-time instrumentation for injection of single event upsets (soft errors) is described in [30]. One disadvantage of compile-time reconfiguration is that the circuit must be re-synthesized for each reconfiguration, which can impose a severe overhead on the time it takes to conduct a fault injection campaign. In order to avoid re-synthesizing the target circuit, a technique for run-time reconfiguration is proposed in [6]. This technique relies on directly modifying the bit-stream that is used to program the FPGA-circuit. A tool for conducting hardware emulation-based fault injection called FADES is presented in [4, 5].

3.2.2. Techniques for injecting or emulating software faults

There are two fundamental approaches to injecting software faults into a computer system: fault injection and error injection [42].

- In the **error injection** approach, there are two common techniques for emulating software faults by error injection: program state manipulation and parameter (involving changing variables, pointers and other data stored in main memory) or CPU-registers corruption (modification of function parameters, procedures and system calls). Several tools, developed for hardware fault injection through software-implemented fault injection, can potentially be used to emulate software faults since they are designed to manipulate the system state (e.g., FIAT, FERRARI, FTAPE, DOCTOR, Xception and MAFALDA).
- **Fault injection** imitates mistakes of programmers by changing the code executed by the target system, while error injection attempts to emulate the consequences of software faults by manipulating the state of the target system. In the fault injection approach, software faults are injected into a system by mutations (i. e., by manipulating the source code, the object code or the machine code. FINE [71] and DEFINE [70] were among the first tools that supported emulation of software faults by mutations. The mutation technique used by these tools requires access to assembly language listings of the target program. SESAME is another tool using mutation as the target fault model, allowing fault injection into software written in assembly languages, procedural languages (Pascal, C), data-flow languages (LUSTRE), as well as declarative languages [32]. A technique, referred to as Generic Software Fault Injection Technique, for emulation of software faults by mutations at the machine-code level is presented in [41]. This technique can be used as a basis for faultload definition for dependability benchmarking [42].

3.2.3. Message-Based Fault Injection

Message-based fault injection technique is dedicated to test protocols in fault-tolerant distributed systems. The aim of this type of testing is to reveal design and implementation faults in the tested protocol. The tests are performed by manipulating the content and/or the delivery of messages sent between nodes in the target system. In this context, several fault injection tools and frameworks have been developed.

The experimental environment for fault tolerance algorithms [43] is an early example of a fault injector for message-based fault injection. The tool inserts fault injectors in each node of the target system and implements different fault types, including message omissions, sending a message several times, generating spontaneous messages, changing the timing of messages, and corrupting the contents of messages. A similar environment is provided by the DOCTOR tool [53], which can cause messages to be lost, altered, duplicated or delayed.

Specifying test cases is a key problem in testing of distributed fault-handling protocols. A technique for defining test cases of protocols from Petri-net models is described in [44]. An approach for defining test cases from an execution tree description is presented in [12].

A tool for testing distributed applications and communication protocols called ORCHESTRA is described in [35, 36]. This tool inserts a probe/fault injection layer between any two consecutive layers in a protocol stack. Results from message-based fault injection assessment of several implementations of CORBA middleware are reported in [79].

3.3. Summary and Relationships Between the Assessment Approaches

Assessment of comprehensive dependability measures can be carried out, even before system building, based on analytical modeling. Modeling requires the knowledge of the following:

- System functions and structure.
- System nominal and degraded operational modes, system failure modes, as well as component failure modes.
- The most significant error detection and fault-tolerance capabilities.
- Maintenance facilities (e. g., on-line repair and backup possibilities, maintenance policy).

For modeling purposes, the above sets of information are expressed in terms of event occurrence rates and conditional probabilities. Events refer for example to fault occurrence and activation, error propagation and system repair or restart. Conditional probabilities are usually related to system behavior after event occurrence. For example, if the event is fault occurrence and activation, the conditional probabilities describe the reaction of the system to faults (i.e., probability of error detection, probability of error containment or probability of error recovery).

The evaluation of dependability entails additionally the knowledge of the numerical values of the model parameters (event rates and conditional probabilities). From a practical point of view:

- Fault occurrence and activation rates are generally obtained from field measurement related to the same system if it is already in operation or, most probably, from similar previous systems that have already been in operation for a long time or in a large number.
- Conditional probabilities can be obtained from experimentation: from field data, whenever possible or from controlled experiments.

When the parameter values are not available, approximate values can be attributed in a first step. Sensitivity analyses allow identification of the most salient ones for the considered measure(s), to be evaluated from controlled experiments such those presented in Section 3.2.

4. Benchmarking Concepts and Examples

Benchmarking is widely used to measure computer performance in a deterministic and reproducible manner, allowing users to appreciate the capacity of their computer systems properly, while benchmarking of system dependability is hardly emerging. Benchmarking the dependability of a system consists in evaluating dependability or performance-related measures in the presence of faults, in a well-structured and standardized way. A dependability benchmark is intended to objectively characterize the system behavior in the presence of faults.

To be meaningful and objective, a dependability benchmark should satisfy a set of properties. In particular, a benchmark must be at least representative and cost-effective, to be accepted by the community. In addition, when the benchmark is based on experimentation, the experiments should be repeatable (in statistical terms), portable, and non-intrusive. These properties have to be addressed from the earliest phases of the benchmark specification as they have a significant impact on the benchmark nature and objectives, and they should be checked after the benchmark design and implementation. These properties contribute to shift the benchmarks from *ad hoc* to standard approaches to dependability assessment.

Performance and dependability benchmarks share a common feature: the definition of an appropriate *workload*, representing a synthetic or a realistic operational profile. The selection of such a workload constitutes a major difficulty as an agreement between system vendors and potential users should be reached

to expect recognition and adoption of any performance benchmark. It is the main (and the only) complexity of performance benchmarks. Actually, several workloads for performance benchmarks have been used as workloads for dependability benchmarks, to which faultloads have been superposed (see e. g., Chapters 5, 6 and 12 of [66]).

Dependability benchmarks characterize the system behavior in the presence of faults, or under abnormal conditions or perturbations. Thus, in addition to the definition of appropriate workloads, they require the definition of suitable "perturbations", referred to as *faultload*. This is not an easy task since the faultload is intended to simulate faults that could be internal or external to the system being benchmarked.

A further difficulty steams from the fact that several benchmark measures may be of interest (cf. Section 2) for characterizing the system either in a comprehensive way (with respect to the service delivered) or according to specific features (e.g., fault tolerance mechanisms). This adds an extra complexity. As a result, a great variety of benchmarks has been defined, e.g., robustness benchmarks, recovery benchmarks, availability benchmarks.

The results of performance and dependability benchmarks can be useful for both the end-users and vendors. Considering the case of dependability benchmarks, the results can help to:

- Characterize the dependability of a component or a system, qualitatively or quantitatively.
- Track dependability evolution for successive versions of a product.
- Identify weak parts of a system, requiring more attention and perhaps needing some improvements by tuning a component to enhance its dependability, or by tuning the system architecture (e.g., adding fault tolerance) to ensure a suitable dependability level.
- Compare the dependability of alternative or competitive solutions according to one or several dependability attributes.

Performance benchmarks are essentially based on experimentation. Performing a performance benchmark consists in executing the workload associated to the benchmark, under particular conditions that are specified within the benchmark.

Dependability benchmarks can be based on modeling or experimentation or both, using the assessment approaches presented in Section 3, depending on the measure to be assessed by the benchmark. More

specifically, comprehensive measures are usually obtained from the processing of dependability models. The latter require the knowledge of numerical values of some parameters, some of which are to be provided by controlled experiments, the others being made available by system providers, or are already known (published data), or can be obtained from field data. Typical measures (or parameters) that can be obtained from controlled experiments have been presented in Section 2.2. Such experimental benchmark measures, for fault tolerant systems may correspond to i) efficiency of the fault tolerance mechanisms, ii) switching time between the primary and the secondary computer of a fault-tolerant system, iii) latency of error detection mechanisms, iv) system restart time or the v) system failure modes.

At the component level, experimental benchmark measures address for example the failure modes of the component, its robustness, or its restart time. In the rest of this section we present few examples of performance benchmarks and dependability benchmarks.

4.1 Examples of performance benchmarks

A performance benchmark is a test that measures the performance of a computer system or a component on a well-defined task or set of tasks. The task or set of tasks is normally defined by a workload and the measures are specific of each benchmark. A set of rules specifies the way the test must be conducted to reach valid benchmark results.

There are three major organizations in the performance benchmarking: EEMBC (Embedded Microprocessor Benchmark Consortium), SPEC (Standard Performance Evaluation Corporation) and TPC (Transaction Processing Performance Council). They are non-profit organizations and their members include most of the major companies in the computer industry. In the following we give an overview of a few benchmark developed by these organizations.

4.1.1 EEMBC Benchmarks

EEMBC [45], the Embedded Microprocessor Benchmark Consortium, was formed in 1997 to develop meaningful performance benchmarks for the hardware and software used in embedded systems. Through the combined efforts of its members, EEMBC benchmarks have become an industry standard for evaluating the

capabilities of embedded processors, compilers, and Java implementations according to objective, clearly defined, application-based criteria.

Since releasing its first certified benchmark scores in April 2000, EEMBC scores have effectively replaced the obsolete Dhrystone mips, especially in situations where real engineering value is important. EEMBC benchmarks reflect real-world applications and the demands that embedded systems encounter in these environments. The result is a collection of "algorithms" and "applications" organized into benchmark suites targeting telecommunications, networking, digital media, Java, automotive/industrial, consumer, and office equipment products. An additional suite of algorithms specifically targets the capabilities of 8- and 16-bit microcontrollers.

EEMBC's certification rules represent another break with the past. For a processor's scores to be published, the EEMBC Technology Center must execute benchmarks run by the manufacturer. EEMBC certification ensures that scores are repeatable and obtained according to EEMBC's rules.

4.1.2 SPEC Benchmarks

SPEC [89] was founded in 1988 and has more than 60 members today, including most of the major computer industry companies. SPEC benchmarks consist of a standardized source code taken form established applications (i.e., real-life applications) and modified by SPEC to improve portability and accommodate some specific requirements of performance benchmarking [46]. In most of the cases, the workload of a SPEC benchmark includes several applications that are run in sequence one after the other. Typically, to run a SPEC benchmark, all what is required is to compile the workload for a specific system and then tune the system for the best results. Two benchmarks are of particular interest, SPEC CPU and SPECjvm, addressing the CPU and JVM (that is used in Section 6).

- SPEC CPU2006 is a suite of compute-intensive benchmarks that measures performance of the computer processor, memory and compiler. It includes two benchmarks suites: CINT2006 for measuring and comparing compute-intensive integer performance, and CFP2006 for measuring and comparing compute-intensive floating point performance. CINT2006 consists of twelve applications and CFP2006 includes seventeen applications. All the applications are real applications and they are written in several languages such as C, C++ and Fortran. Each of these benchmark suites includes four metrics: two speed

metrics, corresponding to the execution time of the applications compiled respectively with conservative and aggressive compiler optimization choices, and two throughput metrics, corresponding to the rate of execution of the applications in a given time using again conservative and aggressive compiler optimization choices.

- SPECjvm2008 is a benchmark suite for measuring the performance of a Java Runtime Environment (JRE), containing several real life applications and benchmarks focusing on core java functionality. The suite focuses on the performance of the JRE executing a single application; it reflects the performance of the hardware processor and memory subsystem, but has low dependence on file I/O and includes no network I/O across machines. The SPECjvm2008 workload mimics a variety of common general purpose application computations. These characteristics reflect the intent that this benchmark will be applicable to measuring basic Java performance on a wide variety of both client and server systems.

4.1.3 TPC Benchmarks

The Transaction Processing Performance Council (TPC) is a non-profit corporation formed by the major vendors of systems and software from the transaction processing and database market. The goal of TPC is to define and disseminate performance benchmarks for transaction processing systems. Detailed and latest information on TPC organization and TPC benchmarks can be obtained from TPC's web site [90].

All TPC benchmarks include two kinds of measures: performance and price/performance measures. The performance measure is a transaction rate (e.g., number of transactions per minute) and the price/performance measures are based on a detailed set of pricing rules that take into account the price of purchasing the system (hardware and software) and the maintenance costs for a given period.

Unlike SPEC benchmarks, that heavily rely on the source code of the workload, TPC transactions are defined by a very detailed specification. So, to run a TPC benchmark it is necessary to implement the specification in the target system, which means that it is necessary to program the transactions and all the aspects defined in a functional way in the benchmark specification.

The TPC has currently four benchmarks: TPC-C for OLTP (On-Line Transaction Processing) systems, TPC-W for transactional Web systems such as e-commerce systems, and two benchmarks for decision support systems, the TPC-H for *ad hoc* decision support queries (queries may not be known in advance) and

the TPC-R for business reporting and decision support queries (when pre-knowledge of the queries is assumed and may be used for optimization).

4.2 Examples of Dependability Benchmarks

Currently, most of the published work on dependability benchmarking resulted from individual effort of institutions, without the involvement of a standardization body. Pioneer work on dependability benchmarking is published in [91] for fault tolerant systems (that has then be followed by [92, 94, 95], and in [83] for operating systems (OSs). Regarding OSs, the dependability benchmarks developed, are robustness benchmark either with respect to faulty applications [73, 87, 61, 62, 69, 65], or with respect to faulty device drivers [40, 2].

As far as we are aware, the only two groups who have worked collectively towards dependability benchmarking are DBench, a European project on Dependability Benchmarking, partially supported by the European Commission [37], and the Special Interest Group on Dependability Benchmarking [88], founded by the IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance.

DBench has developed a framework for defining dependability benchmarks for computer systems, with emphasis on i) *Off-the-Shelf components,* commercial or not, and on ii) systems based on Off-the-Shelf components, via experimentation and modeling. The use of dependability benchmarks in the development/integration of such composite systems seems very useful for component selection and to assess dependability measures of the whole system. To exemplify how the benchmarking issues can actually be handled in different application domains, five examples of benchmarks and their associated prototypes (i.e., actual implementations of the benchmarks) have been developed within DBench. They concern general-purpose operating systems, embedded systems (automotive and space applications) and transactional systems [68]. These benchmarks addressed specifically system integration and share the following common characteristics:

- The benchmark is performed during the *integration phase* of a system including the benchmark target (or when the system is available for *operational phase*).
- The benchmark performer is someone (or an entity) who has no in depth knowledge about the benchmark target and who is aiming at i) obtaining valuable information about the target system

dependability, and ii) publicizing information on the benchmark target dependability in a standardized way.

- The primary users of the benchmark results are the *integrators* of the system including the benchmark target (or the end-users of the benchmark target).

Dependability benchmarking is without doubt a valuable support for system integration, mainly for COTS-based systems.

The work of the Special Interest Group on Dependability Benchmarking led to the publication of the first (and unique) book on dependability benchmarking, published in 2008 [66]. The OS benchmark presented in Section 6 is a subset of a benchmark published in that book in Chapter 12. This book consists of sixteen chapters, prepared by researches and engineers working on the dependability field for several years and more specifically on dependability benchmarking during the more recent years. These chapters illustrate the current multiplicity of approaches to dependability benchmarking, and the diversity of benchmark measures that can be evaluated. Chapters 1 through 6 examine system-level benchmarks, which focus on various aspects of dependability using different measurement methods. Chapters 7 to 16 focus on benchmarks for system components: control algorithms (Chapter 7), intrusion detectors (Chapter 8), fault-tolerance algorithms (Chapter 9), operating systems (Chapters 10 to 15), and microprocessors (Chapter 16). More details on these chapters are given in the Appendix.

5. Case Study: Benchmarking COTS-based Systems

In this Section we illustrate the kind of results that can be obtained based on dependability modeling for benchmarking COTS-based systems. More specifically, our aim is to show how analytical modeling can be used to support benchmarking COTS-based, fault tolerant, alternative architectures. In these architectures fault tolerance can be ensured by three classical replication (or redundancy) techniques, namely passive replication, semi-active replication and active replication.

In a passive replication, only one of the n parallel replicas ($n \geq 2$) processes the input messages and provides output messages (active replica). The other (passive) replicas do not process the input messages. In case of unavailability of the active replica, one of the passive replicas becomes active.

In a semi-active redundancy, only one of the two replicas (i. e., the primary replica) processes all input

messages and provides output messages. The other replica (secondary) is active since it also processes the input messages even though it does not provide any output messages. In case of fault occurrence or activation in the primary, a switch from the primary to the secondary replica is performed.

In an active redundancy, the three replicas process all input messages concurrently so that their internal states are closely synchronized, in the absence of faults. The outputs can be taken from any replica as long as at least two replicas are in the nominal state. In case of fault occurrence or activation, the error is masked.

The case study concerns *instrumentation and control* (I&C) systems available as COTS systems from several providers for power plants (more details can be found in [19]. The starting point of our work was to help, based on dependability evaluation, a stakeholder of an I&C system in selecting and refining systems proposed by various contractors in response to a Call for Tender.

In the rest of this section, we first present the main functions of an I&C system and examples of three different candidate COTS-based hypothetical systems constituting a set of different possible realizations for the considered I&C application. Examples of Petri nets models are given in Section 5.2 and an assessment result is given in Section 5.3.

5.1. Presentation of the considered I&C Systems

The main functions of an I&C system are given in Figure 1, in which the arrows represent the interactions between these functions. The three system architectures considered are depicted in Figure 2. These systems have been selected for their diversity of architectures and of redundancy techniques. For example in System 1, every computer executes a single function, while in System 2 and System 3, some computers execute more than one function.

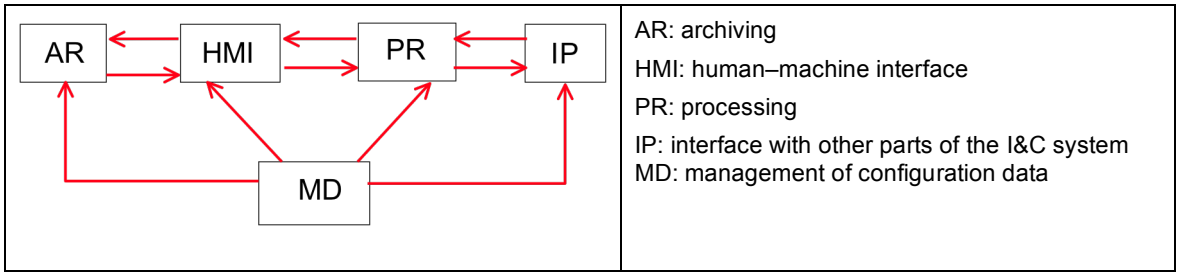


Figure 1: Main Functions of the I&C System

System 1 is composed of 13 nodes (each one executing a single function), connected via a Local Area

Network (LAN). Nodes 1 to 10 are composed of a computer each. Nodes 11, 12 and 13 are fault-tolerant: they are composed of two redundant computers each. Also, nodes 12 and 13 are complementary (i.e., they interface complementary parts of the I&C system).

System 2 is composed of five nodes connected by a LAN. Note that while HMI is executed on four nodes, node 5 runs three functions. Nodes 1 to 4 are composed of one computer each. Node 5 is fault-tolerant: It is composed of two redundant computers.

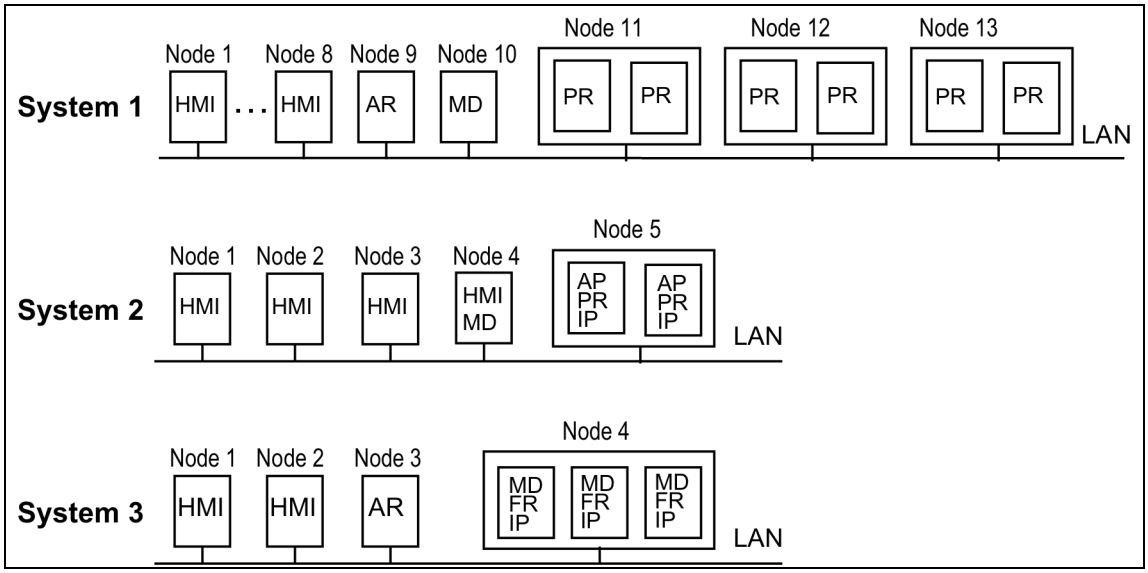


Figure 2: Three Examples of Architectures based on COTS Computers

In System 3, nodes 1 to 3 are composed of a single computer each running a single function. Node 4 is fault-tolerant: it is composed of three redundant computers (i. e., forming a triple-modular redundancy).

Table 1 summarizes the redundancy type of each node. Non-redundant nodes are referred to as single nodes. We assume that each hardware computer hosts a single software component, forming a COTS unit referred to a unit.

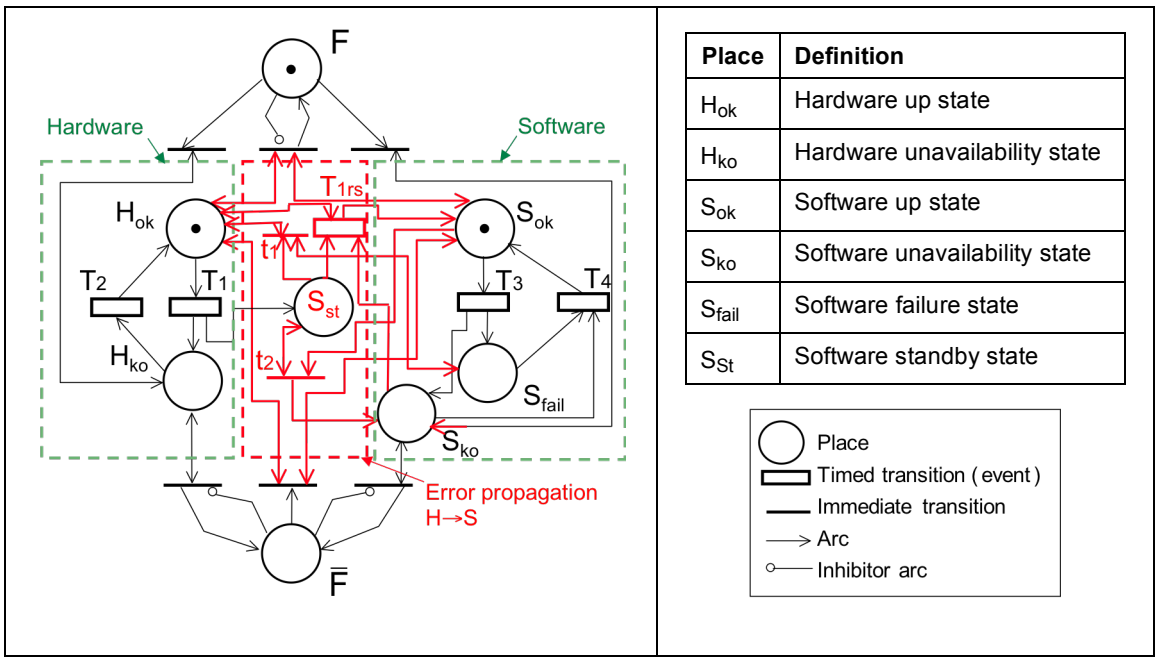
Table 1: Redundancy type of the three architectures

System	Node	Type of redundancy	Recovery
System 1	#1-8 #9-10 #11-12	Passive Single Semi-active	— — switch
System 2	#1-3 #4 #5	Passive Single Semi-active	— — switch
System 3	#1-2 #3 #4	Passive Passive Active	— — masking

5.2. Modeling of the I&C Systems

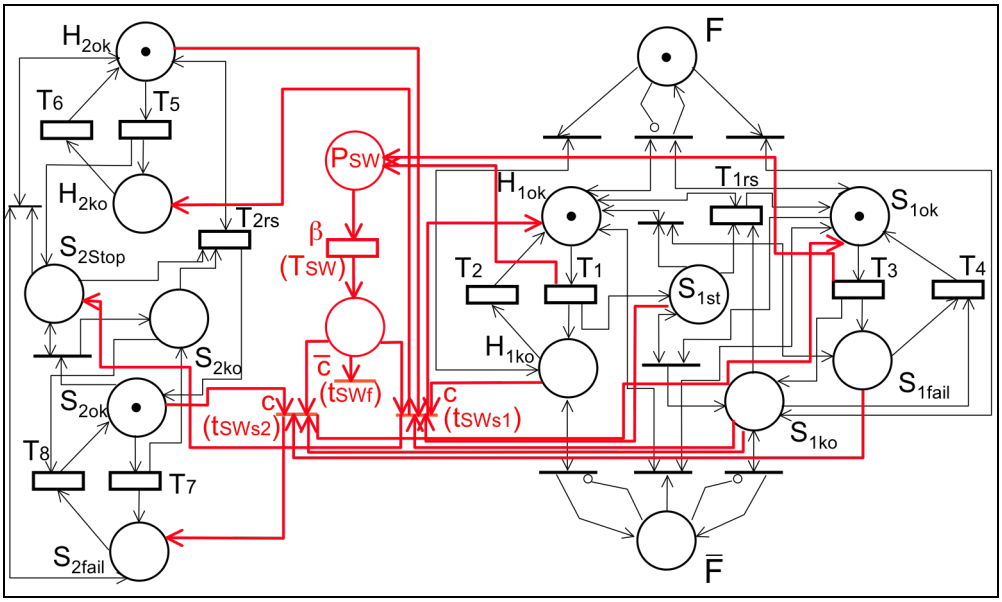
To simplify the model's construction for the three systems, we have built a library of basic models for the efficient modeling and refinement of the given systems, with minimal modifications (see [18]).

Figure 3-a gives as example the Generalized Stochastic Petri Net (GSPN) of a single unit that is then composed with the GSPN of another unit and a switch net to model the semi-active redundancy architecture of Figure 3-b, taking into account the switching of the secondary unit to become the active unit in case of failure of the primary unit.



Place	Definition
H_{ok}	Hardware up state
H_{ko}	Hardware unavailability state
S_{ok}	Software up state
S_{ko}	Software unavailability state
S_{fail}	Software failure state
S_{St}	Software standby state

a – One COTS unit



b – Semi-active redundancy

Transition	Rate	Definition
T1 / T5	λ_h	Primary / secondary hardware fault occurrence
T2 / T6	ν_h	Primary / secondary hardware repair
T3 / T7	λ_s	Primary / secondary software fault activation
T4 / T8	ν_s	Primary / secondary software re-initialization after software failure
T1 / T5	ρ_s	Primary / secondary software restart after hardware repair
Tsw	β	Switch from the primary to the secondary

Figure 3: GSPNs for Single and Semi-active Redundancy Architectures

Figure 3-a assumes that function F is carried out by a hardware component H and a software component S. F and $\bar{F}$ markings depend upon the markings of H and S (F up state is the combined results of H and S up states, while F failure (marking of $\bar{F}$) results from H or S unavailability). For each component, we consider two states: nominal (ok) and unavailable (ko). Transitions between these two states are ruled by events of failure (transitions T1 and T3) and restoration (transitions T2 and T4). It is worth noting that errors can propagate from hardware to software: when the hardware component fails, the software is stopped (immediate transitions t1 and place S_{st}). The software will be restarted only once the restoration of the hardware component is completed. Also, the models consider the case of a hardware failure after the failure of the software. In this case:

- if the software is restored before the hardware's restoration is completed, it will be put on hold until the hardware is up again. Then, and just then, the software will be restarted,
- if the hardware component is restored before the software component, then the token from S_{st} will be removed through the immediate transition t2.

Note that we consider two unavailable states for the software: the fail state S_{fail} (due to the software own failure) and the software stop state after a hardware failure, corresponding to S_{st}. $S_{ko} = S_{fail} \cup S_{st}$.

In the semi-active redundancy GSPN of Figure 3-b, unit 1 corresponds to the primary while unit 2 corresponds to the secondary unit. If the primary unit fails due to the failure of one of its components, the internal fault tolerance mechanisms switch over to the secondary unit that becomes primary. The switching event is represented by transition TSW whose rate is β (the switching time is $1/\beta$). The coverage factor (i.e., the conditional probability that the switch succeeds given the failure of the primary) is c (immediate transitions tSWs1 and tSWs2). Thus, the switch fails and the function is lost with probability $\bar{c}=1-c$ (immediate transition tSWf).

5.3. Comparison of System 1 and System 2

To show the kind of results that can be provided to the stakeholder when using analytical models for benchmarking and comparing alternative solutions, we perform a comparative analysis of System 1 and System 2 unavailability, using the SURF-2 tool [15] to process the models.

Figure 4 presents the annual unavailability of System 1 and System 2, as a function of the switching time (1/β) for c=0.95 and c=0.98, expressed in hours per year. For large values of the switching time, System 1 is more sensitive to this variation than System 2. This can be explained by the fact that System 1 has three components dependent on these two parameters, whilst System 2 has only one.

Also, we notice that for a small switching time (1/β = 30s or 1min), System 1 annual unavailability is smaller than System 2's. However, for larger values of the switching time (1/β=5 or 10min), this trend is reversed.

If we consider the same system for the two values of c, it can be seen that System 1 is more sensitive to c than System 2. Improving c from 0.95 to 0.98, the unavailability is reduced by at least 10 h per year for System 1, while it is only reduced by 2–3 h for System 2. Hence the values of c and 1/β impact more System 1 than on System 2.

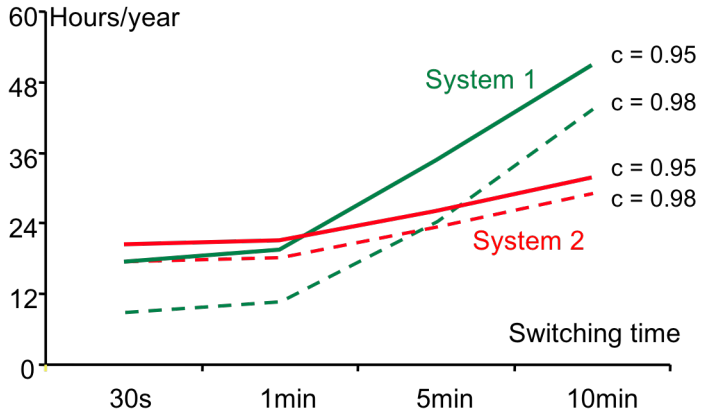


Figure 4: Comparison of System 1 and System 2 Annual Unavailability

5.4. Integration of the Results in the Dependability Benchmarking Process

Assuming that the considered values of the failure and repair rates are not far from reality (they are either provided by the COTS providers or obtained from similar systems), the kind of conclusions the stakeholder can make are, for example:

- If the switching time, 1/β, is around 10min, then for all values of the c parameter, the safest case is obtained for System 2.
- If the switching time is around 1min, then for all values of the c parameter, the safest case is obtained for System 1.

The above analysis shows that it is really important to have the most realistic and accurate numerical values for parameters c and β, to allow a fair and relevant comparison of the systems.

Referring to Section 2, unavailability is a comprehensive measure; c and β are specific measures characterizing the fault-tolerance mechanisms, respectively the coverage factor and the duration of switching from secondary to primary in case of failure of the primary. The best way to assess accurately the two parameters is to perform controlled experiments, based on fault injection, where faults simulate hardware and software faults of the components, aiming at activating the recovery mechanisms to collect information related to their coverage and duration. This calls for a specific benchmark. Unfortunately, we have not performed this benchmark, and we hope that the stakeholder for whom we performed the study did such analysis to refine the offers.

6. Case Study: Benchmarking Operating Systems

COTS Operating systems (OSs) are more and more used even in critical application domains. Choosing the operating system that is best adapted to one's needs is becoming a necessity. In this section, we consider OSs belonging to Windows and Linux families. In our previous work we have considered three workloads: the TPC-C Client performance benchmark for transactional systems in [61], Posmark that is a file system performance benchmark in [69] and the Java Virtual Machine (JVM) in [65]. In this section, we summarize part of the results obtained with the JVM workload. Before presenting the results, we first describe the specification of the OS benchmark and their implementation for Windows and Linux families.

6.1. Specification of the Benchmark

A dependability benchmark is specified through the definition of i) the benchmark target, ii) measures to be evaluated, iii) benchmark execution profile to be used to activate the operating system, iv) guidelines for conducting benchmark experiments and implementing the benchmark. The benchmark results are useful and interpretable only if all the above benchmark elements are supplied together with the results. These elements are summarized hereafter.

6.1.1. Benchmarking Target

An OS is a generic software layer providing basic services to the applications through the API, and communication with peripherals devices via device drivers. The benchmark target corresponds to the OS with the minimum set of device drivers necessary to run the OS under the benchmark execution profile. However, the benchmark target runs on a hardware platform whose characteristics impact the results. Thus, all benchmarks must be performed on the same hardware platform.

Although, in practice, the benchmark measures characterize the target system and the hardware platform, we state simply that the benchmark results characterize the OS as a COTS.

Our benchmark addresses the user perspective, i.e., it is intended to be performed by (and to be useful for) someone who has no thorough knowledge about the OS and whose aim is to improve her/his knowledge about its behavior in the presence of faults. In practice, the user may well be the developer or the integrator of a system including the OS.

As a consequence, the OS is considered as a "black box". The only required information is its description in terms of system calls and in terms of services provided.

6.1.2. Benchmark Measures

The OS receives a corrupted system call. After execution of such a call, the OS is in one of the states defined in Table 2.

Table 2: OS States

SEr (Error code)	The OS generates an error code that is delivered to the application.
SXp (Exception)	In the user mode: the OS processes the exception and notifies the application. For some critical situations, the OS aborts the application. In the kernel mode: an exception is automatically followed by a panic state (e.g., blue screen for Windows and oops messages for Linux). The latter exceptions are included in the panic state and the term exception refers only to the first case of user mode exception.
SPc (Panic)	The OS is still "alive" but it is not servicing the application. In some cases, a soft reboot is sufficient to restart the system.
SHg (Hang)	A hard reboot of the OS is required.
SNS (No Signaling)	The OS does not detect the erroneous parameter and executes the erroneous system call. SNS is presumed when none of the previous situations is observed.

It is worth to mention that *Panic* and *Hang* situations (SPc, SHg) are actual states in which the OS can stay for a while. SEr and SXp characterize events. They are easily identified when the OS provides an error code or notifies an exception.

The benchmark measures include a robustness measure and two temporal measures. They are defined in Table 3.

Table 3: Benchmark Measures

OS Robustness, OSR	The percentages of experiments leading to any of the states listed above. OSR is a vector composed of 5 elements
OS Reaction Time, Treac	The average time necessary for the OS to respond to a system call in presence of faults, either by notifying an exception or by returning an error code or by executing the required instructions.
OS Restart Time, Tres	The average time necessary for the OS to restart after the execution of the workload in the presence of faults.

Although under nominal operation the OS restart time is almost deterministic, it may be impacted by the corrupted system call. The OS might need additional time to make the necessary checks and recovery actions, depending on the impact of the fault applied.

The OS *reaction time* and *restart time* are also evaluated by experimentation in absence of faults for comparison purposes. They are respectively denoted $\tau reac$ and τres.

6.1.3 Benchmark Execution Profile

In the current benchmark, the workload is JVM, solicited through a small program activating 76 system calls for Windows family and 31 to 37 system calls for Linux Family.

The faultload consists of corrupted parameters of system calls. For Windows, system calls are provided to the OS through the Win32 environment subsystem. In Linux OSs, these system calls are provided to the OS via the POSIX API. During runtime, the system calls activated by the workload are intercepted, corrupted and re-inserted.

The parameter corruption technique relies on thorough analyses of system call parameters to define selective substitutions to be applied to these parameters (similarly to the one used in [74]). A parameter is either a data or an address. The value of a data can be substituted either by an out-of-range value or by an incorrect (but not out-of-range) value, while an address is substituted by an incorrect (but existing) address

(that could contain an incorrect or out-of-range data). We use a mix of these three techniques. More details can be found in [69].

6.1.4. Benchmark Conduct

A benchmark controller is required to control the benchmark experiments, mainly in case of OS Hang or Panic states or workload hang or abort states, as such states cannot be reported by the machine hosting the benchmark target itself. Hence, we need at least two computers as shown in Figure 5. The Target Machine hosts the benchmarked OS and the workload, and ii) the Benchmark Controller is in charge of diagnosing and collecting part of benchmark data.

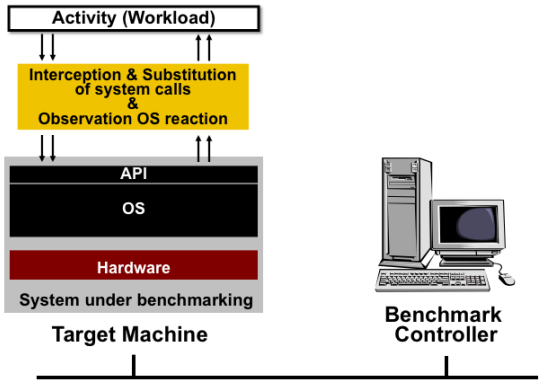


Figure 5: Benchmark Environment

The two machines perform the following: i) restart of the system before each experiment and launch of the workload, ii) interception of system calls with parameters, ii) corruption of system call parameters, iii) re-insertion of corrupted system calls, vi) observation and collection of OS states. The experiment steps in case of workload completion are illustrated in Figure 6 and will be detailed in the next section. In case of workload non-completion state (i.e., the workload is in abort or hang state), the end of the experiment is governed by a watchdog timeout, fixed to 3 times the workload execution time without faults.

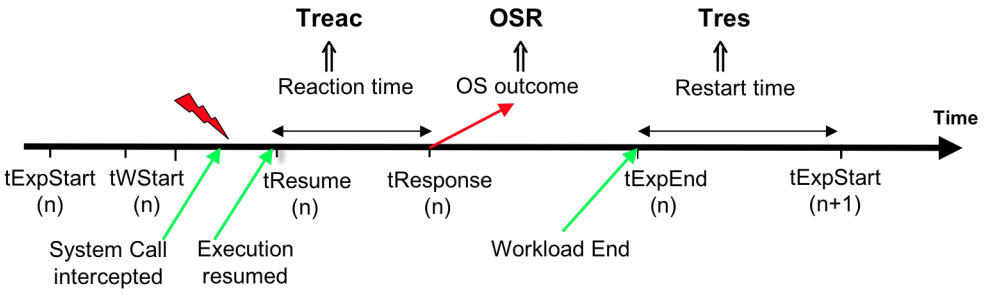


Figure 6: Benchmark Execution Sequence in Case of Workload Completion

6.2. Benchmark Implementation

In order to obtain comparable results, all the experiments are run on the same target machine, composed of an Intel Pentium III Processor, 800 MHz, and a memory of 512 Mega Bytes. The hard disk is 18 Giga Bytes, ULTRA 160 SCSI. The benchmark controller in both prototypes for Windows and Linux is a Sun Microsystems workstation.

To intercept Win32 functions, we use the Detours tool [55], a library for intercepting arbitrary Win32 binary functions on X86 machines. We added three modules for i) substituting parameters of system calls by corrupted values ii) observing the reactions of the OS after execution of a corrupted system call, and iii) collecting the required measurements.

To intercept POSIX system calls, we used another interception tool, Strace [80] to which we added modules similar to those added to Detours.

Before each benchmark run (i. e., execution of the series of experiments related to a given OS), the target kernel is installed, and the interceptor is compiled for the current kernel (interceptors are kernel-dependent both for Windows and Linux). Once the benchmarking tool is compiled, it is used to identify the set of system calls activated by the workload. Parameters of these system calls are then analyzed and a database of corrupted values is built accordingly.

At the beginning of each experiment, the target machine records the experiment start instant tExpStart and sends it to the benchmark controller along with a notification of experiment start-up. The workload starts its execution. The Observer module records, in the experiment execution trace, the start-up instant of the workload, tWStart, the activated system calls and their responses. This trace also collects the relevant data concerning states SEr, SXp and SNS. The recorded trace is sent to the benchmark controller at the beginning of the next experiment.

The parameter substitution module identifies the system call to be corrupted. The execution is then interrupted, a parameter value is substituted and the execution is resumed with the corrupted parameter value (t_{Resume} is saved in the experiment execution trace). The state of the OS is monitored so as to diagnose SEr, SXp, SNS. The corresponding OS response time ($t_{Response}$) is recorded in the experiment execution trace. For each run, the OS reaction time after the experiment is calculated as the difference between ($t_{Response}$) and

t_{Resume}. At the end of the execution of the workload, the OS notifies the end of the experiment to the benchmark controller by sending an end signal along with the experiment end instant, t_{ExpEnd}. If the workload does not complete, then t_{ExpEnd} is governed by the value of a watchdog timer. If, at the end of the watchdog timer, the benchmark controller has not received the end signal from the OS, it then attempts to connect to the OS. If this connection is successful, and if the soft reboot is successful, then a workload abort or hang state is diagnosed. If the soft reboot is unsuccessful, then a panic state, SPc, is deduced and a hard reboot is required. Otherwise SHg is assumed.

At the end of a benchmark execution, all files containing raw results corresponding to all experiments are on the benchmark controller. A processing module extracts automatically the relevant information from these files (two specific modules are required for Windows and Linux families). The relevant information is then used to evaluate automatically the benchmark measures (the same module is used for Windows and Linux).

6.3. Benchmark Results

Three versions of Windows OSs are benchmarked: Windows NT4 Workstation with SP6, Windows 2000 Professional with SP4, and Windows XP Professional with SP1[1]. Windows 2000 Professional and Windows NT4 Workstation will be referred to as Windows 2000 and Windows NT4 respectively. Four Linux OSs (Debian distribution) are benchmarked: Linux 2.2.26, Linux 2.4.5, Linux 2.4.26 and Linux 2.6.6. Each of them is a revision of one of the stable versions of Linux (2.2, 2.4, 2.6).

For Windows, the number of system calls activated by the workload is 76 for the three versions, and the number of experiments ranges from 1282 to 1294. For Linux, the number of system calls activated by the workload ranges from 31 for Linux 2.6.6 to 37 for Linux 2.6.26, and the number of experiments ranges from 408 for Linux 2.4.X to 457 for Linux 2.2.26 (it is 409 for Linux 2.6.6).

The robustness measure, OSR, is given in Figure 7. It shows that all OSs of the same family are equivalent. It also shows that none of the catastrophic states (Panic or Hang OS states) occurred for all Windows and Linux OSs. Linux OSs notified more error codes (58-66%) than Windows (25%), while more

---

[1] In our previous work we have also considered the server versions of the same Operating Systems.

exceptions were raised with Windows (22-23%) than with Linux (7-10%). More no-signaling cases have been observed for Windows (52-54%) than for Linux (27-36%).

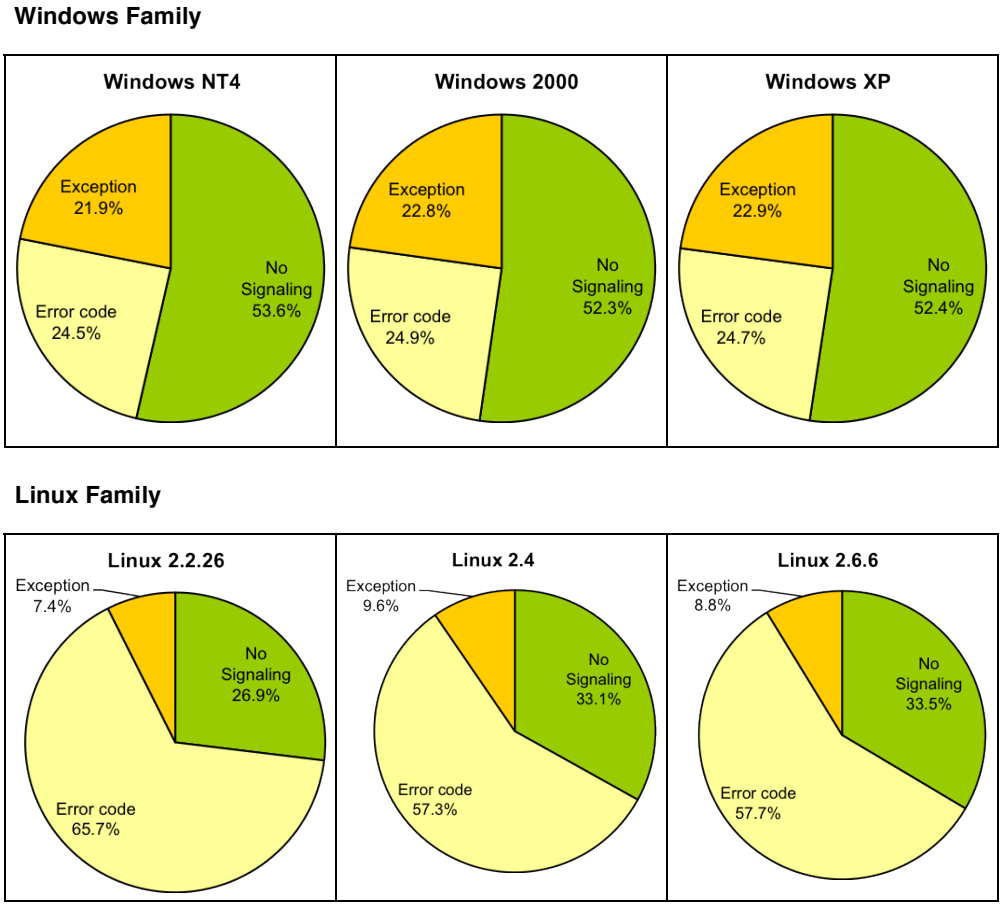


Figure 7: OS Robustness, OSR

These results are in conformance with our previous results, related to Windows using TPC-C Client [61] and to Windows and Linux using PostMark [69]. In [87] it was observed that on the one hand Windows 95, 98, 98SE and CE had a few catastrophic failures and on the other hand Windows NT, Windows 2000 and Linux are more robust and did not have any catastrophic failures as in our case.

The reaction times in the presence of faults (and without fault) are given in Table 4. Note that for the Windows family, XP has the lowest reaction time, and for the Linux family, 2.6.6 has the lowest one. However, the reaction times of Windows NT and 2000 are very high. A detailed analysis showed that the large response time for Windows NT and 2000 are mainly due to system calls LoadLibraryA, LoadLibraryExA and LoadLibraryEXW. Not including these system calls when evaluating the average of the reaction time in the presence of faults leads respectively to 388µs, 190µs and 214µs for NT4, 2000 and XP (the associated average reaction times without fault become respectively 191µs, 278µs and 298µs). For

Linux the high values of the reaction times in presence of faults are also due to three system calls (execve, getdents64, nanosleep). Not including the reaction times associated to these system calls leads respectively to 88µs, 241µs, 227µs and 88µs for Linux 2.2.26, 2.4.5, 2.4.26 and 2.6.6.

Table 4: OS Reaction Times

Windows Family

	Windows NT4		Windows 2000		Windows XP	
	Mean	Standard deviation	Mean	Standard deviation	Mean	Standard deviation
τreac	44035 µs		9504 µs		564 µs	
Treac	12543 µs	154317 µs	306 µs	1047 µs	318 µs	940 µs

Linux Family

	Linux 2.2.26		Linux 2.4.5		Linux 2.4.26		Linux 2.6.6	
	Mean	Standard deviation	Moyenne	Standard deviation	Mean	Standard deviation	Mean	Standard deviation
τreac	2951 µs		2847 µs		2255 µs		865 µs	
Treac	202 µs	583 µs	557 µs	2257 µs	544 µs	2223 µs	505 µs	2198 µs

The restart times are shown in Table 5. The average restart time without faults, τres, is always lower than the benchmark restart time (with faults), Tres, but the difference is not significant. The standard deviation is very large for all OSs. Linux 2.2.26 and Windows XP have the lowest restart time (71 seconds, in the absence of fault) while Windows NT and 2000 restart times are around 90 seconds and those of Linux versions 2.4.5, 2.4.26 and 2.6.6 are around 80 seconds.

It is worth to mention that the average restart times mask interesting phenomena. Detailed analyses show that all OSs of the same family have similar behavior and that the two families exhibit very different behaviors.

For Windows, there is a correlation between the restart time and the workload state at the end of the experiment. When the workload is completed, the restart time is almost the same as the average restart time without substitution. On the other hand, the restart time is statistically larger for all experiments with workload abort/hang. Moreover, statistically, the same system calls lead to workload abort/hang. This is illustrated in Figure 8 in which the benchmark experiments are executed in the same order for the three Windows versions. Similar behaviors have been observed when using TPC-C [61] and PostMark workloads [69].

Table 5: OS Restart Times

Windows Family

	Windows NT4		Windows 2000		Windows XP	
	Mean	Standard Deviation	Mean	Standard Deviation	Mean	Standard Deviation
τres	90 s		88 s		67 s	
Tres	91 s	3 s	89 s	5 s	71 s	5 s

Linux Family

	Linux 2.2.26		Linux 2.4.5		Linux 2.4.26		Linux 2.6.6	
	Mean	Standard Deviation	Mean	Standard Deviation	Mean	Standard Deviation	Mean	Standard Deviation
τres	64 s		74 s		79 s		77 s	
Tres	71 s	37 s	79 s	23 s	83 s	24 s	82 s	27 s

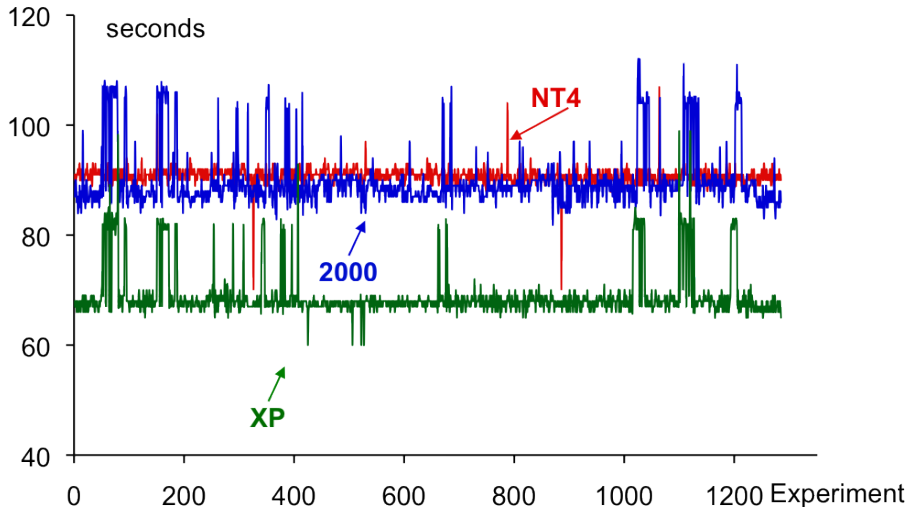


Figure 8: Detailed Restart Time for Windows

Linux restart time is not affected by the workload state. Detailed restart time analyses show high values appearing periodically. These values correspond to a check-disk performed by the Linux kernel every 26 restarts (which explains the important standard deviation on this measure). This is illustrated in Figure 9 for Linux 2.2.26, as an example. The same behavior has been observed when using the PostMark workload [69].

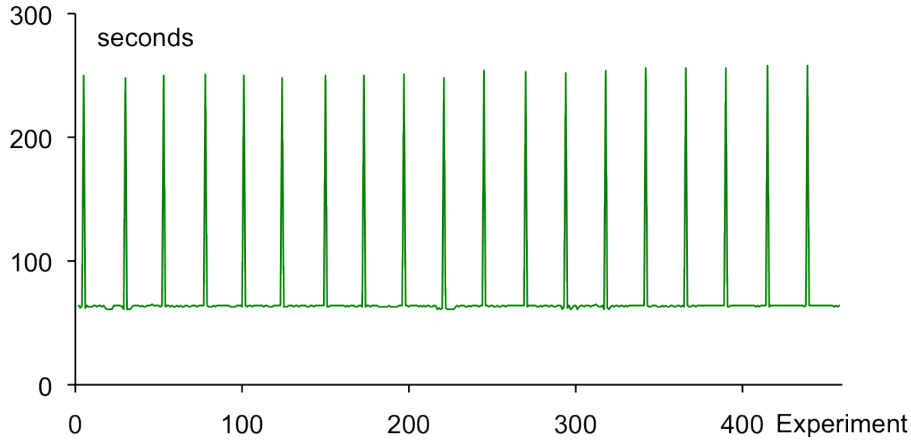


Figure 9: Detailed Restart Time for Linux 2.2.26

7. Conclusion

This chapter was dedicated to system dependability characterization and benchmarking, based on modeling and on measurement. We put emphasis on COTS-based fault tolerant systems and on COTS systems. Before presenting the benchmarking concepts, we gave an overview of the kind of benchmark measures and features that can be used to characterize and benchmark a system. Then we presented the basic approaches that can be used to assess the benchmark measures. When ever appropriate we made reference to performance benchmarks as they can provide good starting means for building dependability benchmarks. To illustrate the various concepts, techniques and results involved in dependability benchmarking we presented two case studies: one addressing the system and service level of a COTS-based fault tolerant system, and one dedicated to a COTS software component. The first case study shows how dependability modeling can be used to benchmark alternative architectural solutions of instrumentation and control systems for nuclear power plants. The benchmarked measure corresponds to system availability. The second case study shows how controlled experiments can be used to benchmark operating systems. The benchmark measures are robustness, reaction time and restart time.

To sum up, dependability benchmarks allow characterization of system dependability. They can provide a good means for comparison between alternative systems. They can also be used for guiding development efforts of system providers, and for supporting acquisition choices of system purchasers, or for comparing the dependability of new versions of a system with respect to previous ones.

While performance benchmarking is already an established discipline, dependability benchmarking is an emerging area, despite the fact that basic valuable techniques are available for dependability assessment, by means of modeling and experimentation. Performance benchmarks are mature enough to be accepted as competition benchmarks. On the other hand, even though current work on dependability benchmarking allowed the definition of several dependability benchmarks, still some work is needed to push towards mature dependability benchmarks to be accepted and supported by the community. What is really missing is the necessary agreement among the various players for the establishment of well-accepted dependability benchmarks, and the establishment of supporting organizations, as for performance benchmarks.

References

[1] J. Aidemark, J. Vinter, P. Folkesson and J. Karlsson, "GOOFI: Generic Object-Oriented Fault Injection Tool", *in Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2001),* Göteborg, Sweden, July 2001, pp. 83-88.

[2] A. Albinet, J. Arlat and J.-C. Fabre, "Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel", *in Int. Conf. on Dependable Systems and Networks,* (Florence, Italy), pp. 867-876, 2004.

[3] H. H. Ammar and S. M. Rezaul Islam, "Time Scale Decomposition of a Class of Generalized Stochastic Petri Net Models", *IEEE Transactions on Software Engineering*, vol. 15, no. 6, pp. 809-820, 1989.

[4] D. de Andrés, J. C. Ruiz, D. Gil and P. Gil, "Run-Time Reconfiguration for Emulating Transient Faults in VLSI Systems", *in Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-2006),* Philadelphia, Pennsylvania, USA, June 2006, pp. 291-300.

[5] D. de Andrés, J. C. Ruiz, D. Gil and P. Gil, "Fault Emulation for Dependability Evaluation of VLSI Systems", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 16, no. 4, April 2008.

[6] L. Antoni, R. Leveugle, and B. Feher, "Using Run-Time Reconfiguration for Fault Injection Applications", *IEEE Transactions on Instrumentation and Measurement,* vol. 52, no. 5, pp. 1468-1473, 2003.

[7] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins and D. Powell, "Fault Injection for Dependability Validation — a Methodology and Some Applications", *IEEE Transactions on Software Engineering,* Feb.1990, vol. 16, pp. 166-182.

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

[9] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs and G. H. Leber, "Comparison of Physical and Software-Implemented Fault Injection Techniques", *IEEE Transactions on Computers,* vol. 52, pp. 1115-1133, 2003.

[10] M. Assaf, S. Das, E. Petriu, L. Jin, C. Jin, D. Biswas, V. Groza and M. Sahinoglu, "Hardware and Software Co-Design in Space Compaction of Cores-Based Digital Circuits", *in Proc. of the 21st IEEE Instrumentation and Measurement Technology Conference (IMTC 04),* Como, Italy, May 2004, vol. 2, pp. 1503-1508.

[11] A. Avizienis, J.-C. Laprie and B. Randell, *Fundamental Concepts of Dependability*, LAAS Research Report, N°1145, April 2001.

[12] D. Avresky, J. Arlat, J. C. Laprie and Y. Crouzet, "Fault Injection for Formal Testing of Fault Tolerance", *IEEE Transactions on Reliability,* Sept. 1996, vol. 45, no. 3, pp. 443-455, 1996.

[13] J. H. Barton, E. W. Czeck, Z. Z. Segall and D. P. Siewiorek, "Fault Injection Experiments Using FIAT", *IEEE Transactions on Computers,* vol. 39, pp. 575-582, 1990.

[14] A. Benso and P. Prinetto, "Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation", Boston: Kluwer Academic Publishers, 2003.

[15] C. Béounes, M Aguera, J. Arlat, S. Bachmann, C. Bourdeau, J.-E. Doucet, K. Kanoun, J.-C. Laprie, S. Metge, J. Moreira de Souza, D. Powell and P. Spiesser, "SURF-2: A Program for Dependability Evaluation of Complex Hardware and Software Systems," *23rd International Symposium on Fault-Tolerant Computing (FTCS- 23),* Toulouse, France, pp. 668–673, 1993.

[16] S. Bernardi and S. Donatelli, "Building Petri Net Scenarios for Dependable Automation Systems," in *10th International Workshop for Petri Nets and Performance Models (PNPM'2003),* Urbana-Champaign, IL, USA, 2003, pp. 72-81.

[17] S. Bernardi, *Building Stochastic Petri Net models for the Verification of Complex Software Systems,* PhD, Universita di Torino, 2003.

[18] C. Betous-Almeida and K. Kanoun, "Construction and Stepwise Refinement of Dependability Models," *Performance Evaluation*, vol. 56, 277-306, 2004.

[19] C. Betous-Almeida and K. Kanoun, "Dependability Modeling of Instrumentation and Control Systems: A Comparison of Competing Architectures," *Safety Science*, vol. 42, 457-480, 2004.

[20 A. Bobbio and K. Trivedi, "An Aggregation Technique for the Transient Analysis of Stiff Markov Chains," *IEEE Transactions on Computers*, vol. C-35, no. 9, pp. 803-814, August 1986.

[21] A. Bondavalli, M. Nelli, L. Simoncini and G. Mongardi, "Hierarchical Modeling of Complex Control Systems: Dependability Analysis of a Railway Interlocking," *Journal of Computer Systems Science and Engineering*, 16(4), pp. 249-261, 2001.

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

[23] A. Brown, L. C. Chung and D. A. Patterson, "Including the Human Factor in Dependability Benchmarks", *in Proc. of the 2002 DSN Workshop on Dependability Benchmarking,* (Washington, DC, USA), 2002.

[24] P. Buchholz, "A Notion of Equivalence for Stochastic Petri Nets," *16th Int. Conf. on Application and Theory of Petri Nets,* Torino, Italy, 1995, pp. 161-180.

[25] J. Carreira, H. Madeira and J. G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers", *IEEE Transactions on Software Engineering,* vol. 24, pp. 125-136, 1998.

[26] P. Chevochot and I. Puaut, "Experimental Evaluation of the Fail-Silent Behavior of a Distributed Real-Time Run-Time Support Built from COTS Components", *in Proc. Int. Conference on Dependable Systems and Networks (DSN-2001),* (Göteborg, Sweden), pp. 304-313, IEEE CS Press, 2001.

[27] [Choi *et al.* 1992] G. S. Choi and R. K. Iyer, "FOCUS: An Experimental Environment for Fault Sensitivity Analysis", IEEE Transactions on Computers, vol. 41, pp. 1515-1526, 1992.

[28] A. Chou, J. Yang, B. Chelf, S. Hallem and D. Engler, "An Empirical Study of Operating Systems Errors", *in Proc. 18th ACM Symp. on Operating Systems Principles (SOSP-2001),* (Banff, AL, Canada), pp. 73-88, ACM Press, 2001.

[29] G. Ciardo and K. S. Trivedi, "Decomposition Approach to Stochastic Reward Net Models," *Performance Evaluation*, vol. 18, no. 1, pp. 37-59, 1993.

[30] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda and M. Violante, "New Techniques for Efficiently Assessing Reliability of SOCs", *Microelectronics Journal,* vol. 34, pp. 53-61, 2003.

[31] P.-J. Courtois, *Decomposability - Queueing and Computer System Applications*, New York: Academic Press, 1977.

[32] Crouzet *et al.* 2006] Y. Crouzet, H. Waeselynck, B. Lussier and D. Powell, "The SESAME Experience: from Assembly Languages to Declarative Models", *in Proc. 2nd Workshop on Mutation Analysis (Mutation'2006),* IEEE, Raleigh, USA, Nov. 2006, 10p.

[33] Y.-S. Dai, Y. Pan and X. Zou, "A Hierarchical Modeling and Analysis for Grid Service Reliability," *IEEE Trans. on Computers*, vol. 56, no. 5, pp. 681-691, 2007.

[34] D. Daly and W. H. Sanders, "A Connection Formalism for the Solution of Large and Stiff Models," *34th Annual Simulation Symposium,* 2001, pp. 258-265.

[35] S. Dawson, F. Jahanian and T. Mitton, "A Software Fault Injection Tool on Real-Time Mach", *in Proc. 16th IEEE Real-Time Systems Symposium,* 1995, Pisa, Italy, Dec. 1995, pp. 130-140.

[36] S. Dawson, F. Jahanian and T. Mitton, "Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection", *in Proc. Int. Symp. on Fault Tolerant Computing,* Sendai, Japan, Jun. 1996, pp. 404-414.

[37] European Project on Dependability Benchmarking, IST-2000-25425, http://www.laas.fr/DBench

[38] T. A. Delong, B. W. Johnson and J. A. Profeta, III, "A Fault Injection Technique for VHDL Behavioral-Level Models", *IEEE Design & Test of Computers,* vol. 13, pp. 24-33, 1996.

[39] S. Donatelli and G. Franceschinis, "The PSR Methodology: Integrating Hardware and Software Models," *17th Int. Conf. on Application and Theory of Petri Nets, ICATPN '96,* Osaka, Japan, 1996, (Springer-Verlag).

[40] J. Durães and H. Madeira, "Characterization of Operating Systems Behavior in the Presence of Faulty Drivers through Software Fault Emulation", *in Proc. 2002 Pacific Rim Int. Symp. on Dependable Computing (PRDC-2002),* (Tsukuba City, Ibaraki, Japan), pp. 201-209, 2002.

[41] J. Durães and H. Madeira, "Emulation of Software Faults by Educated Mutations at Machine-Code Level", *in 13th Proc. Int. Symp. on Software Reliability Engineering (ISSRE'02),* Annapolis, Maryland, USA, Nov. 2002, pp. 329-340.

[42] J. Durães, "Faultloads Based on Software Faults for Dependaility Benchmarking", *PhD Thesis,* Department of Information Engineering, University of Coimbra, 2006.

[43] K. Echtle and M. Leu, "The EFA Fault Injector for Fault-Tolerant Distributed System Testing," *in IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems,* Amherst, MA, USA, 1992, pp. 28-35.

[44] K. Echtle and M. Leu, "Test of Fault Tolerant Distributed Systems by Fault Injection," *in Fault-Tolerant Parallel and Distributed Systems,* D. Pradhan and D. Avresky, Eds.: IEEE Computer Society Press, 1995, pp. 244-251.

[45] Embedded Microprocessor Benchmarking Consortium, Web Site: http://www.eembc.org/

[46] Rudolf Eigenmann (Editor), *Performance Evaluation and Benchmarking With Realistic Applications,* MIT Press, 2001.

[47] N. Fota, M. Kâaniche and K. Kanoun, "Incremental Approach for Building Stochastic Petri Nets for Dependability Modeling," in *Statistical and Probabilistic Models in Reliability,* (Ionescu and Limnios, Eds.), pp. 321-335, Birkhäuser, 1999.

[48] N. Fota, M. Kâaniche and K. Kanoun, "Dependability Evaluation of an Air Traffic Control Computing System," *Performance Evaluation*, vol. 35 (3-4), 553-573, 1999.

[49] K. Goswami, "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis", *IEEE Transactions on Computers,* vol. 46, pp. 60-74, 1997.

[50] W. Gu, Z. Kalbarczyk and R. K. Iyer, "Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors", *in Int. Conf. on Dependable Systems and Networks,* (Florence, Italy), pp. 887-896, 2004.

[51] U. Gunneflo, J. Karlsson and J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-Ion Radiation," *in Proc.19th International Symposium on Fault-Tolerant Computing (FTCS-19),* Chicago, IL, USA, 1989, pp. 340-347.

[52] S. Haddad and P. Moreaux, "Approximate Analysis of Non-Markovian Stochastic Systems with Multiple Time Scale Delays," *12th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* Volendam, NL, 2004.

[53] S. Han, K. G. Shin,and H. A. ´Rosenberg, "DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems", *in Proc. Int. Computer Performance and Dependability Symposium,* Erlangen, Germany, 1995, pp. 204-213.

[54] M.-C. Hsueh, T. K. Tsai and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, pp. 75-82, 1997.

[55] G. Hunt and D. Brubaher, "Detours: Binary Interception of Win32 Functions", *3rd USENIX Windows NT Symp.,* Seattle, Washington, USA, pp. 135-144, 1999.

[56] IEEE Std 1149.1-2001 - IEEE Standard Test Access Port and Boundary-Scan Architecture: IEEE, Piscataway, NJ 08854 USA, 2001.

[57] IEEE-ISTO 5001 - the Nexus 5001 ForumTM Standard for a Global Embedded Processor Debug Interface: IEEE-ISTO, Piscataway, NJ 08854 USA, 2003.

[58] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson and J. Karlsson, "Fault Injection into Vhdl Models: The Mefisto Tool", *in Proc. 24th Int. Symposium on Fault-Tolerant Computing,* Pasadena, CA, USA, 1994, pp. 66-75.

[59] M. Kaâniche, K. Kanoun and M. Rabah, "Multi-Level Modeling Approach for the Availability Assessment of e-Business Applications," *Software: Practice and Experience*, vol. 33, no. 14, pp. 1323-1341, 2003.

[60] M. Kaaniche , P. Lollini , A. Bondavalli , K. Kanoun, "Modeling the Resilience of Large and Evolving Systems", *International Journal of Performability Engineering,* Vol.4, N°2, pp.153-168, 2008.

[61] A. Kalakech, T. Jarboui, J. Arlat, Y. Crouzet and K. Kanoun, "Benchmarking Operating System Dependability: Windows 2000 as a Case Study", in *Proc. 2004 Pacific Rim Int. Symp. on Dependable Computing,* (Papeete, Tahiti, French Polynesia), pp. 261-270, 2004.

[62] A. Kalakech, K. Kanoun, Y. Crouzet and J. Arlat, "Benchmarking the Dependability of Windows NT, 2000 and XP", *in Proc. Int. Conf. on Dependable Systems and Networks (DSN-2004),* Florence, Italy, pp. 681-686, 2004.

[63] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties", *in Proc. 22nd Int. Symp. on Fault-Tolerant Computing,* Boston, MA, USA, July 1992, pp. 336-344.

[64] K. Kanoun and M. Borrel, "Fault-Tolerant System Dependability — Explicit Modeling of Hardware and Software Component-Interactions," *IEEE Transactions on Reliability*, vol. 49, no. 4, pp. 363-376, December 2000.

[65] K. Kanoun and Y. Crouzet, "Dependability Benchmarking for Operating Systems", in *International Journal of Performance Engineering,* vol. 2, no. 3, pp. 275-287, 2006.

[66] K. Kanoun and L. Spainhower (Eds), "Dependability Benchmarking for Computer Systems", Wiley-IEEE Computer Society Press, 2008, 362 p,00 http://eu.wiley.com/WileyCDA/WileyTitle/productCd-047023055X.html.

[67] K. Kanoun, M. Borrel, T. Morteveille and A. Peytavin, "Modeling the Dependability of CAUTRA, a Subset of the French Air Traffic Control System," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 528-535, 1999.

[68] K. Kanoun *et al.,* Project full final report 2004, Project Reports Section, http://www.laas.fr/DBench

[69] K. Kanoun, Y. Crouzet, A. Kalakech, A.-E. Rugina and P. Rumeau, "Benchmarking the Dependability of Windows and Linux using Postmark Workloads", *in Proc. 16th Int. Symposium on Software Reliability Engineering (ISSRE-2005),* (Chicago, USA), 2005.

[70] W.I. Kao and R. K. Iyer, "Define: A Distributed Fault Injection and Monitoring Environment," in Fault-Tolerant Parallel and Distributed Systems, D. Pradhan and D. Avresky, Eds.: IEEE Computer Society Press, 1995, pp. 252-259.

[71] W. I. Kao, R. K. Iyer and D. Tang, "Fine: A Fault Injection and Monitoring Environment for Tracing the Unix System Behavior under Faults", *IEEE Transactions on Software Engineering,* vol. 19, pp. 1105-1118, 1993.

[72] J. Karlsson, U. Gunneflo, P. Liden and J. Torin, "Two Fault Injection Techniques for Test of Fault Handling Mechanisms", *in Proc. Int. Test Conference,* Nashville, TN, USA, Oct. 1991, pp. 140-9.

[73] 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), pp. 30-37, IEEE CS Press, 1999.

[74] P. Koopman, J. Sung, C. Dingman, D. Siewiorek and T. Marz, "Comparing Operating Systems using Robustness Benchmarks", *in Proc. 16th Int. Symp. on Reliable Distributed Systems,* Durham, USA, pp. 72-79, 1997.

[75] C. Kwang-Ting, H. Shi-Yu and D. Wei-Jin, "Fault Emulation: A New Methodology for Fault Grading," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 18, pp. 1487-1495, 1999.

[76] J.-C. Laprie, "Dependable Computing: Concepts, Limits, Challenges", *25th Int. Symp. on Fault-Tolerant Computing (FTCS-25),* Special Issue, (Pasadena, CA, USA), pp.42-54, IEEE Computer Society Press, 1995.

[77] P. Lollini, A. Bondavalli and F. Di Giandomenico, "A Modeling Methodology for Hierarchical Control Systems and its Application," *Journal of the Brazilian Computer Society*, vol. 10, no. 3, pp. 57-69, 2005.

[78] H. Madeira, M. Rela, F. Moreira and J. G. Silva, "A General Purpose Pin-Level Fault Injector", *in Proc. European Dependable Computing Conference,* Berlin, Germany, 1994, pp. 199-216.

[79] E. Marsden, J.-C. Fabre and J. Arlat, "Dependability of CORBA Systems: Service Characterization by Fault Injection", *in Proc. 21st IEEE Symposium on Reliable Distributed Systems (SRDS 2002),* Osaka, Japan, Oct. 2002, pp. 276-285, 2002.

[80] R. McGrath and W. Akkerman, "Source Forge Strace Project", http://sourceforge.net /projects /strace, 2004.

[81] J. F. Meyer and W. H. Sanders, "Specification and Construction of Performability Models," in *Int. Workshop on Performability Modeling of Computer and Communication Systems,* Mont Saint Michel, France, 1993, pp. 1-32.

[82] Aad van Moorsel, E. Alberdi, R. Barbosa, R. Bloomfield, A. Bondavalli, J. Durães, R. Esposito, L. Falai, J. Karlsson, P. Lollini, H. Madeira, I. Majzik, Z. Micskei, L. Montecchi, G. Pinter, V. Stankovic, L. Strigini, M. Vadursi, M. Vieira, K. Wolter, H. Zhang, "State of the Art", Deliverable no. D2.2 of the Project AMBER (Assessing, Measuring, and Benchmarking Resilience), FP7-216295, June 2009, 250p. http://www.amber-project.eu

[83] A. Mukherjee and D. P. Siewiorek, "Measuring Software Dependability by Robustness Benchmarking", *IEEE Transactions of Software Engineering,* vol. 23 no. 6, pp. 366-376, 1997.

[84] M. Nelli, A. Bondavalli and L. Simoncini, "Dependability Modeling and Analysis of Complex Control Systems: an Application to Railway Interlocking," *European Dependable Computing Conference (EDCC-2),* Taormina, Italy, 1996, pp. 93-110, (Springer-Verlag).

[85] M. Rabah and K. Kanoun, "Performability Evaluation of Multipurpose Multiprocessor Systems: the "Separation of Concerns" Approach," *IEEE transactions on Computers*, vol. 52, no. 2, pp. 223-236, 2003.

[86] M. Rebaudengo and M. Sonza Reorda, "Evaluating the Fault Tolerance Capabilities of Embedded Systems via BDM", *in Proc. 17th IEEE VLSI Test Symposium,* Dana Point, CA, USA, Apr. 1999, pp. 452-457.

[87] C. Shelton, P. Koopman and K. D. Vale, "Robustness Testing of the Microsoft Win32 API", *in Proc. Int. Conference on Dependable Systems and Networks (DSN-2000),* (New York, NY, USA), pp. 261-270, IEEE CS Press, 2000.

[88] Special Interest Group on Dependability Benchmarking of the IFIP Working Group 10.4 http://homepages.laas.fr/kanoun/ifip_wg_10_4_sigdeb/

[89] Web Site of the Standard Performance Evaluation Corporation: http://www.spec.org/

[90] Web Site of the Transaction Processing Performance Council: http://www.tpc.org/

[91] 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, pp. 314-323, IEEE CS Press, 1996.

[92] D. Wilson, B. Murphy and L. Spainhower. "Progress on Defining Standardized Classes of Computing the Dependability of Computer Systems," in *Proc. DSN 2002 Workshop on Dependability Benchmarking,* pp. F1-5, Washington, D.C., USA, 2002.

[93] P. Yuste, D. de Andres, L. Lemus, J. Serrano and P. Gil, "INERTE: Integrated NExus-Based Real-Time Fault Injection Tool for Embedded Systems", *in Proc. Int. Conf. on Dependable Systems and Networks (DSN-2003)*, San Francisco, CA, USA, June 2003, pp. 669-669.

[94] J. Zhu, J. Mauro, and I. Pramanick, "Robustness Benchmarking for Hardware Maintenance Events", *in Proc. Int. Conf. on Dependable Systems and Networks (DSN 2003),* pp. 115-122, San Francisco, CA, USA, IEEE CS Press, 2003.

[95] J. Zhu, J. Mauro and I. Pramanick. "R3 - A Framework for Availability Benchmarking," *in Proc. Int. Conf. on Dependable Systems and Networks (DSN 2003),* pp. B-86-87, San Francisco, CA, USA, 2003.

[96] H. Ziade, R. Ayoubi and R. Velazco, "A Survey on Fault Injection Techniques", *International Arab Journal of Information Technology,* vol. 1, no. 2, July 2004, pp. 171-186.

Appendix:

Brief overview of the Chapters[2] in

Dependability Benchmarking for Computer Systems

Chapter 1 describes an autonomic computing benchmark which measures system resiliency. In common with other benchmarks, it has a quantitative throughput metric to capture the impact of disturbances. The autonomic computing benchmark also has a qualitative metric representing the level of human interaction needed to detect, analyze, and recover from disturbances.

Chapter 2 contains three benchmarks which analytically measure reliability, availability and serviceability of a system. All three are intended to allow a consistent dependability feature versus cost analysis for widely varying system architectures.

Chapter 3 covers two quantitative benchmarks which concentrate on the recovery of standalone systems and enterprise clusters. Recovery is broadly defined to be the time required for a system to return to service. For each benchmark, a scoring and weighting methodology is described.

The benchmark in Chapter 4 changes the physical environment around the system in order to quantitatively measure susceptibility to silent data corruption.

Chapters 5 and 6 include performance measurements in the presence of faults, providing performance degradation (due to faults) and availability measures to end-users of On-Line Transaction Processing and Web Serving systems. Chapter 5 discusses a benchmark for On-Line Transaction Processing which focuses on availability using direct experimental measures. Chapter 6 includes a measurement-based benchmark for Web Servers which encompasses availability, accuracy, resilience and service degradation.

---

[2] The description is as it appears in the Preface of the book itself [66].

Chapter 7 is dedicated to the benchmark of automotive control applications running inside engine control units. It addresses the safety of the software applications from the perspective of the engine they control. It is aimed at supporting the selection or the purchase of automotive applications for electronic controller units.

In Chapters 8 and 9, the systems under benchmarks are respectively the intrusion detection mechanisms and fault-tolerance algorithms.

Chapter 8 discusses issues that should be addressed when benchmarking intrusion detectors in the cyber domain. It is worth to mention that no intrusion detector benchmark is currently available, and that this chapter does not develop a benchmark per se.

Chapter 9 addresses Byzantine protocols dedicated to Byzantine-fault tolerance. The benchmark is aiming at assessing the effectiveness of Byzantine-fault tolerance implementation. It is illustrated on a specific protocol, the Castro-Liskov protocol.

Chapters 10 to 15 exemplify the multi-faceted feature of dependability benchmarking, illustrated on operating systems. The benchmarks presented mainly differ by the nature of faults considered: Chapter 10 addresses internal faults, while chapters 11 to 15 are concerned with external faults. Chapters 11, 12, and13 address faults in the software application, Chapter 14 faults in device drivers, and Chapter 15 faults in the hardware platforms.

Chapter 10 defines a benchmark that captures the user's expectations of product reliability during common user experiences. While the benchmark can be applied to any software product, this chapter focuses on the application of this benchmark for operating systems development. Its aim is to help the operating system developer to improve, during its development, the operating system's reliability.

Chapters 11 to 14 present benchmarks based on experimentation. They consider the operating system as a black box, solicit it only through its available inputs and observe its behavior only through the operating system outputs. The ultimate objective of end-user's benchmarks is to improve the application software, or the device drivers, or the hardware platform. However the benchmark results may indicate weaknesses in the operating system that the operating system developer may use, with further analyses, to improve the operating system.

Chapter 11 is dedicated to the Ballista project, the first operating system robustness testing approach. It reexamines the design decisions made, and the lessons learned from a decade of robustness testing research.

Chapter 12 elaborates on the Ballista approach and develops a benchmark evaluating, in addition to the operating system robustness, two measures, the system response time and restart time in the presence of faults.

Chapter 13 focuses on real-time kernels. It characterizes the response time predictability, based on the divergence of the response time in the presence of fault, and the frequency of out-of-boundary responses times.

Chapter 14 concentrates on failures modes of the operating systems induced by faulty drivers. Additionally, it evaluates three complementary measures: responsiveness, availability, and workload safety.

Chapter 15 considers in a first step the operating system as a black box, and develops an end-user benchmark, then shows how this benchmark can be complemented to help the operating system developer to improve the operating system's dependability.

Chapter 16 is dedicated to microprocessors benchmarks, with respect to electrical charge induced by high energy particles, referred to as soft errors. As the benchmark presented does not require any specific knowledge about the microprocessors. It can be used by the manufacturers or end-users.