UNIVERSITÀ DEGLI STUDI DI FIRENZE
Dipartimento di Sistemi e Informatica
Dottorato di Ricerca in Ingegneria Informatica e dell'Automazione
Cicle XXIV
Disciplinar Sector INF/01

ANALYSIS OF CRITICAL SYSTEMS THROUGH RIGOROUS, REPRODUCIBLE AND COMPARABLE EXPERIMENTAL ASSESSMENT

ANDREA CECCARELLI

Supervisor: *Prof. Andrea Bondavalli*

PhD Coordinator: *Prof. Luigi Chisci*

January, 2012

Andrea Ceccarelli: *Analysis of critical systems through rigorous, reproducible and comparable experimental assessment,* Dottorato di Ricerca in Ingegneria Informatica e dell'Automazione, © January, 2012

INTRODUCTION

The world of computer systems today is composed of very different kinds of critical architectures: from embedded safety-critical sensors and safety equipment (e.g., train on-board equipment), to large, highly dependable multi-computers (e.g. plant control systems), to smart resilient components for ubiquitous networks (e.g., biometrics monitoring applications). The common trend for all of them is to become open and part of an integrated cyber world; still, each of them brings specific challenges that need to be addressed for their assessment, possibly leading to the different validation solutions [157], [171].

The evaluation of the behavior of critical systems is important in V&V (Verification and Validation) and in fault forecasting [12], because it allows to estimate the adequacy of a system with respect to the requirements given in its specification. The scientific literature and the industrial practice show that validation of critical systems through quantitative evaluation of resilience and dependability attributes is a key issue.

Several approaches have been proposed to evaluate the dependability properties of a computer-based system. These are generally classified into three categories: analytic, simulative and experimental. The analytical approach is based on the construction of a parametric model of the execution environment, while following a simulative approach we execute algorithms in a simulated execution environment (usually based on a stochastic model) [12].

Finally, in the experimental approach, the process of evaluating a system is based on observations of the behavior of the system under analysis. Experimental measurement is an attractive option for evaluating an existing system or prototype, because it allows to observe the real execution of the system to obtain (hopefully, highly accurate) measurements of the system in its usage environment [130].

Surveying the state of the art on experimental approaches, several methodologies exist for experimental evaluation of critical systems, which share to a large extent similar foundations, and refer or include a huge set of techniques explored and presented through years. To mention only a few representative examples, we cite [8], [10], [61], [73], [111] and [131].

This Thesis proposes a general methodology for the experimental evaluation of critical systems, born from an attentive analysis of the state of the art. Two key aspects are tackled, that we consider major milestones of experimental evaluation but that surveying the state of the art have been often underestimated. These two aspects are:

- Application of principles of metrology (i.e., measurement theory) to perform a metrological assessment of results and measuring systems [24], [26]. In fact, while the attention to quantitative evaluation based on measurements of dependability attributes is renown, as well as the common sensibility in the identification of the measurands, there is seldom attention to characterize tools and results for what they really are i.e., *measuring instruments* and *measurement results*. The application of knowledge available in a recognized discipline such as metrology can result in increased trust in the observations performed while assessing critical systems.
- The adoption of efficient techniques for data retrieval and sharing, to improve reuse, comparison and sharing of measurement results [122], [48]. Despite the fact that sharing results and comparing them is of paramount importance in the current dependability research community, only few solutions actually exist, and are not widely used. Also, the approach followed to quantitatively assess algorithms and systems is generally not univocal, and varies from one work to another: this makes comparison among results reported in different works difficult, if not meaningless. Finally, data collection, sharing, analysis and comparison is complicated by the fact that there is no widely accepted format to describe the results collected and there exist no widespread common guidelines on what data to collect and how.

The methodology proposed in this Thesis is general for the experimental evaluation of critical systems; that is, it is independent from the kind of system or specific technique selected. Also, it specifically targets the industrial context, because it supports the experimental evaluation activities performed as part of standard-compliant V&V processes in industrial practices. The industrial perspective of the approach is discussed examining the possible interplay of the methodology with a framework for the support of V&V processes.

The conceptual methodology is applied to five case studies that embrace five different categories of system, from COTS (Commercial Off-The-Shelf) testing to testing of highly distributed and adaptive systems. In the following table the five case studies are summarized, ordered by increasing systems' distributedness and dimension.

In details, this dissertation is organized in three parts as follows.

Part (i) describes the background, the objectives and the motivations at the base of this work, and the state of the art on related topics. This part is comprised of two chapters, Chapter 1 and Chapter 2.

Chapter 1 contains an overview on basic notions and definitions that constitute the foundation of the research work presented. It shows basic concepts of dependability and security, describes monitoring and experimental evaluation activities, and discusses fundamentals of measurement theory.

	Chapter 4	**Chapter** 8	**Chapter** 9	**Chapter** 5	**Chapter** 6
Kind of system	middleware service	hw COTS	embedded systems	distributed algorithms	Service Oriented Architectures (SOAs)
Details	software clock for resilient timekeeping Resilient and Self-Aware Clock (R&SAClock)	low cost GPS devices	train-borne equipment Driver machine Interface (DMI)	algorithms for distributed systems and tools for rapid prototyping and testing	SOA, with focus on dinamicity and interoperability aspects.
Testing activity	functional, fault-injection, performance testing	on-field testing	fault injection testing	functional, fault-injection testing	robustness testing

Overview of the case studies presented in the Thesis.

Chapter 2 identifies challenges, motivations and state of the art methodologies for the experimental evaluation of critical systems. The objective of this Chapter is to point out the relevance of i) the role targeted by the metrological assessment of instruments and results and ii) (trustworthy) data comparison, sharing, and archiving. The intended outcome of this analysis is to show to the reader the relevance of defining a new methodology for experimental evaluation which applies basics from metrology and guidelines for data comparison and sharing.

Part (ii) of the Thesis contains the conceptual methodology we propose for the experimental evaluation, and its application to three case studies developed in the academic context (the two remaining case study, that are closer to the industrial context, are presented in the successive Part (iii)). Part (ii) is composed of Chapter 3, Chapter 4, Chapter 5 and Chapter 6.

Chapter 3 describes the conceptual methodology developed, which stems from the observations and challenges raised in Part (i). The methodology presented aims to overcome the limitations already mentioned. The methodology is described, and then the five case studies that will be detailed in the following of the Thesis are summarized to point out the peculiarities and different aspects of the application of the methodology to the various case studies.

Chapter 4 describes the first case study. It is related to the experimental evaluation of a middleware component, namely the software clock Reliable and Self Aware Clock (R&SAClock, [23]). The Chapter focuses on the validation methodology and the assessment of the measuring system, including the analysis of faultload and workload intrusiveness and representativeness, and providing a set-up which can be reused for different instantiation of the tested software clock.

The second case study (Chapter 5) focuses on the design, implementation and metrological assessment of a new tool for dependability measurements in distributed protocols, which allows the user evaluating the uncertainty of measurement results involving time interval measurements. The tool is presented along with an experimental campaign to measure distributed round-trip delays on a Wide Area Network (WAN). This case study focuses more on the design and implementation of the measuring instrument and on the collection of trusted measurement results than on the execution of experiments and presentation of results.

The third case study is instead reported in Chapter 6. The argument of this case study is testing of SOAs (Service Oriented Architectures). In particular, the focus of this Chapter is on lifelong testing of SOAs [70], [16], to present a testing tool and related testing methodology for the continuous experimental evaluation of SOAs, which is performed during both offline and online execution. The approach for the design and use of such tool is based on the methodology of Chapter 3, and defines service discovery and testing actions that are applied via the introduction in the SOA of a *testing service*. In this case study emphasis is put on the sharing of information between providers, both in terms of knowledge of the SOA and results (outputs) of the testing activity performed.

The third and final part (Part (iii)) of this Thesis instead addresses the problem of the applicability of our experimental evaluation methodology to industrial V&V processes. Two case studies that result from cooperations with industries are presented. This part is comprised of Chapter 7, Chapter 8 and Chapter 9.

Chapter 7 is devoted to discuss the relevance and applicability of our work to industrial V&V processes. The need of strict methodological process in industrial practices for the validation of critical systems is presented, then a novel framework for V&V processes is introduced, and its relations to the testing methodology is shown. The framework is named RACME (Resiltech Assessment Certification MEthodology, [40]) and it is conceived to supports and guide a V&V expert through a whole V&V process; it is customizable for different systems and V&V processes. The Chapter first shows the RACME solution for the management of *any* V&V and certification process, and then illustrates that our methodology *as it is* can successfully fit processes managed by RACME.

The case study in Chapter 8 is developed in cooperation with Ansaldo STS to experimentally evaluate cheap GPS devices, that are components of a safety-critical system. The experiments performed aim to quantify the localization error of cheap COTS GPS devices, to provide feedbacks of their behavior to the system designers. The activity performed is part of the testing activities executed as a support to the designers, rather than final assessment activities performed on the prototype. These evaluation also allowed to accurately validate the behavior of one fundamental component of the whole system, that is the GPS device.

The fifth case study presented in this Thesis is instead reported in Chapter 9. This case study is devoted to the validation through fault injection of the prototype of a safety-critical railway trainborne equipment, the Driver Machine Interface (DMI). This activity, performed in the context of a broader project and in cooperation with Ansaldo STS, is executed as part of the planned set of V&V activities for the final validation of the prototype.

RELATED PUBLICATIONS

This Thesis is based on works presented in the following publications:

I. A. Bondavalli, A. Ceccarelli, L. Falai, M. Vadursi. Foundations of metrology in the observation of critical systems. To appear in *Resilience Assessment and Evaluation*, book chapter.

II. A. Ceccarelli, M. Vieira, A. Bondavalli. A testing service for lifelong validation of dynamic SOA. In 13*th IEEE International High Assurance Systems Engineering Symposium (HASE)*, November 2011.

III. A. Bondavalli, A. Ceccarelli, F. Gogaj, A. Seminatore, M. Vadursi. Localization errors of low-cost GPS devices in railway worksite-like scenarios. In *IEEE Workshop on Measurements & Networking (M& N)*, October 2011.

IV. A. Ceccarelli, L. Vinerbi, L. Falai, A. Bondavalli. RACME: a framework to support V&V and certification. In *Latin-American Symposium on Dependable Computing (LADC)*, April 2011.

V. A. Ceccarelli, M. Vieira, A. Bondavalli. A Service Discovery Approach for Testing Dynamic SOAs. In *Second IEEE Workshop on Self-Organizing Real-Time Systems (ISORCW-SORT)*, March 2011.

VI. A. Bondavalli, F. Brancati, A. Ceccarelli. Experimental Validation of a Synchronization Uncertainty-Aware Software Clock. In 29*th IEEE Symposium on Reliable Distributed Systems (SRDS)*, October 2010.

VII. Bondavalli, A. Ceccarelli, P. Lollini. Architecting and validating dependable systems: experiences and visions. Book chapter in *Architecting Dependable Systems 7 (ADS7)*, Editors: A. Casimiro, R. de Lemos, C. Gacek, 2010.

VIII. A. Ceccarelli, J. Gronbaek, L. Montecchi, H.-P. Schwefel, A. Bondavalli. Towards a framework for self-adaptive reliable network services in highly-uncertain environments. In 13*th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, May 2010.

IX. A. Bondavalli, A. Ceccarelli, L. Falai, M. Vadursi. A new approach and a related tool for dependability measurements on distributed systems. *IEEE Transactions on Instrumentation and Measurement*, April 2010.

X. A. Bondavalli, F. Brancati, and A. Ceccarelli. Safe estimation of time uncertainty of local clocks. In *IEEE Symposium on Precision Clock Synch. for Measur., Contr. and Comm. (ISPCS)*, October 2009.

XI. A. Bondavalli, F. Brancati, A. Ceccarelli, and L. Falai. An experimental framework for the analysis and validation of software clocks. In *LNCS Software Technologies for Embedded and Ubiquitous Systems (SEUS)*, November 2009.

XII. A. Ceccarelli, A. Bondavalli, and D. Iovino. Trustworthy evaluation of a safe driver machine interface through software-implemented fault injection. In *IEEE 15th Pacific Rim International Symposium on Dependable Computing (PRDC)*, November 2009.

and the currently submitted works:

XIII. A. Bondavalli, A. Ceccarelli, F. Gogaj, M. Vadursi, A. Seminatore. Experimental assessment of low-cost GPS-based localization in railway worksite-like scenarios. *Submitted to Special Issue on IEEE Transactions on Instr. and Measurements.*

XIV. A. Bondavalli, F. Brancati, A. Ceccarelli, L. Falai, M. Vadursi. Resilient estimation of synchronization uncertainty through software clocks. *Submitted to IEEE Transactions on Computers.*

The following book chapters (in Italian) for the didactic book "L'analisi quantitativa dei sistemi critici" (Quantitative analysis of critical systems), Editor: A. Bondavalli contain arguments that constitute the basic notions of this Thesis:

XV. P. Lollini, A. Ceccarelli, M. Vadursi. Richiami di probabilità e metrologia. In *L'Analisi Quantitativa dei Sistemi Critici*, Editor A. Bondavalli, Società Editrice Esculapio, First edition, 2011.

XVI. A. Bondavalli, F. Brancati, A. Ceccarelli. Monitoring di Sistemi. In *L'Analisi Quantitativa dei Sistemi Critici*, Editor A. Bondavalli, Società Editrice Esculapio, First edition, 2011.

The following publications are inherent to the Thesis' topic but were published before the beginning of the PhD course.

XVII. A. Bondavalli, A. Ceccarelli, J. Gronbaek, D. Iovino, L. Karna, S. Klapka, T.K. Madsen, M. Magyar, I. Majzik, and A. Salzo. Design and evaluation of a safe driver machine interface. *International Journal of Performability Engineering (IJPE)*, January 2009.

XVIII. A. Bondavalli, A. Ceccarelli, and L. Falai. Assuring resilient time synchronization. In *The 27th IEEE Symposium on Reliable Distributed Systems*, October 2008.

XIX. A. Bondavalli, A. Ceccarelli, L. Falai, and M. Vadursi. Foundations of measurement theory applied to the evaluation of dependability attributes. In *IEEE Int. Conference on Dependable Systems and Networks (DSN)*, June 2007.

XX. A. Bondavalli, A. Ceccarelli, L. Falai, and M. Vadursi. Towards making NekoStat a proper measurement tool for the validation of distributed systems. In *8th International Symposium on Autonomous Decentralized Systems (ISADS)*, March 2007.

XXI. A. Bondavalli, A. Ceccarelli, and L. Falai. A self-aware clock for pervasive computing systems. In *The 15th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, February 2007.

Finally, the following publications and currently submitted works are marginally or not related to the topic of this Thesis.

XXII. A. Ceccarelli, I. Majzik, D. Iovino, F. Caneschi, G. Pinter, and A. Bondavalli. A resilient SIL 2 driver machine interface for train control systems. In *IEEE Third International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX)*, June 2008.

XXIII. J. Gronbaek, H.-P. Schwefel, A. Ceccarelli, A. Bondavalli. Improving robustness of network fault diagnosis to uncertainty in observations. In *9th IEEE International Symposium on Network Computing and Applications (IEEE NCA)*, July 2010.

XXIV. A. Seminatore, L. Ghelardoni, A. Ceccarelli, L. Falai, M. Schultheis, B. Malinowsky. ALARP (A Railway Automatic Track Warning System Based on Distributed Personal Mobile Terminals). *Submitted to Transport Research Arena (TRA)* 2012.

XXV. A. Bondavalli, F. Brancati, A. Ceccarelli, L. Falai. Providing safety-critical and real-time services for mobile devices in uncertain environment. *Submitted to Book Chapter of Self-Organization in Embedded Real-Time Systems, Springer*.

XXVI. B. Malinowsky, J. Gronbaek, H.P. Schwefel, A. Ceccarelli, A. Bondavalli, E. Nett. Realization of Timed Broadcast via Off-the-Shelf WLAN DCF Technology for Safety-critical Systems. *Submitted to European Dependable Computing Conference (EDCC)* 2012.

XXVII. F. Brancati, A. Ceccarelli, A. Bondavalli, I. Tempestini, M. Puccio. Improving security of Internet services through continuous and transparent user identity verification. *Submitted to the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* 2012.

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Andrea Bondavalli for the continuous support of my Ph.D study and research. Working with him was a great experience, both from the professional and personal side.

Plenty of gratitude also goes to Prof. Marco Vieira from University of Coimbra for his advices and the people at DEI where I spent three very enjoyable months and put the basis of some of the research works which constitute an important part of this Thesis.

I would also like to express my gratitude to all collaborators of my research works, primarily Francesco Brancati, Lorenzo Falai, Florjan Gogaj, Danilo Iovino, Jesper Gronbaek and Michele Vadursi.

Sincere thanks goes to Paolo Lollini and Leonardo Montecchi of the RCL group for the enjoyable time spent together, people at Resiltech S.r.l., and my colleagues working in the PhD room.

Also a thank goes to the many people which collaborated in the National and International projects which made possible the availability of most of the case studies presented in this Thesis; primarily, the FP6 project SAFEDMI, the FP7 project ALARP, and the national project PRIN DOTS-LCCI.

Finally, I would like to thank Lucia and everybody who was important to the successful realization of this Thesis, as well as expressing my apology that I could not mention personally one by one.

CONTENTS

INTRODUCTION iii
RELATED PUBLICATIONS ix
ACKNOWLEDGMENTS xiii

I OBJECTIVES, MOTIVATIONS AND STATE OF THE ART 1
1 Context of the research 3
1.1 Dependability (and security) concepts 3
1.1.1 System, service, faults, errors and failures 3
1.1.2 A taxonomy of faults, failures and errors 4
1.1.3 Dependability and its attributes 7
1.1.4 Means to attain dependability 8
1.2 Basics on monitoring and experimental evaluation 10
1.2.1 Monitoring 11
1.2.2 Experimental evaluation 15
1.3 Foundations of metrology 20
1.3.1 Basic concepts 20
1.3.2 Uncertainty in measuring computing systems 23
2 The need of methodologies for the experimental evaluation 25
2.1 Challenges in monitoring and testing 25
2.2 Methodologies for experimental evaluation 27
2.2.1 RODS 27
2.2.2 The NetLogger methodology 28
2.2.3 Fault injection methodologies 29
2.2.4 Methodologies in software engineering 30
2.2.5 Standards for testability and diagnosability 31
2.3 Trustworthy observation of critical systems 32
2.3.1 Metrology and dependability 32
2.3.2 Measurements properties in tools and experiments for dependability evaluation 36
2.4 Analysis and sharing of experimental results 42
2.4.1 Data acquisition and manipulation techniques 42
2.4.2 Data repositories 44

II PROPOSED METHODOLOGY AND THREE CASE STUDIES 47
3 A methodology for the experimental evaluation of critical systems 49
3.1 Overall view of the methodology 50
3.2 PHASE 1: planning and definition 53
3.2.1 Objectives definition 53

3.2.2 System definition 53
3.2.3 Quantities to assess 54
3.2.4 Workload and faultload definition 55
3.2.5 Experiments definition 55
3.2.6 Architecture of the measuring system 56
3.2.7 Results planning 57
3.2.8 Analysis and planning 57
3.2.9 Data processing tools 58
3.3 PHASE 2: design instrumentation 58
3.3.1 Design and implementation of the monitor 60
3.3.2 Design and implementation of the fault injection instrument 60
3.3.3 Design and implementation of the workload generator 61
3.3.4 Design and implementation of the controller 61
3.3.5 System integration 61
3.3.6 Instrument data parsing and analysis tools 62
3.3.7 Assessment of metrology properties 62
3.4 PHASE 3: execution and data collection 64
3.4.1 Execution of the next experiment 65
3.4.2 Online data processing and feedbacks 65
3.5 PHASE 4: analysis and recommendations 65
3.5.1 Data staging and analysis 65
3.5.2 Data comparison 67
3.5.3 Conclusions 68
3.6 Methodology applied to the case studies 69
4 Evaluation of a middleware service 73
4.1 The Reliable and Self-Aware Clock 74
4.1.1 Basic notions of time and clocks 74
4.1.2 Specification of R&SAClock 74
4.1.3 The Statistical Predictor and Safety Margin (SPS) 76
4.2 Planning and instrumentation 77
4.2.1 System under test 78
4.2.2 Workload, faultload and experiments design 78
4.2.3 Definition of the measuring instrument 79
4.2.4 Preparation of the structure of the results 83
4.3 Analysis of results 83
4.3.1 Effects of the synchronization period 85
4.3.2 Effects of the algorithm memory depth 86
4.3.3 Effects of the synchronization uncertainty confidence level 86
4.3.4 Summary of the results 87
4.3.5 Experiments in the presence of faults 88
4.3.6 Behavior under stress conditions 89

4.4 Conclusions and recommendations 89

5 Time measurements in distributed protocols 91

5.1 The Neko and NekoStat tool 91

5.2 Improvement of the measuring instrument NekoStat 94

5.3 Evaluation of a sample distributed protocol 95

5.3.1 Objective of the experiment 95

5.3.2 The target system 95

5.3.3 The experiment planning and the measuring instrument set-up 96

5.3.4 Execution and analysis of results 96

5.4 Conclusions and recommendations 98

6 Evaluation of dynamic SOAs 99

6.1 Environment definition 100

6.2 Architecture of the testing service 101

6.2.1 Execution and interactions of the testing service 103

6.3 Discovery and testing algorithms 104

6.3.1 SOA discovery and update algorithm 104

6.3.2 Test schedule algorithm 104

6.3.3 Implementation of the testing technique 105

6.4 Evaluation of the jSeduite SOA 105

6.5 Conclusions and recommendations 110

III The methodology in the industrial context and two case studies 111

7 Industrial prospectives and needs 113

7.1 Methodologies and tools supporting V&V 114

7.2 The RACME approach and architecture 116

7.2.1 The RACME workflow 116

7.2.2 A logic view of RACME 118

7.2.3 The RACME architecture and prototype 119

7.3 A testing methodology in V&V and certification processes 123

7.3.1 Interplay of the methodology and RACME 124

7.3.2 Methodology applied to industrial case studies 127

8 Evaluation of COTS GPS devices 129

8.1 Context, objectives and target system 130

8.1.1 The ALARP ATWS 130

8.1.2 Localization in a railway worksite 131

8.1.3 The target system 131

8.2 The measuring system and the experiments plan 131

8.2.1 The measuring system 132

8.2.2 Experiments description 132

8.2.3 Planning of the results 133

8.3 Analysis of results 133

8.4 Conclusions and recommendations 137
9 Evaluation of a railway embedded system 139
9.1 The SAFEDMI system 139
9.1.1 The DMI operational modes 140
9.1.2 DMI architecture and mechanisms 140
9.2 Planning of the measuring system and the experiments 141
9.3 Measuring system design, implementation and assessment 144
9.3.1 Design and implementation 144
9.3.2 Assessment of the measuring system and of the confidence of the results 146
9.4 Analysis of the results of the DMI 147
9.5 Conclusions and recommendations 150
Conclusions 151
BIBLIOGRAPHY 153

LIST OF FIGURES

Figure 1.1 Fault-error-failure chain [12]. 4
Figure 1.2 Elementary fault classes [12]. 5
Figure 1.3 The dependability and security tree [12]. 9
Figure 1.4 Target system a) as black box, and b) with probes. 11
Figure 1.5 A measuring system for the experimental evaluation of critical systems. 16
Figure 2.1 The RODS framework [30]. 28
Figure 3.1 Activity diagram of the methodology. 50
Figure 3.2 WBS view of the methodology. 52
Figure 3.3 PHASE 1: Planning and definition. 54
Figure 3.4 PHASE 2: Design and instrumentation. 59
Figure 3.5 PHASE 3: Execution and data collection. 64
Figure 3.6 PHASE 4: Analysis and recommendations. 66
Figure 3.7 Data staging and analysis. 66
Figure 4.1 Time and clock. 75
Figure 4.2 The measuring instrument and the system under test. 80
Figure 4.3 Controller, Client and R&SAClock interactions to execute the workload. 81
Figure 4.4 Time interval containing T_3. 82
Figure 4.5 Results presented by a star-schema. 83
Figure 4.6 Variable synchronization period. 85
Figure 4.7 Results for different values of the memory depth. 86
Figure 4.8 Results for different values of the coverage level. 87
Figure 4.9 Results of injection experiments. 88
Figure 5.1 The Neko architecture [179]. 92
Figure 5.2 Scheme of NekoStat functioning. 93
Figure 5.3 Instrumenting the system with the R&SAClock. 94
Figure 5.4 OWD measurement results. 97
Figure 5.5 OWD measurement results characterized by synchronization uncertainty lower than 8 ms. 98
Figure 6.1 A sample view of the testing service architecture. 102
Figure 6.2 The subset of the jSeduite SOA selected as case study. 106
Figure 6.3 Services discovered at Step 1 of the case study. 107
Figure 6.4 Services discovered at Step 2 of the case study. 108
Figure 6.5 Cached Feed Reader: a screenshot of the results. 109
Figure 7.1 The RACME workflow for a generic V&V process. 117
Figure 7.2 Logic view of RACME. 118
Figure 7.3 High level view of the RACME architecture. 120

Figure 7.4 The RACME prototype. 122
Figure 7.5 The experimental evaluation methodology and RACME. 126
Figure 8.1 The ALARP scenario. 130
Figure 8.2 The star-schema designed for the experiments. 134
Figure 8.3 Results of Exp_1. 135
Figure 8.4 Results of Exp_2. 136
Figure 8.5 Results of Exp_3. 136
Figure 8.6 R8, GS1, GS2 and G18 roving on an hybrid path (Exp4). 137
Figure 9.1 DMI, EVC Packet Generator and D360. 142
Figure 9.2 Structure of the data organized following a star schema. 144
Figure 9.3 The measuring system and its connections with the target system. 144
Figure 9.4 Results of Exp_1 and Exp_2. 148
Figure 9.5 Results of Exp_3 and Exp_4. 149
Figure 9.6 Results of Exp_5 and Exp_6. 149

Part I

OBJECTIVES, MOTIVATIONS AND STATE OF THE ART

1 CONTEXT OF THE RESEARCH

This Chapter presents an overview on basic notions that constitute the foundation of the research work presented in this Thesis. We start introducing in Section 1.1 basic concepts of dependability (and security). Then, Section 1.2 is devoted to introduce monitoring activities and experimental evaluation activities, together with areas where forms of monitoring and experimental evaluation are applied. Finally, we discuss in Section 1.3 fundamentals of measurement theory (metrology), a science that proposes standards and good practices for the assessment of measuring systems and results and that we believe of utmost importance in the experimental evaluation of critical systems.

1.1 DEPENDABILITY (AND SECURITY) CONCEPTS

When dealing with dependability and security of computing and communication systems, the reference taxonomy and definitions are those given in [12]: this work results from a previous work originated in 1980, when a joint committee on "Fundamental Concepts and Terminology" was formed by the TC on Fault-Tolerant Computing of the IEEE Computer Society and the IFIP (International Federation for Information Processing) WG (Working Group) 10.4 "Dependable Computing and Fault Tolerance" with the intent of merging the distinct but convergent paths of the dependability and security communities.

1.1.1 *System, service, faults, errors and failures*

The starting point is the definition of system, environment and system boundary.

Definition 1. *A* system *is an entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena. These other systems are the* environment *of the given system. The system* boundary *is the common frontier between the system and its environment.*

We present the definition of function of a system, behavior, total state and finally the definition of service and service interface.

Definition 2. *The* function *of a system is what the system is intended to do. The* behavior *of a system is what the system does to implement its function and is described by a sequence of states.*

Definition 3. *The* service *delivered by a system is its behavior as it is perceived by its user(s); a user is another system that receives provided service. The part of the provider's system boundary where service delivery takes place is the provider's* service interface*.*

We are using the singular for function and service, but a system can implement more than one function, and deliver more than one service.

Definition 4. Correct service *is delivered when the service implements the system function. A service* failure*, also abbreviated with failure, is an event that occurs when the service delivered deviates from correct service.*

The deviation from correct service may assume different forms that are called service failure modes and are ranked according to failure severities. The definition of errors and faults is as follows:

Definition 5. *The part of the system state that is liable to lead to subsequent failure is called an error.*

Definition 6. *The adjudged or hypothesized cause of an error is called a fault.*

A fault is *active* when it causes an error, otherwise it is *dormant*. A service failure occurs when an error is propagated to the service interface and causes the service delivered by the system to deviate from correct service. Service failure of a system causes a permanent or transient fault for the other system(s) that receive service from the given system.

This set of mechanisms constitutes the *fault-error-failure* chain of threats shown in Figure 1.1.

$$\begin{tikzcd}\cdots \arrow[r] & \text{fault} \arrow[r, "activation"] & \text{error} \arrow[r, "propagation"] & \text{failure} \arrow[r, "causation"] & \text{fault} \arrow[r] & \cdots\end{tikzcd}$$

Figure 1.1: Fault-error-failure chain [12].

When the functional specification of a system includes a set of several functions, the failure of one or more of the services implementing the functions may leave the system in a degraded mode that still offers a subset of needed services to the user.

1.1.2 *A taxonomy of faults, failures and errors*

We detail the discussion on faults, failures and errors [12]. We start with the presentation of the taxonomy for faults. All faults that may affect a system during its life are classified according to eight basic viewpoints, leading to the elementary fault classes, as shown in Figure 1.2.

If all combinations of the eight elementary fault classes were possible, there would be 256 different combined fault classes. However, not all criteria are applicable to all fault classes; for example, natural faults cannot be classified by

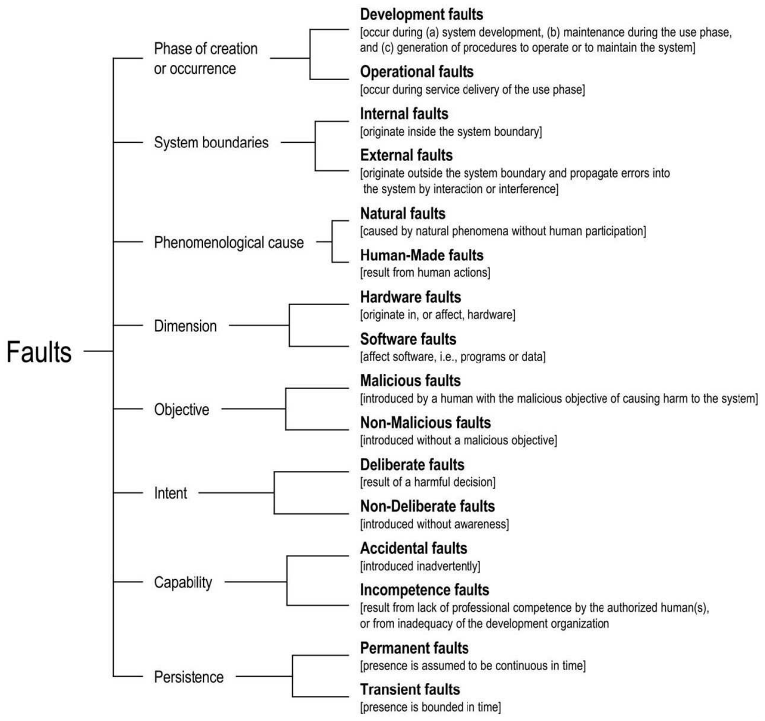

Faults
Phase of creation or occurrence
Development faults
[occur during (a) system development, (b) maintenance during the use phase, and (c) generation of procedures to operate or to maintain the system]
Operational faults
[occur during service delivery of the use phase]
System boundaries
Internal faults
[originate inside the system boundary]
External faults
[originate outside the system boundary and propagate errors into the system by interaction or interference]
Phenomenological cause
Natural faults
[caused by natural phenomena without human participation]
Human-Made faults
[result from human actions]
Dimension
Hardware faults
[originate in, or affect, hardware]
Software faults
[affect software, i.e., programs or data]
Objective
Malicious faults
[introduced by a human with the malicious objective of causing harm to the system]
Non-Malicious faults
[introduced without a malicious objective]
Intent
Deliberate faults
[result of a harmful decision]
Non-Deliberate faults
[introduced without awareness]
Capability
Accidental faults
[introduced inadvertently]
Incompetence faults
[result from lack of professional competence by the authorized human(s), or from inadequacy of the development organization
Persistence
Permanent faults
[presence is assumed to be continuous in time]
Transient faults
[presence is bounded in time]

Figure 1.2: Elementary fault classes [12].

objective, intent, and capability. Knowledge of all possible fault classes allows the user to decide which classes should be included in a dependability and security specification.

The combined fault classes shown in Figure 1.2 belong to three major partially overlapping groupings:

1. *development faults*, that include all fault classes occurring during development;

2. *physical faults*, that include all fault classes that affect hardware;

3. *interaction faults*, that include all external faults.

We move now to describe a taxonomy of failures. The different ways in which the deviation of a failed service is manifested are its service failure modes. Each mode can have more service failure severities. The service failure modes characterize incorrect service according to four view-points:

1. the failure domain;

2. the detectability of failures;

3. the consistency of failures;

4. the consequences of failures on the environment.

The *failure domain viewpoint* leads us to distinguish between content (or value) failures (the content of the information delivered at the service interface deviates from implementing the system function) and timing failures (the time of arrival or the duration of the information delivered at the service interface i.e., the timing of service delivery, deviates from implementing the system function). These definitions can be specialized. In fact, the content can be in numerical or non-numerical sets (e.g., alphabets, graphics, colors, sounds), and a timing failure may be early or late, depending on whether the service is delivered too early or too late. Failures when both information and timing are incorrect fall into two classes:

1. *halt failure*, or simply halt, when the service is halted (system activity, if there is any, is no longer perceptible to the users); a special case of halt is *silent failure*, or simply silence, when no service at all is delivered at the service interface.

2. *erratic failures* otherwise, i.e., when a service is delivered (not halted), but is erratic (e.g., babbling).

The *detectability viewpoint* addresses the signaling of service failures to the user(s). Signaling at the service interface originates from detecting mechanisms in the system that check the correctness of the delivered service. When the losses are detected and signaled by a warning signal, then signaled failures occur. Otherwise, they are unsignaled failures. The detecting mechanisms themselves have two failure modes: 1) signaling a loss of function when no failure has actually occurred, that is a false alarm, and 2) not signaling a function loss, that is an unsignaled failure. When the occurrence of service failures result in reduced modes of service, the system signals a degraded mode of service to the user(s). Degraded modes may range from minor reductions to emergency service and safe shutdown.

The *consistency* of failures leads us to distinguish:

- consistent failures: the incorrect service is perceived identically by all system users;

- inconsistent failures: some or all system users perceive differently incorrect service (some users may actually perceive correct service); inconsistent failures are usually called Byzantine failures [112].

Grading the *consequences* of the failures upon the system environment enables failure severities to be defined. The failure modes are ordered into severity levels, to which maximum acceptable probabilities of occurrence are generally associated. The number, the labeling, the definition of the severity levels, and the acceptable probabilities of occurrence are application-related, and involve the dependability and security attributes for the considered application(s).

Generally speaking, two limiting levels can be defined according to the relation between the benefit (in the broad sense of the term, not limited to economic considerations) provided by the service delivered in the absence of failure, and the consequences of failures:

- *minor failures*, where the harmful consequences are of similar cost to the benefits provided by correct service delivery;
- *catastrophic failures*, where the cost of harmful consequences is orders of magnitude, or even incommensurably, higher than the benefit provided by correct service delivery.

Systems that are designed and implemented so that they fail only in specific modes of failure described in the dependability and security specification and only to an acceptable extent are *fail-controlled* systems. Instead a system whose failures are to an acceptable extent halting failures only is a *fail-stop* system.

Finally, after presenting the taxonomy of faults and failures, we briefly discuss here the errors.

A convenient classification of errors is to describe them in terms of the elementary service failures that they cause, reusing most of the terminology previously presented. Consequently, we can identify content versus timing errors, detected versus latent errors, consistent versus inconsistent errors, minor versus catastrophic errors.

Finally, some faults (e.g., a burst of electromagnetic radiation) can simultaneously cause errors in more than one component. Such errors are called multiple related errors; on the contrary, single errors are errors that affect one component only.

1.1.3 *Dependability and its attributes*

We have defined what is a system, its correct service and the threats which can affect the service. All elements are now set to introduce the definition of *dependability*. The original definition of dependability is as follows:

Definition 7. Dependability *is the ability of a system to deliver a service that can justifiably be trusted.*

This definition stresses the need for justification of trust. An alternate definition, that provides the criterion for deciding if the provided service is dependable, is:

Definition 8. Dependability *is the ability of a system to avoid service failures that are more frequent and more severe than is acceptable.*

Dependability is an integrating concept, which encompasses the following attributes:

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

When addressing security, an additional attribute needs to be considered: *confidentiality* i.e., the absence of unauthorized disclosure of information. Security is a composite of the attributes of confidentiality, integrity, and availability, requiring the concurrent existence of:

1. availability for authorized actions only,
2. confidentiality,
3. integrity with "improper" meaning "unauthorized".

The dependability and security specification of a system must include the requirements for the attributes in terms of the acceptable frequency and severity of service failures for specified classes of faults and use environment. One or more attributes may not be required at all for a given system.

1.1.4 *Means to attain dependability*

Means to attain various attributes of dependability and security can be grouped into four categories:

1. *Fault prevention* means to prevent the occurrence or introduction of faults.
2. *Fault tolerance* means to avoid service failures in the presence of faults; systematic introduction of fault tolerance is often facilitated by the addition of support systems specialized for fault tolerance.
3. *Fault removal* means to reduce the number and severity of faults.
4. *Fault forecasting* means to estimate the present number, the future incidence, and the likely consequences of faults; it is conducted by performing a qualitative or quantitative evaluation of the system behavior with respect to faults occurrence or activation.

Fault prevention and fault tolerance aim to provide the ability to deliver a service that can be trusted, while fault removal and fault forecasting aim to reach confidence in that ability by justifying that the functional and the dependability and security specifications are adequate and that the system is likely to meet them.

The schema of the complete taxonomy of dependable and secure computing is finally reported in the Figure 1.3.

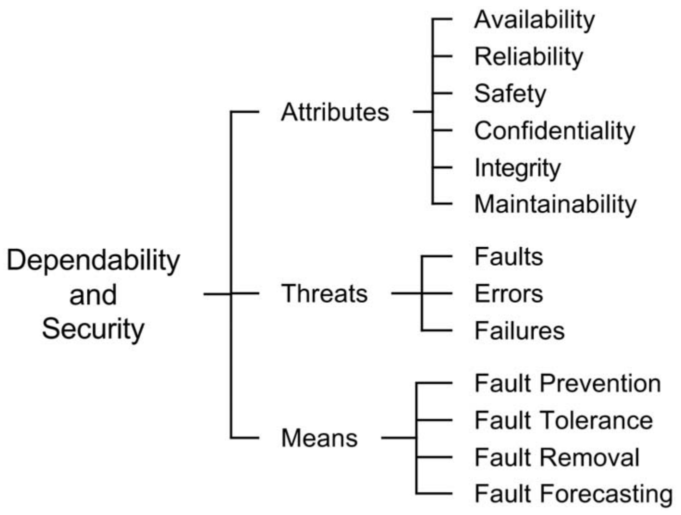


Figure 1.3: The dependability and security tree [12].

Since the context of this Thesis (experimental evaluation of computing systems) falls in the category of fault forecasting, we detail fault forecasting in the following.

Fault forecasting

Fault forecasting is conducted by performing an evaluation of the system behavior with respect to fault occurrence or activation. Evaluation has two aspects:

- qualitative, or ordinal, evaluation, that aims to identify, classify, and rank the failure modes, or the event combinations (component failures or environmental conditions) that would lead to system failures;
- quantitative, or probabilistic, evaluation, that aims to evaluate in terms of probabilities the extent to which some of the attributes are satisfied; those attributes are then viewed as measures.

The quantitative evaluation of performance and of dependability-related attributes is an important activity of fault forecasting, since it aims at probabilistically estimating the adequacy of a system with respect to the requirements given in its specification. Quantitative system assessment can be performed using several approaches, generally classified into three categories: *analytic*, *simulative* and *experimental* [12]. Each of these approaches shows different peculiarities, which determine the suitableness of the method for the analysis of a specific system aspect. The most appropriate method for quantitative assessment depends on the complexity of the system, the development stage of the system, the specific aspects to be studied, the attributes to be evaluated, the accuracy required, and the resources available for the study [164], [120].

Analytic and simulative approaches are generally efficient and timely, and they have proven to be useful and versatile in all the phases of the system life cycle. They are typically based on a parametric model of the analyzed system and on a set of assumptions concerning the behavior of the system and/or of the system environment.

Analytic approach is usually cheap for manufacturers. The accuracy of the results obtained through an analytic approach is strongly dependent on the accuracy of the values assigned to the model parameters and of the assumptions on which the model is based.

Similarly to the analytic approach, the accuracy of the evaluation obtained with the *simulative approach* depends on the accuracy of the assumptions made for the system, on the behavior of the simulation environment, and on the simulation parameters.

Experimental evaluation is an attractive option for quantitative assessment of an existing system or prototype. A very high level of interest is paid to the quantitative evaluation based on measurements, with special attention to evaluation of Quality of Service (QoS, [91]) metrics of systems and infrastructures. Experimental evaluation allows to monitor the real execution of a system to obtain highly accurate measurements of the metrics of interest. However, it may turn out to be quite expensive e.g., when the interest is in very rare events, and the obtained results are often difficult to generalize. In this case, appropriate techniques based on active measurements and controlled experiments can be adopted. In the experimental measurement-based approach, the required measures are estimated from measured data using statistical inference techniques, and the data are measured from a real system or from its prototype. It is usually an expensive approach, since it requires building a real system, performing the measurements and statistically analyzing the data.

1.2 BASICS ON MONITORING AND EXPERIMENTAL EVALUATION

We present basics on monitoring and experimental evaluation of critical systems. The objective of this Section is to present an exhaustive introduction of experimental evaluation, which is at the base of this Thesis. We note that a discussion on experimental evaluation, in order to be complete, should not disregard foundations on monitoring. That is why we first present basics and some significant application fields (identified mainly from [180]) for monitoring, and then for experimental evaluation.

1.2.1 *Monitoring*

Monitoring can be defined as the process of dynamic collection, interpretation and presentation of information concerning objects or software processes under scrutiny [126].

A general view of monitoring can be summarized as follows. The behavior of the system is observed and monitoring information is gathered; this information is used to take decisions and perform the appropriate control actions on the system. Monitoring involves observing the execution behavior or performance of a target application in order to gain an insight into the current operation of the system [186].

In general, when we deal with the monitoring of computing systems we have one or more monitored elements (e.g., computers, operating systems, application packages) send out signals or measurements of their well-being or distress to a monitoring site, where thus various forms of processed information can be extracted and decisions made.

In order to observe a target system, elements of a monitoring system, termed *probes*, are attached to the system to provide information about the system's internal operation. The probes provide an intermediate output from the system in addition to its end output, allowing the system to be seen as more than a *black box* (see Figure 1.4). Such probes can be actual hardware probes that monitor internal system signals; alternatively, some code to perform monitoring can be added to the target software in order to output information about the system. The added code can be regarded as *software probes* [186].

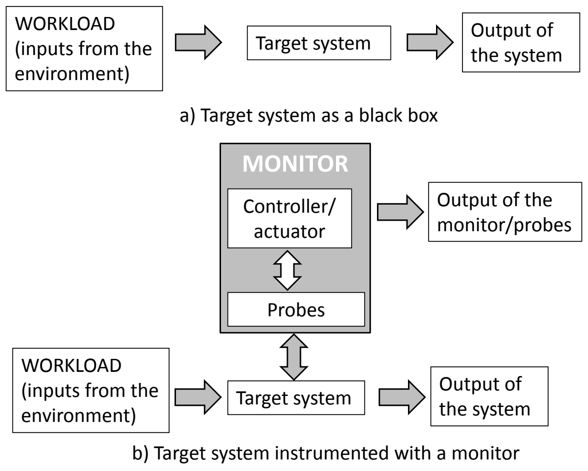

WORKLOAD (inputs from the environment)
Target system
Output of the system
a) Target system as a black box
MONITOR
Controller/ actuator
Probes
Output of the monitor/probes
WORKLOAD (inputs from the environment)
Target system
Output of the system
b) Target system instrumented with a monitor

Figure 1.4: Target system a) as black box, and b) with probes.

Monitoring may serve either or both of the following two purposes [180]:

- Supporting direct feedback control: the monitoring data trigger decisions about, for example, dispatching maintenance technicians, or switching in stand-by components.
- Providing data for assessing the dependability or resilience of the monitored system.

We continue our discussion presenting *forms of online monitoring* according roughly to areas of use: we present the use of monitoring in the diverse areas of network and QoS monitoring, intrusion detection and prevention systems, embedded system telemetry, monitoring of large-scale enterprise software, runtime verification, and automatic failure reporting.

Network and QoS monitoring

Network monitoring is part of network management functions. It includes means for the identification of problems caused by network failures and overloaded or by failed devices, to enable corrective management actions or to produce long term dependability measurements. Also, network monitoring has the purpose of alerting network administrators to virus or malware infections, questionable user activity and power outages [180].

Network monitoring systems typically use "network monitoring interface cards" (NMIC), that are similar to standard NICs, but able to passively listen on a network. NMICs have no MAC (Media Access Control) layer addresses and are invisible to other devices on the network. Network monitoring systems may listen to different application level protocols like for example FTP or SMTP. As for communicating data, many protocols can be used, but the standard is the Simple Network Management Protocol (SNMP, [35]). The protocol prescribes an architecture in which a software component (denoted as agent) executes on managed systems (network elements such as hosts, gateways, terminal servers, and the like), collecting the desired network information, that is in turn communicated to one or more managing systems.

Several works describe how monitoring can be used for an on-line control of the Quality of Service of a system, as surveyed in [91], [11]. The general model for approaching QoS monitoring problems can be summarized in the following functional components [91]:

1. *Monitoring application*. This component serves as an interface. Its functions include retrieving traffic information from monitors, analyzing such information and providing analysis results to users.
2. *QoS monitoring*. This module is an additional one compared with the model of traditional network monitoring systems. It provides mechanisms for

QoS monitoring, to retrieve information from relevant monitors and derive QoS-related parameters.

3. *Monitor*. This module gathers and records traffic information and communicates the information to the monitoring application. In particular, this module performs real-time measurement of real-time flows and reports the measured information to the monitoring application.

4. *Monitored objects*. These are the information such as attributes and activities that are to be monitored in the network.

Intrusion detection and intrusion prevention systems

Intrusion detection is the process of monitoring the events occurring in a computer system or network and analyzing them for signs of possible incidents, which are violations or imminent threats of violation of computer security policies, acceptable use policies, or standard security practices. Intrusion prevention is the process of performing intrusion detection and attempting to stop potential incidents detected [158].

Intrusion detection and prevention systems (IDPSs, [158]) are primarily focused on identifying possible incidents, logging information about them, attempting to stop them, and reporting them to security administrators. In addition, organizations use IDPSs for other purposes, such as identifying problems with security policies, documenting existing threats, and deterring individuals from violating security policies.

IDPSs are important mechanisms in the assessment of network security. Detecting both attempted and successful intrusions would of course allow measurements of security (measurements of both absolute frequency of successful intrusions and of the fraction of intrusions that are successful). However this potential is limited by the fact that the IDPSs themselves are inevitably imperfect, and exhibit false negatives (failing to raise an alarm following a real intrusion), false positives (raising an alarm due to legitimate traffic) or event misclassification (raising an alarm following a real intrusion, but failing to classify it correctly).

Embedded systems telemetry

The use of embedded systems has grown continuously for decades. The trend required advances in monitoring of these systems to support hardware and software maintenance, as well as dependability and performance assessment. Consequently, embedded system telemetry acquired increasing interest. Telemetry is a technology that allows remote measurement and reporting of information. This type of monitoring application is particularly valuable for embedded systems that cannot be suspended to examine variables values and programs flows.

For example, let us consider compressors used to pump natural gas through a pipeline: bringing an engine online is a very complex task and the variables have to be examined at runtime to validate the working of the control system. On the other hand, such an approach carries an overhead (with respect to performance and resource usage) due to additional computation needed for communication between the tool and the target embedded application [180].

As an example to illustrate current practices in telemetry of embedded systems, we mention the Web Network Management Protocol (WNMP, [187]) from Micro Digital Inc. WNMP aims to ease and simplify administration and monitoring of embedded systems using ordinary web browsers on a PC, mobile phone, etc. without requiring a special program running on the client.

Monitoring large-scale enterprise software

Today's enterprise software systems have severe availability requirements. Satisfying stringent dependability requirements is, however, hard - enterprise software is growing in size and complexity, and the necessary updates are becoming more frequent. Each component in a system requires monitoring of its resources, performance and state variables. This frequently leads to generating an overwhelming amount of data, which are hard to collect and store, and are complicated to analyze [134].

Despite the availability of large amounts of information regarding many aspect of modern software systems, effectively managing their operation is difficult. These systems can generate an overwhelming amount of information that can be costly to collect and difficult to handle and analyze [108]. Each software in a system can be monitored via numerous performance, activity, resource utilization, and state-related metrics. Sources include log files, trace files, event notifications, existing instrumentation and related interfaces, dynamic instrumentation, etc. [134].

An effort in this respect is autonomic computing [108] and automated adaptive monitoring, in which the monitoring system continuously assess current conditions by observing the most relevant data, to promptly detect anomalies and to help identifying root-causes of problems.

Runtime verification

Runtime verification techniques are means for runtime failure reporting that typically rely on "formal methods" for designing the functions that detect failures (and are thus not limited to crash failures). These techniques are motivated more with verifying and improving software than by measuring its dependability attributes.

In a nutshell, runtime verification works as follows. A correctness property ϕ, usually formulated in some linear temporal logic, such as LTL [150], is given and a so called monitor that accepts all models for ϕ is automatically generated.

The system under scrutiny as well as the generated monitor are then executed in parallel, such that the monitor observes the system's behavior. System behavior which violates property ϕ is then detected by the monitor and an alarm signal is returned [7].

In experimental evaluation, one approach is to generate a test monitor that checks whether the application works correctly. Then, the system under test is executed with typical inputs and it is observed whether the monitor complains. The monitor generation as used for runtime verification is applicable in the domain of testing as well.

Monitoring for software failures during software runtime confirms (or disproves) whether an actual run of the software conforms to the requirements specification, by checking whether it preserves certain formally specified properties.

Automatic failure reporting for software

Automatic Failure Reporting focuses on the problems of capturing failure data from customer sides. Commercial and open-source software increasingly employs automatic failure reporting features. In fact there are many ways to measure quality before and after the software is released; for commercial and internal-use-only products, the most important measurement is the user's perception of product quality. Unfortunately, perception is difficult to measure, so companies attempt to quantify it through customer satisfaction surveys and failure/behavioral data collected from its customer base [135].

Collecting failure data requires a monitoring process resident on the customer's computer that detects processes and transmits the data. For example in Windows XP this process is enabled by a dialog with the user as part of the installation process. Once a system administrator logs onto a system for the first time following a system reboot, the operating system automatically checks if the cause of the system outage was a system crash. If so, it processes the system dump file, generating a mini dump and an XML file containing versions of all drivers on the system. These data are subsequently compressed. A prompt then appears on the screen requesting the user's permission to send these data to Microsoft. If the user agrees, the data are sent via http *post*. This method also allows the process to send a response back to the user by redirecting the http request to a Web page containing possible solutions [135].

1.2.2 *Experimental evaluation*

Observations of a system can be done with purposes of *experimental evaluation* (or *testing*) for V&V. We define in Figure 1.5 a typical measuring system for testing purposes. The figure is based mainly on [81], but this architecture is general and compatible with the architectures of most testing tools and frameworks (similarities can be found in several works, to mention a few [2], [33], [34], [44], [92], [100], [104], [105], [114], [154], [183], [189], [190]).

Most of the components shown in the figure and many relations amongst them are optional, and depend on the objectives of the evaluation. We place no restrictions on the implementation of the components (they can be hardware or software components, located on the target system or on a different system, etc.) and on the number of instances of each component (depending on the target system and on the objectives of the testing, one or more instances of a component may be necessary).

We detail the components shown in Figure 1.5, starting from the *target system* (or *system under test*). The target system is the subject of the experimental evaluation; it can be a centralized or a distributed system, a mechanism, an application, a set of functions, an hardware component, etc.

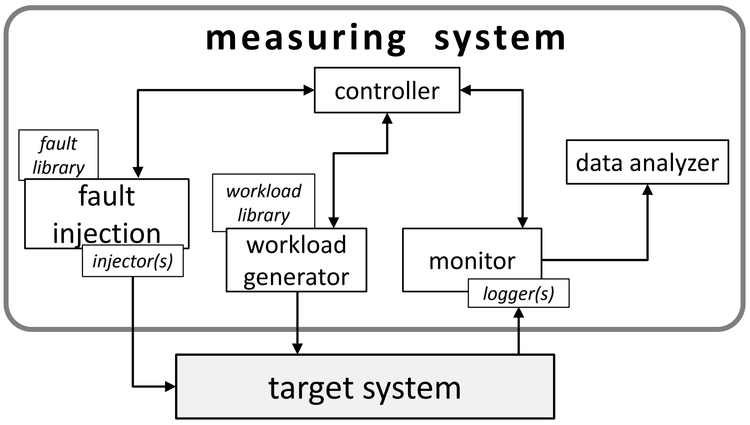


Figure 1.5: A measuring system for the experimental evaluation of critical systems.

Instead, the *measuring system* (or *measuring instrument*) is the tool or instrument used for the experimental evaluation. It can be located entirely on the target system, or (partially) placed on a different system. Contact points with the target system are needed.

The measuring system is composed of the controller, the monitor, the fault injector, the logger or data collector, the data analyzer, the workload generator, the fault library and the workload library, detailed in what follows.

1. The *controller* controls the experiment. For example, it can be a program that can run on the target system or on a separate computer.

2. The *monitor* tracks the execution of the commands through its probes and initiates data collection whenever necessary. The *logger* collects the monitored data.

3. The *workload generator* generates the set of instructions (workload) that the target system will execute during the tests (can be applications, bench-

marks, or synthetic workloads). It reads the workload from a *workload library*.

4. The *fault injection* instrument (can also be an instrument for error/failure injection, depending on the characteristics of the experiments) is required only in case of injection experiments. The injectors perform injections into the target system as it executes commands from the workload generator. The fault injector can be custom-built hardware or software. The fault injector itself can support different fault types, fault locations, and fault times, the values of which are drawn from a *fault library*.

5. The *data analyzer*, which can execute offline, performs data staging and analysis.

As we did for monitoring, in what follows we survey main fields and related techniques for experimental evaluation. We present and discuss the major approaches and techniques that can be considered for obtaining experimental measures relevant for dependability analysis, via:

1. collecting the erroneous behaviors and failure data via measurements carried out in the field;

2. conducting controlled experiments where the target system is explicitly subjected to the application of artificial inputs.

The areas surveyed are not comprehensive of all the possible testing activities, but are particularly significant to introduce the contribution of this Thesis. Fault injection, robustness testing and field measurements are techniques applied in the case studies shown in the following of this Thesis. Also, dependability benchmark is a significant approach to improve comparison of critical system, and although not applied in the case studies of this Thesis it is strictly correlated to its intention and topics. Extensive discussions on experimental evaluation approaches can be found in [180], [9], [170], [120].

Fault injection

Fault injection is an important experimental technique for the assessment and verification of fault-handling mechanisms, as it permits to analyze a system's response to exceptional conditions. Fault injection allows to study how computer systems react and behave in the presence of faults; it can serve different purposes, such as assess the effectiveness of software- and hardware-implemented fault-handling mechanisms, study the error propagation and error latency, verify failure mode assumptions. However, fault injection is suitable for studying emulated faults only, and typically requires a degree of knowledge on the inner parts of the component, and the possibility to operate with them (e.g., adding software handle). It also fails to provide dependability measures such as mean time between failures and availability [81].

Faults are injected either at the hardware level (logical or electrical faults) or at the software level (code or data corruption) and the effects are monitored, considering the system as a white-box. Choosing between hardware and software fault injection depends on the type of faults we are interested in and the effort required to create them. Hardware-implemented fault injection uses additional hardware to introduce faults into the target system's hardware. Depending on the faults and their locations, hardware-implemented fault injection methods fall into two categories [81]:

- *hardware fault injection with contact*. The injector has direct physical contact with the target system, producing voltage or current changes externally to the target chip. Examples are methods that use pin-level probes and sockets.
- *hardware fault injection without contact*. The injector has no direct physical contact with the target system. Instead, an external source produces some physical phenomenon, such as heavy ion radiation and electromagnetic interference, causing spurious currents inside the target chip.

To perform fault injection in software, there are two fundamental approaches:

- *injection of faults that imitate mistakes of programmers*, which is typically done by changing the code executed by the target system;
- *injection of software errors*, to emulate the consequences of software faults by manipulating the state of the target system.

In the following of this Thesis, we will apply fault injection principles in the case study shown in Chapter 4 (applied to a middleware service), Chapter 5 (related to a measuring instrument for distributed measurements) and Chapter 9 (applied to an embedded system).

Robustness testing

Robustness measures the behavior of the system under non-standard conditions. Robustness is defined in the IEEE Standard $610.12-1990$ as the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions [85].

The goal of *robustness testing* is to activate those faults (typically design or programming faults) or vulnerabilities in the system that result in incorrect operation i.e., robustness failure. Robustness tests stimulate at black-box the target system in a way that triggers internal errors, and in that way exposes both programming and design errors [183].

Robustness failures are typically classified using the CRASH criteria [111]:

- *Catastrophic*: the whole system crashes or reboots.
- *Restart*: the application has to be restarted.
- *Abort*: the application terminates abnormally.
- *Silent*: invalid operation is performed without error signal.
- *Hindering*: incorrect error code is returned (note that returning a proper error code is considered as robust operation).

The measure of robustness can be given as the ratio of test cases that exposes such faults, or, from the system point of view, as the number of robustness faults exposed by a given test suite.

In the case study shown in Chapter 6, in the context of testing of Service-Oriented Architecture, a tool for robustness testing of Web Service shall be used.

Field measurements

Measuring a real-life system consists in recording naturally occurring errors and failures in the system while it is running under typical user workload. Analysis of such field data can provide valuable information on actual errors or failures behavior, quantify dependability measures, and identify system bottlenecks.

Although field data (field measurements) are highly relevant to the research community to understand and improve computer-based systems robustness and reliability, the availability of such data remain hard to obtain. Often, the few data available are based on open-source projects and published research works [48]. One important initiative to mitigate the scarcity and fragmented view of field data is the development of public repositories, to store data and results based on that data originating from many sources and teams [184]. In this direction, several initiatives have been proposed and are currently available; we delegate such discussion to Section 2.4. We just mention here that despite some initiatives on public repositories are available, the raw data from the vast majority of research works on experimental dependability evaluation and on field failure data are not available.

This Thesis devotes great attention to sharing and storing of data, which will be explored in the methodology described in Chapter 3. In the case study of Chapter 8 we report an example involving COTS GPS (Global Positioning System, [106]) devices, where field data are collected in a real-case scenarios under typical usage conditions.

Dependability benchmarking

The goal of *benchmarking the dependability* of computer systems [103], [53] is to provide generic ways for characterizing their behavior in the presence of faults.

Benchmarking must provide a uniform and repeatable way for dependability characterization. A benchmark must be as representative as possible of a domain. The objective is to find a representation that captures the essential elements of the domain and provides practical ways to characterize the computer dependability to help system manufacturers and integrators improving their products and end-users in their purchase decisions [123].

Various results and initiatives are present on dependability benchmarking as the IFIP WG 10.4 SIG (Special Interest Group) on Dependability Benchmarking [1] and the EU project IST-2000-25425 DBENCH (Dependability Benchmarking) [103]. In this Thesis we do not explicitly address benchmarking, but we address comparison and experiments repeatability, which are important aspects related to benchmarking. In the case studies presented in this Thesis (especially Chapter 4 and Chapter 8), analysis of data structures and measuring instruments shall allow to guarantee experiments repeatability and systems comparison.

1.3 FOUNDATIONS OF METROLOGY

Metrology (measurement theory, [96]) has developed standards, theories and good practices to make measurements, to evaluate measurements results, and to characterize measuring instruments. We present fundamental concepts to characterize measurement systems and methods according to metrological criteria. Complete digests of metrological terms and concepts can be found in [96], [95], [94] and [97].

1.3.1 *Basic concepts*

Measuring a quantity (namely the *measurand*) consists in quantitatively characterizing it. The procedure adopted to associate quantitative information to the measurand is called *measurement*. The measurement result is expressed in terms of a measured quantity value and a related measurement uncertainty.

Accuracy is a concept that is often badly used; in metrology it must be intended only in a qualitative way. It was formerly defined as the difference between the measure and the true value of the measurand. As it is now commonly accepted that the true value of the measurand can not be exactly known, the *qualitative* concept of accuracy represents closeness of the measure to the best available estimate of the measurand value.

Considering quantitative concepts, the *measurement error*, or error of measurement, is the measured quantity value minus a reference quantity value (used as a basis for comparison with values of quantities of the same kind).

Uncertainty provides *quantitative* information on the dispersion of the quantity values that could be reasonably attributed to the measurand. Uncertainty has to be included as part of the measurement result and represents an estimate of the degree of knowledge of the measurand. It has to be evaluated according to conventional procedures, and is usually expressed in terms of a confidence interval, that is a range of values where the measurand value is likely to fall. The probability that the measurand value falls inside the confidence interval is named confidence level. Uncertainty can also be expressed in terms of *relative uncertainty*, which is the ratio of uncertainty to the absolute value of the estimate of the measurand. *Indirect measurements* are instead performed when the measurand value is not measured directly, but it is rather determined from direct measurements of other quantities.

Uncertainly of indirect measures can be obtained in principle following several ways. To give answer to the need for a univocal way of evaluating uncertainty, which offers the opportunity of comparing results from different methods and instruments, the Guide to the expression of Uncertainty in Measurements (GUM, [95]) has been published in 1993, and amended in 1995, after years of discussions within, but not limited to, the scientific community. Actually, since then, some supplements to the GUM are being discussed, and some questions are still open (e.g. alternative ways of evaluating uncertainty in indirect measurements in particular cases) [94], [97].

According to the GUM, *standard uncertainty*, usually indicated as u, that is uncertainty expressed as a standard deviation, can be evaluated in two ways:

1. statistically, as an estimate of the standard deviation of the mean of a set of independent observations (*Type A uncertainty*).
2. on the basis of a scientific judgment using all the relevant information available, which can include previous measurement data, knowledge of the behavior and property of relevant materials and instruments, manufacturer's specifications, data provided in calibration and other reports, and uncertainties assigned to reference data taken from handbooks (*Type B uncertainty*). This second way of evaluating uncertainty can be as reliable as the first, especially when the independent observations are very few.

Uncertainty evaluation becomes more critical for measurand estimated through indirect measurements. Specifically, let Y be the measurand, which is determined through N other quantities $X_1, X_2, ..., X_N$, according to the functional relation

$$Y = f(X_1, X_2, ..., X_N) \tag{1.1}$$

which is also called *measurement equation*. An estimate of Y, denoted by y, is achieved from equation 1.1 using input estimates $x_1, x_2, ..., x_N$ for the values of the N input quantities $X_1, X_2, \ldots, X_N$, that is

$$y = f(x_1, x_2, ..., x_N) \tag{1.2}$$

First of all, the *standard uncertainty* (i.e., the uncertainty expressed as a standard deviation) $u(x_i)$ of each estimate x_i $(i = 1, ..N)$ involved in the measurement function has to be evaluated. Then, we have to compose such standard uncertainty to obtain the combined standard uncertainty $u_c(y)$; according to [95]:

$$u_c(y) = \sqrt{\sum_{i=1}^{N} \left(\frac{\partial f}{\partial x_i}\right)^2 u^2(x_i)} \tag{1.3}$$

where the partial derivatives $\frac{\partial f}{\partial x_i}$ are equal to $\frac{\partial f}{\partial X_i}$ evaluated in $X_i = x_i$. Equation 1.3, referred to as the law of propagation of uncertainty, is based on a first order Taylor's approximation of equation 1.1. Actually, equation 1.3 is the simplified form to be used when the estimates x_i can be assumed to be not correlated. Otherwise, a further sum involving the estimated covariances associated with each pair (x_i, x_j) is needed under the square root in equation 1.3.

Singling out the most significant quantities of influence is important when we have to characterize a measurement system. Those are quantities that are not object of the measurement, but whose variation determines a modification in the relationship between measurand and instrument's output. Their presence can significantly degrade the measure, as they can represent a major cause of uncertainty. With regard to this, *selectivity* of a measurement system corresponds to its insensitiveness to quantities of influence. In other words, the less variable the output of a measurement system is due to the variability of the quantities of influence, the more selective is the system.

Resolution is the ability of a measuring system to resolve among different states of a measurand. It is the smallest variation of the measurand that can be appreciated i.e., that determines a perceptible variation of the instrument's output.

Repeatability is the property of a measuring system to provide closely similar indications in the short period, for replicated measurements performed i) independently on the same measurand through the same measurement procedure, ii) by the same operator, and iii) in the same place and environmental conditions.

Stability is defined as the property of a measuring system to provide closely similar indications in a defined time period, for measurements performed independently on the same measurand through the same measurement procedure and under the same conditions for the quantities of influence.

To characterize a measurement system, and draw a comprehensive comparison with alternative systems, some other indicators should also be taken into account, such as measuring interval, measurement time, intrusiveness and compatibility.

The *measuring interval* of a measurement system is the range of values of the measurand for which the measurement system is applicable with specified measurement uncertainty under defined conditions.

The importance of taking into consideration *measurement time* for a complete characterization is intuitive: besides being directly linked to measurement costs (in terms of resources occupation), the reciprocal of the measurement time gives an upper bound to the number of measurements that can be performed in the unit of time.

It is well known that any measurement system perturbs the measurand, determining a modification of its value. Minimizing such perturbation, that is minimizing the system's *intrusiveness*, is therefore desirable when designing a measurement system.

Finally, as measurement results are expressed in terms of ranges of values, intervals measured through different instruments ought to be compared rather than single values. Specifically, if results are expressed with the same confidence level, they are said to be *compatible* if the related intervals overlap.

1.3.2 *Uncertainty in measuring computing systems*

We complete this discussion on metrology and uncertainty presenting a classification of main measurements that can be performed on computing systems. We identify two classes of measurements: measurements with *negligible uncertainty* and measurements with *non-negligible uncertainty* [26].

The first class includes static quantities which depend on the static characteristics of the system (e.g., software quality measurements as number of source code lines), as well as countable dynamic quantities which depend on a particular execution of the system (e.g., number of packets re-transmissions, or number of queuing operations). Measurements belonging to this class are typically characterized by very low (negligible) uncertainty.

The second class is identified by measurements with non-negligible uncertainty, which generally refer to the dynamic behavior of the system and involve the estimation of continuous quantities. Few examples are: end-to-end communication delays, quality of clock synchronization, Mean Time To Failure, Mean Time Between Failures. It is easy to note that this latter class includes quantities whose measurement presents more challenges than those belonging to the former one. Looking closer at this class, we identify the crucial role of *time measurements*, that are typically the most critical ones to face when designing and testing systems and services. However we acknowledge that the analysis should not restrict only to time measurements: depending on the kind of system and service, different kind of measurements which suffer of non-negligible uncertainty may be involved and influence critical aspects of the system.

For the similar reasons, also relevant amongst measurements with non-negligible uncertainty are *spatial measurements* (the uncertainty in spatial measurements is a critical issue in algorithms for reliable localization and tracking) and *climate measurements*.

2 THE NEED OF METHODOLOGIES FOR THE EXPERIMENTAL EVALUATION

This Chapter focuses on identifying current state of the art and related challenges on methodologies for the experimental evaluation of critical systems. Additionally, this Chapter identifies the role targeted by the assessment of instruments and results according to principles of measurement theory, and it describes state of the art approaches to trustworthy data comparison and results sharing. The objective is to understand the current status and progress of the experimental evaluation of critical system and to identify gaps which offer room for improvements. In fact, the methodology that will be proposed in Chapter 3 derives from challenges and observations identified in the course of this Chapter.

The remaining part of this Chapter is organized as follows. In Section 2.1 we present main challenges for the monitoring and experimental evaluations, and in Section 2.2 we describe the state of the art on methodologies for the experimental evaluation of systems, with a particular focus on critical systems. Then in Section 2.3 we show tools and practical case studies for the evaluation of dependable systems taken from the literature, reviewed at the light of metrology concepts and rules, and in Section 2.4 we illustrate alternatives for the sharing and analysis of experimental results.

2.1 CHALLENGES IN MONITORING AND TESTING

The following is a discussion on the typical challenges encountered during experimental evaluation activities in centralized and distributed systems. First it explains the typical concerns that are raised when monitoring systems (monitors are typically included in a measuring system for testing purposes: consequently, the challenges here presented also apply to experimental evaluation), and then it explains concerns that are specific to experimental evaluation.

We start showing typical challenges that are raised when monitoring systems. The highlighted challenges are [77], [98], [127]:

- *Direct and indirect observations.* The behavior of some systems can be directly observed, thereby making the process of monitoring relatively straight forward. In computer systems most events of interest cannot be observed directly without special facilities, thereby requiring the incorporation of a monitoring infrastructure.

- *Complete and incomplete observations*, referring to whether the information necessary in order to construct a particular model of an observed system is available or not. Incompleteness can cause problems when it is not intended or not catered for; e.g., when information cannot be obtained thereby hiding certain aspects of the system from the observer.

- *Presentation problems.* In many cases it is necessary to modify the information from the observed events in a system to overcome problems due to i) events which appear in a form not amenable for immediate use by the observer, ii) events occurring at a rate which cannot be easily used by the observer, iii) overwhelming volume of observed events, and (in distributed systems) iv) necessity of structures and processes which collect and order the information from the observed events.

- *Monitor intrusiveness (perturbation).* Every system is affected by being monitored. The extent of the perturbation caused by the monitor may be not negligible. There is typically a relation between the flexibility of the monitoring facilities, the cost of the implementation, and the intrusiveness of the monitor.

- Monitoring in distributed system raises specific challenges: i) no central point of control (requires to consider more "threads of control"), ii) no central set of observation (requires the collection of events observed locally in order to construct global views), iii) no central source of monitoring information (source of the monitoring information is not a single source, and events will not necessarily be directly sent to the user or to a local file), iv) no central point of decision making (the process of making decisions in a distributed system may itself be distributed, having more than one manager in a monitoring session), v) incomplete observability (it is not possible to observe certain parts of the system at all or only partially), vi) encapsulation and security (ensuring that the state of objects and their associated procedures are protected from external observation and interference creates a conflict with monitoring needs e.g., the usage of monitoring facilities may clash with security requirements), and finally vii) huge amount of data to be processed, possibly in real-time, to perform short-term management or filter the logs.

In addition to the previous challenges, experimental evaluation of systems (especially for quantitative evaluation) needs to take account of the following challenges [120]:

- *Variety of threats.* A variety of threats and possible faults characterize modern systems, as the fault model is becoming more and more complex.

- *Costs.* The experimental measurement-based approach is usually expensive, since it requires building a real system (or a prototype), performing the measurements and analyzing the data statistically. For this reason often only a subset of the whole system is analyzed.

- *Intrusiveness of the measuring instrument*. In the case of controlled experiments aimed at active measurements (e.g., fault injection experiments), the various components of the testing tool may interfere with the target system. Intrusiveness relates to both probes and injectors. It is evident that such probes should be non intrusive; rather, the target system should evolve as in absence of the measuring system.

- *Semantic gap*. Frequently there is a semantic gap between the data collected during experimental evaluation and the relevant measures of interests to the user (the evaluator). Often low-level data (e.g., memory accesses of a processor) are collected, that shall be translated to a higher level quality of service characteristics (e.g., expected service delay) or dependability measures. In the field of fault injection experiments, a related issue is the representativeness of the data affecting the input domain, like the workload and the faultload.

- *Non-determinism (affects repeatability)*. Distributed, asynchronous systems are inherently non-deterministic. Thus, two executions of the same experiment may produce different, but nevertheless valid, ordering of events. In field measurement or controlled experiments it may be difficult to reproduce results of experiments. For example, we can consider dynamic and adaptive systems where despite the geographic mobility of the nodes is very carefully described and followed during the course of the experiments, the results of the experiments may differ since the propagation conditions are strongly influenced by changing environment conditions.

2.2 METHODOLOGIES FOR EXPERIMENTAL EVALUATION

In the following of this Section we survey methodologies for the experimental evaluation of (critical) systems.

The analysis presented in this Section includes general methodologies, tools and techniques ([62], [30], [73], [10], [8], [131]), notions from the field of software quality metrics ([86], [101], [66]) which also devoted a high level of attention on collection and analysis of relevant information, and finally standards for testing and testability ([87], [89], [163]).

2.2.1 *RODS*

In [62], [30] the conceptual framework RODS (Rigorous Observation of Distributed Systems) for the experimental evaluation and monitoring of distributed systems is presented. The design of RODS is made using concepts from metrology science. To support experimental evaluation and monitoring of heterogeneous systems, the framework proposed is composed of a sequence of steps and a set of tools that can be used for the evaluation and monitoring of several kinds of nodes composing a distributed system.

Figure 2.1 describes the process of analysis of a distributed system using the RODS methodology: the circles represent functions, the boxes represent input/output.

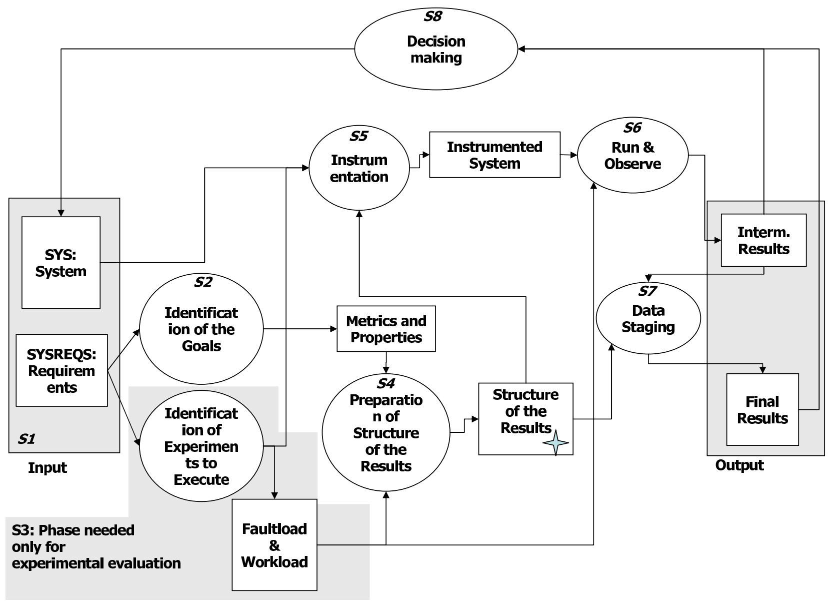


Figure 2.1: The RODS framework [30].

2.2.2 *The NetLogger methodology*

NetLogger is both a methodology for analyzing distributed systems and a set of tools to help implement the methodology. The main concept of the NetLogger methodology for experimental evaluation can be summarized as follows [73]:

1. All components must be instrumented to produce monitoring: application software, middleware, operating system and networks. The more components are instrumented, the better is.

2. All monitored events must use a common format and a common set of attributes. Monitored events must also contain a precision timestamp which is in a single timezone and globally synchronized via a clock synchronization method.

3. Log all interesting events, for example entering and exiting any program or software component, or begin/end of I/O operations.

4. Collect all log data in a central location.

5. Use event correlation and visualisation tools to analyze the monitored events logs.

2.2.3 *Fault injection methodologies*

Several tools and related methodologies for fault injection have been proposed through years. In general, key principles are the definition of the system under test, the workload, the faultload, injectors and probes. Amongst the many works available, we survey three which devote a particular care to the methodology; these are [10], [8] and [131]. The first two works explicitly focus on fault injection methodologies, and the third is a representative example of the adaptation of a basic methodology to a specific area.

The paper [10] discusses possible approaches to capture experimental data and record them for exploitation in a repository, so that the research community is able to exploit the recorded data. The guidelines to fault injection experiments are organized in three steps:

- *Planning the experiments*. Two different ways to plan a campaign (i.e., the injection of a series of faults) can be distinguished. In the first, a campaign is made up of a series of independent experiments, in which a fault pattern is injected and the target system is observed during a specific timeframe. In such case, it is necessary to check the target system for possible residual errors. In the second case, a series of faults are injected during the execution of the workload. However, in that case the end of each experiment is characterized by the occurrence of a specific failure event (e.g., a crash).

- *Collecting the outcomes*. During the conduction of the experiments, much attention is required to the target system and its operational profile, to record data characterizing the target system itself, the production process and the environment.

- *Archiving the experimental data*. Specific care needs to be taken in managing and storing the outcomes of fault injection experiments. A popular format for recording and analyzing data sets corresponds to the use of spreadsheet tables. This phase may benefit of the support of databases.

In [8], a methodology for fault injection is proposed. According to the authors, fault injection experiments correspond to a form of *test campaign*, characterized by an *input* domain and an *output* domain. The input domain corresponds to a set of injected *faults* F and a set A that specifies the *activity* (workload) of the target system and thus, the activation profile for the injected faults. The output domain corresponds to a set of *readouts* R that are collected to characterize the target system behavior in the presence of faults and a set of *measures* M that are derived from the analysis and processing of the FAR sets. Together, the FARM sets constitute the major attributes that can be used to fully characterize a fault injection test sequence. A fault injection test sequence consists of a series of *experiments*; each experiment specifying a particular point in the $\{F \times A \times R\}$ space. In practice, it is often the case that the definition of M has an impact on the selection of the other attributes.

During each experiment in a fault injection test campaign, a fault from the F set is injected that, in conjunction with the activity of the target system, determines an error pattern. For increased confidence in the estimates obtained, it is necessary to carry out a large number of experiments; for minimum bias in the estimation, it is further recommended to select both the F and A sets by *statistical sampling* among the expected operational fault and activation domains of the target system [8].

In [131], an approach is proposed to evaluate the potential risk of using a given (OTS) software component in a larger system. The approach to experimentally measure the risk of reusing a given component C in a system S is represented by the equation:

$$\mathrm{Risk}_c = \mathtt{prob}(f_c) \cdot \mathtt{cost}(f_c).$$

The term $\mathtt{prob}(f_c)$ represents the likelihood of the existence of residual software faults in the component C, estimated through well-established software complexity metrics, and the term $\mathtt{cost}(f_c)$ represents the impact of the activation of faults in the component C measured by software fault injection.

The proposal is based on three key elements [131]:

- The estimation of $\mathtt{prob}(f)$ by using complexity metrics of the target component, following a model based on logistic regression [80].
- The experimental evaluation of $\mathtt{cost}(f)$ through the injection of software faults in the target component and measuring its impact in the system under analysis. The fault activation probability is evaluated during the fault injection experiments.
- The use of a real workload and operational profile during the fault injection experiments. The main intended use of the real workload is the comparison of components for integration in a system for an application scenario.

2.2.4 *Methodologies in software engineering*

We present some results of assessment methodologies from software engineering and software measurements. The IEEE 1061 Standard for a software quality metrics methodology [86] proposes a methodology for software quality metrics. The methodology is a systematic approach to establish quality requirements and identifying, implementing, analyzing and validating the process and product of software quality metrics for a software system. It comprises five steps:

1. Establish software quality requirements.
2. Identify software quality metrics.
3. Implement the software quality metrics.
4. Analyze the software metrics results.
5. Validate the software quality metrics.

In [101] a framework for evaluating metrics in software engineering is proposed. The evaluation framework is based on *questions*, which range from the purpose and the scope of the measure, to the validity of the measures and the expected measurement error.

Finally, we mention [66] in which the approach recommended aims to handle key factors as causality and uncertainty, combining different (often subjective) evidence. According to the roadmap on software metrics presented in [66], the way forward for software metrics research lies in causal modelling, empirical software engineering, and multi-criteria decision aids. The authors stress the need of combining and successfully using (at the very end, to solve a decision problem) information from testing measurement data, empirical data, subjective information about the process/resources (e.g., the quality and experience of the staff), and specific pieces of evidence such as the existence of a trustable proof of correctness of a critical component.

2.2.5 *Standards for testability and diagnosability*

Testability according to [163] deals with those aspects of a system that allow the status (operable, inoperable, or degraded) or health state to be determined. According to [163], the writing of test procedures cannot and should not be done separately from testability analysis.

The IEEE Artificial Intelligence Exchange and Service Tie to All Test Environments (AI-ESTATE, [163]) standards are product information exchange standards for test and diagnosis. The standards of the IEEE 1232 series [87], [89] developed a means of exchange of information between diagnostic reasoners. As the information models for the IEEE 1232 standards were developed, it became apparent that these models could be used for standardizing testability and diagnosability metrics.

The AI-ESTATE architecture is a conceptual model in which AI-ESTATE *applications* may use any combination of components and inter-component communication [163]. The intent of AI-ESTATE is to provide a formal, standard framework for the exchange of diagnostic information (both static and dynamic) in a test environment.

The objective of the successive IEEE P1522 standard is to provide notionally correct, inherently useful, and mathematically precise definitions of testability metrics and characteristics. It is expected that the metrics may be used to either measure or predict the testability of a system. The intent is not to restrict the number or type of metrics, but to provide a sound, understandable, and repeatable basis for measurements [163].

2.3 TRUSTWORTHY OBSERVATION OF CRITICAL SYSTEMS

In this Section measurement tools for the evaluation of critical systems present in the literature are analyzed at the light of metrology concepts and rules. Without expecting to be exhaustive, this Section i) investigates if and how deeply such tools have been validated in accordance to measurement theory, and ii) tries to evaluate (if possible) their measurement properties.

Note that issues with the way measurement is applied in assessing computer dependability were first raised with respect to software reliability assessment. Problems were identified separately in two communities of research and practice: software reliability [32] and software metrics [14], [64]. There were three sets of inter-related issues: confusion about the meaning of a measure (leading for instance to redefining software "reliability" as a count of bugs in a piece of code, or to seeking scalar measures for inherently multi-dimensional attributes), confusion between problems of measurement and of prediction (leading for instance to naïve methods for inference from observed failures to future reliability), and insufficient fitness for purpose of the metrics [65].

In the following of this Section, dependable computing systems are looked at with the eye of the metrologist, with the intention of highlighting the peculiarities of such modern systems, with particular regard to the quantitative assessment of dependability and QoS metrics in accordance to measurement theory [24]. Section 2.3.1 proposes a classification of computer systems, from a metrological point of view, and singles out the most important properties to be evaluated for tools and, in general, for experimental campaigns of computing systems. Section 2.3.2 describes a number of well-known measurement tools for the analysis of dependable systems, which are critically evaluated along the lines traced by metrology concepts and rules.

2.3.1 *Metrology and dependability*

To adapt the metrology concepts of Section 1.3 to the field of dependable systems is not trivial. In this Section, a classification of computing systems, and of measurements that can be of interest on such systems are drawn. Then, on the basis of such classification, the most significant measurement properties that should characterize measurement tools designed to operate on the different kinds of systems are highlighted.

We start by classifying the computing systems whose QoS or dependability attributes are to be measured along the following dimensions [182], [164]:

- *Real-time*: along this paths we start from *time-free systems*, characterized by the absence of timing constraints or temporal requirements, to reach the case of so called *hard real-time systems*, characterized by well-defined constraints on their temporal behavior. A system is time-free when there is no deadline for its operations, whereas it is hard real-time when the correctness of its behavior is defined not only based upon the logical correctness of the operations performed but depends also upon the time frame in which such operations are performed.

- *Criticality*: along the criticality dimension at one extreme we find *non-critical systems*, while the other extreme is represented by *X-critical systems*, which may take many forms, e.g. *safety-critical systems*, or *mission-critical systems* or *life-critical systems*. While the failure of the former does not imply any significant damage, for the latter a failure could result in very dangerous events such as loss of life, significant property damage, or damage to the environment. There are many examples in different areas which fit this definition, such as medical devices, aircraft flight control, or nuclear systems.

- *Centralized/Distributed*: starting from the so called centralized systems we may find several forms of distributed systems. While a centralized system is made of a unique node, which may eventually be decomposed in non-autonomous and closely-coupled parts, a distributed system is a set of distinct nodes, with minor and even unstable coupling constraints, interconnected by any kind of network, cooperating for common objectives.

The aforementioned dimensions constitute a quite simple categorization of computing systems to which the concepts of metrology introduced in Section 1.3 should be applied. Depending on the category a system belongs to, the metrology properties and indicators may be less or more *important* to be taken into account and less or more *difficult and costly* to apply and to assess.

Among the fundamental properties that should be taken into consideration for a significant characterization of measurement systems, those of major concern to dependability evaluation can be identified in: *uncertainty*, *repeatability*, *resolution* and *intrusiveness*.

Table 2.1 enlists the abbreviations used in the other tables presented in this and in the next Section.

Table 2.2 describes the importance of metrological properties for the different categories of systems introduced above, which should be considered to design

CE	Centralized	Unc	Uncertainty
DI	Distributed	Int	Intrusiveness
RT	Real-time	Res	Resolution
¬RT	Non real-time	Rep	Repeatability
CR	Critical		
¬CR	Non-critical		

Table 2.1: Abbreviations used in Tables.

	Unc	**Int**	**Res**	**Rep**
CE-¬RT-¬CR	X	X	X	
CE-¬RT-CR	X	X	X	X
CE-RT-¬CR	XX	XX	X	
CE-RT-CR	XX	XX	XX	X
DI-¬RT-¬CR	X	X		
DI-¬RT-CR	X	X		X
DI-RT-¬CR	XX	XX	X	
DI-RT-CR	XX	XX	X	X

Table 2.2: Summary on important metrological properties to consider in order to perform confident measurements on computing systems.

measurement tools that can provide reliable results. A rank of the importance of assessing a metrological properties in each configuration is provided; *X* stands for recommended, *XX* for mandatory. We briefly explain the choices performed in what follows.

Uncertainty. A quantitative evaluation of uncertainty is necessary to appreciate the quality of the measurement. Such need is not only theoretical but has an important practical implication. Let us consider, for example, a safety critical system with hard real-time requirements; in such system there can be cases when uncertainty is essential to state whether the system is compliant with its requirements or not. If an indicator has to be below a given threshold, and the measurements results confirms it is below that threshold, one would be convinced that the system meets its requirements. What if after evaluating uncertainty, the interval expressing the measurements results is, even partially, over the threshold? In this case the available knowledge of the measurand does not allow to state that the system meets its requirements with sufficient confidence.

Uncertainty is even more important (and needs to be evaluated) in the case of distributed systems. Time interval measurements carried out on such systems can be significantly affected by offset and drift among distributed clocks. Another case in which uncertainty is very important is when indirect measurements are performed by combining results of several direct measurements. In such cases, the uncertainty of direct temporal measurements propagates on uncertainty of indirect measurements.

Intrusiveness. In general, performing measurements alters (to different extents) the state and the behavior of the system under test. In computer science, we can think of a target process which acts as the measurand, and of measures collected with a process scheduled on the same CPU hosting the target process; the schedulability of the entire system might be compromised, with consequent harmful effects on measurement results.

Performing an analysis of the intrusiveness of a measurement system is particularly important when measurements are carried out on computer systems or infrastructures, since this often implies loading the system and, ultimately, influencing its behavior in a non negligible way. The importance of intrusiveness in computer systems is clear and well understood, although it is difficult to quantify it. It should be evaluated as the impact of the measurement system on the performance of the computer system, expressed in terms of memory usage, CPU usage and/or operating system relative time. Intrusiveness is a parameter of fundamental importance for all the cases of interest of this Chapter. This is particularly true for real-time systems: a tool able to collect sufficiently reliable data in a non real-time environment may behave very differently in a hard real-time environment. Intrusiveness is thus particularly critical in hard real-time systems, where timing predictability may be altered by the additional overhead of monitoring tasks, or other mechanisms, e.g. fault injection probes.

Intrusiveness and *uncertainty* are related to each other since intrusiveness has consequences on uncertainty. This explains why in Table 2.2 all the rows in which *intrusiveness* is important exhibit the same importance for *uncertainty*.

Resolution. Resolution may be critical in real-time systems since it needs to be much lower than the imposed time deadline to allow useful quantitative evaluations of time or dependability metrics. In computing systems it can be generally assumed that resolution of the measurement system for time interval evaluation is equal to the granularity of the clock used in the experiment. In a centralized context it can happen that resolution is of the same order of magnitude of the measure, and it is thus of great importance to evaluate the resolution. On the other hand, when experiments are performed on distributed systems, uncertainty is usually far greater than resolution; in such cases, the evaluation and the control of resolution may be less crucial.

Repeatability. Repeatability is often not achievable when measurements are carried out on computer systems. The same environmental conditions can, in fact, hardly be guaranteed. This is especially true with regard to distributed systems, where differences among local clocks, in addition to the problems of thread scheduling and timing of events, enormously increase the difficulty of designing repeatable experiments. Critical systems are a class of systems in which repeatability is very important. In such cases, great efforts to grant the highest possible degree of repeatability are required and motivated. When performing experimental validation of a critical system, in fact, it is necessary to observe the same behavior triggered by the same trace of execution.

2.3.2 *Measurements properties in tools and experiments for dependability evaluation*

Tools and experiments used for experimental quantitative evaluation of computing systems are now described (the discussion extends and complements results of [24] with the addition of some recent works), with the purpose of understanding if they have considered and respected the aforementioned criteria and to which extent this has been accomplished. With no intent to criticize any individual experiment or experience, the objective is to investigate the general consciousness about metrological properties addressed in the previous Sections. The observations are based on an objective analysis of the considered works, and no attempt to numerically quantify measurement properties on these works is done. We do not want to question the results presented in the literature, but just to show that underestimating or neglecting factors such as uncertainty, intrusiveness, resolution and repeatability can easily reduce the trust in the achieved measures or in the developed measurement system.

In the left part of Table 2.3 (columns *Tool*, *Exp* and *System classification*) the works taken into consideration are classified according to the criteria introduced related to the dimensions of systems. Note that the columns *tool* and *experiments* are marked depending on whether the main focus was the tool or the experiments performed (in some cases, both).

The considered works cover some very different situations in which dependability measures have been collected. Such difference stems either from the type of analysis performed or from the kind of system under study, covering a spectrum of eight different systems typologies. Note that some works belong to more than just one category of systems.

Let us now consider the right part of Table 2.3 (columns *Relevant Properties* and *Awareness*). In the column *Relevant Properties*, the most relevant metrological properties that should have been addressed are singled out for each paper. The *Awareness* part of the table reports marks related to the measurement properties for which some concern (often with quite good observations) has been shown. Due to the very different, non-uniform, and often partial (or missing) approaches we have observed (even the name of the four measurement properties are different from a work to another), it has been actually difficult to identify these elements in the surveyed works. Therefore, in some cases the ticks in the table are the result of our own interpretation and understanding. This does not necessarily mean that measurement properties have been ignored by the related authors when designing tools and experiments, but we just note that these properties have often not been given the adequate emphasis.

Surveyed Work	Tool	Exp	System classification								Relevant Properties				Awareness			
			CE $\neg$RT $\neg$CR	CE $\neg$RT CR	CE RT $\neg$CR	CE RT CR	DI $\neg$RT $\neg$CR	DI $\neg$RT CR	DI RT $\neg$CR	DI RT CR	Unc	Int	Res	Rep	Unc	Int	Res	Rep
XCEPTION [34]	√		√								√	√	√			√		√
GOOFI [2]	√		√	√	√	√					√	√	√	√		√		
AVR-INJECT [46]	√	√	√								√	√	√			√		√
MAFALDA [61]	√	√		√							√	√	√	√		√		
MAFALDA-RT [154]	√	√			√	√					√	√	√	√		√		
MESSALINE [8]	√	√		√				√			√	√	√	√			√	√
FTAPE [177]	√	√					√				√	√				√		
Loki [44] [49]	√	√					√				√	√			√	√	√	
Neko/NekoStat [179] [63]	√	√					√				√	√					√	
ORCHESTRA [52]	√	√							√		√	√	√			√		
PFI Tool [51]	√	√					√				√	√				√		
OS dependability benchmark [99]		√	√								√	√	√					
Impact Analysis on Real-Time Sys [90]		√				√					√	√	√	√				
Evaluating COTS [58]		√				√					√	√	√	√				
Injection tools Comparison [165]		√				√					√	√	√	√	√	√	√	√
Fault-Tolerant Commercial Systems [178]		√					√				√	√				√		√
Log-based FFDA [149]		√					√				√	√				√	√	
FORTRESS [67]		√								√	√	√	√	√	√		√	
PLATO [15]		√						√			√	√		√			√	
OLSR [54]		√					√				√	√			√			

Table 2.3: Surveyed works classification and metrological properties.

Let us start analyzing *uncertainty*. Although a full consciousness of all measurement properties is not achieved in most of the surveyed tools, it is important to highlight that a quite exhaustive analysis of measurement parameters is, in some cases, performed.

Loki [44], [49] is a tool for software fault injection; it makes a post-runtime analysis, using offline clock synchronization, to place injections on a single global timeline and determine whether the intended faults were properly injected; there is a significant attempt to evaluate the uncertainty of the time instant at which faults were injected, even though it is not referred to as uncertainty. Such a deep analysis is performed only for time-stamping fault injection. Although the approach to uncertainty is quite informal, i.e., far from uncertainty as dealt with by the GUM [95], this example denotes a significant and remarkable interest in quantitatively evaluating the dispersion of the values that can reasonably be attributed to the measurand. Uncertainty-related issues are taken into account in the experiments related to the testing/development of FORTRESS [67]. FORTRESS is a support system for designing and implementing fault-tolerant distributed real-time systems that use COTS components. In the test case performed, the FORTRESS fail-aware datagram service gives an upper bound on the transmission delay of each delivered message.

The experiments reported in [54] evaluate the proactive routing protocol Open Link State Routing (OLSR), to identify the parameters which strongly affect the overall performance. Although the term "uncertainty" is not explicitly mentioned, uncertainty issues are attentively considered: statistical methods for the ANOVA (ANalysis Of Variance), also mentioned in the GUM, are applied and the range of variability of collected data is carefully described.

Finally, the three experiments reported in [165] attentively analyze sources of uncertainty affecting measurements results collected using three different fault injection techniques (test port-based injection, exception-based injection, and instrumentation-based injection). In most cases, the outcome of injecting a given error resulted very similar for the three techniques, increasing the confidence that the intended error is correctly injected, analyzed, and classified. Although the main focus of the work is uncertainty, the paper debates also on repeatability, intrusiveness and resolution, presenting a complete characterization of the measuring instrument [165].

It has to be highlighted that, for some time, the measurement relative uncertainty can be very low. This could explain the absence of assumptions or concerns about uncertainty. A few examples of such situations can be found in [177] and [99]. In [177], the FTAPE tool is used to measure the execution time with/without faults. Such time interval is greater than 1000 seconds. In [99], a dependability benchmark for operating systems (OSs) is developed and tested on Windows 2000 OS; the restart time of the OS is measured, which is equal to a dozen of seconds. Relative uncertainty here is very small, and in cases like these the absence of assumptions or concerns about uncertainty can therefore be justified.

Regarding *intrusiveness*, we have observed that a large subset of the surveyed tools show great consciousness of the importance of designing a non-intrusive measuring instrument (Loki [44], ORCHESTRA [52], XCEPTION [34], MAFALDA [61], MAFALDA-RT [154], FTAPE [177], GOOFI [2], PFI Tool [51] and AVR-INJECT [46]).

In Loki, awareness about intrusiveness is clear: to be as non-intrusive to the system as possible, at runtime the system does not block while notifications about the system state are in transit. A post-runtime check is made to correct possible problems due to non-compatible views of the system state.

In ORCHESTRA, a fault injection tool for distributed real-time systems, a deep analysis about intrusiveness is performed. ORCHESTRA deals with real-time systems with strict time requirements. It is designed to explicitly address the intrusiveness of fault injection on a target distributed system. This operation is performed by exploiting operating system support to quantitatively assess the intrusiveness of a fault injection experiment on the timing behavior of the target system and to compensate for it whenever possible.

In XCEPTION, a tool for software fault injection, an important attempt to evaluate tools characteristics using system performance monitoring facilities is made.

In MAFALDA, a fault injection tool for safety critical systems, the used fault injection technique is chosen with awareness of problem of intrusiveness of the tool.

In MAFALDA-RT, a tool for fault-injection in real-time systems (it is a completely new version of MAFALDA), the authors focus on the problems of temporal intrusiveness. The authors identify the main causes of intrusiveness in: i) the time related to the injection of faults, and ii) the time related to the observation of the system behavior.

In FTAPE, the authors recognize the problem of intrusiveness of the fault injection component and of the workload monitoring component, and they try to estimate the time overhead comparing the time the workload requires to execute with and without the fault injection and monitoring components.

In the fault injection Java tool GOOFI, it is recognized that logging is a time-consuming operation, thus GOOFI makes available two different logging modes: a detailed (time-consuming) mode and a normal (less time-consuming) mode.

In [51], the fault injection tool PFI Tool (Probe/Fault Injection Tool) is presented. The authors recognize that their approach can be more intrusive than others, but, despite this awareness, a metrological characterization of the tool is missing.

In AVR-INJECT [46], a tool designed to automate the fault injection on nodes of Wireless Sensor Networks (note that the target system has been classified as centralized in Table 2.3 because the tool allows to study a single node and not the whole network), attention to lightweight fault injection is addressed through very short perturbation functions executed at the user level.

In the experiments we have been considering, the intrusiveness of the measurement tool used is considered in [178], [149] and [165]. In the experiment in [178] a dependability benchmark for commercial systems is proposed and studied on TMR-based prototype machines (using the FTAPE tool). The time overhead of the fault injection tool used is accounted, even if intrusiveness is not looked closely as in [177]. In [149], the potential intrusiveness of a log-based Field Failure Data Analysis (FDDA) activity is acknowledged and possible distortions on dependability measurements caused by collisions (two independent faults which just occur coincidentally i.e., they are triggered near the same time [149]) are assessed through comparison of results obtained with both the proposed heuristic and log tuples. Finally, in [165] the expected intrusiveness of the three different fault injection techniques adopted is attentively discussed.

In most cases, missing observations about intrusiveness of the measurement tool may be an acceptable approach for single experiments, still allowing reliance on results, when a complete estimation of the uncertainty of the experiment results is provided (assuming that intrusiveness of the monitoring tool was already evaluated).

Resolution is usually the easiest parameter to estimate. However, it is frequently not considered at all: the reason is probably that it is often considered not important, at least if compared with intrusiveness and uncertainty. As an example on resolution, we observe that PLATO [15] and Neko/NekoStat [179], [63] use Java system calls to collect timestamps: this way the resolution of the system is the granularity of the Java clock used, usually greater than granularity of the system clock.

Finally, let us consider *repeatability*. The difficulty in reaching a satisfactory level for repeatability has been taken into account in the experiments on computer systems described in [178], even if the word *repeatability* is not explicitly used. The authors show consciousness that, due to the aforementioned limits on accurate timestamping, many executions of the same run will probably not bring exactly the same results, because the event (i.e., the injected fault) may not be signaled at the exact time instants when it is intended to occur. This explains why a second execution of the same run does not necessarily recreate a catastrophic incident that can, for instance, occur in the first execution. Among the surveyed works, the problem of creating repeatable experiments is discussed also in XCEPTION [34], MESSALINE [8], AVR-INJECT [46] and in the experiments in [165].

In XCEPTION it is highlighted that good results are achieved in experiments performed by using the spatial method for fault triggering (a spatially-defined fault is injected when the program accesses a specified memory address, either for data load/store or instruction fetch [34]), not in the temporal trigger methods, due to execution time uncertainties. This is an obvious limit, common to all tools.

In MESSALINE it is observed that in distributed system it is really difficult to perform repeatable experiments. Moreover, the type of the architecture has usually a major impact on the difficulty to set up a reliable testbed and on the repeatability of the experiments.

In AVR-INJECT [46] it is acknowledged that controllability and repeatability of the experiments is improved by the knowledge of the target instruction (i.e., the fault trigger).

In [165] repeatability is discussed and solutions to improve repeatability (as trying to recreate the exact same staring conditions) are proposed.

To complete the review two more works from Table 2.3 are briefly considered. In [58] a comparison of the dependability of different real-time Java virtual machines in the spacecraft software context is made. Some case studies are performed to evaluate dependability of COTS components. The authors show awareness of the problems encountered in Java and they use a Real-Time Java [18], which surely fits better in real-time context. However, although the great interest shown in precise timestamping, no information about measurement properties is provided.

In [90] the behavior of a real-time system running applications under operating systems that are subject to soft-errors is studied. Although errors due to real-time problems are recognized, no estimation about the quality and trustability of the presented results is shown.

A further remark concerns comparison among measurement results provided by different tools or experiments. In the surveyed tools, result comparison is rarely dealt with in terms of compatibility. Actually while expressing measurement results as intervals of values is the practice often followed in simulation studies, it is not as common in experimental dependability evaluations (with few exceptions as [165]). Comparison of results carried out in terms of compatibility can, in fact, be carried out only after evaluating uncertainty.

To summarize, the main findings of this brief survey on tools and experiments developed to assess dependability properties show that some consciousness about the metrology properties is present, but the approaches are quite intuitive, and usually quite incomplete as well. In particular, while there is a diffused consciousness about intrusiveness, there is rarely a real effort to try to estimate uncertainty and to determine solid bounds on the reliability and trustability of the measures collected with the tools.

In the experiments, less attention is paid to these themes. This does not mean that experiments are badly designed, nor that the measurement systems used for the experiments are not properly constructed, but more explanations about the measurement system should have been provided in order to allow to appreciate or understand what level of uncertainty may be associated to the obtained measures.

2.4 ANALYSIS AND SHARING OF EXPERIMENTAL RESULTS

Developing tools and experiments for quantitative analysis and collecting experiments results is a widespread activity. It is well-known that the analysis of resilience assessment data is a complex task, as the evaluation experiments usually produce large amounts of raw data, or data that are difficult to interpret. The importance of tools or techniques that support the data analysis is evident; several of such tools exist and are commonly applied, for example the very well-known Microsoft Excel and Matlab.

Supporting tools ease data analysis, but this is not sufficient to guarantee harmonization and comparability of results collected in different experiments and by different people. In fact different tools (used in different ways by different teams) lead to results that are presented very differently, making comparison difficult (the approach followed to quantitatively assess algorithms and systems is usually not univocal, varying from a work to another) [122], [10]. A related aspect specific to the research community that is worthy to mention here, and that we also briefly debated in Section 1.2.2, is that even if the final results and the conclusions are usually presented in papers, raw data are not shared, despite their availability would support the statements contained in the work and other teams working on related topics. Such problems are well-known, as it is witnessed by research initiatives and workshops devoted to the topic [166], [83], [5].

We present in what follows approaches for the analysis and sharing of experimental results. The solutions proposed are mainly based on data repositories, which are to the author knowledge the best way today to share results of experiments and to organize data, allowing further confrontation and analysis by other teams. We divide the discussion in two parts: first we show techniques to acquire and manipulate large quantities of data produced during experiments (Section 2.4.1), then we present existing public repositories and related toolsets to store, compare and analyze experimental data (Section 2.4.2).

2.4.1 *Data acquisition and manipulation techniques*

The OLAP and data warehouse approach

OLAP (On-Line Analytical Processing, [110]) is a technique to perform complex analysis over the information stored in a multidimensional data structure, typically a *data warehouse*, which is a global repository that stores large amounts of data extracted from heterogeneous systems. A data warehouse coupled with OLAP enable decision makers to analyze and understand business trends and to transform raw data into strategic decision making information.

The data are organized in the data warehouse using a *star schema*, which consists of a single *fact* table and a single table for each *dimension*. Facts are numeric or factual data that represent a specific business or process activity and each dimension represents a different perspective for the analysis of the facts, and it is described by a set of attributes. Normally, there are more than three dimensions and the dimension attributes represent a detailed description of the dimensional data.

The approach proposed in [122] for the experimental evaluation and analysis of critical systems consists in collecting the raw data produced during the experimental evaluation and storing them in a data warehouse structured as a star schema. In the case of experimental evaluation, facts tables of the star schema contain readouts collected during the experiments, and dimensions tables contain the key features of performed analysis. The dimensions include all the different perspectives that may be used to analyze those numerical facts.

For example, raw data representing things such as error detection efficiency or error recovery time are facts, while sets of attributes describing the target systems, the different configurations, the workloads, the faultloads, etc. represent the dimensions.

Once the data warehouse is populated, the OLAP tools can be used to analyze the data and compute the measures. As most of the tools allow analysis trough the web, this is the natural way to share the raw data. This methodology allows to analyze the usually large amount of raw data produced in dependability evaluation experiments and to compare results from different experiments or results from similar experiments across different systems.

Complex Event Processing (CEP)

When the purpose of the analysis is monitoring a distributed system, lighter supports than OLAP tools and data warehouse may be preferable to collect intermediate results for decision making phases. Complex Event Processing (CEP, [121]) is a defined set of tools and techniques for analyzing and controlling the complex series of interrelated events that drive modern distributed information systems [121].

As an example, we mention StreamBase's Event Processing Platform [168], which is a software for rapidly building systems that analyze and act on real-time streaming data. StreamBase provides a rapid application development environment, a low-latency high-throughput event server, and connectivity to real-time and historical data. The server queries and analyzes data streaming into the systems and delivers results on-the-fly.

2.4.2 *Data repositories*

The Software Reference Fault and Failure Data

The Error, Fault, and Failure Data Collection and Analysis Project [59], held at the National Institute of Standards & Technology (NIST), aimed at helping industries to assess software system quality by collecting, analyzing, and providing error, fault, and failure data of software systems. The project maintains a repository on software fault data, available upon request. Analytical and statistical use of the data is possible through a tool developed within the project. Unfortunately, a discouragingly low rate of data set submission led to a premature termination of the project [48].

Computer Failure Data repository

The Computer Failure Data Repository (CFDR, [159]) is a recent initiative supported by USENIX to create public repository on computer failure data. It aims to accelerate research on system reliability by filling the nearly empty collection of public data with detailed failure data from a variety of large production systems. It contains raw data of experiments, a description of the systems data, a description of data format, and results of data analysis, including possible reports or papers presenting such data [159].

Data & Analysis Center for Software

The Data & Analysis Center for Software (DACS, [50]), established in the late 1970s, serves as the authoritative US Department of Defense (DoD) source for state-of-the-art software engineering and technology information to support the software community. It provides a centralized hub for collecting and distributing software data and information and offers technical support for acquiring, developing, testing, validating, and transitioning software technology and processes. The DACS website lists over 40 software technical and management areas where users can get information on best practices, topic-area experts, tools, literature, service providers, training and other related resources [50].

The AMBER RAW data repository

The AMBER Raw Data Repository (ARDR, [4]) represents an effort to integrate and coordinate the European research and practice on assessment of critical systems, promoting the exploitation of results and the dissemination of knowledge. It provides users with raw data storage and OLAP analysis tools, while affording access to third-party data stored in an online data warehouse. This allows potentially any researcher either to compare their own data with data from other research teams, or to further analyze and investigate the experimental raw data made available in the OLAP repository by other researchers, possibly uncovering aspects of the results that were missed or not focused by the authors of the experiment [4].

Part II

PROPOSED METHODOLOGY AND THREE CASE STUDIES

3 A METHODOLOGY FOR THE EXPERIMENTAL EVALUATION OF CRITICAL SYSTEMS

In the previous Part (i), several methodologies were introduced for experimental evaluation, which share to a large extent similar foundations, and refer or include a huge set of techniques explored and presented through years in the state of the art. We propose in this Chapter our own methodology for the experimental evaluation of critical systems, born from the analysis of the state of the art performed in Chapter 2, which aims to overcome some of the common limitations that we identified. In particular, the following aspects are tackled that we believe deserve attention and need improvements:

- The necessity of a *general methodology* for the experimental evaluation of critical systems i.e., a methodology that is independent from the kind of system or specific technique selected. This is also due to the necessity to target different contexts, from academic research to industrial practices, to support both the planning and execution of tests and the writing of documentation produced as part of V&V and certification processes (see for example the IEEE Standard for Software Test Documentation [88] or the standard IEC 61508 [84]).
- As support of the methodology, we identified that the exploitation of principles from *measurement theory* could be used to assess results and measuring systems thus increasing trust in the experimental evaluation.
- Finally, solutions to improve the *reuse, comparison and sharing* of data could be included in the methodology. Should structured, fully depicted and trusted results be provided, then tools and experiments could be better compared. This calls for an approach to data sharing that is *general* (as any existing tool and experimental setup should be used), *eases comparison* and cross-exploitation of raw results from different experiments, and *eases sharing* raw results within an organization or even world wide.

Consequently, we present the task and steps required, without contextualizing them for specific systems or techniques. The application of the methodology to various techniques and systems is demanded to the case studies reported in Chapter 4 to Chapter 9.

The following of this Chapter is organized as follows. In Section 3.1 we present an overview of the proposed methodology, while from Section 3.2 to Section 3.5 the various phases of the methodology are described. Finally, in Section 3.6 the case studies that shall be presented in the following of this Thesis are briefly introduced to identify which are the main aspects of the methodology explored in each case study.

3.1 OVERALL VIEW OF THE METHODOLOGY

The methodology is subdivided in four *phases* (numbered from 1 to 4), each of them organized in *steps*, or *tasks*. We present the main phases and the lists of steps that should be taken into account by the experts performing the experimental evaluation.

A high level view of the overall methodology is presented through an UML activity diagrams in Figure 3.1, which shows a comprehensive view of the four phases and their relations.

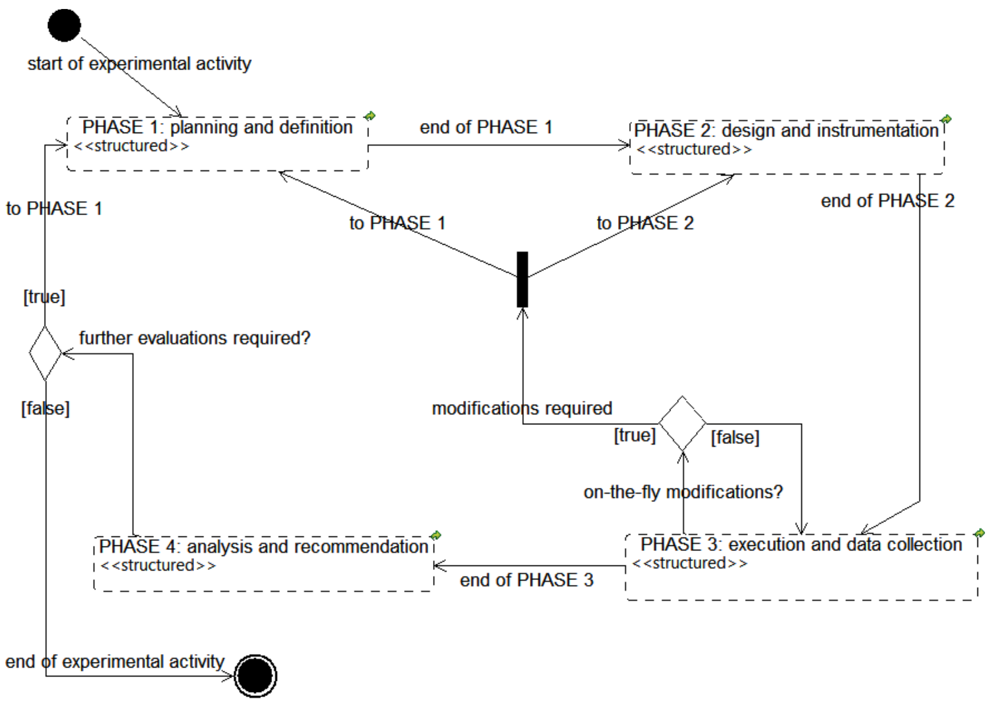

start of experimental activity
PHASE 1: planning and definition
<<structured>>
end of PHASE 1
PHASE 2: design and instrumentation
<<structured>>
to PHASE 1
to PHASE 1
to PHASE 2
end of PHASE 2
[true]
further evaluations required?
[false]
modifications required
[true]
[false]
on-the-fly modifications?
PHASE 4: analysis and recommendation
<<structured>>
end of PHASE 3
PHASE 3: execution and data collection
<<structured>>
end of experimental activity

Figure 3.1: Activity diagram of the methodology.

As experimenting is typically iterative, the methodology is organized in four iterative phases:

- *PHASE 1: Planning and definition*. This phase constitutes the planning and the definition of the experiments. The objectives of the experiments, the target system, the workload, the faultload and finally the experiments are defined. Also, the organization of results can be defined in this stage, since it allows to identify in an unambiguous way the purposes and contexts of the analysis. The envisioned architecture of the measuring system should be sketched in this phase, to be refined and implemented in the following phase 2.

Executing this phase attentively increases the possibility that the overall experimental evaluation is successful; we recommend that at the end of this phase a conclusive analysis is carried out to identify potential lacks and major issues that may happen during the phases devoted to implementation and execution (phase 2 and phase 3), and to decide if re-iteration of phase 1 is required. At the end of phase 1, phase 2 is started. Details on phase 1 are shown in Section 3.2.

- *PHASE 2: Design and instrumentation*. Once phase 1 is completed, it is necessary to build the required instruments for the experimental campaign, and to integrate them in the system. Activities that are part of this phases are the design and implementation of the measuring system (the monitor, the fault injection instrument, the workload generator, and the controller), and finally the integration with the system under test (e.g., the system under test is instrumented with the fault injection tool and the required probes and loggers). At the end of this phase, the measuring system (with the possible exception of the parts related to data analysis) is defined, implemented and the system under test instrumented. If possible, preliminary sample *runs* (*executions*) of experiments to verify the set-up are executed. A metrological assessment of the measuring instruments and of preliminary results is necessary at the end of this phase, to estimate if the quality of results will be acceptable. This phase should be iterated until feedbacks from the metrological assessment are positive; at this point, phase 3 is started. Details on phase 2 are shown in Section 3.3.

- *PHASE 3: Execution and data collection*. Once the system is instrumented, the experiments are executed. During the execution of the experiments some small modifications may be useful or necessary. For example, such modifications may be in terms of new experiments, a different configuration of the set-up or even a different order in the execution of the planned experiments. These modifications may be suggested by an analysis on-the-fly of the results.

  In Figure 3.1, this phase is also connected to phase 1 and phase 2. When the experiments execution concludes, phase 3 terminates and phase 4 is started. Details on phase 3 are shown in Section 3.4.

- *PHASE 4: Analysis and recommendations*. The objective of this fourth phase is to analyze, share and cross-exploit measurements results collected. Data staging and analysis is performed on the measurement results collected to draw conclusions, which are typically in form of summarizing results and recommendations. From the conclusions, further experiments or modifications to the experiments may be derived, thus requiring the re-execution of the methodology (that is, restart from phase 1). Otherwise, the evaluation activity terminates. Details on phase 4 are shown in Section 3.5.

Our methodology aims to provide a set of guidelines expressed through activities and relations between activities. We are aware that in many experiments, some of the activities proposed are trivial or not relevant, and consequently it is not required to address them explicitly or devote a particular care. Also, there is a large set of techniques available to implement some of the activities proposed: it is not our intent to enlist them here, nor to define classifications. There are very relevant works in the dependability community which provide exhaustive classifications and description of specific techniques, and to which we refer (e.g., see [164], [124], [170], [180]). We demand the selection of techniques to the comprehensive reviews of the state of the art for the experimental evaluation of critical systems.

The four phases and the list of tasks (also called *steps* in what follows) that are part of each phase are explored in Figure 3.2 through a Work Breakdown Structure (WBS, [116]).

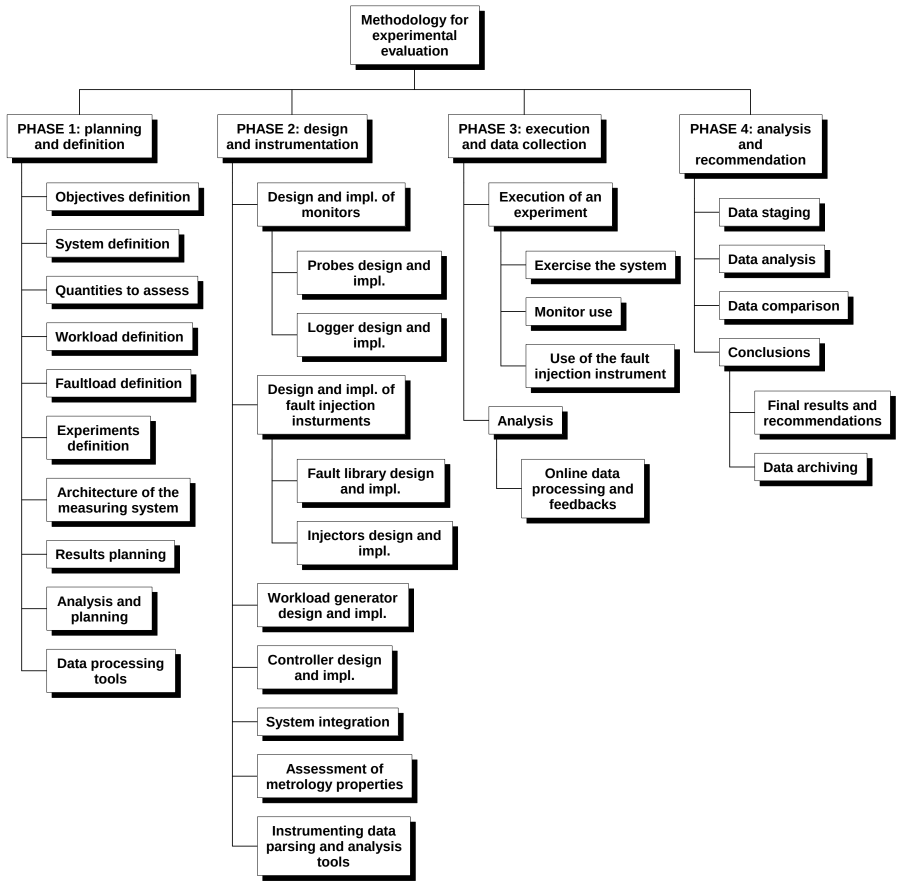

Methodology for experimental evaluation
PHASE 1: planning and definition
Objectives definition
System definition
Quantities to assess
Workload definition
Faultload definition
Experiments definition
Architecture of the measuring system
Results planning
Analysis and planning
Data processing tools
PHASE 2: design and instrumentation
Design and impl. of monitors
Probes design and impl.
Logger design and impl.
Design and impl. of fault injection insturments
Fault library design and impl.
Injectors design and impl.
Workload generator design and impl.
Controller design and impl.
System integration
Assessment of metrology properties
Instrumenting data parsing and analysis tools
PHASE 3: execution and data collection
Execution of an experiment
Exercise the system
Monitor use
Use of the fault injection instrument
Analysis
Online data processing and feedbacks
PHASE 4: analysis and recommendation
Data staging
Data analysis
Data comparison
Conclusions
Final results and recommendations
Data archiving

Figure 3.2: WBS view of the methodology.

The WBS is a model typically used in project management which allows to "break" the project down into discrete components or tasks). Note that the WBS does not allow to appreciate the parallelism of activities; this will be identified in the detailed description of each phase.

3.2 PHASE 1: PLANNING AND DEFINITION

Phase *planning and definition* is composed of the steps reported in Figure 3.3. At the end of this first phase, a description document should be provided which summarizes the outputs of each step i.e., defines the plan of the experimental activity.

3.2.1 *Objectives definition*

The definition of the *objectives* (or *targets*) of the experiments is the first step to start an experimental activity. The objectives should be defined clearly, through a precise description of what is the target of the experimental evaluation. This description may also discuss what are the expectations of the experiments, because these are often at the base of the establishment of experimental campaigns.

3.2.2 *System definition*

After defining the key objectives, the *system under test* (or *target system*) and its *boundaries* are defined. Boundaries are particularly relevant as they allow to separate in an unambiguous way the system under test from the other components of the measuring system (see also the definition of system under test and measuring system in Section 1.2.2). This distinction is fundamental to understand events or components that may influence the system under test, through the interfaces offered by its boundary elements, and may alter the system behavior (e.g., in terms of repeatability, measurement uncertainty and intrusiveness).

Finally, defining the boundaries allows to identify the functions and structures whose behavior does not compromise the quality of measurements. Such functions and structures can be used to support the other components of the measuring system (e.g., act as loggers), or to select the components that are not required to be monitored or controlled during the experiments execution.

Note that the various configurations of the system may be classified as different system*s* under test (target system*s*).

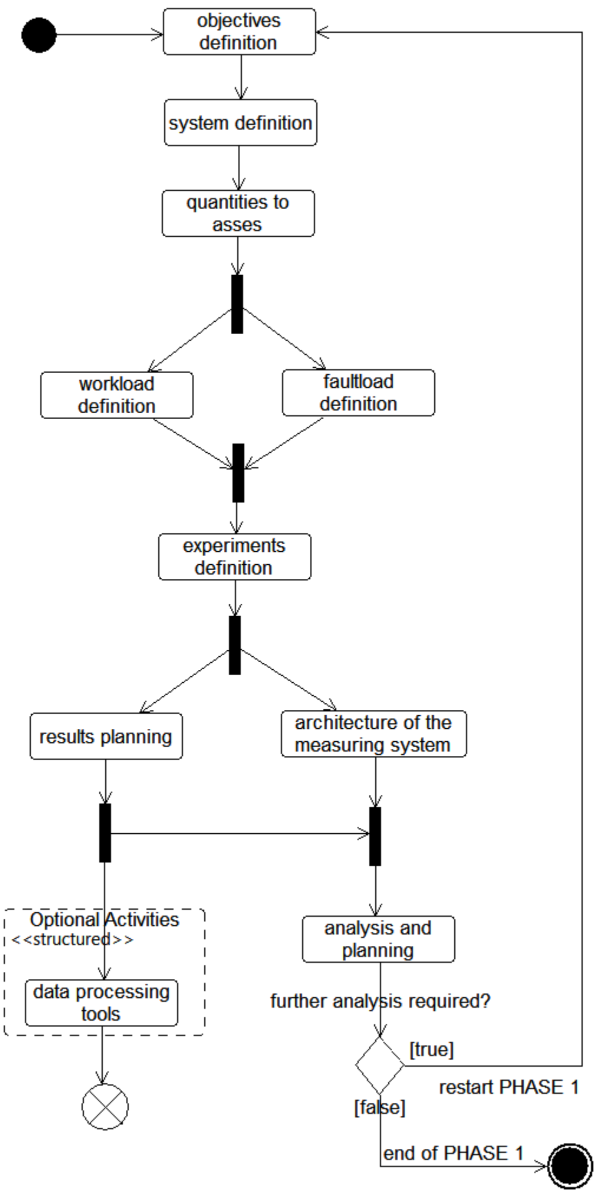


Figure 3.3: PHASE 1: Planning and definition.

3.2.3 *Quantities to assess*

When performing experimental measurements, the *quantities to assess* should be clearly identified. In fact, the faultload, workload, experiments and measuring system will be defined to tackle the analysis of the quantities defined in this step.

The definition of a quantity can be intuitive, especially in case of well-known and countable quantities, or complex (e.g., for derived quantities which are indicators of QoS). Different approaches for the definition of the quantities can be found in literature. For example in [62], [31] several fields are proposed to describe a quantity, including *motivations* or *applicability*; instead in [102] quantities are identified on the basis of their nature (qualitative/quantitative), type (dependability/performance related), extent (comprehensive/specific) and the assessment method (experimental/modeling).

To maintain our methodology intuitive, we only recommend few key fields to define a quantity, which are *name*, *acronym*, *description*, and *measurement unit* (if applicable). Obviously we do not restrict the possible usage of other fields.

3.2.4 *Workload and faultload definition*

The selection of an appropriate workload and faultload (*workload definition* and *faultload definition*) can be performed in parallel (see Figure 3.3). A workload represents a typical execution profile for the considered application area; that is, the workload is the computational load for the system under test. A faultload represents a typical set of faults affecting the operation of the system. A faultload is described by the set of faults and their type, their intended location, insertion time and distribution in time and space to be inserted into the system under test. Previously collected field data measurements can be used for the definition of the workloads and faultloads, in order to define experiments that are representative of the system behavior in its real execution environment [45].

Finally, we note that the so-called *stressload* [45] can be derived combining workload and faultload.

A discussion on how to select the proper workload (i.e., application, benchmark or synthetic workload [81]) and kind of faultload and fault injection technique (i.e., hardware fault injection with or without contact, and software fault injection at compilation or runtime [81]) would be inappropriate here, and is demanded to works devoted to such topic as [45].

3.2.5 *Experiments definition*

The experimental evaluation activity is characterized by one or more experiments to be executed on the target system. The *experiments* can be defined combining the different workload, faultload and systems under test (the various configurations of that target system) that were identified in the previous steps. Additional information may be necessary, as the experiments duration; note that in some cases such information may be derived from the description of the workload, faultload or system under test.

3.2.6 *Architecture of the measuring system*

An high level view of the architecture of our measuring system should be defined during this step. This high level view should identify the main components, and a generic view of their structure and connections. Also, their connections to the system under test should be defined here. This high level view is required in phase 1 as it is the first step to contextualize the experimental evaluation from a conceptual work towards the practical execution of the target system. Knowledge on the overall architecture, together with an investigation of possible problems of the architecture and of the experimental campaign (see the step *analysis and planning*), helps determining the completeness of the analysis performed during phase 1 and the feasibility of the planned experimental activity.

In our methodology, we identify the following components as key parts of a measuring system (see also Section 1.2.2):

1. A *controller*, which controls the experiment execution. In some cases, it may correspond to the operator.
2. A *workload generator* and the related *workload library*, to generate the workload.
3. A *monitor*, composed of probes and loggers.
4. In fault injection experiments, a *fault injection instrument*, that includes injectors and a fault library. Whenever possible, the specific technique to be applied for fault injection should be decided in this phase, while designing the high level view of the system (see also Section 3.3.1).
5. *Data staging and analysis* tools, to manage and analyze data.

These components will be detailed in successive phases of the methodology. The controller, the workload generator, the monitor and the fault injection instrument are explored in phase 2, the data staging and analysis tools are optional steps in phase 1 and phase 2, and mandatory steps of phase 4.

Note that fundamentals from metrology should be taken into account in the design of these components, following principles from Chapter 1 and observations from Chapter 2. Although the architecture design proposed in this step is intuitive and may be refined afterwards, it is required that its design considers basics from metrology. In fact, not only specific solutions for very pointwise problems, but also the overall architecture design, may affect the quality of measurement results. For example, a simple, well-known approach to reduce intrusiveness is to execute on the target system only the smallest possible set of operations, while demanding the *logging* and *controlling* activity to a separate (distributed) node, connected to the target system.

As general and broad observations and guidelines, we mention that the quality of measurements is generally increased by common practices as i) execute the data logging activities and the fault injection controller on a node different than the system under test, in order to allocate on the system under test only light probes and injectors [2], [34], ii) optimize as much as possible the logging activities [2], since they are time-consuming, iii) control the perturbation of the system [52], and iv) keep clocks tightly synchronized to reduce the possibility of biased data due to poorly synchronized clocks [79]. Further information and suggestions are not explored here, but can be found in the case studies reported in the following of this Thesis.

3.2.7 *Results planning*

The planning of the results can help to structure and highlight the objectives, the results and the key elements of our evaluation, thus it helps to define the purposes and context of the analysis [122].

The planning of results can be defined in this stage, since it allow to identify in an unambiguous way the purposes (and contexts) of the analysis. From the analysis in Section 2.4, as possible technique to structure and organize results we candidate the *star schema* presented in Section 2.4.1. Following [122], we contextualize it for the organization of results in the experimental evaluation of critical systems.

Note that in our methodology, differently from what is typically done in data warehousing [110], the star schema is not necessarily bounded to the underlying relational database. In other words, we can create a star schema even if we have no intention of using a highly structured database for the management of measurement results, but only to help defining in an (as much as possible) unambiguous way the purposes and contexts of the analysis. This way, the star schema simply acts as a reference diagram to verify the completeness of the analysis and to explain what are the key components identified during phase 1 (consider that the star schema is usually intuitive).

Examples of star schemas for experimental evaluation can be found in the case studies described in the following of this Thesis.

3.2.8 *Analysis and planning*

The *analysis and planning* considers as inputs all previous steps and related outcomes. The objective of this step is to understand if phase 1 can conclude or further iterations are needed. To decide this, an overview summarizing the planned actions for the experiments execution is performed, analyzing potential risks and possible recovery actions. This analysis consists basically in:

- understand the feasibility, also considering time and costs, of the planned experiments;

- perform a preliminary assessment of the expected metrological properties of the envisioned architecture and the related problems. This analysis is based on the conceptual work done during phase 1. In particular it considers the high level definition of the measuring system, the knowledge on the system under test, and considerations on workload, faultload and experiments. Some examples on the observations based on knowledge on the measuring instruments were reported in the step *architecture of the measuring system*. Further considerations can be performed considering also the specificity of experiments, workload and faultload, for example on reproducibility of results and experiments.

The final objectives of this step are to understand possible lacks, estimate feasibility of the planned actions, and make a plan for the successive phases. Such plan may include estimation on expected timings, costs, required instrumentation, and workplan; an execution plan of the experiments may be also defined (e.g., the test specification required in many industrial contexts [42], [57]). Additionally, outcomes of this step may require to re-iterate some of the previous steps.

3.2.9 *Data processing tools*

The optional step that we present here is the step *data processing tools*. Activities as building script for data parsing and database loading can be performed in this step to ease import of the results. This step is not mandatory, but can turn useful during the successive phases. In fact, if data can be easily imported and analyzed (or if a preliminary analysis can be performed), then quick feedback on (ongoing) experiments can be achieved to decide on corrective actions.

Note that data processing is also optional in phase 2 and phase 3. It is instead mandatory in phase 4. In fact, despite this step is actually needed only in phase 4, in which conclusive results are achieved, having preliminary data processing (and analysis) allows to tune the experiments, the faultload, the workload and the measuring system, and it helps timely identifying possible criticality in the experimental activity or in the instantiation and set-up of the measuring instrument.

3.3 PHASE 2: DESIGN INSTRUMENTATION

The phase *design and instrumentation* is composed of the steps reported in Figure 3.4 and described in what follows.

In this phase the measuring system is built and the system under test is instrumented with the measuring instrument. Four parallel steps are identified at the beginning of this phase: these are the design and implementation of i) the monitor (probes and loggers), ii) the workload generator, iii) the fault injection instrument (fault library and injectors), and iv) the controller. These components are designed and built on the basis of the outcomes of phase 1. The integration of these four components in the system under test follows.

Note that foundations from metrology should be taken into account during the design, the implementation and the integration of these components. A metrological assessment is in fact performed to evaluate if the required metrological properties are satisfied. Optionally, tools for data processing and analysis can be instrumented with the measuring system.

At the end of this phase, all the instruments required to execute the experiments are ready. Output of this phase is the complete design and implementation of the measuring system, and the updated metrological assessment.

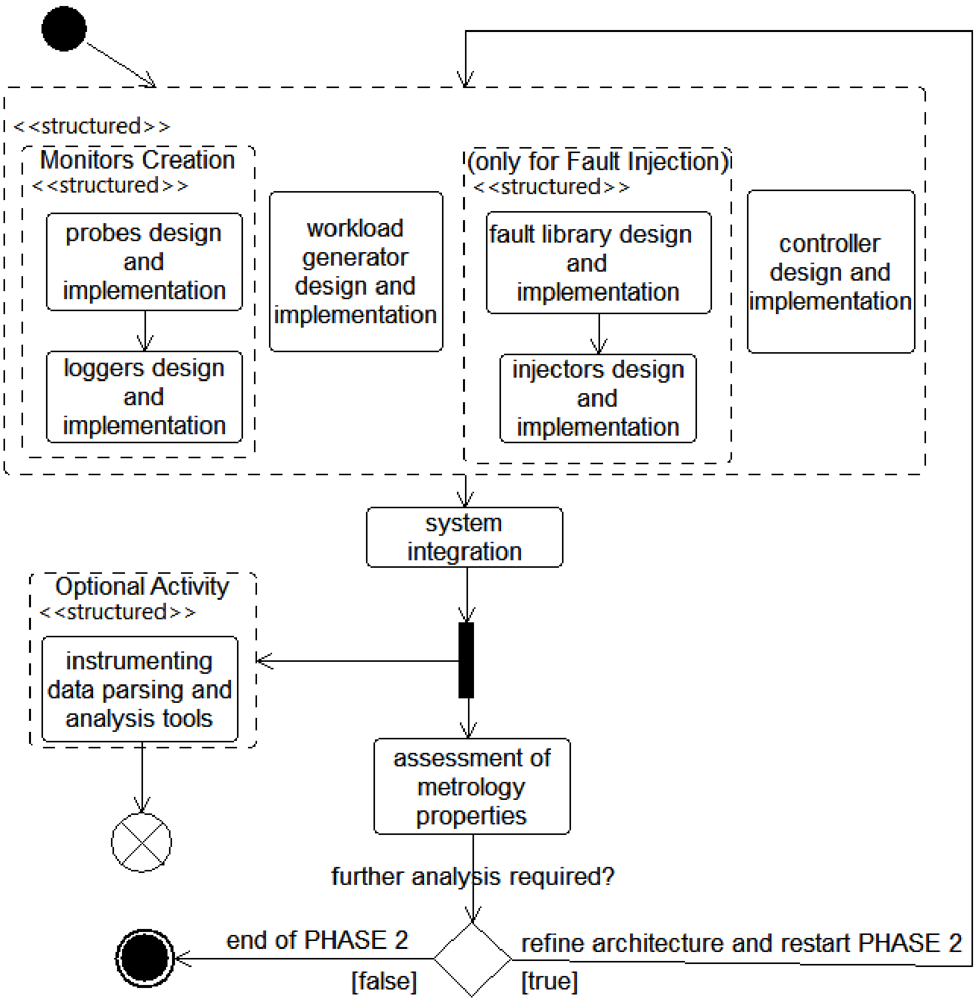

<<structured>>
Monitors Creation
<<structured>>
probes design and implementation
loggers design and implementation
workload generator design and implementation
(only for Fault Injection)
<<structured>>
fault library design and implementation
injectors design and implementation
controller design and implementation
system integration
Optional Activity
<<structured>>
instrumenting data parsing and analysis tools
assessment of metrology properties
further analysis required?
end of PHASE 2
[false]
refine architecture and restart PHASE 2
[true]

Figure 3.4: PHASE 2: Design and instrumentation.

3.3.1 *Design and implementation of the monitor*

The monitor tracks the execution of the system and initiates data collection whenever necessary. We divide the monitoring part in two components: probes and loggers.

Probes

In order to observe the target system, (hardware or software) probes are attached (instrumented) to the system to provide information about the system's internal operation.

The probes must be capable of observing the internal operation of the system to fulfill the purpose of the monitoring i.e., allow to measure the quantities of interests.

A classification of monitoring systems can be performed according to the three approaches used for probes: hardware, software and hybrid (i.e., hardware/-software) [186]. We do not detail here specific probing solutions, as choosing the right approach and implementing it is beyond the scope of this dissertation; detailed discussions can be found in [186], [126].

Loggers

The *loggers* collects data from the probes. A logger can execute on the same system where the probes are located or on a separate system (e.g., to reduce intrusiveness). Data stored by the loggers constitutes the measurement results collected during the experiments execution, and can be further analyzed to derive the experiments results or to compute indirect quantities.

Information on loggers and logging techniques can be found in [73], [160].

3.3.2 *Design and implementation of the fault injection instrument*

For those experiments which include fault injection, the instrument in charge of performing injections is designed and implemented in this step. It is composed of a fault library and one or more injectors.

Fault library

The *fault library* describes the faults (including at least fault types, locations, and times) that the injectors are able to inject [81]. The specific information to be contained in the fault library depends on the characteristics of the fault injection experiments and on the injectors.

Injectors

The *injectors* inject faults into the target system as it executes commands from the workload generator. An injector can be hardware or software. Each injector can support different fault types, fault locations, and fault times, the values of which are drawn from the fault library [81].

3.3.3 *Design and implementation of the workload generator*

The system under test executes a workload. The workload may derive from the typical execution of the system in its environment (e.g., an application workload which simply consists in setting up the system and let it operate in its usual scenario), or may be designed ad-hoc for the execution of specific experiments. In this second case, it is necessary to design and implement a component, called *workload generator*, which is in charge of executing the workload.

The kind of events and parameters that need to be included in the workload generator depends on the specific experiment and system under test; however, generally the workload generator allows to generate events at specific time instants or following a specific distribution [45].

3.3.4 *Design and implementation of the controller*

The *controller* controls the experiments to execute. The controller is typically a program that can run on the target system or on a separate computer. Note that in some experiments the controller may not be necessary, or may simply be the expert (the operator) which guides the execution of the experiments (for example, switches on/off a machine, starts or stops the monitored application, etc.).

3.3.5 *System integration*

Once the previous components are implemented, they must be *integrated* and plugged to the target system.

Note that some of the required components may not be available until the last moment (e.g., when integration is performed right before going on-field; an example is in the case study in Chapter 8, where the measuring instrument and the system under test were provided by two different groups, and integrated only when the two groups gathered on-field right before starting the execution of the experiments). In such cases phase 2 is strictly connected to phase 3, also from a temporal viewpoint.

3.3.6 *Instrument data parsing and analysis tools*

An additional component that can be integrated in this phase is the data parsing and analysis tool, for example a preliminary version developed during phase 1. In the optional step *instrument data parsing and analysis tools*, instruments and tools for parsing data and extracting results are defined and instrumented. This step allows online analysis of the data collected on-field while an experiment executes or at the end of each of the planned experiments, as described in phase 3. Extensive discussions and literature on online analysis can be found in [180], [77], [167].

3.3.7 *Assessment of metrology properties*

As lengthily debated in this Thesis, a rigorous investigation of the fundamental metrological properties is mandatory. The last step we plan for phase 2 is the process to assess the quality in terms of metrological properties of the measuring instrument and of the expected measurement results. That is, the objective of this step is to acquire knowledge on how much confidence we should put in the results of our experiments.

To perform this analysis, information acquired from data sheet, knowledge on the measuring and target system and on their implementation, or preliminary runs of ad-hoc experiments can be used. An attentive analysis of the system under test, its boundaries and the measuring system is necessary. Phase 2 is iterated until this step is performed successfully.

To ease the execution of this step, we present some guidelines and key references focusing on resolution, intrusiveness, uncertainty and repeatability:

- *Resolution*. Of the four properties enlisted here, it is typically the easiest to quantify. Often resolution is explicitly reported in the data sheet of the system under test. For example, in case of time measurements, resolution is typically the resolution of the local clock. In distributed time interval measurements, the resolution is the maximum between the resolution of the source node and that of the destination node. In case of position measurements, it is typically the smallest variations in position that the system is able to detect.

- *Intrusiveness*. Understanding the intrusiveness of a system is not trivial. Golden runs (a golden run is a trace of the system executing without any injections being made, hence, this trace can be used as reference [76]) can allow to understand the impact of the components of the measuring instruments, to perform an offline analysis before the beginning of an experiment. In many cases, it can be sufficient to bring evidence by reasoning that the intrusiveness is negligible (with respect to the dimension of measurement results).

- *Uncertainty*. It is important and useful to evaluate the measurement uncertainty in order to be aware of the quality of measurement results. This is particularly true whenever we want to obtain direct or indirect measurements in which time intervals related to different nodes of a distributed system are involved. In fact amongst measurements, the most common sources of uncertainty are in time measurements, where a significant cause of uncertainty is usually represented by poor synchronization between distributed clocks. Misalignment of distributed clocks may affect the quality of measurements: such misalignment is hard to estimate and predict, and may vary due to many causes such as unexpected network delays, temperature variations or even faults in clock synchronization mechanisms. For example, exploiting clock synchronization protocols like the Network Time Protocol (NTP, [129]) does not totally prevent from collecting some severely incorrect data due to transient perturbations which cause an increase in the dispersion of the values that can reasonably be attributed to the measurand (e.g., a time interval); this can be unacceptable for some applications. In Chapter 4 we will further debate on this specific example.
- *Repeatability*. Determinism of the target system is needed to ensure repeatability, including the starting state of the system: for example, in order to completely ensure repeatability of every experiment, a fault injection tool would have to copy the entire state of memory at start-up and restore it in each experiment [165]. Repeatability is probably the most critical issue to face, especially when performing time measurements in distributed systems, due to limits on collecting accurate time values (executions of the same run will probably not bring the same exact results [96]).

Extensive notions on how to assess a measuring system and the measurement results collected are reported in [95], [94], [97], [96]. Examples on their application can be found in the case studies of this Thesis.

We single out and recommend some basic guidelines, i.e. operative rules that should be kept in mind - and if possible put into practice - when measuring dependability-related attributes [24]:

- the measurand should be clearly and univocally defined;
- all sources of uncertainty should be singled out and evaluated;
- some attributes of major concern for dependability measurements, such as intrusiveness, resolution and repeatability should be evaluated;
- measurement uncertainty should be evaluated (according to the GUM);

- comparison of measurement results provided by different tools/experiments should be made in terms of compatibility [96].

3.4 PHASE 3: EXECUTION AND DATA COLLECTION

The phase *execution and data collection* is composed of the steps reported in Figure 3.5 and described hereafter. This phase is iterated for each experiment executed. Once all experiments have been executed, this phase terminates.

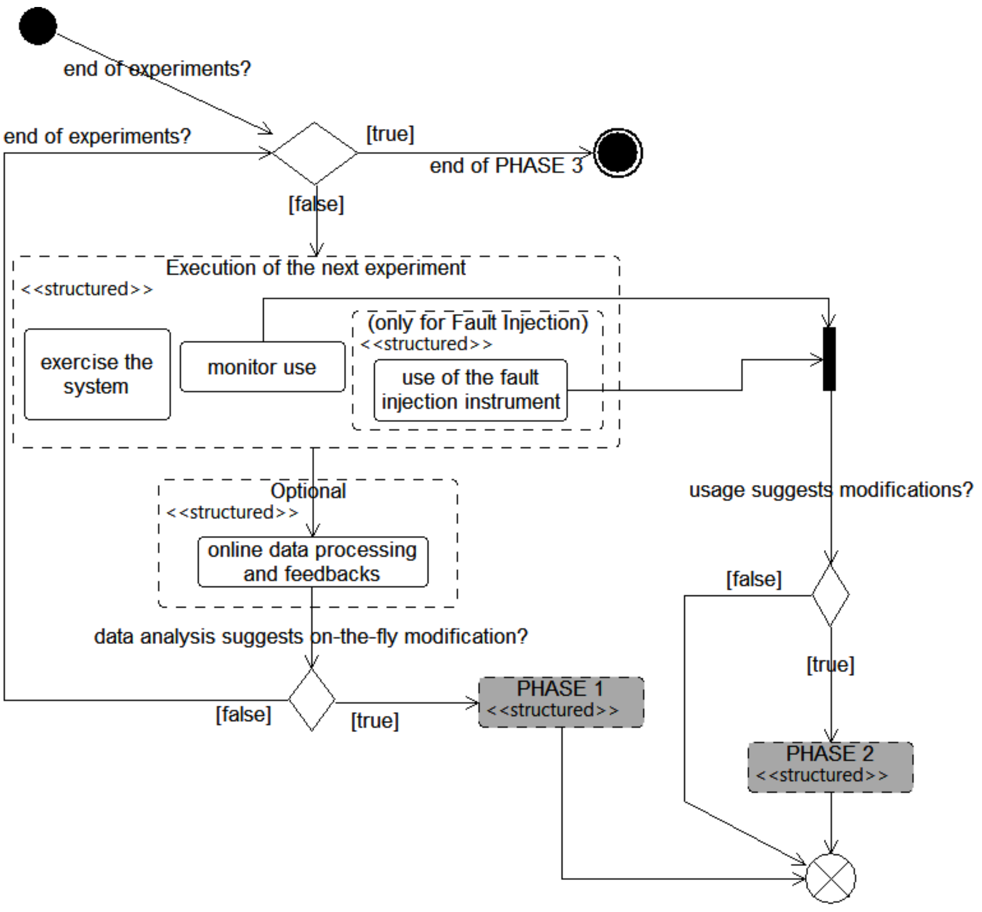


Figure 3.5: PHASE 3: Execution and data collection.

Note that during the execution of the experiments, it may be required to perform some (small) adjustments that is, to modify some of the outcomes of previous phases. Consequently, from phase 3 it is possible to move back to phases 1 or directly to phase 2.

In Figure 3.5 the possible transition to phase 1 is proposed at the end of each experiment; the analysis of the collected data can lead to the decision of going back to phase 1. Instead a potential transition to phase 2 is activated through feedbacks from the usage of the measuring instrument.

3.4.1 *Execution of the next experiment*

The core of phase 3 are three parallel steps, that are *exercise the system*, *monitor use* and (in case of fault injection experiments) *use of the fault injection instrument*. For each experiment, these steps are executed.

3.4.2 *Online data processing and feedbacks*

Objective of this optional step is to analyze the measurement results while still executing the experiments. The objective is to provide quick feedback on the results and the behavior of the measuring instruments, and to identify possible modifications required to the experiments description or to the measuring system. A more detailed analysis of the collected data is instead performed in phase 4.

3.5 phase 4: analysis and recommendations

The phase *analysis and recommendation* is composed of the steps reported in Figure 3.6 and discussed in what follows. In this phase, analysis of measurement results is performed and conclusions are drawn. This is the last phase of the experimental evaluation; the outcomes of this phase determine if the experimental activity terminates or restarts.

3.5.1 *Data staging and analysis*

We subdivide the data staging and analysis in two steps:

- a *data staging* step to structure and integrate the logged events in a database
- the analysis of measurement results (the step *data analysis*).

We represent these two steps in Figure 3.7.

Data staging

The inputs of data staging are events, data and intermediate results collected by the loggers in phase 3 during the execution and observation of the system (the purpose of data staging is to get data ready for loading into a presentation server [110]). Data staging is quite a complex task that can be heavily intrusive. That is why we plan it at the end of the experiments, when all data are collected and available.

Any existing tool and experimental set-up can be used; the only information required is the data format of the raw results produced during experiments, in order to load them into the database. OLAP technologies, which include

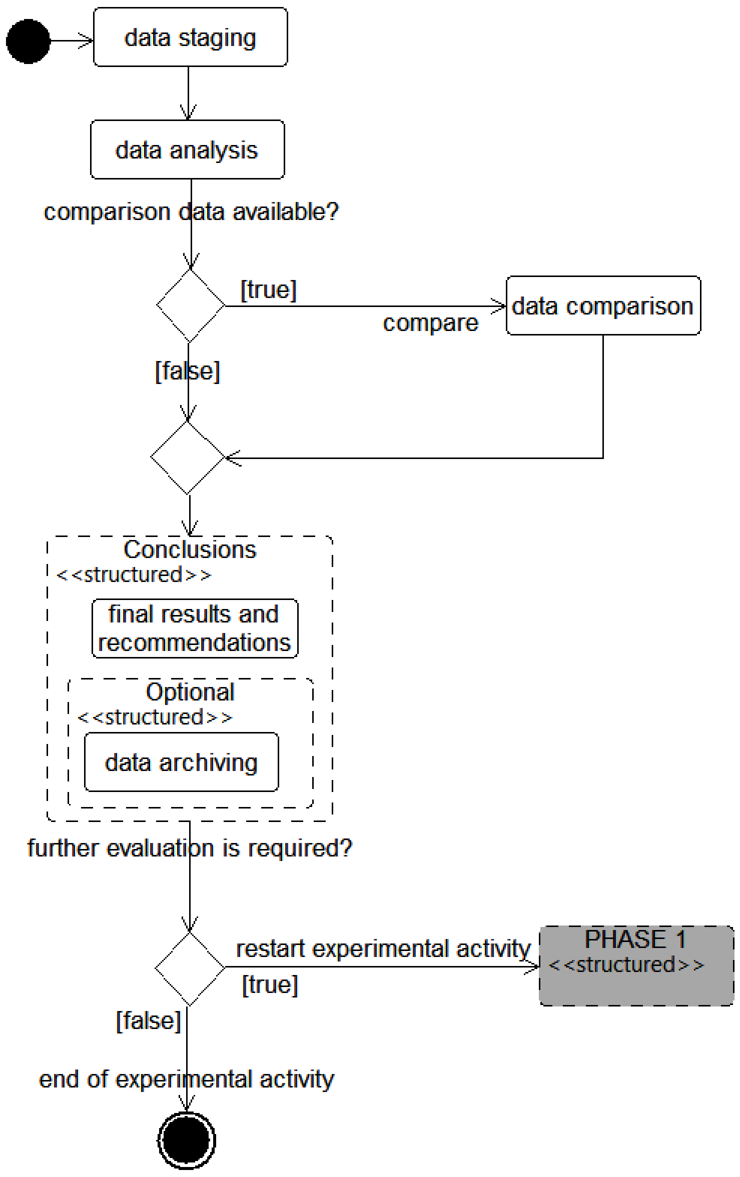


Figure 3.6: PHASE 4: Analysis and recommendations.

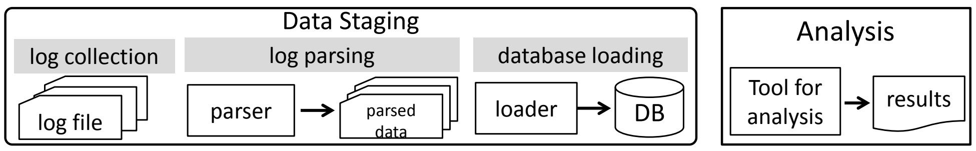


Figure 3.7: Data staging and analysis.

general-purpose tools for the analysis of datasets, can be profitably used here. Usage of OLAP technologies is also strictly connected to the structure of the database used to store results, which shall be defined according to the star schema planned in phase 1.

The step *data staging* is composed of three parts, that are log collection, log parsing and database loading (in Figure 3.7):

- *Log collection*. In log collection the logged events are collected (for example, these can be files located on distributed computers) and merged, creating a unique data log which contains the raw data.
- *Log parsing*. In log parsing the data log is parsed; raw data are extracted from the data log and transformed in parsed data, for example in file formats as CSV (Comma Separated Value; it is a standard data file format used for storage of data structured in table form [162]), that are easier to handle than the raw data. Log parsing include cleaning of the data (detect and correct, or remove, corrupted or inaccurate records), fill missing data, and put the data in a standard format easy to load in a database or manage. Also, integration of data taken from several sources (e.g., transforming timestamped events collected during the experiment execution into appropriate metrics) is performed. Note that log parsing may also include actions to make the data *anonymous*; this is particularly relevant in public repositories where industrial or research groups providing data do not want to disclosure their identity.
- *Database loading*. In database loading we create SQL (Structured Query Language, [71]) queries starting from the content of the parsed files to populate a data repository (e.g., a database or a data warehouse). As the trace files of experiments can be difficult to handle due to their size, and the information stored in each file can be structured in different ways, a structured data repository can ease storing, manipulation and retrieval of data.

Data analysis

Once experiments have been run, log files have been parsed and data have been loaded into the database, the data can be analyzed to investigate the results. As previously mentioned, a significant support can be provided by the usage of approaches based on OLAP and data warehousing to analyze and share the raw results stored in a common multidimensional structure (a star schema) [110].

3.5.2 *Data comparison*

Comparison of both raw data and final results is an important step as it allows to compare different solutions. A typical challenge is comparing data (collected analyzing the same or a different component) that were collected in different periods and/or by different groups. Often, quantities selected are different and consequently effective comparison requires the availability of the raw data.

OLAP tools and techniques (including the star-schema introduced previously), together with an attentive application of the methodology and the metrological characterization of the results, ease comparison and cross-exploitation of raw results from different experiments. In fact, results are stored in a common data warehouse (that can be available to the entire scientific community, or within the personnel of an organization), the experiments performed are attentively described, and the metrological characterization improve confidence in results and investigation of their *compatibility*.

3.5.3 *Conclusions*

After the attentive analysis of the data collected is performed, conclusions on the experimental campaign are drawn. This step is organized in two parts: *final results and recommendations*, to show a summary of results and main achievements of the experimental campaign and recommendations that can be derived and are proved by the results, and *data archiving*, which includes actions to store data (for example, sharing to the community or archiving them in the database of a factory, for further reuse).

These two parallel activity conclude phase 4; in case the outcome of the final recommendations establishes that further experiments are required, phase 1 is restarted.

Final results and recommendations

At the end of the experiments, final results are presented. These show summarizing information and key aspects of the analysis. In particular, given the objectives selected at the beginning of the experimental campaign, in this step it should be presented if and why the initial objectives are satisfied. As an example, a possible result is the completed tests plan and tests report according to the IEEE standard 829-1998 for Software Test Documentation [88].

Recommendations may be generic observations or specific indications. For example, i) they may indicate the need of additional experiments, requiring in this case a new execution of the experimental campaign (back to phase 1), or ii) they may provide inputs or feedbacks to the designers of a specific protocol, or iii) they can be in form of outcomes that are part of a V&V process, and consequently are part of a testing campaign performed to assess the proper behavior of the target system.

Data archiving

At the end of the analysis, measurement results collected can be archived. Sometimes, data can be archived in public repositories (see Section 2.4), while often they are internal to organizations (a factory or a research institution). However, in both cases it is important that measurement results are stored in an organized way, to be retrieved (and reused) with reasonably low effort whenever needed. Note that data may be required after years, or by people different from those which collected and stored the data: in such scenarios, reusing the data is efficient or feasible only if they were stored using a well-defined and documented structure. This is the reason why this dissertation put emphasis on data archiving and sharing. Independently from being public or not, carefully archived data can be further reused and explored, for example they constitute the technical background for future works (possibly not even envisioned at the time the experiments were performed) or they are used for comparison.

In this direction, in Chapter 7 the RACME (Resiltech Assessment and Certification MEthodology) solution is presented. RACME merges challenges of data archiving and retrieval, together with a methodological support to V&V processes. Briefly, RACME is an industrial framework that copes with the previous issues and proposes a methodology and related instruments to support V&V processes and certification activities. The RACME solution aims to support the V&V experts during the overall V&V process, showing the set of activities to be performed, organizing the relevant inputs and outputs (e.g., experiments results and tests report), detecting inconsistencies amongst different documents or missing elements, and finally helping the documents construction for certification purposes [40].

3.6 METHODOLOGY APPLIED TO THE CASE STUDIES

In the remainder of this Thesis, our methodology shall be applied to five different case studies, reported in Chapter 4, Chapter 5, Chapter 6, Chapter 8, and Chapter 9. Each of them presents different criticality, and consequently different care is devoted to the various aspects of the methodology. To give evidence of the application of the methodology, in this Section we discuss which steps have been deemed particularly critical for each case study.

The five case studies span on different areas, and are i) the experimental evaluation of a middleware service (a software clock) for resilient timekeeping (Chapter 4), ii) the design, implementation and exercise of an improvement to a measuring instrument for the analysis of distributed protocols (Chapter 5), iii) the design, implementation and exercise of a testing service for dynamic, highly adaptive Service Oriented Systems (Chapter 6), iv) the experimental evaluation of COTS GPS devices used for localization (Chapter 8), and finally v) the experimental evaluation of an embedded safety-critical system for train-borne equipment (Chapter 9). Case studies i) to iii) are "academic" case studies, built in our *lab* and specifically targeting different aspects of the methodology (i.e., the experimental evaluation process, the improvement of a pre-existing tool, and the construction of a completely new tool), while case study iv) and case study v) have been defined and carried out with tight cooperation of industries.

In the first case study, the methodology is applied to the *experimental evaluation* of a software clock. As this case study represents a complete process for the experimental evaluation (from the definition of the objectives till the conclusion of the evaluation activity), all steps and aspects of the methodology have been considered. Amongst them, the discussion in Chapter 4 devotes particular attention to the analysis of the metrological properties of the measurement tool, and of the quality of the expected results. Also, reuse of the measuring instrument, and sharing and comparison of results are relevant topics discussed; in fact results populate a database based on a star-schema, and scripts and tools are used to automatize data staging and analysis. This ease re-executing the experiments and comparing the results (in fact, in [21] the same set-up is used to evaluate and compare different versions of the target system). We remark that the experimental evaluation activity required some iterations, because preliminary runs were executed to understand generic trends and correct small problems (in particular, these allow to identify a systematic error, noticed during the first runs, see Section 4.3).

The second case study focuses on the *improvement of a pre-existing measuring instrument* for the evaluation of distributed protocols, rather than on the execution of an experimental evaluation campaign (although a case study is presented, it is mainly intended to show the effectiveness of the improvement to the measuring instrument and not the application of the whole methodology). The methodology here is applied mainly for the steps and phases related to the measuring instrument design and instrumentation. The main aspect to note is the particular care in the metrological analysis of the measuring instrument and of the expected results. The case study also proves the relevance of devoting attention to such topics; in fact it shows that underestimating potential sources of uncertainty may lead to misleading conclusions and recommendations.

The third case study focuses on the *design, implementation and exercise of a measuring instrument* for the experimental evaluation of Service Oriented Architectures (SOAs). This measuring instrument (a *testing service*) is designed to evaluate highly adaptive and dynamic systems; it is composed of several different parts, and its complexity is on a different scale than the previous measuring instruments. However, it still classifies as a measuring instrument for the experimental evaluation of critical system and consequently the methodology still holds. An important aspect reported in this case study is the proposed approach for the sharing of tests results; in fact, a database is maintained and accessed by people which use the testing service and benefit of the information it maintains.

The fourth and the fifth case study are *performed in cooperation with industries*. In this two case studies, the Genova and Torino bases of Ansaldo STS made available respectively the measuring instrument in the fourth case study and the target system in the fifth case study. This implies that the duration of phase 3 (and partially phase 2) was limited by the time we had access to the instruments and the prototype. Consequently phase 1 of the methodology acquired particular relevance as carefully planning the experimental campaign reduced the risk of (unexpected) difficulties during phase 2 and phase 3.

Going into details of the fourth case study, it is devoted to the experimental evaluation of GPS devices to understand their measurement error. The analysis presented in Chapter 8 allows to provide preliminary results and feedbacks to the designers of the overall system in which the GPS devices are used and for its V&V. Another aspect covered in this case study is the attention devoted to build a highly reusable database (again, the database was structured as a star schema) and scripts for its automatic population with the data collected.

Finally, the fifth case study is devoted to the experimental evaluation of a prototype of a safety-critical embedded system. The four phases of the methodology are attentively applied and described, as well as the metrological characterization of results. Given the short time slot (one week) in which the prototype was available for our tests (the prototype was held in an industry and not in our lab), there was limited opportunity to iterate on phase 2 and phase 3. Consequently, we devoted a particular care to the preliminary steps of the methodology (those steps that could be performed without having the prototype at our disposal), in order to optimize as much as possible our time in phase 2 and phase 3. Finally, we remark that amongst conclusive results of the analysis, an important recommendation was provided to the system designers to notify a potential flaw in the system (a duration slightly exceeding the timeout, see Section 9.5) and a related correction.

4 EVALUATION OF A MIDDLEWARE SERVICE

Clock synchronization to an external timebase is a common requirement for many pervasive and distributed systems. In some cases, a good clock synchronization is a crucial requirement, as failing to fulfill it can have a severe effect on the performance or even the safety of a system. Experimental evaluation of clocks and clock synchronization mechanisms is mandatory to prove formal theories [145], and typically requires a high quality clock for measures comparison and a particular care when defining the measuring system [181]. Different solutions are possible [60], [142], [181], [79], [118] depending for example on the environment and system in which clocks operate, the affordable costs for the equipment, the desired quality of results, etc.

In this Chapter we describe the experimental evaluation of a middleware component, namely the software clock Reliable and Self Aware Clock (R&SAClock, [23], [19]). We focus on the validation methodology and the assessment of the measuring system, including analysis of faultload and workload intrusiveness and representativeness, providing a set-up which can be reused for different instantiation of the tested software clock [20], [22].

The R&SAClock [23], [19] is designed to be self-aware of its synchronization offset from the reference time. When asked to provide the time, R&SAClock replies with an *enriched time value*, which also gives information on the confidence that can be associated to the time value. Thus, the system can at any time reliably predict its current offset. In the implementation considered here, the R&SAClock is a $C++$ middleware service with regard to external synchronization.

Experimentally verifying its performance is a key factor of success for a clock that is intended to be reliable in providing a tight upper bound to the offset synchronization. However, to validate the R&SAClock, which provides a synchronization uncertainty sometimes lower than 1 ms (millisecond), an accurate methodology is needed. Consequently, the validation test bed shown in this Chapter is based on a detailed analysis of which is the most suitable reference time instant to compare with R&SAClock output. Moreover, the experimental plan covers a relevant set of cases, including different values of the software clock parameters and different types of workload, and takes into consideration the possible occurrence of faults in the system under test and/or in the underlying synchronization mechanism.

The rest of the Chapter is organized as follows. Section 4.1 introduces the R&SAClock and the algorithm Statistical Predictor and Safety Margin (SPS, [19]) that computes synchronization uncertainty; these notions shall also be required in Chapter 5. Section 4.2 and Section 4.3 report the experimental evaluation methodology and the analysis of results, while Section 4.4 reports a discussion on the validation achieved in terms of degree of satisfaction of requirements, or identification of directions for improving the R&SAClock implementation. Further details can be found in [20], [22].

4.1 THE RELIABLE AND SELF-AWARE CLOCK

4.1.1 *Basic notions of time and clocks*

Let us consider a distributed system composed of a set of nodes. We define *global time* as the unique time view shared by the nodes of the system, *reference clock* as the clock that always holds the global time, and *reference node* as the node that owns the reference clock. Given a local clock *c* and any *time instant t*, we define $c(t)$ as the *time value* read by local clock c at time t. The behavior of a local clock c is characterized by the quantities *offset*, *accuracy*, *precision* and *drift*. The offset $\Theta_c(t) = t - c(t)$ is the actual distance of local clock c of the node n from the global time at time t [129]. This distance may vary through time. Accuracy A_c is an upper bound of the offset [182]; accuracy is often adopted in the definition of system requirements and therefore targeted by clock synchronization mechanisms. Drift $\rho_c(t)$ describes the rate of deviation of a local clock c at time t from global time [182]. Finally, precision π describes how closely local clocks remain synchronized to each other at any time. Figure 4.1 exemplifies the concepts of accuracy, drift, offset, precision and clock synchronization; the outside thick dashed lines represents the bound in the rate of drift, a fundamental assumption of clock synchronization mechanisms, since it allows to predict the maximum deviation after a given time interval.

Despite their theoretical importance, accuracy and offset are usually of practical little use for systems. Synchronization mechanisms typically compute an *estimated offset* $\widetilde{\Theta}_c(t)$ (and an *estimated drift* $\widetilde{\rho}_c(t)$), without offering guarantees and only at synchronization instants. We define the *synchronization uncertainty* (or simply *uncertainty*) $U_c(t)$ as an adaptive and conservative evaluation of offset $\Theta_c(t)$ at any time t; uncertainty is such that $A_c \geqslant U_c(t) \geqslant \Theta_c(t) \geqslant 0$ [23], [19].

4.1.2 *Specification of R&SAClock*

R&SAClock is a software middleware service that provides to users (e.g., system processes) both the time value and the synchronization uncertainty associated to the time value. When a user asks the current time to R&SAClock (by invoking the function *getTime*), R&SAClock provides an enriched time value:

$$[likelyTime, minTime, maxTime, FLAG].$$

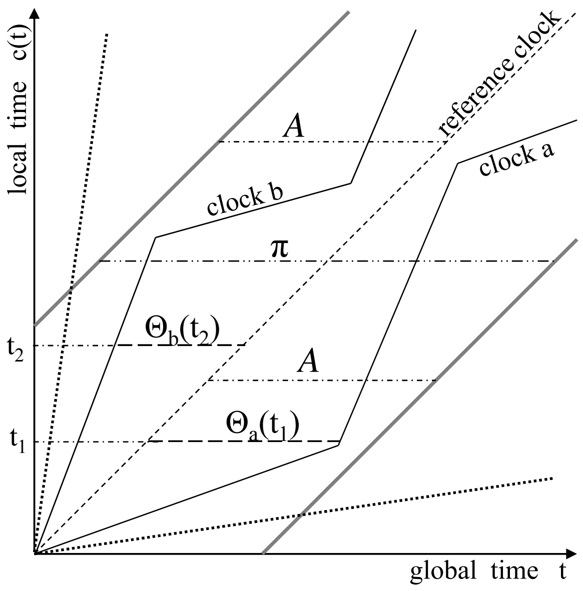


Figure 4.1: Time and clock.

LikelyTime is the time value computed reading the local clock $c(t)$. *MinTime* and *maxTime* are based on the synchronization uncertainty provided by the internal mechanisms of R&SAClock. *FLAG* is a Boolean value indicating whether the current synchronization uncertainty is within an accuracy bound set as a requirement by the user (the *FLAG* value is not considered in this Chapter; instead it shall be exploited in Chapter 5). Details on R&SAClock and its implementation can be found in [23], [21].

From the perspective of a user of R&SAClock, the main expectations are the following:

- a request for the time value should be satisfied quickly, and
- the enriched time value should include the true time.

These can be more formally expressed as:

- **REQ1.** The service response time provided by R&SAClock is bounded: there exists a maximum reply time Δ_{RT} from a *getTime* request made by a user to the delivery of the enriched time value (the probability that the *getTime* is not provided within Δ_{RT} is negligible).
- **REQ2.** For any *minTime* and *maxTime* in any enriched time value generated at time t, it must be $minTime \leqslant t \leqslant maxTime$ with a coverage Δ_{CV} (by coverage we mean the probability that this equation is true). In other words, given $likelyTime = c(t)$, the true time t must be guaranteed within the interval $[minTime, maxTime]$ with a coverage Δ_{CV}.

4.1.3 *The Statistical Predictor and Safety Margin (SPS)*

We briefly describe the SPS algorithm for the computation of synchronization uncertainty for a local software clock c that is disciplined by an external clock synchronization mechanism. A complete description of the algorithm can be found in [19]. To ease the readability of the notation, the subscript c is omitted from the expressions presented in the rest of this Chapter. In Table 4.1 the main quantities involved in the SPS are shown and explained.

Symbol	**Definition**
t_0	time in which the most recent synchronization is performed
$\widetilde{\Theta}(t_0)$	estimated offset at time t_0
$\widetilde{\rho}(t_0)$	estimated drift at time t_0
M, m	maximum and current number of (most recent) samples of the estimated drift that the UEA collects ($0 < m = M$)
N, n	maximum and current number of (most recent) samples of the estimated offset that the UEA collects ($0 < n = N$)
p_{ds}	probability that the population variance of the estimated drift is smaller than a safe bound on such variance
p_{dv}	a safe bound of the drift variation since t_0 is computed with probability p_{dv}
$p_{ds} \circ p_{dv}$	the joint probability of these two values represents the coverage of the prediction function
p_{os}	probability that the population variance of the estimated offset is smaller than a safe bound on the variance
p_{ov}	a safe bound of the offset at t_0 is computed with probability p_{ov}
$p_{os} \circ p_{ov}$	the joint probability of these two values represents the coverage of the safety margin function

Table 4.1: Main SPS quantities and parameters.

The set-up parameters used by SPS (detailed in Table 4.1) are: the four probabilities p_{ds}, p_{dv}, p_{os}, p_{ov} and the memory depth M and N. The coverage and the performance (how much the synchronization uncertainty is effectively tight to the estimated offset) achieved by SPS depend on these parameters.

We assume t_0 the time in which the most recent synchronization is performed: at time t_0 the synchronization mechanism computes the estimated offset $\widetilde{\Theta}(t_0)$ and possibly the estimated drift $\widetilde{\rho}(t_0)$ (if not provided by the mechanism, it can be easily computed by the R&SAClock itself).

Briefly, SPS computes the uncertainty at a time t with a coverage, intended as the probability that $A_c \geqslant U_c(t) \geqslant \Theta_c(t) \geqslant 0$ holds. The computed uncertainty is composed by three quantities: i) the *estimated offset* (computed by the synchronization mechanism), ii) the output of a *predictor* function P which predicts the behavior of the oscillator and continuously provides bounds that constitute a safe (pessimistic) estimation of the oscillator drift and iii) the output of a *safety margin* function SM which aims at compensating possible errors in the prediction and/or in the estimation of the offset. The computation of synchronization uncertainty requires a right uncertainty $U_r(t)$ and a left uncertainty $U_l(t)$: consequently, SPS has a right predictor and a right safety margin for right uncertainty, and a left predictor and left safety margin for left uncertainty. The output of the SPS at $t = t_0$ is constituted by the two values:

$$U_r\,(t) = \max(0, \widetilde{\Theta}\,(t_0)) + P_r\,(t) + SM_r(t_0) \tag{4.1}$$

$$U_l\,(t) = \min(0, \widetilde{\Theta}\,(t_0)) + P_l\,(t) + SM_l(t_0). \tag{4.2}$$

The *estimated offset* $\widetilde{\Theta}(t_0)$ is computed by the synchronization mechanism and can contain errors. If the estimated offset is positive, it influences the computation of an upper bound on the offset itself and consequently is considered in equation 4.1. If it is negative, it is ignored. A symmetric reasoning holds for equation 4.2.

The *predictor* functions (left and right) predict the behavior of the oscillator and continuously provide bounds (lower and upper) which constitute a safe (pessimistic) estimation of the oscillator drift and consequently a bound on the offset. The oscillator drift is modeled with the random walk frequency noise model, one of the five canonical models used to model oscillators (the power-law models [17]), that we considered as appropriate and used. Obviously the parameters of this random walk are unknown and depend on the specific oscillator used. We compute them resorting to the observation of the last m samples of the drift (where m smaller or equal to the set-up parameter M), and using a safe bound on the population variance of the estimated drift values. The coverage of this safe bound depends on the set-up probabilities p_{ds} and p_{dv} (see Table 4.1).

The *safety margin* functions (left and right) aim at compensating possible errors in the prediction and/or in the estimation of the offset. The safety margin function is computed starting from the collection of the last n samples of the estimated offset (where n is smaller or equal to the set-up parameter N). A safe bound to the population variance of the estimated offset is computed. The coverage of this safe bound depends on the set-up probabilities p_{os} and p_{ov} (see Table 4.1).

4.2 PLANNING AND INSTRUMENTATION

We present the main outcomes and results of the application of the methodology for phase 1 and phase 2.

4.2.1 *System under test*

The target system consists of an R&SAClock prototype, which is installed as a software component on a computer, namely PC_R&SAC. In the system under test, the local software clock is synchronized through the Network Time Protocol (NTP, [129]). An NTP client (process daemon) running on PC_R&SAC synchronizes the local clock using information from the NTP server(s). PC_R&SAC is connected to one or more NTP servers through the Internet.

4.2.2 *Workload, faultload and experiments design*

A wide set of experiments are required in order to derive a meaningful validation. The parameters to set for the experiments are:

1. the numbers of samples N and M, which represent the memory depth of the algorithm and are chosen as $M = N$ within the set $[10, 40, 80]$;
2. the probabilities p_{ds}, p_{dv}, p_{os} and p_{ov}, which represent the confidence levels of the synchronization uncertainty computed by R&SAClock and are always chosen as $p_{ds} = p_{dv} = p_{os} = p_{ov} = p$;
3. the workload, which is expressed in terms of *getTime* requests per second to R&SAClock;
4. the faultload.

The selection of a set of meaningful faults is based on the analysis of the criticality of the algorithm: either the synchronization mechanism fails in estimating the offset or the poor quality of the local clock. The first criticality may be due either to:

1. a failure of the communication channel, which is emulated by closing the port 123 that is used by NTP;
2. network delays and variable latency in the communication with the NTP server(s), which are emulated by retaining the NTP packets and artificially introducing a random delay on them, or
3. a crash of the process that manages the synchronization, which are emulated by killing the NTP process running on the client or on the server.

The second criticality can stem from extreme operating conditions, such as quick temperature variations, emulated through the Linux primitives *adjtime*. Possible synchronization attacks are not explicitly considered, but are treated as asymmetry in the communication delay with the NTP server(s).

4.2.3 *Definition of the measuring instrument*

The key of the validation methodology is to trigger R&SAClock asking the time and then verify that the time t is actually within the interval (*minTime*, *maxTime*) defined by the enriched time stamp. The final goal is to evaluate an experimental coverage level for the implementation of R&SAClock under test (i.e., on a specified node in a given network situation) under various operating conditions involving: (i) different parameter values; (ii) different types of workload for the node where R&SAClock is installed and (iii) for R&SAClock itself (intended as number of time requests in the unit of time) and, finally, (iv) the possible occurrence of faults of different nature in the system under test, including NTP. We are also interested in verifying the performance of R&SAClock, and therefore we also need to evaluate the time it takes to answer to a time request.

The basic components of a system for the validation of a software clock are:

- the system under test in which the clock is installed;
- a reference clock;
- a controller of the experiments which decides the workload and the faultload to use, and controls the experiments;
- a monitor that collects the results (i.e., the information on the current time) given by the system under test and verifies their concurrence with the time provided by the reference clock;
- the probes inserted in the system under test.

Being this a software clock, the probes are basically software instructions inserted in the code of the node where the software clock is running, which are mandated to capture certain events of interest and timestamp them.

Each time a client program makes a *getTime* request to R&SAClock, a software probe timestamps the request and another probe timestamps the reply of R&SAClock. Then, the *enriched time value* is compared to the time provided by the reference clock.

Before describing the experimental set up, it is worth discussing which time instant the *enriched time value* given by R&SAClock has to be compared to.

The correct way to proceed is not to think at R&SAClock as a (software) device designed to answer the question "what time is it?", regardless of the practical use of the output it gives. Performing a meaningful validation of R&SAClock means verifying if R&SAClock works properly and to what extent it is useful. This means taking the time from the reference clock when R&SAClock provides its answer, rather than when the question is made. In fact, we have to verify the fulfillment of the requirement REQ2 at the moment the time information is made available by R&SAClock to the client program.

The design and implementation of the validation testbed follow three basic rules: granting a time resolution sufficiently lower than that of the system under test and keeping the software probes as simple as possible in order to reduce the intrusiveness on the system under test and ultimately the uncertainty of the measurement results.

The measuring instrument and the system under test are shown in Figure 4.2. The choice of keeping the monitoring system and R&SAClock on different nodes is justified by the need of minimizing the intrusiveness of the monitoring system on the operative system of the node the R&SAClock is installed on. For the same reason, the option of having the reference clock as a second clock on the same node of R&SAClock is not considered. In addition, when multiple validations are to be made, it is more practical not to install a second (reference) clock on all the systems under test.

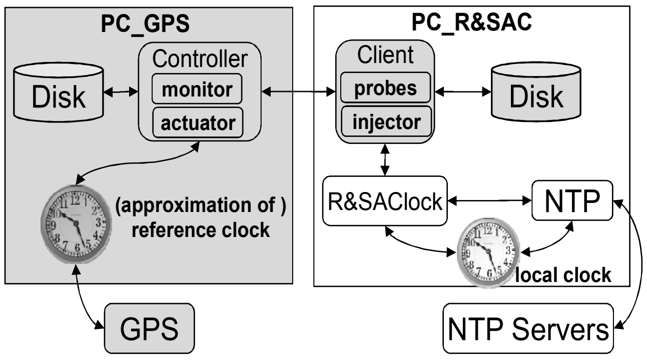


Figure 4.2: The measuring instrument and the system under test.

Our choice is to have a node (PC_GPS) including the reference clock and the monitoring system. It is a HP Pavilion desktop. Being synchronized to a high quality GPS receiver, the PC_GPS local clock is suitable to act as the metrological reference clock as it is orders of magnitude closer to the true value of current time than the clock of the target system.

The monitoring system consists of a software component for the control of the experiment (*Controller* hereafter) that is composed of an actuator that triggers the Client to make it request the enriched time value to R&SAClock, and of a monitor that receives and processes the information received from the Client about the completion of the *getTime* requests, accesses the reference clock, and writes data on the disk. The Client is a software component located on the target system, which performs injection functions to inject the faultload and to generate the workload, and probing functions to collect the relevant quantities and write this data on the disk.

PC_R&SAC is a Linux PC connected to (one or more) NTP servers by means of an Internet connection and to the PC_GPS (another Linux-based PC) by means of an Ethernet crossover cable.

The Controller and the Client are two high-priority processes that communicate using a socket. Figure 4.3 shows their interactions to execute the workload. The Client waits for Controller's commands. At periodic time intervals, the Controller sends a message containing a *getTime* request and an identifier ID to the Client, and logs ID and the *Controller.start* timestamp. When the Client receives the message, it logs ID and the *Client.start* timestamp and performs a *getTime* request to R&SAClock. When the Client receives the enriched time value from R&SAClock, it logs the enriched time value, the *Client.end* timestamp and ID, and sends a message containing an acknowledgment and ID to the Controller. The Controller receives the acknowledgment and logs ID and the *Controller.end* timestamp. At the end of the experiment, the log files created on the two machines are combined pairing entries with the same ID and data are processed.

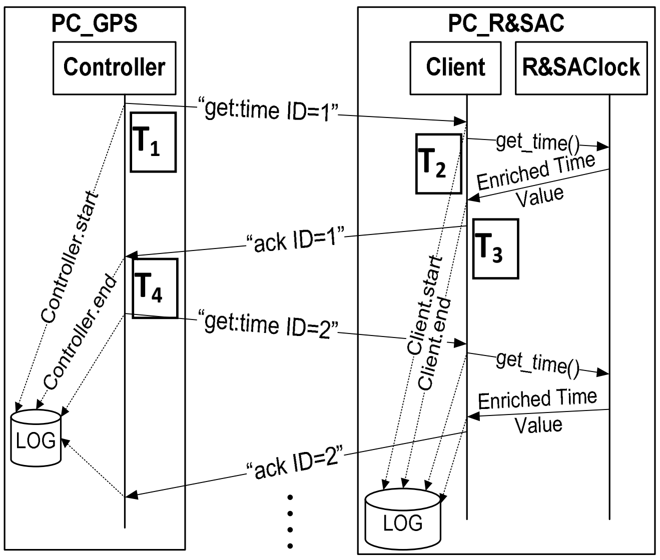


Figure 4.3: Controller, Client and R&SAClock interactions to execute the workload.

Controller and Client interact to execute the faultload as follows. The Controller sends to the Client the commands to inject the faults (that are enlisted in the following), and logs the related events. The Client executes the received command and logs the related event. Data logging is handled by NetLogger [73], which guarantees negligible intrusiveness.

As stated before, the enriched time value should be compared to the reference time at the instant T_3 i.e., $t(T_3)$. For $t(T_3)$, the following relation holds (see also Table 4.2 for a definition of the symbols used):

$$t(T_3) \in (t(T_1) + \delta_1 + \Delta_{23}, t(T_1) + \Delta_{14} - \delta_2 - \tau). \tag{4.3}$$

Symbol	**Definition**
T_1	time instant when the getTime request is started from the PC_GPS
T_2	time instant when the getTime request is started from the Client towards R&SAClock
T_3	time instant when R&SAClock completes the evaluation of the enriched time value
T_4	time instant when the enriched time value reaches the PC_GPS i.e., the PC_GPS comes to know that R&SAClock has calculated the enriched time value
$t(T_i)$	reference time at the time instant T_i
δ_1, δ_2	minimum transmission time (respectively, from PC_GPS to PC_R&SAC and vice-versa)
τ	time elapsed between $t(T_3)$ and the beginning of the transmission plus the time elapsed between the reception of the ack at PC_GPS and the actual timestamping instant
Δ_{xy}	time interval $\lvert t(T_y) - t(T_x) \rvert$

Table 4.2: Time instants and time intervals involved in a *gettime* request.

If the hypothesis of $\delta_1 + \delta_2$ being much smaller than Δ_{23} is correct, it is possible to reduce the uncertainty on $t(T_3)$ to a small interval. Thus, the fulfillment of REQ2 can be verified by comparing the output (*minTime*, *maxTime*) with the interval $(T_1 + \delta_1 + \Delta_{23}, T_4 - \delta_2 - \tau)$ shown in Figure 4.4. T_3 is assumed to be the median of this interval. The main contribution to the uncertainty on T_3 is given by the resolution, that is the amplitude of the interval where T_3 falls. In the experiments such interval has come out to be of the order of $100\mu s$ (microseconds).

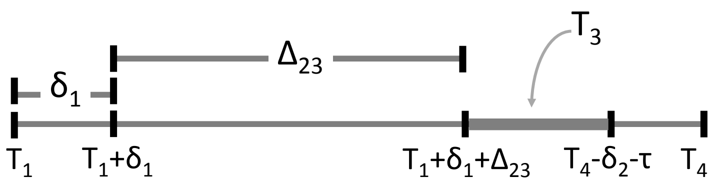


Figure 4.4: Time interval containing T_3.

A reasonable hypothesis underlying equation 4.3 is that the delay between any T_i and the time its corresponding timestamp is taken, is the same for any i. Moreover, it should be noted that Δ_{23} and Δ_{14} are measured on different machines and, therefore, the interval in equation 4.3 could come out to be empty (due to severely different drifts). In such a case, the monitoring system can estimate $t(T_3)$ by subtracting δ_2 and τ from the time provided by its clock (i.e., the reference clock) at T_4, when it receives the ack:

$$t(T_3) = t(T_4) - \delta_2 - \tau \qquad (4.4)$$

4.2.4 *Preparation of the structure of the results*

In the star-schema of Figure 4.5, the facts table is R&SAClock_FACTS: the table contains an entry for each enriched time value request performed by the monitoring system. Each entry is composed of the following values: *MONITOR.start*, *MONITOR.end*, *Client.start*, *Client.end*, *likelyTime*, *minTime*, *maxTime*, and *flag*.

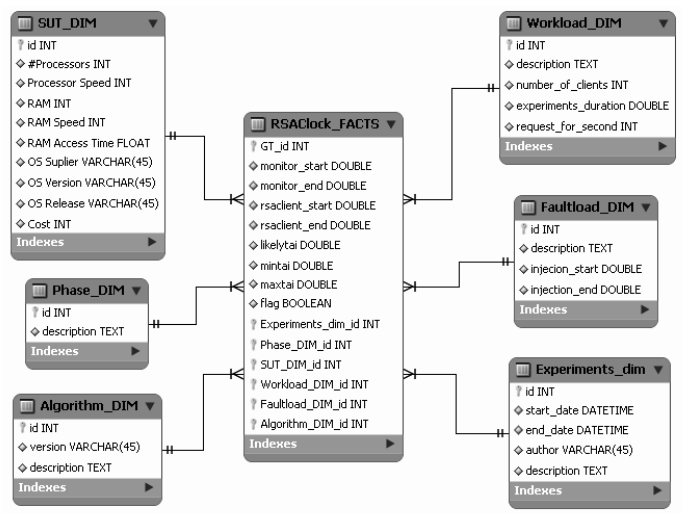


Figure 4.5: Results presented by a star-schema.

The dimensions tables are six: Phases_DIM, Workload_DIM, Faultload_DIM, Experiment_DIM, SUT_DIM and Algorithm_DIM. Phases_DIM contains the identified scenarios, Workload_DIM contains the workload parameters (e.g., experiments duration and number of *getTime* requests per second), Faultload_DIM contains the faultload used in the experiments, Experiment_DIM contains information on the experiments, SUT_DIM contains the description of the system under test, and Algorithm_DIM contains the description of the synchronization uncertainty algorithm in use.

4.3 analysis of results

We describe the offline analysis of measurement results collected. We subdivide the analysis in two phases: i) a data staging phase to structure and integrate the logged events in a database, and ii) the analysis of measurement results.

The data staging phase is composed of three steps: log collection, log parsing and database loading. In log collection the logged events are merged in a unique log file using NetLogger's API (Application Programming Interface). In log parsing we use an AWK script (AWK is a programming language for processing text-based data [153]) to parse raw data and create CSV files, that are easier to handle than the raw data. In database loading we create SQL (Structured Query Language [71]) queries starting from the content of CSV files to populate the database. This structure allows comparison of data collected from different experiments and target systems. For example, in [21] it is used to support comparison of different versions of the R&SAClock.

In the following, after discussing some preliminary results that allow to single out the optimal choice for the synchronization period, which is one of the parameters of NTP, some results are commented with the aid of figures trying to put in evidence their dependence from the parameters of the SPS algorithm and of the experimental setup. Then, the behavior of R&SAClock when some realistic faults are injected and under stress conditions is evaluated.

The results are expressed in terms of *coverage* and *performance*. The former, which permits to evaluate the fulfillment of REQ2, is calculated as the ratio of the time the algorithm is wrong (i.e., the reference time instant does not fall within the interval provided by the algorithm) to the observation time. The latter is evaluated as the ratio of the average distance between *likelyTime* and *globalTime* to the average width of the interval provided by the algorithm. As regards the fulfillment of REQ1, Section 4.3.6 includes an analysis of the response time of R&SAClock.

For each experiment, both the coverage and the performance are separately evaluated for the first 6 hours and the last 4 hours of each experiments (*transient* and *steady-state*, respectively).

We performed the experiments using as PC_R&SAC three different PCs (from our lab) of different factories and quality, executing different Linux versions and different NTP versions (4.0 or newer). The three PCs show a drift that varies from 22 ppm (parts-per-million) to 61 ppm.

As stated before, we have to check REQ2 with respect to the time instant T_3 i.e., REQ2 is fulfilled if equation 4.3 is verified. Actually, R&SAClock reads its local clock at an instant between T_2 and T_3, namely T^*, and then processes its data to compute the enriched time stamp and make it available to the client at T_3. This introduces a systematic error in the *likelyTime* equal to the deterministic component of $T_3 - T^*$, which depends from R&SAClock implementation. The results of our very first experiments gave evidence of this systematic error.

Therefore, according to the GUM [95], which states that any systematic error should be estimated and corrected, the R&SAClock has been modified as follows. Just before giving its output result, R&SAClock reads the local clock once again to estimate $T_3 - T^*$ and adds this corrective term to the enriched time value provided to the client.

4.3.1 *Effects of the synchronization period*

Preliminary executions of the SPS algorithm made evident that the SPS fails to satisfy its coverage requirement when NTP is configured to perform sparse synchronization (an average of one synchronization every 10 minutes). The problem is due to the assumption that the offset (and the estimated offset computed by NTP) fluctuates around zero.

While in a long-term observation period it is reasonable to model the offset as a zero-mean normally distributed random variable, when NTP performs sparse synchronizations it is likely that the short-time average of the offset is different from zero. The fact that NTP keeps the assumption of the offset fluctuating around zero thus leads to an error in the offset estimation. Though small (hundreds of microseconds in our runs of the experiments), this error is sufficient to make the global time continuously outside of the interval (*minTime*, *maxTime*).

In Figure 4.6 an example of this behavior is given. The configuration parameters are $p = 1 - 10^{-6}$, $M = N = 40$, 1 request per second (req/s). The central line, marked as *likelyTime*, is the offset, the upper and lower lines, marked as *maxTime* and *minTime*, are respectively the difference between *maxTime* and the reference and between *minTime* and the reference time. NTP is set to have variable synchronization periods between 16 s, which is the minimum possible value, and 128 s.

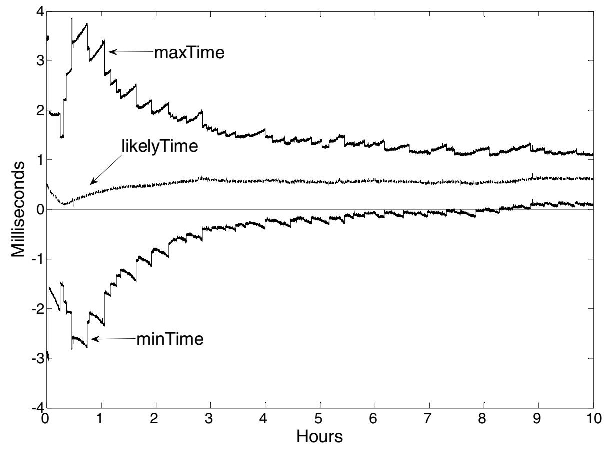
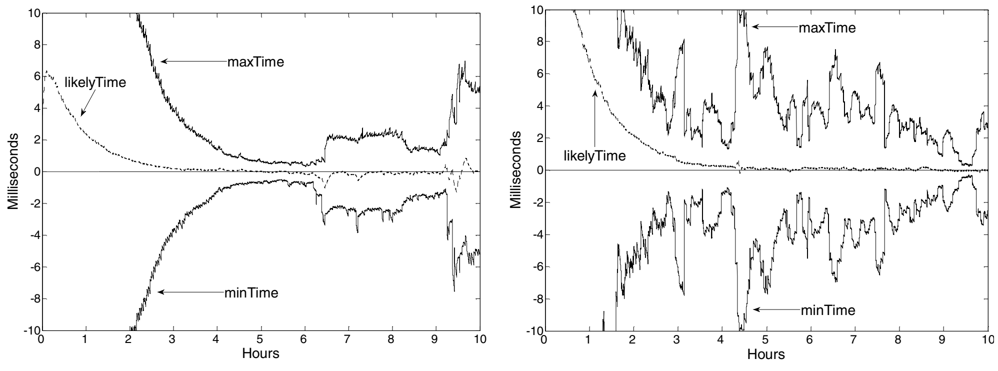


Figure 4.6: Variable synchronization period.

At the beginning, *minTime* and *maxTime* converge towards *likelyTime*. After hour 8, the global time is continuously outside of the interval (*minTime*, *maxTime*). In order to avoid such a poor performance, in all the subsequent experiments, the synchronization period of NTP has been chosen equal to 16 s.

4.3.2 *Effects of the algorithm memory depth*

Figure 4.7 shows how R&SAClock behavior varies with its memory depth (we vary the parameters M and N). The other configuration parameters are $p = 1 - 10^{-6}$, while the workload is 1 req/s.



(a) Memory depth $M = N = 80$. (b) Memory depth $M = N = 10$.

Figure 4.7: Results for different values of the memory depth.

Figure 4.7a shows the results when the algorithm processes the last 80 offset and drift samples ($M = N = 80$), whereas Figure 4.7b refers to $M = N = 10$. It is evident that a deeper memory permits to keep the synchronization uncertainty low when the local clock is slightly unstable (i.e., when the *likelyTime* line is close to the zero). In fact, the distance between *minTime* and *maxTime* is much lower in Figure 4.7a than in Figure 4.7b. When only 10 samples are processed, in fact, any small variation in the offset or in the drift provokes a great increase in the synchronization uncertainty computed by R&SAClock.

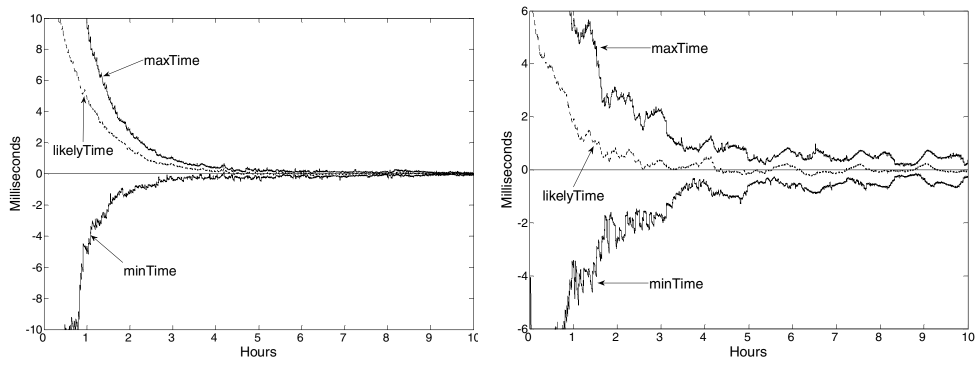
However, after the transients, that is either at the beginning or after the local clock has experienced some instability (see hour 9 in Figure 4.7a), the algorithm takes more time to reduce the computed uncertainty when it has a deeper memory, which may not be desirable.

This is the reason why 40 samples ($M = N = 40$) appears to be the most adequate choice, unless some *a priori* knowledge on the clock behavior is available.

4.3.3 *Effects of the synchronization uncertainty confidence level*

As stated before, the choice of the probability p is directly linked to the confidence level for the synchronization uncertainty computed by R&SAClock. Figure 4.8 shows the results for $p = 0.99$ and $p = 0.9999$, the workload being 1 req/s, and the memory depth being 40 samples.

When the confidence level p is 0.99, the interval $[\mathtt{minTime}, \mathtt{maxTime}]$ is much narrower than when p is 0.9999. However, this implies that here and there such interval may fail to contain the reference time i.e., it does not include the zero. On the contrary, a wider interval more likely includes the zero, but represents a wider synchronization uncertainty.



(a) Coverage level $p = 0.99$. (b) Coverage level $p = 0.9999$.

Figure 4.8: Results for different values of the coverage level.

4.3.4 *Summary of the results*

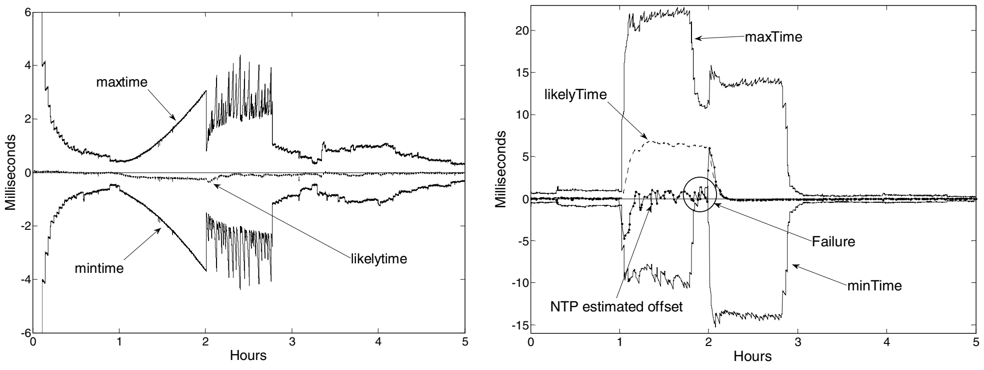
Table 4.3 summarizes the results of the experiments carried out with one of the PCs, after fixing the NTP synchronization period equal to 16 s. For the sake of brevity, similar outcomes that are achieved with the two other PCs are not enlisted. As expected, the coverage decreases with the confidence level p. In all the cases, the coverage is definitely very good, as it is higher than 0.99 even for $p = 0.99$. Thus, REQ2 can be considered as fulfilled with success. Regarding the performance index P.I., it increases when p is lower. As regards the choice of the number of samples to process, the figures confirm that 40 is by far the best option, as it grants good performance both in the transient and the steady-state phases, whereas a choice of $M = N = 80$ has extremely poor performance in the transients, and only a slight enhancement in the steady-state.

p	M, N	Transient (start-up) phase		Steady-state phase	
		C	P.I. $[\cdot 10^{-3}]$	C	P.I.$[\cdot 10^{-3}]$
0.999999	10	1	0.403	1	11.63
0.999999	40	1	73.095	1	25.90
0.999999	80	1	0.012	1	29.10
0.9999	40	1	5.618	1	111.1
0.999	40	1	52.225	0.9999	52.44
0.99	40	0.992	68.758	0.9949	32.40

Table 4.3: Coverage and performance results.

4.3.5 *Experiments in the presence of faults*

Failure in the communication. After the experiment has been running for one hour, the port 123 is closed, thus the NTP client is still active on the PC and disciplines the clock, but on the basis of "old" data about the offset and the drift. In such a case, R&SAClock increases the synchronization uncertainty until fresh data are available, but keeps on providing correct results, since the reference time is within the output interval (see Figure 4.9a, which is related to $p = 0.9999$, from hour 1 to hour 2). After hour 2, the port is re-opened and NTP has fresh data to process. Thus, after about 45 minutes, that is the transient needed to collect $M = N = 40$ new samples (considering the actual sample collection period is about 4 times the NTP synchronization period), the synchronization uncertainty dramatically decreases.



(a) A failure occurs in the NTP communication. (b) A temporary channel congestion occurs.

Figure 4.9: Results of injection experiments.

While R&SAClock can go on working properly even if the communication between the NTP client and the server(s) goes down for one hour, killing the NTP process implies a quick loss of synchronization, even with $p = 1 - 10^{-6}$. This happens also when the NTP process is killed after one hour of correct functioning. Such behavior could be easily expected, as now the clock is not conditioned anymore, whereas in the previous case it went on being conditioned, even if on the basis of old data.

Channel congestion. In this experiment, packets incoming on the port 123 are randomly delayed of a time interval uniformly distributed between 10 ms and 20 ms. In other words, we are making NTP work in the conditions of an asymmetric channel.

What happens in this case is that NTP gets convinced that it has to correct the clock of an amount of time which is around 7 ms, not surprisingly about half the average time we have delayed the packets. R&SAClock is unable to keep the reference time within its output interval: Figure 4.9b shows the results obtained for $p = 1 - 10^{-6}$, when after one hour we start to delay the incoming packets. The dot-marked line is the offset estimated by NTP. NTP assumes the communication channel is symmetrical and corrects the clock in the wrong way. Then, the confidence interval becomes narrower and narrower, until after 1 hour 50 minutes from the beginning of the experiment, the reference time goes outside R&SAClock output interval (the circle labeled as *failure*). Then, after hour 2, we stop delaying the packets. R&SAClock first widens the confidence interval as a consequence of the transient, but finally converges.

Poor quality of the local clock. In this experiment, after the first hour, the drift of the local clock is modified every 60 s of a random quantity between −25 ppm and 25 ppm. Then, after a further hour, the local clock is left to its own behavior till the end of the experiment. With a confidence level $p = 1 - 10^{-6}$, R&SAClock succeeds even with such a poor clock ($coverage = 1$). With $p = 0.99$, the coverage is 0.999.

4.3.6 *Behavior under stress conditions*

We conclude our analysis investigating the behavior of R&SAClock under stress conditions, created by means of a suitable workload generator [169]. The goal is to evaluate the ability of R&SAClock to satisfy very frequent requests and thus these experiments focus on the implementation of R&SAClock software prototype itself. In experiments of the duration of 30s, when multiple clients are making *getTime* requests to R&SAClock, the mean response time of R&SAClock showed a regular behavior, which doubles when the system CPU is stressed. Regarding the throughput i.e., the number of *getTime* request completed in the unit of time, it is seriously reduced when the CPU is stressed. However, for a given operating condition of the CPU (idle or stressed), the throughput does not seem to be affected by an increase in the number of concurrent clients, as long as this is below 50.

4.4 CONCLUSIONS AND RECOMMENDATIONS

We applied the methodology for experimental evaluation to validate a software clock that is designed to be aware of its synchronization uncertainty. The proposed methodology permits to achieve a resolution of the order of hundreds of microseconds.

The coverage of the software clock as it comes out from the experiments is excellent, except when the communication between the NTP client and its server(s) is of extremely poor quality. What is really satisfying is the 100% coverage experienced when the confidence level is above 0.999. This is, R&SAClock is suitable for applications needing a very high coverage, provided the parameters are adequately chosen.

As regards the requirement of a prompt response by the clock, it is usually of the order of 1 ms for a single client.

Future work should be oriented to modify R&SAClock in order to reduce its processing and execution time, and to adapt the validation set-up in order to validate R&SAClock in scenarios when it is synchronized through a GPS receiver instead of NTP, calling for a further reduction of the validation methodology resolution.

The measuring instrument developed and the available solution for data parsing, loading and analysis can be reused to ease comparison of data, re-execution of the experiments, and (well-organized) data archiving. An example can be found in [21], where two different algorithms to compute synchronization are compared.

5 TIME MEASUREMENTS IN DISTRIBUTED PROTOCOLS

In this Chapter we focus on the design, implementation and metrological assessment of a new tool for dependability measurements in distributed protocols, which allows the user evaluating the uncertainty of measurement results involving time interval measurements, and a related experimental campaign performed to collect distributed round-trip delays on a Wide Area Network (WAN).

The tool comes from the integration of Neko/NekoStat, a powerful highly portable Java framework for the analysis of distributed algorithms [179], [63], with a Java/C++ version of the R&SAClock. Details on the R&SAClock and fundamentals on time and clocks applied in this Chapter are presented in Chapter 4. The tool is presented along with an experimental case study [26], [25].

With respect to the methodology presented in Chapter 3, this case study focuses more on the design and implementation of the measuring instrument and on the collection of trusted measurement results than on the execution of experiments and presentation of results. This is why in the following of the Chapter we do not detail some of the phases that are part of the methodology, but instead we concentrate on tool design, implementation and assessment.

The following of this Chapter is organized as follows. In Section 5.1 the Neko and NekoStat tool, that are the measuring instruments used, are presented; in Section 5.2 the improvement of NekoStat to be aware of uncertainty in time measurements is shown, and in Section 5.3 a case study where the measuring instrument is exercised to evaluate a simple distributed protocol is illustrated. Conclusions and recommendations are in Section 5.4.

5.1 THE NEKO AND NEKOSTAT TOOL

Neko [179] is a simple but powerful highly-portable Java framework that allows defining and analyzing distributed algorithms. One of its most interesting features is that the same Neko-based implementation of an algorithm can be used for both simulations and experiments on a real network. The architecture of Neko can be divided into three main layers, which are from the top layer to the bottom layer (Figure 5.1):

- *applications*, which are developed by the programmer/user,
- *NekoProcesses*, which are the core of the testing framework, and

- *networks*, which are network interfaces (towards real or simulated networks).

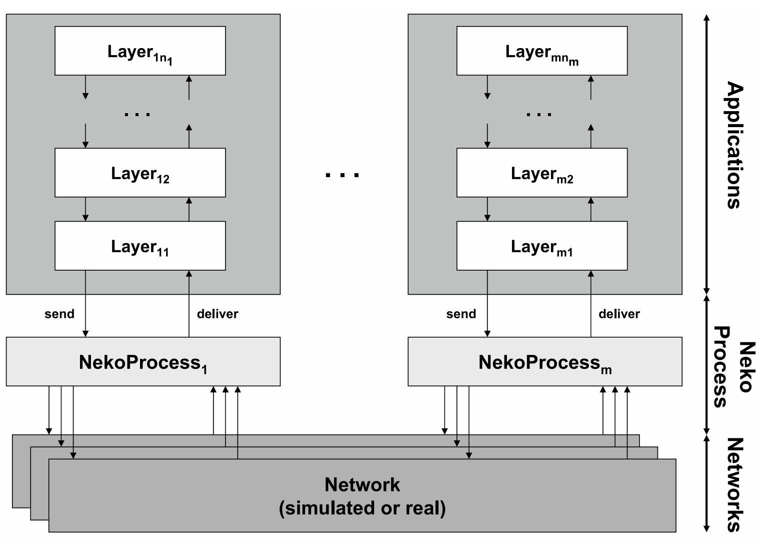


Figure 5.1: The Neko architecture [179].

Neko can be used to tests distributed algorithms and systems. It allows to test a system searching for *qualitative* ("on/off") properties, such as termination, validity, integrity, and agreement.

The tool is equipped with supports to obtain execution traces, both on simulated and on real environments. The potentialities of the Neko tool in the rapid prototyping of distributed algorithms are thus evident: the possibility to use simulated networks allows analyzing the algorithm in different conditions (variable transmission delays, different probabilities of message losses, network congestion, etc.) and, after that, it is possible to test the algorithm in real environments. Neko is thus very useful and versatile to test new algorithms, or to compare already existing ones. Neko also allows performing fault injection experiments at the network level as well as at the level of communications between layers, and thus it can be used to study the behavior of the analyzed algorithm with respect to specific injected faults or exceptional situations.

NekoStat is an extension to Neko that provides it with the ability to collect events and to analyze them by means of statistical and mathematical tools [63]. Following the same idea underlying Neko, through NekoStat it is possible to perform *quantitative* analysis of distributed algorithms, using both simulative and experimental approaches.

The NekoStat functionalities and the related components implementing them can be subdivided into two sets: mathematical functionalities, that handle the numerical quantities, and analysis functionalities, that collect and analyze distributed events. The implementation of the mathematical functionalities is the same both for simulation and real executions, whereas analysis supports are internally different, still with a common interface. The evolution of experimental analysis with NekoStat can be subdivided into different phases. In the first phase the application layers and the EventCollectors are activated; at the occurrence of an event, the application layer calls the StatLogger, which saves the event in the local EventCollector.

At the termination of the experiment run, the StatLogger of the slaves sends the local EventCollector to the master. The master can thus construct the global history, merging all the events of the EventCollectors. At this point the last phase of the analysis can start; the master StatLogger calls repetitively the StatHandler with the information of every event of the global history. The same StatHandler can thus be used both for simulative and experimental analysis of an algorithm (Figure 5.2).

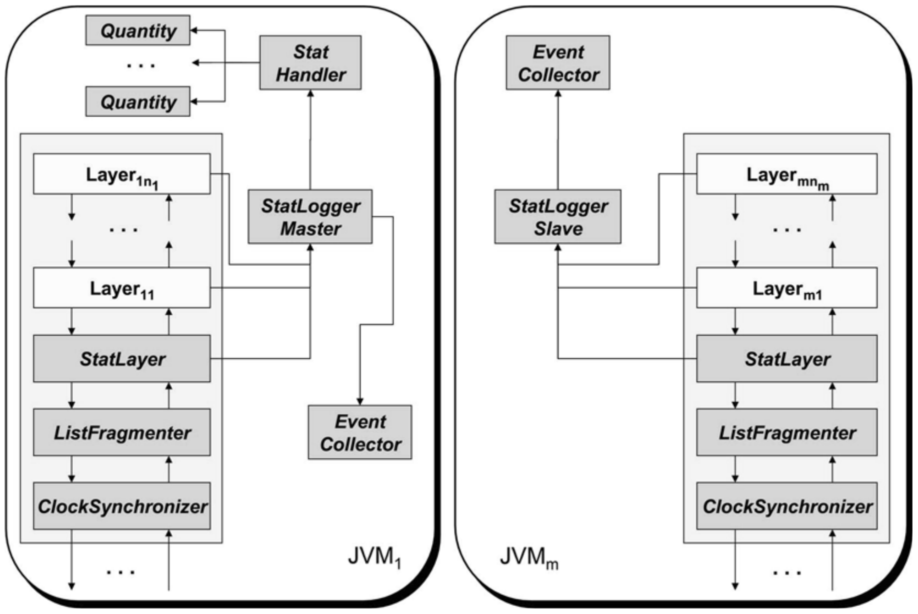


Figure 5.2: Scheme of NekoStat functioning.

The main limitation of NekoStat as an instrument for dependability measurements on distributed systems is that it does not allow evaluating the quality of the measurements it performs; as is, NekoStat cannot be qualified as a suitable measuring instrument. Apart from the formal metrological correctness, there is the practical risk that the tool collects measurement results which are not reliable, without discarding them before the data processing.

In order to enhance NekoStat to overcome the limitations previously mentioned, attention has been focused on time interval measurements, which constitute a large majority of the direct measurements carried out by people working in the field of dependability, as stated in Part i. When measuring time intervals in distributed systems, the most serious threat is represented by poor alignment of distributed clocks to a (unique) global time view.

Simply using a synchronization mechanism like NTP does not allow to fully trust the resulting offset, nor provides guarantees on the stability of the synchronization along time. This is the reason why using time values obtained by local clocks to compute time interval measurements, even if they are controlled by a synchronization mechanism, can be unsatisfactory in many evaluation tools. To face this problem, the NekoStat analyzer has been integrated with the R&SAClock component described in Chapter 4.

5.2 IMPROVEMENT OF THE MEASURING INSTRUMENT NEKOSTAT

The proposed measuring instrument is an integration of NekoStat with the R&SAClock software component. With reference to the classification of the means to attain dependability, the proposed tool can be labeled as a quantitative instrument for fault forecasting that can be also used for fault removal. As depicted in Figure 5.3, the R&SAClock substitutes the typical NekoStat clock, that is a virtual clock instantiated by NekoStat.

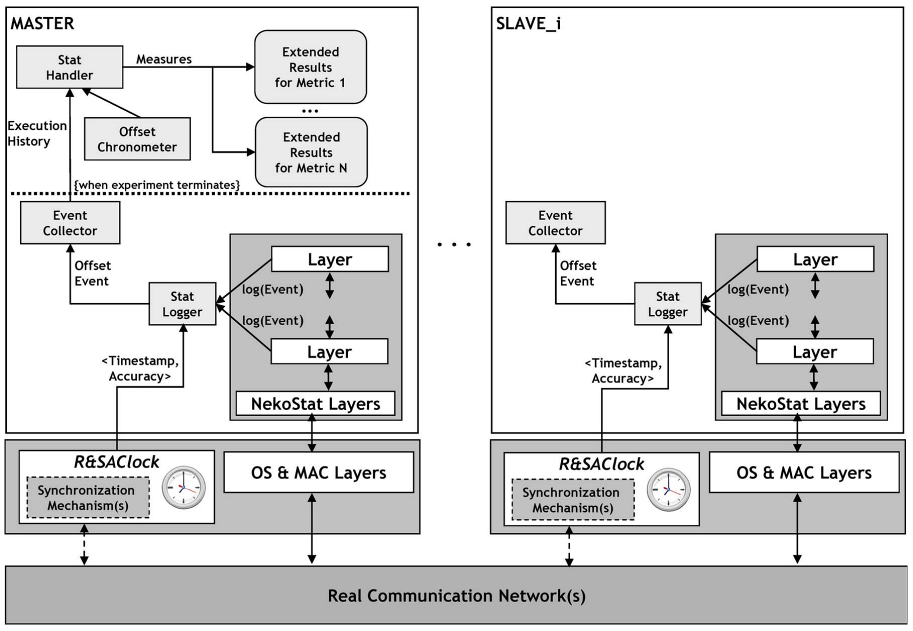


Figure 5.3: Instrumenting the system with the R&SAClock.

A measurement algorithm on NekoStat that exploits the R&SAClock can impose an *accuracy* requirement, that is the worst synchronization uncertainty that the application can accept in order to work correctly. The accuracy required by the application is a time value that is given in input to the enhanced version of NekoStat at the beginning of the experiment through its configuration file. Thus, R&SAClock can give value to its output *FLAG*, which is a Boolean value indicating whether the current synchronization uncertainty is within the accuracy requirement or not.

As a further useful feature, in addition to the evaluation of the uncertainty of time interval measurements, NekoStat can directly exploit *FLAG* to filter measurement results affected by unacceptably high uncertainty, and exclude them from the successive analysis. An example of the application of this feature is given in the next Section.

In the version that has been integrated in NekoStat, R&SAClock has been implemented in Java/C++ for the NTP synchronization protocol and Linux operating system (OS). R&SAClock lays on NTP synchronization protocol and uses data and functionalities provided by NTP (via NTP-related system calls) to get current time from virtual system clock, and values necessary to feed the internal mechanisms that compute synchronization uncertainty. The time resolution of the proposed tool is 1 ms, which is the resolution of the Java virtual machine clock.

5.3 EVALUATION OF A SAMPLE DISTRIBUTED PROTOCOL

In the following, a simple but significant experiment is presented. It shows the value added to NekoStat by the R&SAClock in terms of reliability and quality of measurement results.

5.3.1 *Objective of the experiment*

The purpose is to measure the distribution of one-way delays (OWDs) on an intercontinental end-to-end one-way transmission channel. In distributed computing systems, OWD measurement is important to estimate some fundamental dependability and networking metrics (examples are the Mean Time Between Failure, Mean Time To Failure [12], or bandwidth parameters [151], [6]).

5.3.2 *The target system*

The communication path is an end-to-end User Datagram Protocol (UDP) channel on a Wide Area Network (WAN) between two nodes. It is a basic example of a distributed application. No information on the network path or the competing traffic is available. The hosts used for the experiment are two Linux servers:

- *rcl.dsi.unifi.it*, connected to the network of the University of Firenze (Italy),
- *dmz.crhc.uiuc.edu* connected to the network of the University of Urbana (Illinois, USA).

The clock synchronization mechanism chosen on both servers is NTP.

5.3.3 *The experiment planning and the measuring instrument set-up*

Each transmission delay is measured as the time between two events (detected by NekoStat), the sending of a message by the sender process and the delivery of the message at the receiver's side. The R&SAClock allows associating the synchronization uncertainty of the local clocks at the time the events occurred.

The experiment is built as follows. The NTP daemons of sender and receiver hosts are started at the beginning of the experiment, when the two clocks are about 120 ms apart from each other. Then the NekoStat application is run and collects the events on the sender and receiver processes (at the rate of one event per second; note that some events may have been lost due to the use of the UDP transmission protocol).

Before starting the time interval measurements, R&SAClock is activated on both hosts. R&SAClock is set up on the two systems with the same configuration profile. The initial drift bound on both servers has been set to 20 ppm (this bound has been computed on the basis of our experience on NTP drift manipulation, and our observation on past behaviors of the drifts on the servers). Then, instead of simply including measurement uncertainty, which is related to the synchronization uncertainty, in each measurement result, a feature of the R&SAClock is exploited to process the data. Specifically, the R&SAClock evaluates the uncertainty associated to each one-way delay measurement, compares it to a threshold, represented by the value of *applicationAccuracy* and consequently forces the value of the parameter *FLAG* either to 0 or to 1.

In the experiment, if the *FLAG* value is 0 on both sender and receiver, this implies that the precision π of the global system is lower than 8 ms, otherwise it is greater than 8 ms. The value 8 ms has been arbitrarily chosen; it is about 10% of the delay that we commonly experienced by a packet transmitted on our intended path (see Section 5.3.2). Thus, at the end of the experiment, each one-way delay measurement result is associated to a value of the output parameter *FLAG*. NekoStat gives two output files: one containing all the obtained measurement results, and another including only the results affected by "acceptable" uncertainty, according to the value of *FLAG*. This way, the comparison is made possible.

5.3.4 *Execution and analysis of results*

Operatively the experiment is run only once, using the new version of NekoStat. That is, measurement results are collected only once, but data are then successively analyzed twice; the first time ignoring the information on measurement uncertainty provided by the R&SAClock (as one would do with the previous version of NekoStat), and the second time exploiting the information on measurement uncertainty for the data analysis (done by the new version of

NekoStat). In this case, the major contribution to measurement uncertainty is given by synchronization uncertainty. In addition to that, we have uncertainty due to the logging of events (i.e., the time needed by the operating system to collect information about events); this action has a negligible impact, that is surely under the Java virtual machine clock resolution, which is equal to is 1 ms.

Figure 5.4 shows all the obtained OWD measurement results, corresponding to a total of about 34000 samples. By looking at these results, one may think that the one-way transmission delay follows a multi-modal distribution, apparently suggesting that multi-modes are due to network perturbations – which looks sensible, whereas it is deceptive. Indeed, the multi-modality is only apparent, as it is caused by the poor quality of some measurement results.

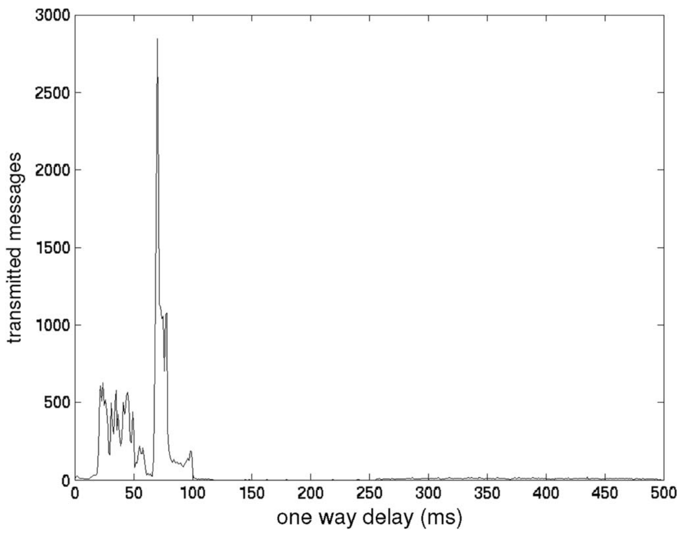


Figure 5.4: OWD measurement results.

In fact, if results are filtered to exclude those characterized by poorer synchronization, the distribution depicted in Figure 5.5 is achieved. Figure 5.5 shows the OWD measurement results affected by "acceptable" uncertainty, which are 9109 out of a total of 34049 results, reported in Figure 5.4. This figure shows that OWD in the path considered for the experiments can be reasonably modeled as either mono- or bimodal. Due to the evaluation of synchronization uncertainty, made possible by the new feature NekoStat has been equipped with, a deceptive and erroneous interpretation of the measurement results has been avoided.

In fact, if it is reasonable that different paths are utilized and/or competing traffic met by each packet varies, and we cannot be 100% confident that the OWD distribution is simply mono- or bimodal, it is also notable that a smoother OWD distribution is attained taking into account uncertainty in the analysis of measurement results.

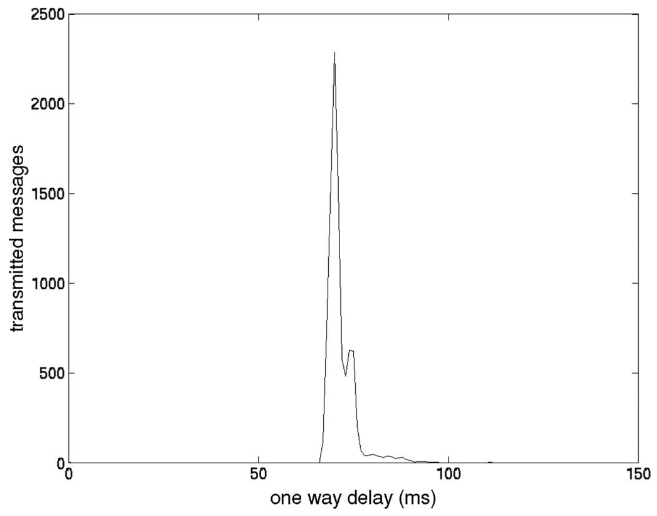


Figure 5.5: OWD measurement results characterized by synchronization uncertainty lower than 8 ms.

In other words, this does not mean that we have measured competing traffic or exactly determined how many network paths are involved, but we have for sure increased the quality of measurement results.

5.4 CONCLUSIONS AND RECOMMENDATIONS

We presented an enhanced version of NekoStat, a tool supporting the analysis of distributed algorithms and systems, which has been equipped with R&SAClock, a software component capable of evaluating the uncertainty of time intervals measurements.

In fact, when dependability measurements involving time interval direct measurements are considered (and this is very often the case with regard to distributed systems), erroneous analysis of measurement results can be avoided thanks to the evaluation of synchronization uncertainty, which has been made possible by the proposed tool. The comparison between the results of the experiment carried out with and without R&SAClock show that following a correct approach when designing a measuring instrument is not just a question of formal correctness, but gives clear practical benefits to the analysis of measurement results.

A final remark is that the R&SAClock solution proposed to improve the experimental evaluation of distributed systems may also be applied to improve time measurements for network and QoS monitoring. Although not explored in this Thesis, examples of such approach can be found in [72], [37] where uncertainty information is exploited to improve diagnosis of congested channels.

6 EVALUATION OF DYNAMIC SOAS

SOA (Service Oriented Architecture, [148]) is a wide-spreading architectural style used by many organizations. A SOA consists of several interacting services that are designed to support the critical backbone infrastructure of organizations. Software resources are packaged as "services" (and especially Web Services, WS, the most popular type of services available nowadays), which provide standard business functionalities and are independent from the state or context of other services [148].

Services have a published interface and communicate with each other by exchanging messages. In the particular case of Web Services, WSDL (Web Service Description Language, [148]) files are exposed and SOAP (Simple Object Access Protocol, [148]) messages are exchanged (the prior describe the network services as a set of endpoints that operate on messages in a cross-platform way and the latter represents requests for the execution of operations).

SOA allows developers to overcome many distributed computing challenges, as software interoperability and reuse, adaptation (modification, reconfiguration and flexibility) and evolution of the infrastructure. Due to their intrinsic characteristics, SOAs are becoming a fundamental support for many organizations to implement *business-critical* processes. However, the flexibility and dynamicity of SOA raise important issues that may prevent the applicability of this architectural style to its full extent [70].

A relevant issue is V&V of SOAs. In fact, the provider (or the stakeholder) of some services may not be aware of the existence of other services (or may not have their control). Additionally, a SOA evolves during in-service, meaning that services may be modified or new services can enter/exit the system while in operation. Solutions for online V&V are deemed necessary as common approaches for traditional, offline V&V conducted before deployment cannot be fully applied in SOAs [70].

Amongst the possible V&V solutions, we focus on *lifelong testing of SOAs* [70], [16], [13], to present a testing tool and related testing methodology for lifelong testing of SOAs. The approach for the design and use of our measuring instrument is based on the methodology of Chapter 3, and defines service discovery and testing actions that are applied via the introduction in the SOA of a *testing service*. Emphasis is put on the sharing of information between providers, both in terms of knowledge of the SOA and results (outputs) of the testing activity performed. A shared database is maintained and accessed by the providers to acquire the information they need.

The rest of this Chapter is organized as follows. Section 6.1 describes the SOA environment that is at the base of our research, Section 6.2 presents the architecture of our testing service and Section 6.3 sketches the main algorithms used. Section 6.4 shows the case study and finally in Section 6.5 conclusions are drawn. Further information can be found in [38], [39].

6.1 ENVIRONMENT DEFINITION

In the context of this work, a SOA is described from the point of view of the providers interested in performing continuous validation.

We propose a representation for the SOA components which later serves to support the testing approach. This view does not yet consider the testing service as part of the SOA (it will be introduced in the next Section).

There are four key components in our approach: *BPEL services* (the WS-Business Process Execution Language [138], or simply BPEL, is a standard for business-critical process definition, which describes the interactions between businesses or elements in some business), *controlled services*, *within reach services*, and *unknown services*. The following paragraphs describe these components.

A *BPEL service* is a service that runs a BPEL process and to which a provider can have full access. We assume that the provider has access to the description (implementation) of the BPEL service.

A *controlled service* is a non-BPEL service owned by the provider or to which the provider has access to extensive information such as the source code or the executable files including debugging information.

A *within reach service* is a service that is part of the SOA and is known by the provider; however the provider is not able to control it (except invoking it). A within reach service may also be a service that runs a BPEL process. In practice, the provider does not have access to the internals of a within reach service and can only access the interface exposed via the WSDL file.

An *unknown service* is a service that is part of the SOA, but whose existence and details are not known by the provider.

Obviously, it is not mandatory to have the four types of services in a given SOA (there may exist zero or more instances of each type).

The BPEL, controlled, within reach and unknown services interact as follow. The unknown services are invoked only by other unknown services and by within reach services (obviously, if BPEL services or controlled services invoke unknown services, they are no longer unknown, becoming instead within reach). On the other hand, unknown services may invoke all the other types of services (they can also act as users). BPEL services and controlled services invoke other BPEL services, controlled services and within reach services. Finally, within reach services may invoke any type of services, including the unknown ones.

6.2 architecture of the testing service

The testing service is part of the SOA architecture itself and is managed by the providers (possibly by more than one) that require the continuous evaluation of the SOA. In practice, providers share the testing service in order to share/coordinate validation efforts and information on the trustworthiness of the architecture. Including the testing service as part of the SOA provides the flexibility needed to perform lifelong testing, overcoming the problems introduced by SOA dynamicity and evolution. In fact, testing tools that are not part of the architecture may not be able to communicate continuously with the existing services (e.g., due to authentication issues) and, consequently, are inadequate for lifelong testing support.

The architecture of the testing service is a *composite Web Service* (i.e., it is made of several coordinated and distributed WSs) that uses basic information provided by one or more providers about the services that each provider knows, owns or manages. This information is used to build an initial description of the SOA and then to automatically discover the other services that operate within it. This view is kept up-to-date at in-service by detecting services that evolve, enter or exit the architecture. The services enlisted in the SOA description are periodically tested by the testing service, according to tests plans that can be properly configured. The approach is provider-centric, meaning that it considers the point of view of a provider that needs to trust other providers' services, and consequently favors extensive testing even at the risk of activating dormant faults or degrading service performance. Note that at in-service, services copies are used to avoid service degradation and error propagation caused by the testing activity. Whenever required (and possible), tests are performed over copies of the services rather than real ones (copies of the real services are automatically deployed, tested in spite of the real services, and then undeployed), to guarantee that the testing activity does not impact service performance or activates bugs leading to service degradation or disruption.

The testing service is composed of the following services: the *testing BPEL*, the *testing controlled* and the *testing core*. The key roles and interactions of these components are exemplified in Figure 6.1. The parameter *SOA status* (0 or 1) allows distinguishing if the testing service is operating in an offline SOA or at in-service.

The *testing BPEL* monitors BPEL services and manages the deploy and undeploy of their copy for testing purposes. An instance of a testing BPEL is created and started for each existing BPEL service. This instance is able to recognize new versions of the BPEL process and to detect the services that are invoked. The first time it is started, the testing BPEL i) reads the description file of the BPEL service it monitors and sends the list of the services discovered to the testing core, and ii) tests the BPEL service. From the description of the BPEL process, the testing BPEL automatically identifies the list of the participant services and

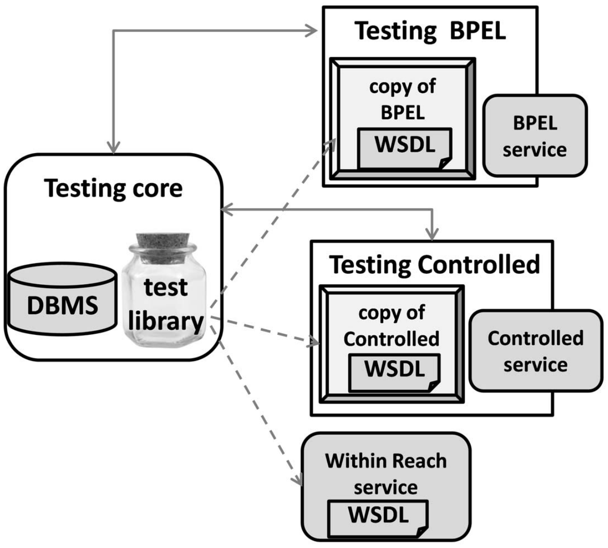


Figure 6.1: A sample view of the testing service architecture.

retrieves their addresses from the WSDL files. Note that this solution is valid for static binding (services to invoke are known at compile time), while for dynamic binding (services to invoke are decided at in-service) more sophisticated approaches are needed, but are currently not realized in our testing service. For example, viable alternatives for dynamic binding are to apply monitoring rules to the BPEL process as in [16], or using Aspect-Oriented Programming (AOP, [109]) to intercept messages as in [133].

When the SOA status is set to 1, and the BPEL service needs to be tested, the testing BPEL creates a copy of the BPEL service by re-deploying it using a different name. If this copy of the BPEL service interacts with some others BPEL services or controlled services, copies of such services are also automatically deployed. The copied BPEL service then interacts with such copies instead of with the real services. Obviously, the invoked within reach services are used as they are. Note that all the copies deployed are automatically undeployed when the test completes.

Similarly to the testing BPEL, an instance of a *testing controlled* is created and started for each existing WS, and the *testing controlled* monitors the controlled service detecting the services it invokes. Additionally, the testing controlled support testing of the controlled service (including the deploy and undeploy of a copy of the controlled service for testing purposes during in-service testing).

The *testing core* is the main component of the testing service as it is in charge of guiding the SOA testing activities: it selects the testing policies (when, what and how to test), it collects and merges results, it traces the SOA evolution using information received from controlled services, from BPEL services and directly notified by the providers, and it starts and halts the *testing BPEL* and *testing controlled*.

6.2.1 *Execution and interactions of the testing service*

The testing core service maintains and updates the SOA description using the information received from the providers and from the testing BPEL and testing controlled services. If new providers join the testing service, then it uses the additional information they may offer.

The testing core selects the services to test in the following way:

- if it is a within reach service, the test is executed by a testing module of the testing core from an external point-of-view (by applying black-box tests through the service interfaces, as proposed in [115]);
- if it is a controlled or BPEL service, then the test can be executed directly by the testing module of the testing core (when considering black-box tests) or by the controlled and BPEL services (when considering white-box tests).

Test results are available to the providers that use the testing service.

The testing core receives information about the *SOA status* (0 = offline, 1 = in-service) from the provider, and sends that information to the testing BPEL and testing controlled services. If the SOA status is 0 (offline), then the services are tested without deploying copies: during offline testing, the services are not open to users and there are no major risks regarding error propagation or service degradation.

On the other hand, if the SOA status is 1 (in-service), the testing BPEL and testing controlled services create and test copies of their target services, in order to avoid, or at least limit, the risk of service degradation and error propagation (a web server typically isolates the various services that it is executing). Note that within reach services cannot be copied (the testing service has no access to the internals of within reach services and consequently it is not able to automatically instantiate a copy). Acting from the point of view of a provider that needs to trust other providers' services, we prefer to test services even at the risk of activating dormant faults or degrading their performance.

Note that, if a service provider offers a testing infrastructure in parallel to the production environment (common in web service scenarios), then it can be used as target of the test, avoiding the deployment of copies.

In our current implementation, all tests are run by the testing module of the testing core, which performs black-box robustness testing; also, it uses the same test generation and execution method for BPEL services, controlled services and within reach services.

6.3 DISCOVERY AND TESTING ALGORITHMS

We describe the services discovery algorithm, the testing policy, and the testing technique of the testing service.

6.3.1 *SOA discovery and update algorithm*

We only sketch the algorithm that describes the SOA discovery; a comprehensive description of the algorithm can be found in [38].

The testing core contains the list of the services discovered (in the database table *services list*). A service can be discovered via different sources: by the user of the testing service (that manually adds it to the services list), by the testing BPEL services, and by the testing controlled services. When the user adds a service to the services list, he must declare it as being a within reach, a controlled or a BPEL service.

When a service is classified as *controlled* or *BPEL*, the corresponding testing BPEL or testing controlled service is queried for additional services it may have discovered. Such newly discovered services are added as being *within reach*. Afterwards, the various testing BPEL and testing controlled services act proactively to notify the testing core of eventual changes and to update the list of interacting services. Also, discovered connections amongst the services are recorded in the database table *services connections* and evolution of the SOA is traced in the *SOA evolution* table. Note that if a service is added by a provider and also discovered by another source (e.g., a testing BPEL which invokes it), it maintains the classification assigned by the provider.

A provider can manually remove from the list only the services that he manually added. Instead, the services that were added through the automatic discovery steps are automatically removed from the list when they leave the SOA (i.e., when they are not required anymore by the corresponding testing controlled or testing BPEL services).

6.3.2 *Test schedule algorithm*

We present the simple test schedule that is currently applied by the testing service. More detailed scheduling policies can be set (configured) in the testing service, but are not further explored in this Chapter.

Services included in the table services list are tested according to the policies presented in this Section. If the SOA status is equal to 0, then no copies of the services are performed, while if the SOA status is equal to 1, copies of BPEL services and controlled services are tested instead of the original ones.

BPEL services and controlled services are first tested when they are added to the services list. They are afterwards re-tested every time they are modified (i.e., when they evolve). On the other hand, within reach services are tested periodically. The reason is that it is not possible to detect when the internals of a within reach service changes (these services are remote and there is no access to the deploy process). Consequently, we believe that periodic testing is a good compromise to assess the service behavior (the time between tests can be configured by the provider).

When a service is not reachable at testing time or is "not testable" (e.g., the service is online, but refuses connections or actions for testing), retrying attempts are periodically performed (again, the time between retries can be configured). Note that even if the service is not accessible, it may still be part of the architecture: failed attempts do not lead to exclude the service from the services list.

6.3.3 *Implementation of the testing technique*

The current version of the testing service executes robustness testing of WSs through the usage of the API of the *wsrbench* tool [115]. Such API provide means to automatically import and analyze the WSDL file of the target WS, generate and execute the workload, and collect and analyze the results of the tests. The robustness tests are invoked by the testing core, which targets the various services one by one, collects results and stores them for further analysis. *Other types of tests can be added* in the testing service by plugging the required library, in the same way we did for robustness testing.

6.4 evaluation of the jseduite soa

Our testing service has been applied to the *jSeduite* SOA [55]. jSeduite is a free SOA that deals with information broadcast inside academic institutions. It is composed of atomic Web Services representing information sources and BPEL orchestrations expressing business processes. jSeduite services use a MySql database.

We started by identifying a subset of the jSeduite SOA, composed of 24 services (see Figure 6.2): 6 BPEL services (represented in light gray in figure) and 18 *JAX-WS* (Java API for XML Web Services), represented in white. Some of the services invoke external WSs and applications, represented in dark grey. We distributed jSeduite on four Glassfish [172] servers (installed on four virtual machines running Linux Ubuntu OS, with addresses ranging from 192.168.81.128 to 192.168.81.131). A client application (producing a workload) is started, cyclically querying the various services. Due to lack of space, we do not provide further details on the experimental set-up and on the role and meaning of the services and orchestrations involved; however an exhaustive description can be found in jSeduite. We organized the case study in two steps, in which two different providers use (and share) the testing service, as described below. We set SOA status to 1.

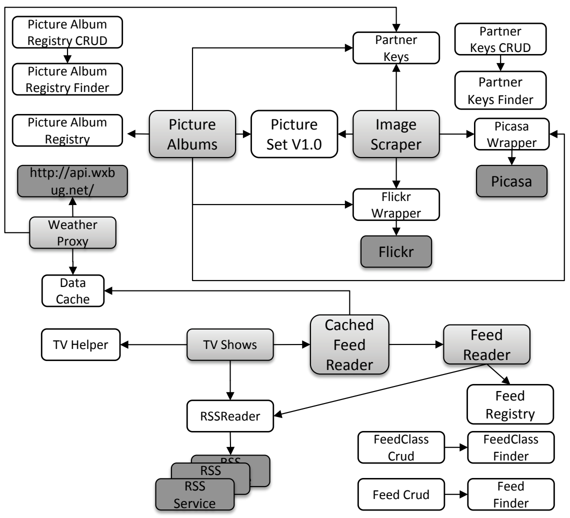
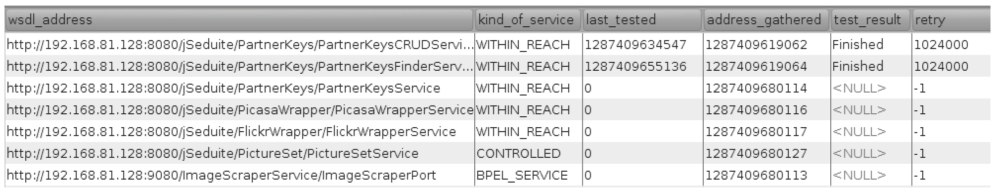


Figure 6.2: The subset of the jSeduite SOA selected as case study.

In **Step** 1, the first provider uses the testing service to test his own services (*Image Scraper* and *Picture Set*), before opening them to external SOA users. Obviously, the services list maintained by the testing core is initially empty. The provider adds the services that he controls or knows to the services list. These are the BPEL service *Image Scraper* (at this point, the corresponding testing BPEL is started), the controlled service *Picture Set v*1.0 (at this moment, the corresponding testing controlled is started) and two within reach services (*Partner Keys Crud*, *Partner Keys Finder*). The testing service performs services discovery based on the available information.

Table 6.1 presents a snapshot of the services list table and Figure 6.3 graphically represents the part of the SOA discovered. The services automatically added to the table are *Flickr Wrapper*, *Partner Keys*, and *Picasa wrapper*. These services are discovered by the BPEL service in charge of monitoring the *Image Scraper* component.



wsdl_address	kind_of_service	last_tested	address_gathered	test_result	retry
http://192.168.81.128:8080/jSeduite/PartnerKeys/PartnerKeysCRUDServi...	WITHIN_REACH	1287409634547	1287409619062	Finished	1024000
http://192.168.81.128:8080/jSeduite/PartnerKeys/PartnerKeysFinderServ...	WITHIN_REACH	1287409655136	1287409619064	Finished	1024000
http://192.168.81.128:8080/jSeduite/PartnerKeys/PartnerKeysService	WITHIN_REACH	0	1287409680114	<NULL>	-1
http://192.168.81.128:8080/jSeduite/PicasaWrapper/PicasaWrapperService	WITHIN_REACH	0	1287409680116	<NULL>	-1
http://192.168.81.128:8080/jSeduite/FlickrWrapper/FlickrWrapperService	WITHIN_REACH	0	1287409680117	<NULL>	-1
http://192.168.81.128:8080/jSeduite/PictureSet/PictureSetService	CONTROLLED	0	1287409680127	<NULL>	-1
http://192.168.81.128:9080/ImageScraperService/ImageScraperPort	BPEL_SERVICE	0	1287409680113	<NULL>	-1

Table 6.1: Services list corresponding to the services discovered at step 1.

As shown in Table 6.1, the table services list contains the following data: the WSDL address of each service (column *wsdl_address*); the type of service

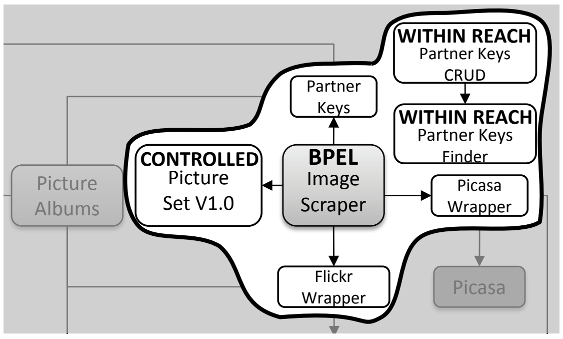


Figure 6.3: Services discovered at Step 1 of the case study.

(column *kind_of_service*, which can contain three values: within reach, BPEL, or controlled); the most recent time it was tested, in milliseconds (column *last_tested*: 0 means that it has not been tested yet); the time when the service was discovered and added to the list, in milliseconds (column *address_gathered*); the test outcome (column *test_result*: acceptable values are "Finished", "Service not reachable", and "Error during test execution"; note that only two services had been completely tested at the time the snapshot was captured); and the time when the test should be retried, as imposed by the test schedule algorithm, in milliseconds (column *retry*). Column *retry* is set after performing the test of the corresponding service, and a retry value is selected depending on the value of the column *test_result.* As controlled and BPEL services are tested only when there are changes (and not periodically), if the test finishes with outcome "Finished", the value of the column *retry* is set to -1.

In **Step** 2 of the case study, the second provider starts using the testing service. The second provider adds a within reach service (*Feed Registry*) and two BPEL services (*Picture Albums* and *Cached Feed Reader*; the corresponding testing BPELs are started) to the list. Additionally, the first provider, that owns the Picture Set service, deploys a new version of it (*Picture Set v*2.0) which, compared to version 1.0, contains new connections to two other services. The part of the SOA that is now known by the testing service is shown in Figure 6.4. For brevity, we do not report the corresponding services list.

Services automatically added during the discovery process in this last step are:

- *Picture Album Registry*, discovered through *Picture Album*;
- *Picture Album Registry CRUD* and *Picture Album Registry Finder*, discovered via *Picture Set v*2.0; and
- *Data Cache* and *Feed Reader*, discovered through *Cached Feed Reader*.

Some services are discovered by more than one source, but they are obviously enlisted in services list only once. These services are *Partner Keys*, *Flick Wrapper*,

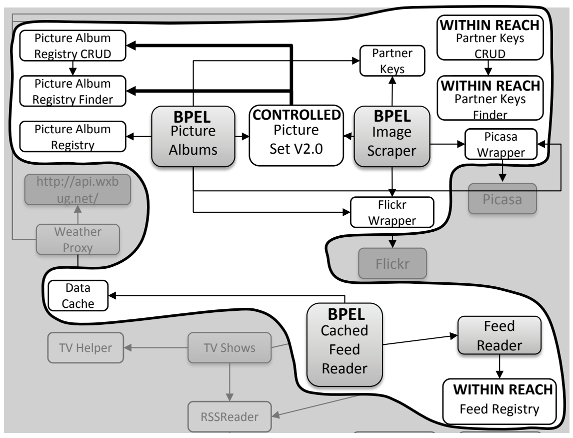


Figure 6.4: Services discovered at Step 2 of the case study.

Picasa Wrapper and *Picture Set*, which are discovered by the two BPEL services, *Picture Albums* and *Image Scraper*. According to the proposed SOA discovery algorithm [38], the first three services are classified as within reach services, while *Picture Set* maintains the classification assigned by the provider (it remains a controlled service). In our case study we discovered 15 out of the 24 services (excluding the external services *Picasa*, *Flickr*, *RSS service* and *api.wxbug.net*, considered as unknown in the context of the case study).

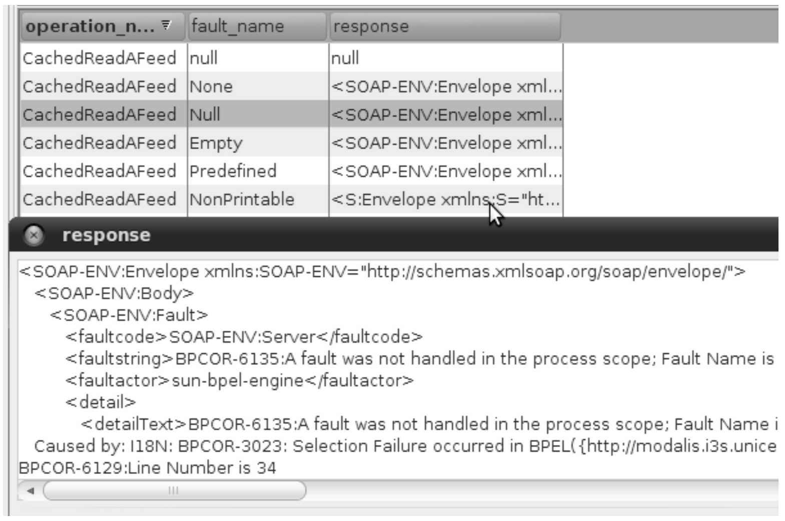
The providers that rely on the testing service can analyze the tests results to identify possible robustness problems in the SOA and its services. We analyzed the testing outcomes stored by the testing core and the most representative robustness problems identified. According to the wsAS classification [115], where failures are classified in ABORT failures (abnormal termination of the execution of the WS by returning an unexpected exception) and SILENT failures (absence of response from the service), jSeduite services are mostly affected by ABORT failures. SILENT failures were identified only when testing the BPEL processes.

The typical behavior of jSeduite Web Services is to reply to robustness tests with messages formed according to SOAP standards. By inspecting the source code, we observed that exceptions are caught (using Java *throws* statements and *try-catch* blocks), and that the answers returned to the client are the descriptions of the exceptions; i.e., no default values compliant to the data type expected by the client are returned (except for some operations as the *isValid* operation of *Data Cache*: in this operation, when an exception is caught, the answer returned to the client is the Boolean value *false* and not the description of the exception).

During the tests we observed that the service *Partner Keys CRUD* crashes when a malformed input value (containing *null*) is provided to the *createPartnerKey* operation. In practice, the service tries to insert *null* values in a MySql table that does not accept *null* values (the *not null* constraint is set). To fix this, we modified the source code of the service, adding data acceptance checks [152]. The modified version was then deployed.

Regarding BPEL processes, in some cases there was no response during robustness tests, leading to SILENT failures (*wsrbench* did not receive an answer within the allowed time interval, and consequently the *wsrbench* timeout expired). This happened for example when submitting the fault *Min Type Minus One* (sends the minimum number valid for the type Integer minus one [115]) in the *Image Scraper* service. The reason for such robustness failures and for the exceptions returned became evident when inspecting the source code of the BPEL processes. In fact, we observed that some lines of execution missed proper data input validation and exception handling: some faults were not handled by the process itself, but the corresponding failure was managed by the BPEL server, which contains specific policies to manage failures and create default message responses.

In Figure 6.5 we provide a small excerpt of the testing results for the BPEL service *Cached Feed Reader* (the third row from the top is expanded in the box below). In this case, the incorrect input sent to the BPEL service is a *null* value, instead of a positive integer. The BPEL process is not able to recognize that the input parameter is an unexpected value, and it first queries the *Data Cache* service (which returns false), and then it invokes the BPEL process *Feed Reader* with an invalid parameter. *Feed Reader* returns an exception to *Cached Feed Reader*. The BPEL process *Cached Feed Reader* does not manage the exception, which is finally captured by the server. The process ended returning an unexpected fault message (ABORT failure).



operation_n...	fault_name	response
CachedReadAFeed	null	null
CachedReadAFeed	None	<SOAP-ENV:Envelope xml...
CachedReadAFeed	Null	<SOAP-ENV:Envelope xml...
CachedReadAFeed	Empty	<SOAP-ENV:Envelope xml...
CachedReadAFeed	Predefined	<SOAP-ENV:Envelope xml...
CachedReadAFeed	NonPrintable	<S:Envelope xmlns:S="ht...

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Server</faultcode>
      <faultstring>BPCOR-6135:A fault was not handled in the process scope; Fault Name is
      <faultactor>sun-bpel-engine</faultactor>
      <detail>
        <detailText>BPCOR-6135:A fault was not handled in the process scope; Fault Name i
  Caused by: I18N: BPCOR-3023: Selection Failure occurred in BPEL({http://modalis.i3s.unice
BPCOR-6129:Line Number is 34
```

Figure 6.5: Cached Feed Reader: a screenshot of the results.

6.5 CONCLUSIONS AND RECOMMENDATIONS

In this Chapter we presented a testing service that can be integrated in the SOA and that is able to perform life-long testing of the architecture, by discovering new services and automatically testing them. Our testing service, implemented as a composite service that is subdivided in a testing core (the main component of the testing service) and other services that are plugged to the SOA services, supports the view of one or more providers during the whole lifecycle of the SOA.

Information and test results are shared in our approach through the cooperation of the providers. In fact, the testing service is mostly effective when information on the SOA and results of the tests are shared amongst the various providers which rely on the testing service. However, we must note that the sharing approach here proposed opens important questions, mainly i) who should own the testing service, ii) which policies to apply for its management and iii) what should be the agreement amongst providers. In fact the testing services shares tests outcomes, and may have access to sensitive services information and testing infrastructures. These questions are not easy to solve and require further investigation, constituting directions for future work.

A final, important remark is that we focused here on BPEL and WSDL, but the approach is general and can also be applied to other equivalent languages, for example YAWL (Yet Another Workflow Language, [56]) and WADL (Web Application Description Language, [74]), respectively. Also, it is important to mention that in the current version of the testing service we are not considering particular services as UDDI (Universal Description Discovery and Integration, [148]) registers or infrastructures as ESB (Enterprise Service Bus, [148]); to introduce them in the environment constitutes direction for future work.

Part III

THE METHODOLOGY IN THE INDUSTRIAL CONTEXT AND TWO CASE STUDIES

7 INDUSTRIAL PROSPECTIVES AND NEEDS

In industrial practices, V&V and certification activities are largely applied, because critical systems need assurance that they meet the specifications and fulfill the intended purpose. The ultimate goal is to provide an assurance recognized by society (and in some cases by law) that a system is deemed safe by the certification body [146].

Several different V&V processes exist and are currently applied, depending on the application field considered [107]. For example, the CENELEC EN 50126 [41] describes the required V&V process for certification of railway equipment, while the DO-178B [57] is devoted to airborne systems and the IEC 61508 [84] instead refers to generic electronic safety equipment. The certification activity performed according to the standards is typically aimed to assess that the procedures enlisted in the standards were correctly and responsibly applied, and a sufficiently large number of activities to guarantee the system's dependability requirements was performed.

As the time to market for critical embedded systems includes also the time needed to obtain the approval from certification authorities, we observed in [40] that the *productivity* of system development can be significantly improved by defining a comprehensive methodology which describes the role played by each task in the whole V&V process. A direction to achieve such improvement is the development of integrated tool-chains which support the automatic application of methodologies to speed up and automatize the systems V&V and certification process.

RACME (Resiltech Assessment Certification MEthodology, [40]) is both a framework and a methodology conceived to reach such goal. RACME supports and guide a V&V expert through a whole V&V process; it is customizable for different systems and V&V processes, and constitutes the chain that links the various V&V activities, providing the required input, structuring information and helping the V&V expert to retrieve such information.

RACME *does not contain built-in applications* to apply a specific V&V technique (e.g., formal methods for software verification), but it constitutes the chain that links the various V&V activities, providing the required input, organizing the outputs and providing the V&V experts with the results they need (e.g., it uses the output of the test execution to identify the requirements that are currently not satisfied).

The main progresses that are intended with RACME methodology are summarized by the following points:

1. identification, monitoring and control of all safety and certification-related activities and information within the system life-cycle;
2. improvement of the traceability, including artifacts from all the system development life-cycle; and
3. leveling of the presentation, data and control layers [185], [173] with a deep enhancement in terms of usability and, consequently, correctness of the information treated.

The experimental evaluation methodology presented in this Thesis can be applied also to the industrial practises for V&V, and complement the usage of RACME and its intended contribution. Such possibility is tackled in this Chapter: we show that the methodology presented in Chapter 3 can be matched to RACME and thus adopted in V&V industrial processes. That is, the objective of this Chapter is first to show the RACME solution for the management of *any* V&V and certification process, and then to illustrate that our methodology can successfully fit such processes supported by RACME.

The rest of this Chapter is organized as follows. In Section 7.1 we present the state of the art on methodologies for V&V and the motivations to the RACME approach. In Section 7.2 we show the RACME approach for the support of V&V processes, and details on the RACME architecture and prototype. Finally, in Section 7.3 we present our testing methodology in the context of V&V and certification processes (supported by RACME).

7.1 METHODOLOGIES AND TOOLS SUPPORTING V&V

The process to support system V&V consists in the collection and elaboration of evidence coming from results provided by (uncorrelated) tools performing some of the V&V tasks and from (a large set of) documents.

Analyzing the state of the art of *industrial* solutions, we identified few results regarding methodologies, platforms or frameworks that support the V&V experts through the overall V&V process, allowing integration of outputs and task coordination, and that do not limit their support to a specific V&V activity or techniques. We identified the following results as relevant for the targets previously enlisted: the DVDT tool [137], [143] and the tool suite IBM Rational [82].

The concept of the documentation tool DVDT (Department of Defense VV&A, Verification, Validation and Accreditation) [137], [143] is to produce standardized VV&A documentation and VV&A XML schemas that facilitate VV&A information sharing, discovering, and retrieving within the Global Information Grid (GIG, [144]) enterprise. The DVDT automates the standardized documentation templates to aid those involved in the Navy's models and simulation efforts in collecting, organizing, and documenting information pertaining to the VV&A of models and simulations.

IBM Rational [82] suite is a broad collection of tools that support the different phases of software life-cycle (e.g., it includes the IBM TeleLogic [82], a family of products for system assessment). There are several different instruments, that focus on many different activities of software life-cycle, for example: analysis, modeling and design, management of software quality, management of software modification, configuration and release, management of processes, projects and portfolio.

This solution shows the following two limitations with respect to V&V and certification issues. First, as its focus is not on the reusability of V&V results and on the customization to different contexts and for different V&V processes, it does not consider certification issues for different V&V standards and processes. Second, there is a lack of integration and cross-check of outputs provided by add-ons and tools that are useful for specific V&V activities.

We believe that V&V experts would benefit from such a support, because it would improve organizing the large quantity of documentation produced, facilitating the handling and retrieval of a large set of information, improving consistency checks and non-regression. As final benefits, we believe that not only V&V time and costs can be optimized, but that also certification evidence can result easier to prove.

Another work we mention is the standard Software Assurance Evidence Metamodel (SAEM, [140]) from the OMG (Object Management Group) for the management of safety assurance evidence. The SAEM is a standard-independent metamodel that is directed towards linking the certification evidence to safety claims and the evaluation of these claims subject to the evidence. SAEM establishes the necessary fine grained models of evidence elements required for detailed compliance and risk analysis. The structure of the SAEM provides the basis for the logical design of tools for storing, cross-referencing, evaluating, and reporting the elements of evidence for systems during the software assurance process. This very recent initiative partially shares the goals of RACME, and can also act as a relevant input for the structuring and organization of the information stored in RACME.

The research community is well-aware of the challenges presented in this Section, and several related works can be identified. We discuss here the works [146], [147], [117] and [47].

In [146] a conceptual model is presented to characterize the evidence for arguing about software safety. The model captures both the information requirements for demonstrating compliance with IEC 61508 and the traceability links necessary to create a seamless continuum of evidence information i.e., a chain of evidence.

In [147] model-driven engineering principles and technology is used to specify and analyze safety evidence in order to show conformance to a safety standard.

The approach establishes a sound relationship between a domain model of a safety-critical application and the evidence model of a certification standard. Briefly, this is realized by capturing the relevant standard as a conceptual model using a UML class diagram and using this as a basis for creating a UML profile. The profile is then augmented with constraints to aid system suppliers in systematically relating the concepts in the standard to the concepts in the application domain.

In [117] it is identified the existence of a dense web of safety-relevant information and relationships. The paper discusses the benefits to both safety case writers and readers of applying information modeling as the basis for creating an electronically-based safety case. The usage of methodologies as GSN (Goal Structuring Notation) is proposed to formally describe relationships between entities.

In [47] the use of electronic formats for safety cases is described to meet the requirements of a number of military and civil standards. An example of electronic safety case is presented and it is discussed how this can be used to manage a safety programme and to produce a safety case that will meet the requirements of the certification authorities.

7.2 THE RACME APPROACH AND ARCHITECTURE

In this Section we describe the RACME approach and architecture: we first present the RACME workflow, then we show a logic view of the RACME framework and finally we present the RACME architecture and the available prototype. Further details on RACME can be found in [40].

7.2.1 *The RACME workflow*

Many different V&V workflows may be applied in RACME, depending on the characteristics of the V&V process, of the activities performed using RACME, and of the activities that are performed without RACME (e.g., done by third parties: consequently inputs to the activity shall not be extracted from RACME; if needed, the activity outputs can be added in RACME).

We define a very general workflow, not specific or customized to a single V&V process: customization may be required to adapt RACME to different standards or different life-cycle. However, these customizations shall only mean a different structure of the workflow, and consequently a different configuration of the V&V process supported by RACME, without requiring any change to the framework. We can note that different grains or parts of the workflow are required and used for the different processes.

The generic RACME workflow for V&V is shown in Figure 7.1. The initial inputs provided are the *RACME configuration* for the specific V&V process in use, the *customer's templates* for documentation and the *requirements* of the target system. These documents are loaded into RACME, and the requirements of the target system are extracted from the documentation.

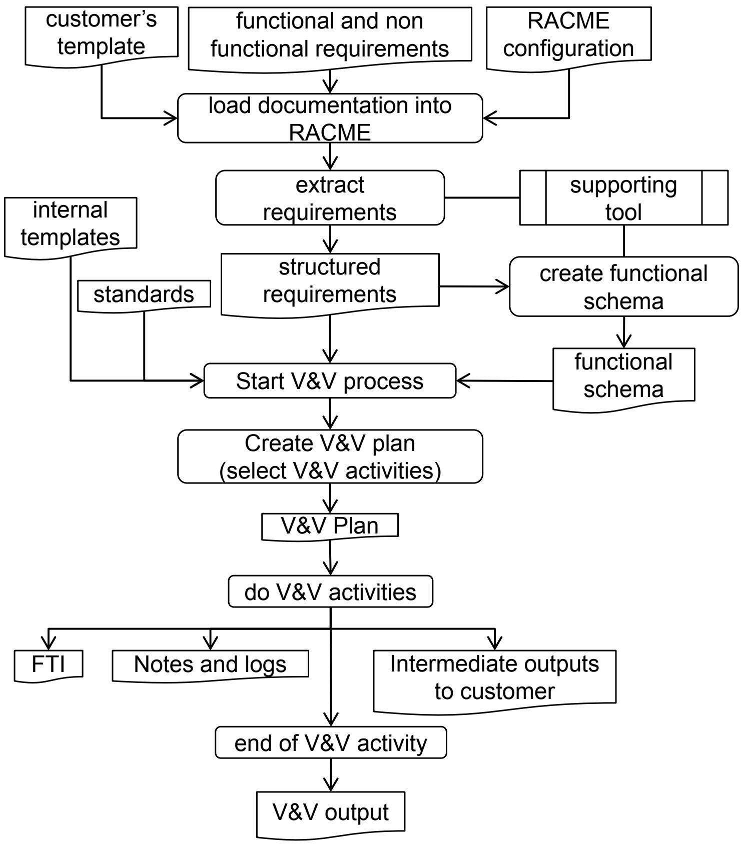


Figure 7.1: The RACME workflow for a generic V&V process.

The requirements provided as input are loaded into RACME and organized according to the RACME internal notation (the *structured requirements* box in Figure 7.1). Such requirements are then used to create a functional representation of the target system. This *functional schema* is created by the V&V expert, using the semi-formal language SysML (System Modeling Language, [139]), that provides diagrams to describe requirements and is used as a mean to match the requirements to the various parts of the system, at different grains and from different perspectives. This matching facilitates cross-inspect of requirements of different parts of the system. A *supporting tool*, external to RACME, to create such SysML functional schema is required.

Once the functional schema is available, the V&V process can start; additional inputs at this point are *templates internal to the company* that uses RACME and *standards* that describe the V&V process. Note that these information are stored in RACME as default information, and do not need to be provided as inputs each time RACME is used.

For any V&V process, the first action is to produce a *V&V Plan*, which enlist the V&V activities that are planned.

Then, the activities identified in the V&V Plan are executed: RACME provides the input for each activity and helps generating and formatting the outputs. In this phases, instruments build-in in RACME are used to: i) trace requirements, and ii) support the merging of results and the creation of the documentation. During the V&V process, RACME helps to produce *FTI*s (Formal Technical Inspections, they are largely used to collect non-conformities in documents/activities), *notes and log files*, and the various *intermediate results and documents*. Finally, at the end of the V&V process, the final *V&V output* (e.g., the safety case) is produced.

7.2.2 *A logic view of RACME*

An abstract view of the RACME framework is depicted in Figure 7.2. A repository stores the provided information and data. An engine provides the functionalities to integrate data and correlate activities. RACME links the various V&V activities, and uses inputs from the V&V experts and from information already stored in itself.

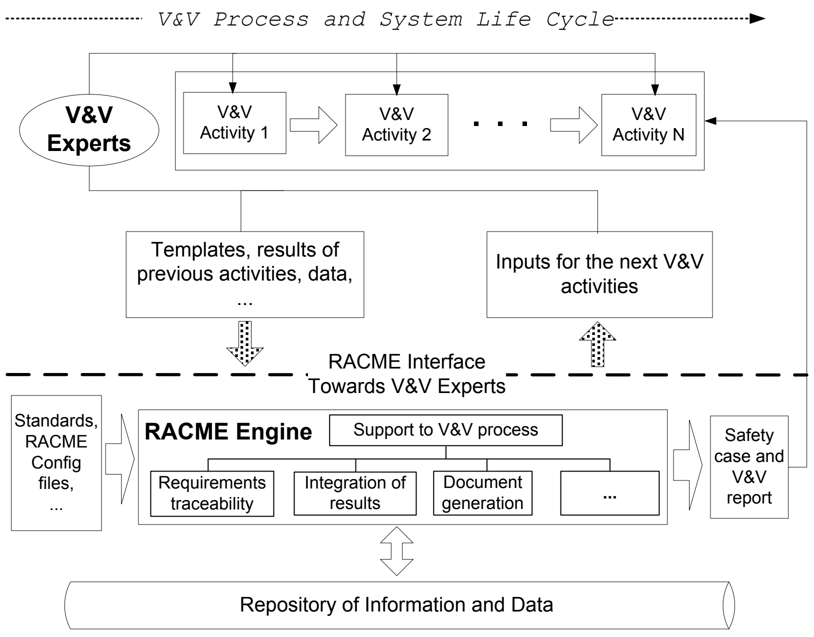

V&V Process and System Life Cycle
V&V Experts
V&V Activity 1
V&V Activity 2
. . .
V&V Activity N
Templates, results of previous activities, data, ...
Inputs for the next V&V activities
RACME Interface Towards V&V Experts
Standards, RACME Config files, ...
RACME Engine
Support to V&V process
Requirements traceability
Integration of results
Document generation
...
Safety case and V&V report
Repository of Information and Data

Figure 7.2: Logic view of RACME.

RACME manages requirements, hazards, mitigation, FTI, tests plans and reports through the following actions:

1. performs consistency checks between the produced output (i.e., output consistency, traceability and non-regression verification),

2. visualizes different information regarding each item, and

3. manages their versioning.

As it can be noted in Figure 7.2, RACME does not support a specific V&V activity (there are plenty of specific tools and techniques for that, surveyed in several works as [93], [176]), nor integrates tools that support a specific activity. It provides inputs to each activity, then it imports the results, and it processes and integrates them to cover non-regression, compatibility and cross-checking issues, as well as (semi-automatic) document generation. RACME allows to integrate data from different and heterogeneous tools and data related with different development life-cycles. Also, it allows to tailor the production of the necessary information and evidences for the certification of embedded systems according to different standards that often prescribe and dictate the structure of the documentation (and of the individual documents) to be prepared for certification approval.

7.2.3 *The RACME architecture and prototype*

We describe the overall RACME architecture and prototype. The implementation of RACME is currently ongoing, consequently only a subset of its functionalities is available in the running prototype.

Overall RACME architecture

Figure 7.3 offers an high level view of the overall architecture of the RACME framework.

Our plan is to implement the RACME architecture progressively, first building a main core (the key parts of the RACME server: the DataBase Management System *DBMS*, the *file system*, and the *V&V Activity Manager*) and then extending and adding functionalities (components to manage specific aspects of the V&V process, as the *requirements manager* and the *tests manager*, and *plugins* to deal with documents I/O).

Such plan is pursued organizing RACME in a way similar to a plug-in architecture [188]. Considering our incremental approach to implement the RACME architecture, the benefit of the plugin architecture is to improve modularization, because the code is cleanly separated into distinct modules, and the time required to replace or add components is reduced. Without reaching the complete independence that characterizes a plugin architecture, our approach is based

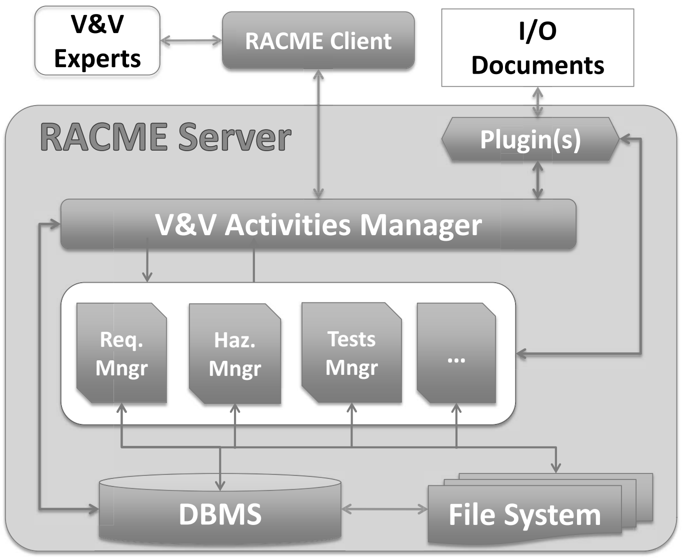


Figure 7.3: High level view of the RACME architecture.

on blocks which implement the different functionalities of the system and that exchange only little key information to each other. Doing this, we find a tradeoff between the advantages of the plugin architecture and its impact on the development costs (a plugin architecture has typically, at least in its first stages, bigger costs with respect to traditional solutions). A further advantage that derives from the modularization is the "natural" interoperability of the system; this plugin-like architecture allows to include the management of different kind of data (as XML or SysML) with little changes: this is particularly important if we consider the need of RACME to deal with functional specification written in SysML and with inputs documents which may contain several different kinds of data.

Each macro-functionality (the grey boxes of Figure 7.3) will be implemented by a set of classes referencing mainly to each other and operating with tables in a database.

Other concerns on the RACME architecture

Since RACME is a support for the V&V experts, its use should be internal to the company, without imposing any standard or restriction to the customers (e.g., imposing a format for the documentation). RACME data acquisition from input document is therefore managed through the concept of plug-in, that has to be seen as a customized "map" to associate a semantic value to different parts of a document, allowing to select information and importing them in the system.

RACME has critical relations with documents, and in particular "*.doc(x)*", "*.xls(x)*" and "*.pdf*" (these are in fact the main formats used by companies).

Facilities to easily interface with these kinds of documents are very important, especially: document import, parsing of the relevant information and database population. The basis to import and parse documents is as follows. In industrial documents, information is typically stored following specific patterns. For example, requirements are univocally identified using a specific format. Consequently, it is relatively easy to define a pattern, parse the document and extract the relevant information.

Also, data extraction is an important topic for the RACME architecture. Extracting data from documentation permits to consider (and consequently manage) information as objects independent from the documents in which they were originally contained. This reduces the inconsistencies that naturally rise in a development process, in which different groups write documentation at different stages. In RACME, information exists besides documents and when a new document is released its contents is checked with data already present in the RACME DBMS. Moreover, extracting data from a document naturally increases data visibility and accessibility.

The natural attitude of many companies to use tools running on Windows makes RACME Windows-oriented. This means that the tool can be developed for Window OS, without a real need to be generic. Also, the user interface is planned to run on a Windows machine and be Windows compliant. Such considerations have led us to implement RACME in C# [119]. This language, developed by Microsoft, eases handling of Microsoft Office documents and offers facilities to implement a Windows-compliant user interface.

The RACME prototype

Our first RACME prototype covers only a subset of the RACME requirements, but it allows to collect "field feedback" to evaluate the overall feasibility of RACME as support to real V&V and certification processes, and consequently to provide indications of possible modification and evolution. The lack of similar tools and consequently of experience in this kind of interaction makes the need of field feedback even more important: for the V&V experts at Resiltech S.R.L. which cooperated with us in the design, implementation and use of RACME this has been the first time that different pieces of V&V information (hazards, audit, requirements, mitigation, actions) have been related to each other not only in a conceptual way, but also through a supporting tool.

Our first prototype implements the features described in what follows, and summarized in Figure 7.4.

Main data structures of the inner RACME database. A big attention is put on the definition of data structures for the management of the requirements, documents, FTIs, tests, semi-formal representation of the system, and hazards.

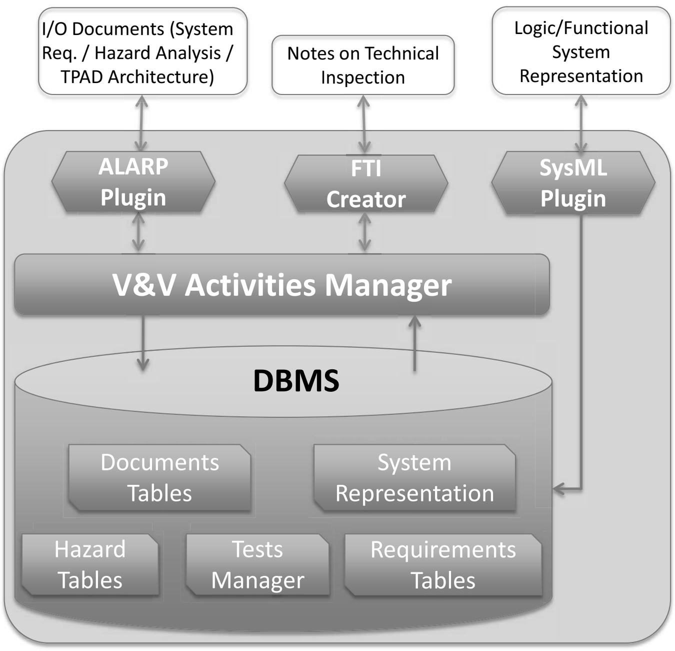


Figure 7.4: The RACME prototype.

This development phase has clarified a lot of "not easy to answer" questions, for example how to differentiate formats used to represent the information. In fact, each piece of information has its own format and can differ significantly from the others. From the analysis of the documentation of past projects, it has been possible to extract the main data needed to represent the key information. Inner tables have been optimized keeping in mind this set; moreover specific spare fields have been inserted to allow the management of anomalous representations. These inner tables have been related to each other and specific rules to guarantee consistency and mandatory fields have been defined and implemented in the database (in Figure 7.4 we show the key tables).

Storing the functional representation of the target system. During the V&V activities it is fundamental to continuously take a look at the physical components under analysis, in order to better evaluate risks and easily identify inaccuracies of documentation. To simplify this goal, in RACME a SysML representation of the target system is stored. This representation is imported through the *SysML plugin* and stored in table *System Representation*. This allows to select data or documentation related to different parts of the system.

Managing documents of the V&V process. Documentation produced in the project is acquired from the RACME prototype, versioned and made easy to recall and access. Information contained in RACME can be exported in order to semi-automatically create documentation from templates.

ALARP plugin. The *ALARP plugin* deals with the documentation format used in the project ALARP (A railway automatic track warning system based on distributed personal mobile terminals, [3]). Primary goal of ALARP is to build a safety-critical ATWS (Automatic Track Warning System) to recall the attention of a workgroup operating on a railway worksite about the presence of trains approaching the worksite. We plan to use ALARP as a pilot project to gain feedbacks from a real use case and to understand how to tailor RACME. Using RACME on a specific project requires a specific plugin to deal with the project documentation templates, formats and data.

Managing Technical Inspection (FTI). Management of the documentation allows to associate information to each document (e.g., document non-conformities). This naturally leads to the creation of FTIs documents. FTIs are created using the *FTI creator* plugin.

Managing Hazard Analysis. The implementation of the main structures of the database is coupled with the management of at least one V&V activity to allow the evaluation of the core features of RACME. In this version of RACME, we decided to implement the management of the hazard analysis process (hazard identification and creation of the Hazard Log).

Managing requirements, mitigation, tests plans and tests reports. One of the goals of RACME is to relate different activities on the basis of common data. The main information stream that links V&V activities is represented by the chain "requirements / mitigation (safety requirements coming from hazard analysis) / actions (safety tasks that have to be monitored) / tests plan and tests report". In the RACME prototype this chain is implemented, making possible to relate each requirement with the proof of its correct implementation in the system (test reports, documents, etc.).

7.3 A TESTING METHODOLOGY IN V&V AND CERTIFICATION PROCESSES

As previously illustrated, RACME does not contain built-in solutions to apply a specific V&V technique, or does not support a specific V&V activity (e.g., it does not implement a workflow devoted to the experimental evaluation). Specific V&V activities are executed in parallel with RACME, using methodologies and tools devoted to their execution. At "synchronization" points, inputs and outputs are exchanged with RACME, and checks are performed on the integrity and non-regression of requirements defined during the different stages of the design process.

RACME can integrate data produced using different and heterogeneous tools and methodologies, because it imports and parses data and documents and relates them, according to the rules specified in the definition of the workflow and specific RACME plugins. As a result, *any* V&V activity performed as part of a V&V process managed in RACME can interact with a proper RACME workflow. Certification evidence for the overall V&V process and the single activity is improved as a natural consequence of the usage of the RACME support.

From the discussion above, it can be noted that i) the methodology for the experimental evaluation can benefit from being complemented with RACME, and ii) RACME can benefit from its usage. Regarding point i), RACME merges challenges of data importing, archiving and retrieval, together with methodological support to V&V activities (including testing activity): archiving and reuse of data is one of the key topics we addressed when defining our methodology. Regarding point ii), RACME can benefit from the usage of a testing methodology because well-structured V&V activities ease their integration in the RACME workflow. Consequently, this eases providing certification evidence.

In this Section we show the potential interplay of our methodology for experimental evaluation and RACME, to bring evidence of the possible application of the methodology to industrial contexts. Then, we briefly introduce two case studies (that shall be explored in Chapter 8 and Chapter 9) where the experimental evaluation is performed in cooperation with industrial partners in the context of standard-compliant processes for V&V of safety-critical systems.

7.3.1 *Interplay of the methodology and RACME*

We start presenting the main usage of RACME when performing experimental evaluation (experimental evaluation may be named differently in the various standards, for example it is called it is called *dynamic analysis and testing* in the railway standard EN 50128 [42]). RACME includes a Test Manager which is devoted to the managing of inputs and outputs and requirements relations for experimental evaluation activities; also, plugins built ad-hoc for a specific project may complement the Test Manager. The main actions performed using RACME during the experimental evaluation activity are the following:

1. Extract the documents templates for tests plan and test reports that are stored in RACME.

2. Feed RACME with the information required to create and update FTIs (e.g., open a new FTI, modify an FTI due to system changes, add clarifications or answers, close the FTI when the request for changes is fulfilled).

3. Insert in RACME the different versions of the tests plan that are produced. RACME can import such versions, parse them to maintain the status of the

test plan and to establish relations between system requirements and tests, using the traceability matrixes included in the tests plan. Such relations are used by RACME to verify the correctness of traceability and to guarantee non-regression of requirements.

4. Similarly to the tests plan, RACME can import, parse and manage the various versions of the tests reports and related requirements traceability, and the tests data to automatically fill the tests report.

5. Through its management functionalities, RACME always maintains the up-to-date version of each documents, the status of the FTIs (typically it is open/close), and information on possible conflicts between requirements (identified through verifications on up-to-date traceability matrixes).

We can reasonably assume that the overall experimental evaluation process starts when the documents templates are downloaded from RACME and terminates when final versions of all reports are in RACME. We revise the 4 phases of our methodology, to understand the interactions of our methodology and RACME during the experimental evaluation process of a critical system in industrial practices. Such interactions are summarized also in Figure 7.5.

In phase 1, the templates are extracted from RACME; these templates shall be used for the rest of the activity. During this phase, the tests plan is produced and imported in RACME, with the support of the specific plugins developed in RACME. Also, information to open and update FTIs can be provided to RACME. The outputs that can be extracted from RACME in this phase are the updated FTIs and the current version of the tests plan (the most updated version of the test plan; a V&V expert may produce and import in RACME different versions of it).

In phase 2 and phase 3, the system is instrumented and the experiments are executed. Preliminary versions of the test report can be drafted in this phases; these versions do not contain the whole set of results, but include the description of the measuring instrument, the set-up used for the experiments, and the metrological characterization of the measuring system and of the expected results. Similarly to the previous phase, information to open and update FTIs (e.g., because errors in the system implementation are detected during the system instrumentation) can be provided to RACME. The outputs that can be extracted from RACME in this phase are the updated FTIs and the versions of the tests plan (e.g., the most updated version provided in phase 1) and tests report. Note that due to the rigor required in V&V and certification processes, possible modifications to the measuring instrument performed while iterating phase 2 and phase 3 must be attentively reported in the tests plan and tests report.

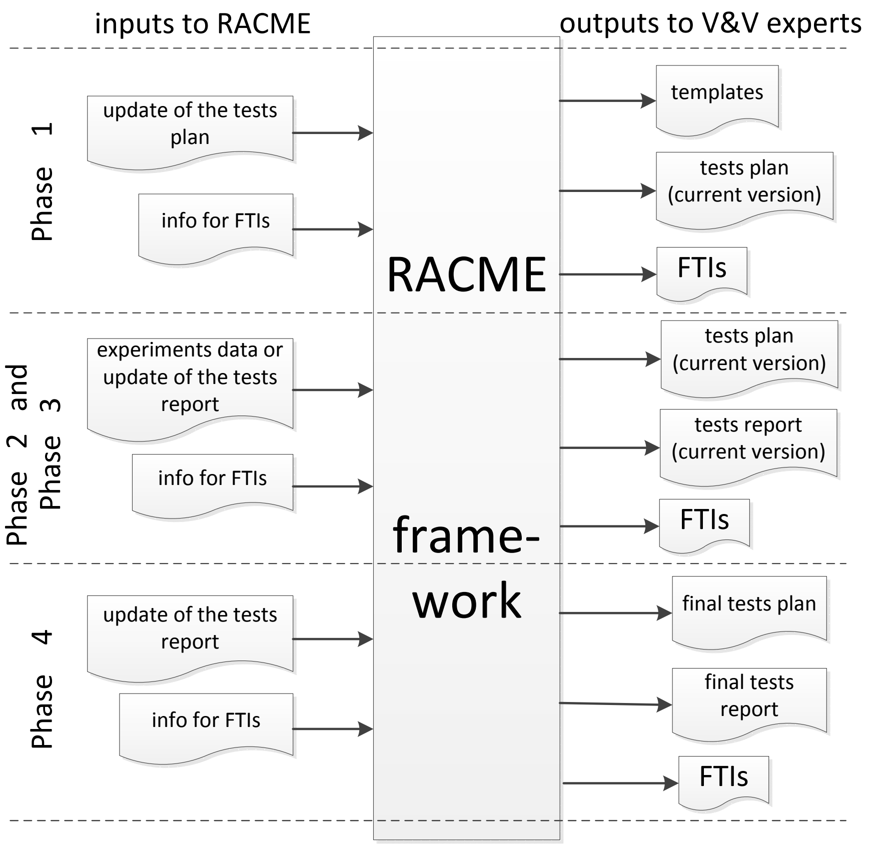


Figure 7.5: The experimental evaluation methodology and RACME.

In phase 4 the test outcomes are completely included in the tests report. Also in this phase FTIs may be opened (e.g., to signal potential problems identified during the analysis of results that are not part of the tests report) or updated.

The experimental evaluation activity concludes when the versions of the tests report and tests plan maintained in RACME are final. FTIs that are still open may be closed in successive phases of the V&V process.

We already discussed in Chapter 3 that experimenting is iterative, and our methodology is consequently designed to support such iterations. This iterative concept also applies to testing for V&V. For example, modifications of a system may require to change the tests plan or to re-execute tests (in fact, for certification purposes, tests are executed on the most up-to-date version of the system, and changes in parts of the systems may require to re-execute the tests related to those parts).

Finally, we note that the experimental evaluation phase terminates when the final versions of tests plan and report are presented: this may require several iterations, and changes in the system over a (reasonably) long span of time e.g., to meet feedbacks included in FTIs.

7.3.2 *Methodology applied to industrial case studies*

Two case studies explored in Chapter 8 and Chapter 9 are introduced to identify their relations to industrial V&V processes for certification purposes. In fact, the experimental evaluation in these case studies is conducted with the cooperation of industries and academia in the context of two broader projects. These case studies refer to two different safety critical systems compliant to the SIL 2 requirements according to the railway standards CENELEC EN 50126 [41], CENELEC EN 50128 [42], CENELEC EN 50129[43] (SIL stands for Safety Integrity Level 2; the mentioned standards propose both qualitative and quantitative classes for the safety of equipments, and SIL 2 quantitatively means that the Tolerable Hazard Rate THR per hour is required to be between $10^{-7} \leqslant \mathsf{THR} \leqslant 10^{-6}$ [43]). Validation processes including experimental evaluation activities are planned and executed for the two systems. Note that the second case study refers to the ongoing ALARP project where the RACME prototype is applied.

The first case study (Chapter 8) is developed in the context of the ongoing EU project ALARP, which focuses on the design, implementation and validation of a SIL 2 ATWS (Automatic Track Warning System) to alert trackside workers on trains approaching the worksite. The case study is devoted to the experimental evaluation of cheap COTS GPS devices, that are adopted to localize railway trackside workers. Since a key requirement in ALARP is the accurate localization of workers, the experiments performed aim to identify the localization error to provide feedbacks for the design of the localization solution [27], [28]. The activity performed here in cooperation with the Italian company Ansaldo STS was part of testing activities executed to support the designers, rather than final assessment activities performed on the prototype. However, these evaluations also allowed to accurately test one fundamental component of the ALARP system, that is the GPS device.

The RACME prototype is used in the ALARP project to support the V&V activity (see also the ALARP plugin presented in Section 7.2.3). The star schema and the related database that were built to support the experimental evaluation for the (fine-grained) storage of results can be uploaded in RACME, and using the ad-hoc plugin the database can be queried to (semi) automatically fill the tests report.

The second case study (Chapter 9) is in the context of the EU project SAFEDMI (Safe Driver Machine Interface for ERTMS automatic train control, [155]). The project focuses on the design, implementation and validation of a SIL 2 Driver Machine Interface (DMI) for trainborne equipment. In the case study we report our experience of fault injection testing on the DMI prototype, performed as part of the planned V&V activities. With respect to the previous case study, the system under test is in this case a (almost final) prototype. The tests plan is executing according to the methodology presented in Chapter 3.

At the time the experiments were planned and executed, the project was in its terminal phase and most of the documentation was consolidated. Consequently, in the application of the methodology, the documentation made available during the course of the project (in particular the system requirements and the design documents) constituted a fundamental input for the definition of the experimental evaluation.

As results of the fault injection tests we performed, the safety mechanisms of the DMI correctly identify the errors and a proper reaction is executed. Only in one case a slight violation of the requirement was observed, analyzed, its caused detected and a simple modification identified to solve the problem (this in a V&V process would correspond to opening an new FTI). Although the RACME prototype was not available at the time this experimental campaign was performed, we present it here as it gives a clear idea of how the experimental evaluation methodology can be applied in industrial practises.

8 EVALUATION OF COTS GPS DEVICES

It is well known that accurate localization in open space is nowadays an open challenge [132]. This fact is the main reason behind several works that aim to design algorithms to integrate, exploit and improve GPS-based technologies. Relevant examples are DGPS (Differential GPS, [106]), StarFire GPS [75], or RTK (Real Time Kinematic, [141]).

Each of these alternatives has both positive and negative aspects, as already pointed out in many comparison works as [132], [113]. Which alternative fits a certain system the best depends on the system requirements, such as the operating environment, the required localization accuracy and the admissible cost of the hardware devices.

This Chapter shows the experimental evaluation of COTS GPS devices, that was performed in the context of the ALARP project [3]. We consider the localization requirements of the ALARP system, which demands that a railway worker in a railway worksite is accurately localized for safety reasons, using low-cost GPS-based techniques. Although the sources of localization errors for GPS are well-known [106] and have already been widely analyzed in different contexts [132], [113], the specificity and different requirements of ALARP make a detailed investigation of localization errors in this context relevant.

The objective of the experimental evaluation here described is to experimentally assess if and to what extent cheap GPS devices can be successfully applied in the ALARP scenario. In particular, our work aims to investigate the localization errors, with the goal of quantifying the contribution of systematic and random errors achieved when using low cost GPS receivers and providing feedbacks to the designer of the localization solution in ALARP.

This Chapter contains results of a first run of the experiments, and conclusions and recommendation to be applied in successive experiments. Further details on this activity can be found in [28], [27].

The remaining part of this Chapter is organized as follows. In Section 8.1 the ALARP system and its localization requirements and devices are presented. In Section 8.2 the planned experimental activity is described; results are instead reported in Section 8.3. Finally, conclusions and remarks for the next iteration of the experiments are in Section 8.4.

8.1 CONTEXT, OBJECTIVES AND TARGET SYSTEM

We present the context where we operate, the objectives of the experimental evaluations, and the target system.

8.1.1 *The ALARP ATWS*

ALARP is a project partially funded by the European Commission whose goal is the study, design and development of a safety-critical Automatic Track Warning System (ATWS) for railway track-side workers.

The primary goal of an ATWS is to recall the attention of a workgroup operating on a railway worksite about the presence of a train approaching the worksite. The ALARP ATWS will be able to inform the trackside workers about approaching trains on the track, emergencies on tracks and tunnels nearby the workers (e.g., fires in a tunnel, toxic smoke, etc.), and escape routes in case of emergencies. Additionally, it will keep track of the status and position of the workers (to identify those at risk, or not responding, or to suggest escape routes).

The ALARP architecture will be based on the following components (see Figure 8.1) [161]:

- the track-side Train Presence Alert Device (TPAD, [161]), able to sense an approaching train on the interested track without interfering with the signaling system, using long-range multi-spectral cameras and eaves-dropping the train-network communication,
- a set of distributed, low-cost, wearable, context-aware, robust, trustable and highly reliable, wireless Mobile Terminals (MTs, [161]) to inform the workers about possible approaching trains and/or other events that could put at risk their safety. The Mobile Terminal (MT) will be able to generate alarms, and to communicate and interact through wireless connections with other MTs and the track-side TPADs.

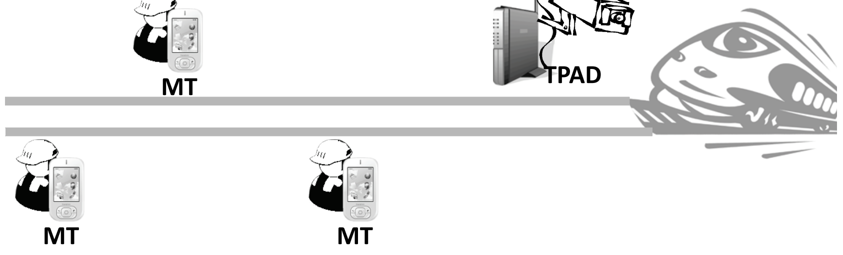


Figure 8.1: The ALARP scenario.

8.1.2 *Localization in a railway worksite*

A typical railway worksite, in which the workers (the MTs) need to be localized, is an operation area of maximum 700 m length, in which workers typically move on foot (slow movement speed). The worksite can be located in place possibly (partially) surrounded by foliage, in canyon, or near buildings (i.e., the MTs may suffer limited visibility of GPS satellites). In ALARP, strict requirements on localization accuracy are presented (in the order of less than one meter); also, the MTs should be low-cost and based on COTS hardware, including the GPS devices.

The objective of the experimental campaign performed is to assess the behavior (and error) of the COTS GPS devices identified for the ALARP prototype (see next Section) and the error correlation of such devices in worksite-like scenarios.

It is reasonable to expect errors due to satellite clocks (errors in the synchronization of the different satellite clocks, typically in the order of 0.8 m to 4 m) and ephemeral satellite orbits (errors in precisely establishing the spacecraft location, typically on the order of 0.8 m.) when receivers use the same satellite set. Also errors are expected due to ionospheric and tropospheric signal perturbation and delays (given by the transition of the signal through the troposphere and ionosphere), and due to the receiver's design (errors due to the specific design of the receiver). All these enlisted errors can be considered as systematic. At the same time, we can expect that errors due the receiver's thermal noise and external interferences exhibits negligible variations from a receiver to the others [106].

Conversely, we expect that multipath (reflection errors, one of the most significant and variable errors incurred in the receiver measurement process) affects randomly the localization error and it is expected as the major issue in localization measurements [106].

8.1.3 *The target system*

The target system is the ND-100S produced by Globalsat [69]. We also use another GPS device, the Garmin 18 LVC [68]. The Garmin 18 LVC is a GPS device of a higher category of price and performance than the Globalsat (it costs around three times the Globalsat ND-100S). Using two receivers of different quality allows to collect information on how the localization error varies depending on the device used and on the tradeoff in performances and costs.

8.2 THE MEASURING SYSTEM AND THE EXPERIMENTS PLAN

Table 8.1 contains key acronyms used in the following of this Chapter.

Acronym	Definition
R7	The Trimble R7 reference station
R8	The Trimble R8 rover
GS1	A GlobalSat ND-100S USB Dongle GPS receiver
GS2	A GlobalSat ND-100S USB Dongle GPS receiver
G18	A Garmin 18 LVC receiver

Table 8.1: Main acronyms used.

8.2.1 *The measuring system*

A reference system allows to compare the data collected using the GPS devices previously mentioned. The reference system is the Trimble system [174], [175] composed of a stationary reference station Trimble R7 [174] and a roving device Trimble R8 [175]. This Trimble system uses the GPS-based RTK [141] technology to calculate the position with accuracy in the order of few centimeters (it is by far more accurate than the others GPS devices considered as target system).

During the experiments, common laptops with OS Windows 7 are used to log the NMEA 0183 sentences (the protocol National Marine Electronics Association 0183 [136] defines the information that GPS devices communicate e.g., the estimated time and position, the visible satellites, the satellites used by the devices to compute their position, etc.) provided every second on USB ports by all devices involved (Globalsat ND-100S, Garmin 18 LVC, Trimble R7 and R8). Note that the NMEA sentences contain the time instant at which the sentence is generated, so we can log them without the need to investigate possible delays of the laptops in timestamping and data collection.

8.2.2 *Experiments description*

We previously discussed the error sources for the ALARP scenario; amongst those, some error sources are bounded to the characteristics of the environment in which the worksite is placed (multipath, interferences) and to the satellite visibility (satellite clock and ephemeris). Here, we focus on the case when devices are close to a cliff (or to the side of a high building), and are consequently subject to interference, multipath and limited satellites visibility. Devices may have partial or no satellites visibility for a short period of time. Each time a receiver loses and re-acquires satellites visibility, it may need to execute a transient phase in which the computation of their position is particularly unreliable. Additionally, the characteristic of the environment may increase multipath and interference errors.

The experimental campaign consists of four experiments, which are summarized in Table 8.2: the first column is the experiment ID, the second column is the description of the experiment.

Exp.	Description
Exp1	R8, GS1, GS2, G18 are stationary and measuring on the same point. R7 is close to them and with a free LOS (line of sight) towards them. The experiment provides information on the error magnitude in presence of big obstacles on one side.
Exp2	R7, GS1 fixed in a relatively good satellite visibility. The others move along an established path close to tall buildings. The experiment aims to capture the effects that tall obstacles have on the position error.
Exp3	Similar to Exp2, but all devices except R7 move along the established path. This experiment also aims to compare the behavior of GS1 and GS2.
Exp4	An *hybrid* path where GS1, GS2, G18 and R8 are roving alternating building and plain scenarios.

Table 8.2: The planned experiments.

The first experiment (Exp1) is conducted near a tall obstacle on one side, and aims to evaluate the accuracy of all the receivers involved in the experiments, doing stationary measurements in a benign scenario. The second and the third experiments (Exp2 and Exp3) refer to a building area's worksite scenario; they are mandated to study the systematic and random error when the GPS signal is potentially obstructed by possible buildings, which can induce high signal reflection and seriously affect the continuity of the GPS signal. Finally, in the fourth experiment (Exp4), the devices follow an hybrid path where GS1, GS2, G18 and R8 are roving alternating building and plain scenarios.

8.2.3 *Planning of the results*

A star-schema is created, and shown in Figure 8.2. This schema contains two dimensions, *Experiment* and *Receiver*, and several fact tables (one for each kind of NMEA 0183 sentence provided by the GPS devices). In other words, it consists of a set of star-schemas with the same two dimensions tables.

This star-schema is implemented in a SQL database, where NMEA 0183 sentences are imported; this will ease retrieving, analysis and comparison of results, for these experiments runs and the future experiments that are planned with the same set-up.

8.3 analysis of results

The experiments were executed during two raining days; consequently, the unstable weather conditions compromised part of the tests plan. Also, we must

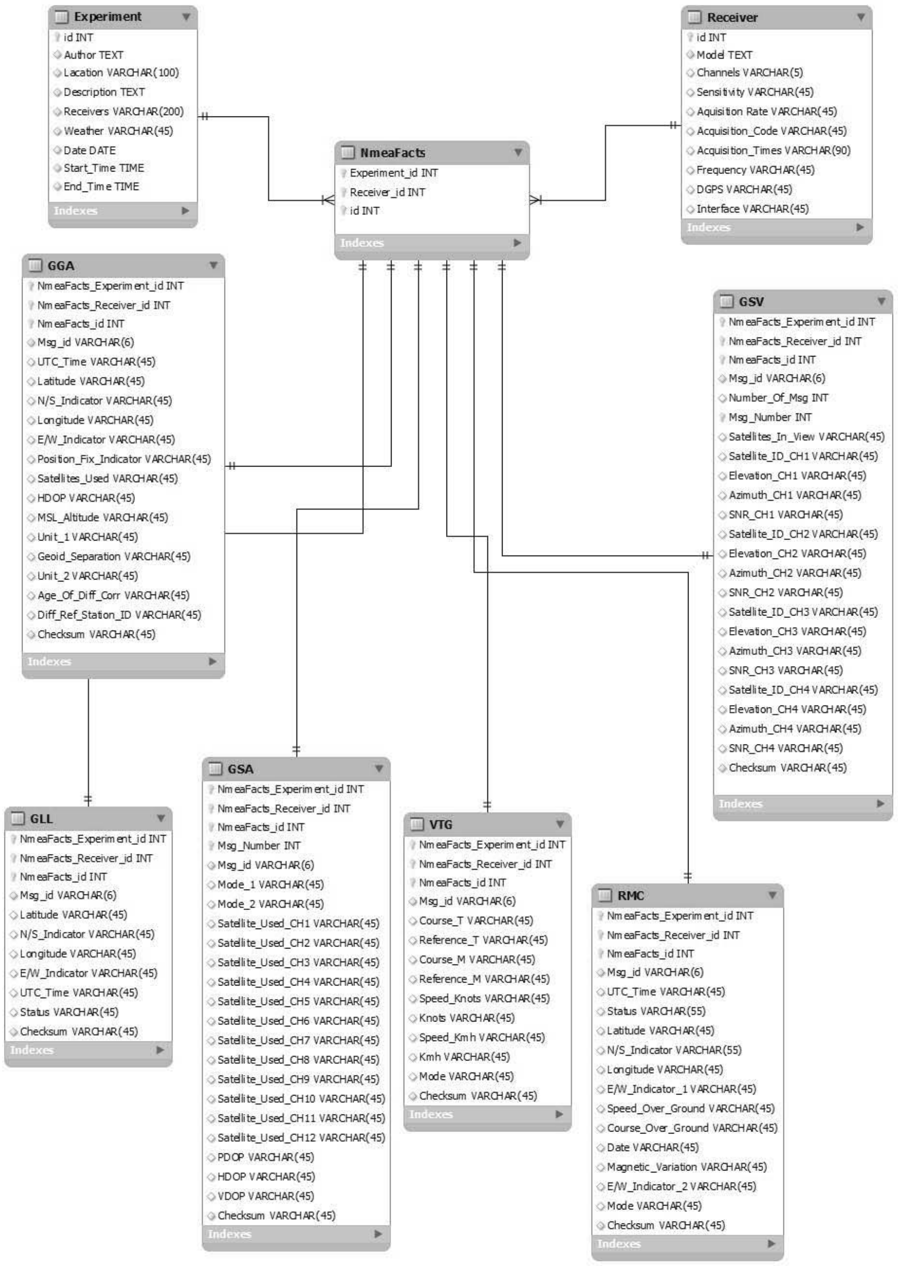
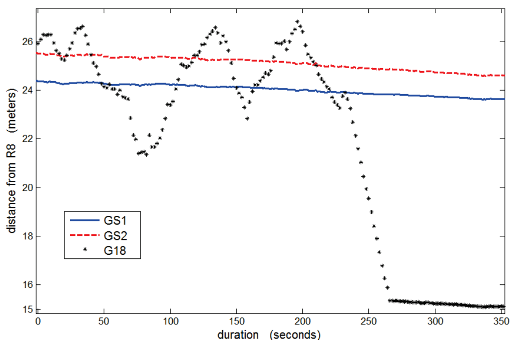


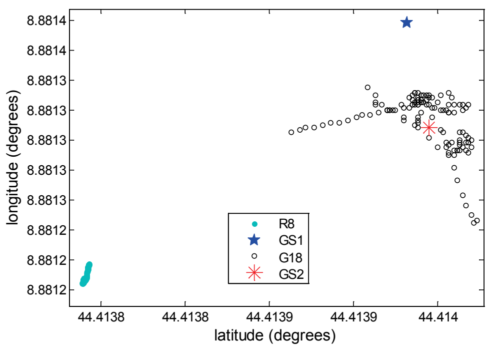
Figure 8.2: The star-schema designed for the experiments.

note that weather condition can influence localization results and constitute an additional source of error, thus making the analysis of the systematic and random contributions to the errors in the collected measures more complex.

We report and discuss the results of the experiments. In Figure 8.3a and Figure 8.3b results of Exp1 are reported (we report a trace of 350 s duration). Figure 8.3a shows distance of GS1, GS2 and G18 with respect to R8 during stationary measurements. GS1 and GS2 show no transient phase (see Figure 8.3a), and stabilize immediately (the slow convergence towards the x-axis is due to small adjustment of R8 in its position calculation; this behavior of R8 can be noted in Figure 8.3b in the bottom left corner). G18 exhibits a longer transient phase, but once it stabilizes, it locates itself far closer to R8 with respect to GS1 and GS2 (this distance is below 16 meters, whereas GS1 and GS2 are more than 20 meters far from R8).



(a) Distance of GS1, GS2 and G18 from R8.

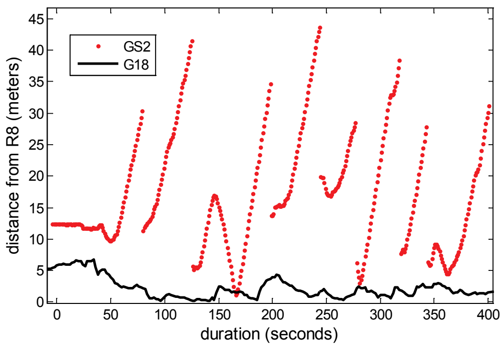


(b) Latitude, longitude of GS1, GS2, G18, R8.

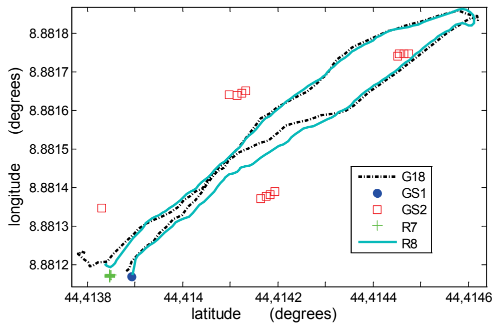
Figure 8.3: Results of Exp_1.

Figure 8.3b shows the latitude and longitude positions of GS1, GS2, G18 and R8, for the same set of data shown in Figure 8.3a. Position estimations of GS1 and GS2 are (relatively) close one to the other (average distance of the two estimations is 7.9 meters); this suggests that a systematic error can be identified in GS1 and GS2 in stationary conditions of measurements, although further investigations are required to confirm that.

Figure 8.4a and Figure 8.4b report results of Exp2. In Figure 8.4a, we present results collected while moving the devices for 400 seconds in a court next to tall buildings. In Figure 8.4b, we map latitude and longitude of the devices used for a duration of 110 seconds (to ease the readability of the figure, we only map a subset of the measures collected). G18 shows a reasonably stable behavior: its distance from R8 is always less than 7 meters. Also, it reacts quickly to directions changes and to speed variations (different walking speeds and direction changes that were inevitable during the experiment). Finally, note that in Figure 8.4b G18 often intersects the path of R8 (that is, our estimation of the true position). Also in Figure 8.4b in some points G18 deviates from R8 (see the lower part of lines of G18 and R8); this corresponds to a change of direction performed while very close to a tall building: G18 is unable to react quickly and to detect the direction change. GS1 and GS2 instead present a very irregular behavior. This is because they update their position estimation very scarcely: GS2 computes only 13 different positions in 110 seconds, as it can be noted from the few squares in Figure 8.4b.



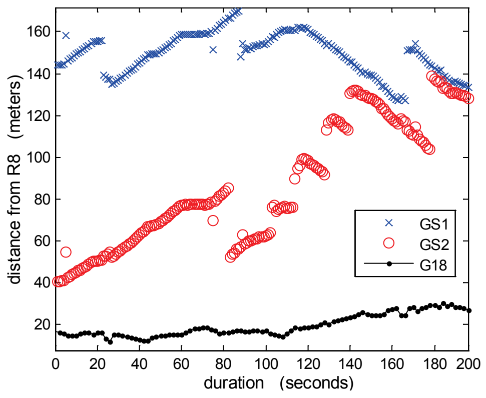
(a) Distance of GS2 and G18 from R8.



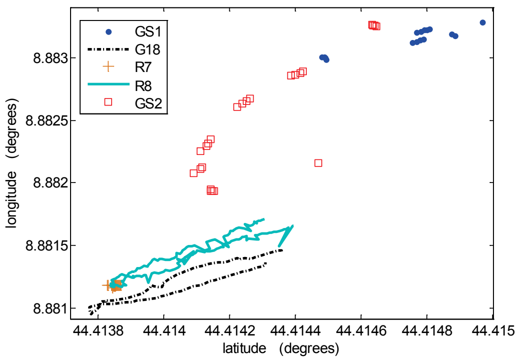
(b) Latitude and longitude of the five devices.

Figure 8.4: Results of Exp_2.

In Figure 8.5a and Figure 8.5b we present results for Exp3. Figure 8.5a shows the distance measured from R8 for GS1, GS2 and G18, during a run of 200 seconds; Figure 8.5b shows the latitude and longitude recorded by R7, R8, GS1, GS2 and G18 during the same run. The results show that G18 locates itself closer to R8 than GS1 and GS2 (the maximum distance of G18 to R8 is slightly above 30 meters). GS1 and GS2 exhibit a similar behavior, but they both locate themselves very far from R8 (from 40 m up to more than 160 m).



(a) Distance of GS1, GS2 and G18 from R8.



(b) Latitude and longitude of the five devices.

Figure 8.5: Results of Exp_3.

Finally, in Figure 8.6 we give the result of Exp4, which lasts 890 seconds and shows an hybrid path where GS1, GS2, G18 and R8 are roving alternating building and plain scenarios. Figure 8.6 reports the distance of GS1, GS2 and G18 from R8. As in the previous cases, G18 performs better than GS1 and GS2. In two parts of Figure 8.6 (around second 200 and second 700), traces significantly deviate from their average distances; this is due to the passage below a bridge.

It is interesting to compare the number of satellites that R8, GS1, GS2, and G18 see during this experiment on an hybrid path. R8, which is able to detect

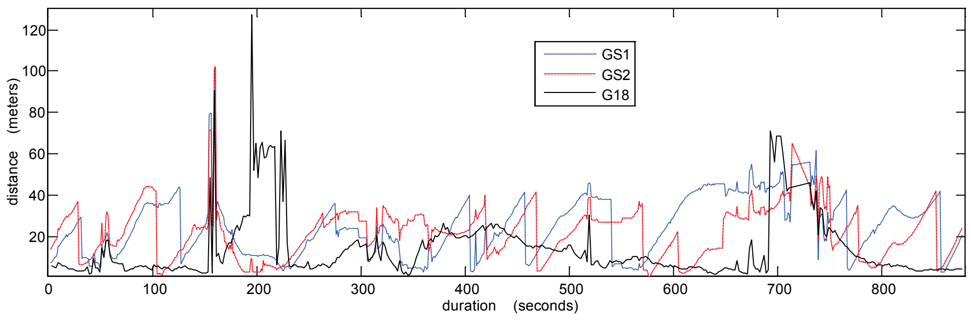


Figure 8.6: R8, GS1, GS2 and G18 roving on an hybrid path (Exp4).

both GPS and GLONASS (GLObal NAvigation Satellite System, [78]) satellites, always sees from 16 to 17 satellites. G18, instead, sees from 7 to 11 satellites (the average is 10.89). GS1 and GS2 respectively show an average of 9.49 and 9.67 satellites; GS1 ranges from 9 to 11 satellites, whereas GS2 ranges from 9 to 12 satellites.

8.4 CONCLUSIONS AND RECOMMENDATIONS

Accurate localization of workers in a railway worksite is a challenging task, due to specific characteristics of the environment and of possible localization errors. We presented a set of experimental measurements aimed to evaluate the localization errors of low-cost GPS devices, and in particular to distinguish the systematic and random contribution to the localization error. This work aims to provide a useful input for the context of the ALARP system in which accurate localization of workers in railway worksites is required using low-cost GPS devices.

The Chapter detailed the motivations and planning of the experiments, the measuring instrument and its set-up, and the results achieved through a first test session to evaluate three popular GPS devices GS1, GS2 and G18.

To summarize the achieved results, the following key observations can be derived. First, possible systematic contributions to the localization error can be identified for GS1 and GS2 when doing stationary measurements; from the figures shown it appears that for a short period of time, in similar environmental conditions and satellite visibility, similar localization errors are measured. On the contrary, the behavior of GS1 and GS2 looks more unstable during motion measurements. In fact, GS1 and GS2 have shown irregular sampling period, where data are not continuously refreshed; if this problem cannot be fixed or mitigated (we cannot here exclude possible configuration or setup errors in GS1 and GS2), GS1 and GS2 may result inadequate for workers positioning in a railway worksite.

Second, during motion measurements, encouraging results were instead given by G18, whose estimated positions always fluctuate around the true values (the values computed by R8).

Results and observations presented constitute the starting point to define new experiments and collect new data, to characterize the localization error that should be expected in a railway worksite and to provide additional feedbacks to the designers of the localization solutions that shall be applied in ALARP. Ongoing activity is aimed to extend the experimental cases considered and, possibly, the number and type of GPS receivers.

9 EVALUATION OF A RAILWAY EMBEDDED SYSTEM

In this Chapter we present the activity for the experimental evaluation of a safe train-borne Driver Machine Interface (SAFEDMI, [29]), that we performed as part of the V&V activities planned in the context of the SAFEDMI [155] project. The measuring system built for the purpose and the results obtained on the assessment of the DMI are scrutinized along basic principles of metrology and good practices of fault injection.

We present a fault injection instrument and some experiments for the evaluation of a safe railway train-borne Driver Machine Interface (SAFEDMI, [155]). The safety requirements of the SAFEDMI are classified as SIL 2: we evaluate the behavior of the SAFEDMI and of its safety mechanisms (mainly in terms of reaction to the faults injected) through software-implemented fault injection (SWIFI). The evaluation activity performed according to our methodology provides experiments that are easy to reproduce, and basics from metrology are applied to improve confidence in the results. Further details are in [29], [36].

The rest of the Chapter is organized as follows. In Section 9.1 we introduce the SAFEDMI system. In Section 9.2 we define our activity, then in Section 9.3 we show the fault injection instrument and in Section 9.4 we depict results. Conclusions are in Section 9.5.

9.1 THE SAFEDMI SYSTEM

In railway train-borne equipment, the SAFEDMI [29] acts like a SIL 2 bridge between the operator (the train driver) and the EVC (European Vital Computer: it supervises the train movement). SAFEDMI communicates with the EVC as a slave; it acquires and manipulates driver's commands (using a keyboard) for the EVC and it transforms EVC commands in graphical and audible information (using an LCD screen and audio devices). The SAFEDMI itself is composed of two components: the Driver Machine Interface (DMI) and the Bridge Device (BD). The DMI is the core of the SAFEDMI: it manages the communication activities with the EVC, with the BD, and with the driver (through a LCD screen and a keyboard). The BD is a wireless access point that allows configuration and diagnostic activities.

In the experimental activity reported in this Chapter we will focus only on the DMI.

9.1.1 *The DMI operational modes*

The DMI has four operational modes (or states): Start-up, Configuration, Normal and Safe mode. In [29], another operational mode that allows automatic restarts of the DMI is presented. This operational mode is not implemented in the prototype that we are evaluating here, and consequently it will not be described here.

In *Start-up mode* the initialization procedures and the thorough testing of all devices are performed; diagnostic functionalities are also available. From Start-up mode, the modes Configuration, Normal and Safe can be entered. In *Configuration mode* safe software upload and configuration are performed by means of wireless communication; after a successful configuration session, a restart of the DMI is needed (transition to Start-up mode). In *Normal mode* the DMI produces graphical and audio information to support train driving, as well as it acquires and processes driver's commands; periodic testing activities are performed and diagnostic functionalities are available. *Safe mode* is entered when a malfunctioning is detected and attempts to restart the DMI have failed. This mode prevents further operations of the DMI. In this mode the LCD backlight lamps are switched off, and the keyboard and EVC communications are disabled.

9.1.2 *DMI architecture and mechanisms*

Regarding the hardware, the DMI has a non-redundant hw architecture including COTS components. Absence of hardware redundancy imposes a more sophisticated software architecture, since safety requirements are accomplished only by software mechanisms.

The software architecture is composed of five C modules (Global monitoring, Checks and tests, Operations, Communications, I/O Manager) that are possibly activated/deactivated when the DMI transits from one mode to another.

The *Global monitoring* module performs the role of execution monitor, software watchdog, log manager, and diagnostic manager; it recognizes the conditions that cause an operational mode change of the DMI and performs the operations needed to reconfigure the system when a mode change happens. The thread *execution monitor* of the Global monitoring module performs transition to Safe mode whenever it receives notification of error detection by any safety mechanism; it is a high priority thread that cyclically executes once every 100 ms. The *Checks and tests* module performs checking and testing activities. The *Operations* module contains the software objects involved in the implementation of the functional requirements of the operational modes, as visualization and audio emission procedures. The *Communications* module handles the EVC-related communications, while the *I/O Manager* module contains the drivers of the DMI hardware devices.

Regarding the safety mechanisms applied to guarantee SIL 2, in SAFEDMI the reactive fail-safety principle was adopted [43]: as a consequence, efficient error detection plays a crucial role. According to the safety standards [43], [42], the DMI error detection addresses random faults that cover both permanent and transient faults and residual systematic faults (mainly software design faults that have not been identified by fault prevention and removal during development). Due to lack of hardware redundancy, active permanent and transient faults in the hardware and residual systematic faults are detected on the basis of their effects on software execution through periodic test procedures and mechanisms for on-line (run-time) error detection, mainly i) data acceptance/credibility checks [152], ii) control flow monitoring [125], and iii) multiple computation and comparison [164].

9.2 PLANNING OF THE MEASURING SYSTEM AND THE EXPERIMENTS

We identify the environment and the target system, the quantities to assess, the workload, the faultload, the experiments to execute and the structure of results.

Environment and target system. The target system is the DMI prototype shown in the left part of Figure 9.1. The DMI OS is the real-time OS VRTX [128]. The DMI prototype we used for the evaluation is connected to a PC with OS Microsoft Windows (shown in the right part of Figure 9.1) where the applications EVC Packet Generator (a simulator of the EVC [156]) and the diagnostic tool D360 (a tool that receives events and information from the DMI and logs them [156]) are executing.

The EVC Packet Generator does not require manual interactions of the operator with the DMI: as a consequence possible variations in different executions of the same experiment due to manual interactions (the operator may commit errors, or may press buttons with different timings in different executions) are avoided. The EVC Packet Generator generates the same exact workload for each experiment execution, and with the same timings (we will further discuss the timings of workload generation in Section 9.4).

Quantities to assess. The relevant quantities to assess are both countable dynamic quantities with (expected) negligible uncertainty and non-countable time-related quantities with non-negligible uncertainty.

Countable quantities are (number of):

- faults injected,
- errors detected,
- safe failures (i.e., after error detection, Safe mode is successfully entered preventing further DMI operations),

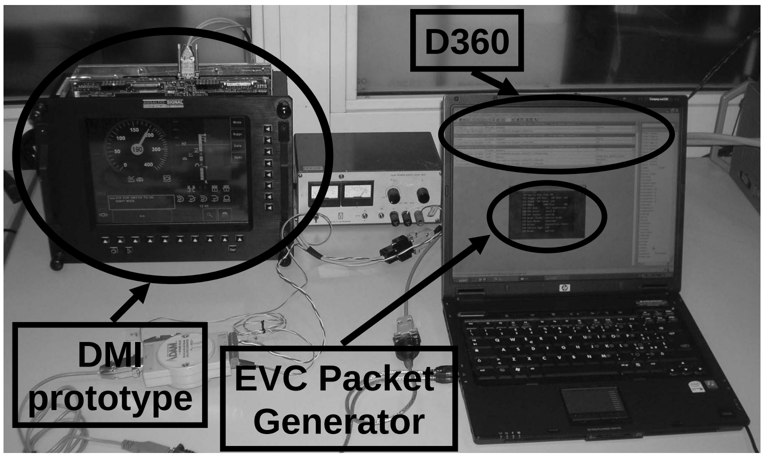


Figure 9.1: DMI, EVC Packet Generator and D360.

- system failures, and
- recoveries.

Non-countable quantities are instead:

- latency (time elapsed from the injection of a fault and its detection), and
- reaction (after detection, time needed to transit to Safe mode).

Workload. The workload is created by the EVC Packet Generator that communicates with the DMI. We distinguish between two different workloads: W_ID Startup and W_ID Normal.

The first one is composed of the communications performed by the DMI from the power-on to train mission successful termination (the train successfully arrives at destination), transition to Safe mode or system failure; note that while the DMI is in Start-up mode the communication with the EVC is not active, and thus only scheduled activities of the DMI are performed (periodic tests and boot operations). The second one is composed of the communications executed since the DMI enters Normal mode up to train mission successful termination, transition to Safe mode or system failure.

This workload is specific for the Normal mode and the Start-up mode. We do not identify a workload specific for the Safe mode and the Configuration mode: in Safe mode the DMI does not perform safety-critical operations (the communication with the EVC, the keyboard and the LCD screen are disabled) and the Configuration mode is not testable in current DMI prototype.

Faultload. We execute software-implemented fault injections (we do not have the tools to consider hardware fault injection as a viable option). We perform run-time injections to inject faults at specific time instants. Due to the limited possibility to operate at the lower levels of the DMI system, we can only insert faults in the application level of the DMI (e.g., to modify the content of a variable or to alter the execution flow); we emulate i) hardware faults by directly injecting the errors and ii) software (systematic) faults.

The faultload we present is the set of faults shown in Table 9.1. For each fault we define an ID (column Fault_ID) and a description of the fault. The selected faultload addresses some of the critical functionalities of the DMI.

Fault_ID	Description and additional comments
CFM_goto	A *goto* is inserted in the function that controls the LCD lamp, and forces an improper transition in the control flow graph.
CFM_check	The thread Checks (it performs checking activities) is killed while the DMI is executing, thus altering the correct execution flow at task level.
DataAcc	Visualization messages received from the EVC are not elaborated correctly by the DMI (the data received are not computed correctly).
DuplExCPU	In the DMI, two identical graphic images are created in two different RAM areas and compared: the CPU makes an error while creating the text message of one of the images.
CFM_signature	An erroneous signature is inserted in the control flow monitoring graph of the audio output management function.

Table 9.1: Definition of the faultload.

Specification of the experiments. The experiments defined are reported in Table 9.2: for each experiment we define an ID (column Exp_ID), the selected workload (column W_ID), the fault injected (column Fault_ID) and the time instant, after power-on of the DMI, in which the injection is performed (column Time).

For each experiment, the system is run until mission successful termination, transition to Safe mode or system failure. In Exp_1, Exp_3, Exp_4, Exp_5, and Exp_6 the injection is performed 30 seconds after the power-on of the DMI (the injection is performed while the DMI is in Normal mode); in Exp_2 the injection is performed 2 seconds after the power-on of the DMI (the injection is performed while the DMI is in Start-up mode).

Structure of the results. As shown in Figure 9.2, results are organized by a star schema: the facts table is table Measurements Results, and the dimensions tables are tables Experiments, Target System, Workload, Faultload and Safety Mechanisms.

Exp_ID	W_ID	Fault_ID	Time
Exp_1	Normal	CFM_goto	30
Exp_2	Startup	CFM_goto	2
Exp_3	Normal	CFM_check	30
Exp_4	Normal	DataAcc	30
Exp_	Normal	DuplExCPU	30
Exp_6	Normal	CFM_signature	30

Table 9.2: Definition of the experiments.

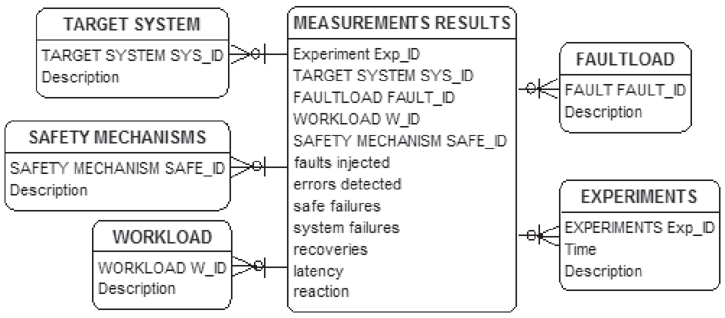


Figure 9.2: Structure of the data organized following a star schema.

9.3 MEASURING SYSTEM DESIGN, IMPLEMENTATION AND ASSESSMENT

We present the measuring system and we investigate its quality and the validity of the results it collects.

9.3.1 *Design and implementation*

The measuring system and its interactions with the target system are shown in Figure 9.3 and explained in what follows.

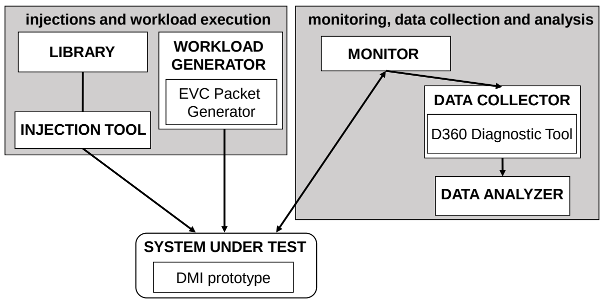


Figure 9.3: The measuring system and its connections with the target system.

We subdivide the measuring system in two functional blocks (the grey blocks of Figure 9.3). The first block, that is composed of the software C components library, injector and workload generator, deals with the injections and the workload execution: its function is to execute the experiments defined in Table 9.2. The functions of the second block are monitoring, data collection and analysis: this block monitors the target system, collects measurements results and analyzes them. The software components monitor (developed in C), data collector and data analyzer belong to this functional block.

Regarding the first block, the *workload generator* is the EVC Packet Generator located on the PC connected to the DMI (described in Section 9.2 and shown in the right part of Figure 9.1); the library and injection tool are instead both located on the DMI.

The *library* is the mean to inject the available faultload: it enlists the available faults as well as the methods to inject them in the DMI. The identified faults are implemented adding extra code in genuine DMI functions or developing additional functions that are not part of the DMI genuine software. The number of instructions needed to inject a fault is always small, and these instructions are fast to execute: the perturbation they introduce on system scheduling and the impact on the overall computational load can be considered negligible.

The *injection tool* allows to perform the run-time injections in the DMI. It is a cyclic, light and low-priority thread active on the DMI. This thread executes cyclically once every 1000 ms with a deadline of 2000 ms. The injection tool reads from a configuration file the instructions about the experiment to execute (Fault_ID and injection Time), and uses the library to select and inject the faults. The injection tool can inject a single fault or a sequence of faults at specific time intervals one from the others.

Regarding the components of the second functional block, the *data collector* (or *logger*) is the D360 diagnostic tool described in Section 9.2 and located on the PC connected to the DMI (this PC is shown in the right part of Figure 9.1). It receives, logs and organizes information received from the *monitor*, which executes on the DMI to timestamp events and to communicate events and related timestamps to the data collector. The data collector and the monitor communicate using a dedicated serial channel, different from the serial channel for the communication between the EVC Packet Generator and the DMI.

The monitor is an extension of the DMI log manager thread, that is a DMI genuine thread used for diagnostic activities (so we do not introduce a new thread in the system). The log manager thread is the thread with the lowest priority in the DMI, and it has no deadlines: it executes only when other threads are not running. As a consequence, to provide precise timestamping of events it is necessary to collect each time instant (by invoking the DMI system call *getTime*) as an atomic action with the raising of the event.

To reduce and control the system perturbation introduced by the monitor, only events that are fundamental to collect the relevant quantities (defined in

Section 9.2) are logged, and detailed traces of activities performed by the DMI are not collected.

Finally, the *data analyzer* is an analyzer (e.g., an OLAP tool) that executes offline to exploit the measurements results collected.

9.3.2 *Assessment of the measuring system and of the confidence of the results*

We assess the quality of the measuring system along the principles of experimental evaluation and fault injection and the confidence of results through principles of measurement theory: in particular, we comment on the resolution, intrusiveness, repeatability and uncertainty showed by the measuring system.

The *resolution* of the measuring system is investigated only for time-related quantities. System resolution for time instants is 2 ms; it is the resolution of the DMI timer used as the base for the activities of the scheduler and of all threads.

Three components of the measuring system may perturb the DMI: the library, the injection tool and the monitor. To investigate *intrusiveness* we need to analyze perturbations in time and memory. Memory perturbation is negligible, since the executable files, the dedicated variables and the dedicated memory areas of library, injection tool and monitor are very small compared to the DMI memory.

Time perturbation needs a deeper investigation. The injections are performed through few, quick instructions (they are not time-consuming operations as erasing memory areas or accessing a significant quantity of data) that are executed at worst in few microseconds. Both the injection tool and the monitor are low-priority threads that execute mainly when other threads are not running, to be as low intrusive as possible. The monitor sends data to the data collector using a completely dedicated communication channel: thus this communication does not alter the communication between the DMI and the EVC Packet Generator.

To further analyze intrusiveness, a schedulability analysis of DMI threads has been performed using the SchedAnalyzer [156] tool (it provides a pessimistic estimation of the CPU computational load of the overall set of threads on the CPU): it resulted that the set of threads is schedulable (threads deadlines are guaranteed to be met, and there is enough CPU free time to guarantee that the injection tool and monitor threads will execute without influencing other threads execution) [156]. Thus, considering that resolution is 2 ms, we can state that intrusiveness is negligible.

Repeatability instead cannot be guaranteed. The injection tool is a low priority thread: this provides low intrusiveness, but it affects repeatability. In fact there are no guarantees that the injections are performed exactly at due time instants. The EVC Packet Generator affects repeatability as well: despite it is supposed to generate always the same workload with the same exact timing, such exact timing is not guaranteed because of the non-real time OS (Microsoft Windows in use.

In fact, the experiments Exp_1, Exp_4, Exp_5, and Exp_6 (Figure 9.4a, Figure 9.5b, Figure 9.6a and Figure 9.6b) show a high standard deviation of latency, being 489 ms in Exp_1, 518 ms in Exp_4, 666 ms in Exp_5 and 953 ms in Exp_6. This high standard deviation indicates somehow low repeatability.

As previously said, we have a limited number of observations: consequently we compute a *Type B uncertainty* through an investigation of the system behavior instead of a Type A uncertainty computed through standard deviation. Type B uncertainty is estimated for time-related measurements as follows. When an event is raised, the *getTime* system call is invoked as an atomic action with the event: the contribution to uncertainty of this block of instructions is orders of magnitude smaller than 2 ms (it is at worst microseconds). The resolution of the target system (2 ms) is the most significant contribution to uncertainty, while other contributions to uncertainty could be considered negligible. According to [95], in such situations the true value is expected equally distributed in an interval given by the measured value and the measured value plus the resolution (e.g., if 10 ms is the measured value and the resolution is 2 ms, the true value is expected within the interval $[10; 12]$ ms). The expected true value should be set as the midpoint of the identified interval with an uncertainty of *at most* half the interval (e.g., if the interval is $[10; 12]$, the expected true value is 11 ± 1 ms and confidence 1). However, our purpose is to estimate the safety of a critical system: we prefer to differentiate from the approach proposed in [95] and to report an uncertainty that is conservative, meaning that it must never err on the side of being too small. Consequently, for each event, we pessimistically consider that the corresponding time instant is collected with uncertainty of ± 2 ms. The uncertainty of the two time intervals latency and reaction is finally set to ± 4 ms.

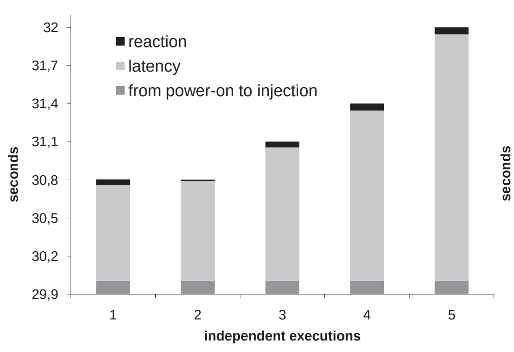
9.4 ANALYSIS OF THE RESULTS OF THE DMI

Having shown that the measuring system is adequate and able to provide trustful results, now we analyze these results in order to understand how satisfactory is the DMI behavior.

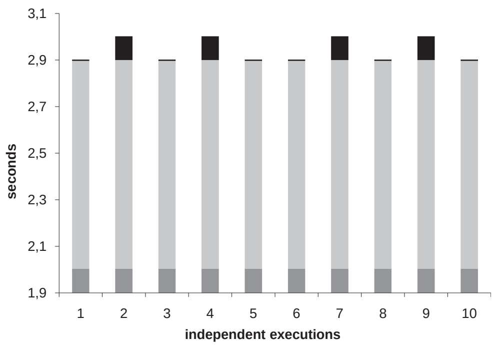
The results of the experiments are shown in Figure 9.4, Figure 9.5 and Figure 9.6. Each experiment consists of five or ten executions. In the figures, the time interval elapsed from power-on of the DMI to the injection is represented in dark grey, latency is represented in light grey and reaction is represented in black. The errors are always detected by the DMI safety mechanisms and the Safe mode is always successfully entered.

According to system requirements, reaction is always expected to be smaller than 100 ms. In the experiments performed, the observed reaction is at worst 102 ± 4 ms (this value of reaction is measured in Exp_2). Although the value is at the limit, and some of the results obtained show a violation of the requirement, nevertheless this has not been considered particularly serious. In fact, the explanation of this violation can be traced considering the execution monitor -in charge of performing transition to Safe mode- which is a high priority thread with a cycle of 100 ms. From here it has been very easy to find how to fix this problem and to provide a guarantee of being able to respect the 100 ms deadline: it is sufficient to slightly shorten the cycle time of the execution monitor (e.g., to 96 ms) and still proving the schedulability of the entire set of tasks. However we are aware that further observations are necessary and need to be performed to confirm that such requirement on the reaction is fully respected.

As previously said, in Exp_1 we note the low repeatability of the five executions. Latency varies significantly from an execution to another. The measured values of reaction vary too. The standard deviation of the latency is 518 ms and the standard deviation of the reaction is 19 ms.



(a) Five executions of Exp_1.

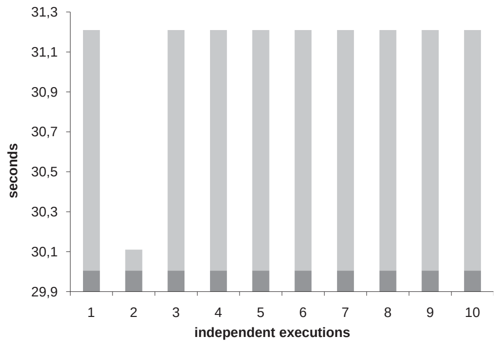


(b) Ten executions of Exp_2.

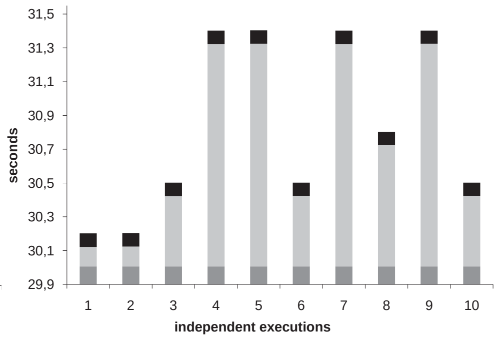
Figure 9.4: Results of Exp_1 and Exp_2.

In Exp_2 the injections are performed while the DMI is in Start-up mode. In Start-up mode there are no interactions between the DMI and the EVC Packet Generator: as a consequence the EVC Packet Generator does not affect the repeatability of the experiment. In fact latency varies slightly in the ten executions of Exp_2 (minimum is 892 ± 4 ms, maximum is 896 ± 4 ms and standard deviation is 2 ms). Instead the reaction varies significantly: reaction is 6 ± 4 ms in six executions (the execution monitor activates slightly after the error is detected i.e., the execution monitor commands transition to Safe mode immediately after error detection) and 102 ± 4 ms in four executions as observed before.

In Exp_3 shown in Figure 9.5a the injections are performed in Normal mode. Ten executions of the experiment are performed: latency and reaction are stable values, with the exception of the second execution of the experiment. In fact latency is 1204 ± 4 ms in nine executions and 102 ± 4 ms in one execution (the second execution).



(a) Ten executions of Exp_3.

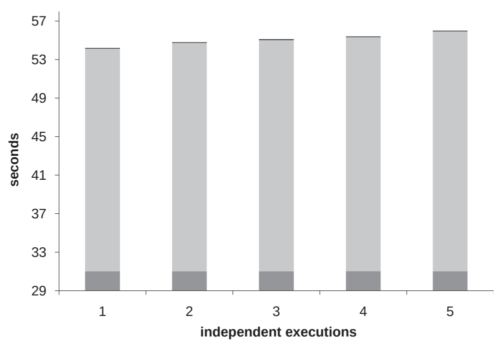


(b) Ten executions of Exp_4.

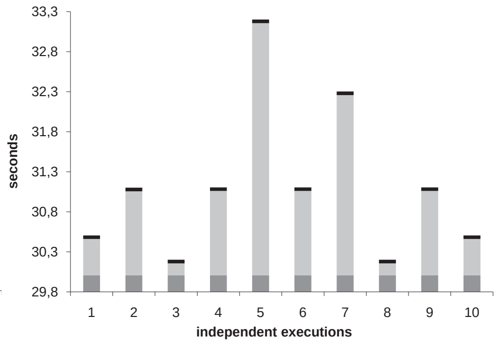
Figure 9.5: Results of Exp_3 and Exp_4.

This regularity appears to be due to the fact that the safety mechanism detecting the CFM_Check fault (the execution monitor) executes independently from the workload. Probably the low latency of the second execution is due to minimal variations in scheduling activities of the DMI. In all executions, the reaction is substantially null (0 ± 4 ms). The execution monitor in this case both performs the detection of the errors and commands the transition to Safe mode: thus the distance between the two events is only of a few instructions.

In Exp_4, Exp_5 and Exp_6 (shown respectively in Figure 9.5b, Figure 9.6a and Figure 9.6b), latency varies significantly from an execution to another, but the reaction is a stable value (in the experiments, standard deviation of reaction is close to zero).



(a) Five executions of Exp_5.



(b) Ten executions of Exp_6.

Figure 9.6: Results of Exp_5 and Exp_6.

The cause is that the safety mechanisms that detect the errors in these experiments are functions of high priority threads that are run just before the execution monitor. These positioning in the schedule to a short, fixed distance determines a reaction time which is not affected by the workload.

9.5 CONCLUSIONS AND RECOMMENDATIONS

In this Chapter software-implemented fault injection has been applied on a DMI prototype. The application of our methodology allowed collecting confident results, which showed an adequate behavior of the safety mechanisms. Trustfulness in results has been estimated satisfactory and the experimental campaign has shown that the safety mechanisms of the DMI correctly identify the faults injected and a proper reaction is executed.

In particular the faults injected were almost always properly detected and Safe mode correctly entered (thus preventing possible unsafe operations of the DMI). Only in one case a slight violation of the requirement was observed, analyzed, its caused detected and a simple modification identified to solve the problem.

CONCLUSIONS

The key role of computing systems and networks in a variety of high-valued and critical applications justifies the need for reliably and quantitatively assessing their characteristics. It is well known that the quantitative evaluation of performance and dependability-related attributes is an important activity of fault forecasting, since it aims at probabilistically estimating the adequacy of a system with respect to the requirements given in its specification.

Quantitative system assessment can be performed using several approaches, generally classified into three categories: analytic, simulative and experimental. Each of these approaches shows different peculiarities, which determine the suitableness of the method for the analysis of a specific system aspect. The most appropriate method for quantitative assessment depends on the complexity of the system, its development stage, the specific aspects to be studied, the attributes to be evaluated, the accuracy required, and the resources available for the study.

Focusing on experimental evaluation, increasing interest is being paid to quantitative evaluation based on measurement of dependability attributes and metrics of computer systems and infrastructures. This is an attractive option for assessing an existing system or prototype, because it allows monitoring a system to obtain highly accurate measurements of the system in execution in its real usage environment.

A mandatory requirement of each experimental evaluation activity is to guarantee a high confidence in the results provided: this implies that the measuring system (the instruments and features used to perform the measurements), the target system and all factors that may influence the results of the experiments (e.g., the environment) need to be investigated and that possible sources of uncertainty in the results need to be addressed.

Current situation is that, even if the measuring systems are carefully designed and actually provide confident results, that are not altered due to an intrusive set-up, badly designed experiments or measurement errors, there is seldom attention to quantify how well the measuring system (the tool) performs and what is the uncertainty of the results collected. Methodologies and tools for the evaluation and monitoring of distributed systems could benefit from the conceptual framework and mathematical tools and techniques offered by metrology (measurement theory), the science devoted to studying the measuring instruments and the processes of measuring. In fact metrology has developed theories and good practice rules to make measurements, to evaluate measurements results and to characterize measuring instruments.

Additionally, well-structured evaluation processes and methods are key elements for the success of the experimental evaluation activity. The approaches to assess algorithms and systems are typically different one from the others and lack commonly applied rules, making comparison among different tools and results difficult. Despite the fact that sharing results and comparing them is acknowledged of paramount importance in the current dependability research community, it is a matter of fact that in the field of dependability the approach to quantitatively assess algorithms and systems is not univocal, but generally varies from a work to another, making the comparison among different tools and results quite difficult, if not meaningless. Should structured, fully depicted and trusted results be shared, then tools and experiments could be better compared.

Starting from these observations, this Thesis proposes a general conceptual methodology for the experimental evaluation of critical systems. The methodology, subdivided in iterative phases, addresses all activities of experimental evaluation from objectives definition until conclusions and recommendations. The methodology tackles two key issues. The first is providing a metrological characterization of measurement results and measuring instruments, including the need to attentively report a description of such characterization. The second is proposing techniques and solutions (mainly from OLAP technologies) for the organization and archiving of measurement results collected, to ease data retrieval and comparison.

The applicability of the methodology to industrial practices and V&V processes compliant to standards is shown by introducing a framework for the support of V&V process, and then discussing the interplay of the methodology and the framework to perform the experimental evaluation activities planned in a generic V&V process.

The methodology is then applied to five case studies, where five very different kinds of systems are evaluated, ranging from COTS components to highly distributed and adaptive SOAs. These systems are (in ascending order of distributedness and complexity) i) the middleware service for resilient timekeeping R&SAClock, ii) low-cost GPS devices, iii) a safety-critical embedded system for railway train-borne equipment (a Driver Machine Interface), iv) a distributed algorithm prototyped and tested with an improved version of NekoStat, and v) a testing service for the runtime evaluation of dynamic SOAs. Case studies i), iv) and v) have been developed exclusively in the academic context (in University labs), while case studies ii) and iii) have been performed in cooperation with industries, to bring evidence of the effectiveness of the methodology in industrial V&V processes. These five case studies offer a comprehensive and exhaustive illustration of the methodology and its insight. They show how the methodology allows tackling the previous issues in different contexts, and prove its flexibility and generality.

BIBLIOGRAPHY

[1] IFIP WG 10.4. Dependability benchmarking SIG (Special Interest Group on dependability benchmarking. (Cited on page 20.)

[2] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: Generic object-oriented fault injection tool. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)*, pages 83–88, Washington, DC, USA, 2001. IEEE Computer Society. (Cited on pages 16, 39, and 57.)

[3] ALARP - A railway automatic track warning system based on distributed personal mobile terminals - Project Contract FP7-SST-2010-234088, http://www.alarp.eu. (Cited on pages 123 and 129.)

[4] R. Almeida, N. Mendes, and H. Madeira. Sharing experimental and field data: the amber raw data repository experience. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems Workshops (ICDCSW '10)*, pages 313 – 320, June 2010. (Cited on pages 44 and 45.)

[5] AMBER - Assessing, Measuring and BEnchmarking Resilience - Project Contract FP7-IST-2008-216295. (Cited on page 42.)

[6] L. Angrisani and M. Vadursi. Cross-layer measurements for a comprehensive characterization of wireless networks in the presence of interference. *IEEE Trans. on Instrumentation and Measurement*, 56(4):1148 –1156, August 2007. (Cited on page 95.)

[7] O. Arafat, A. Bauer, M. Leucker, and C. Schallhart. Runtime verification revisited. Technical report, Institut für informatik - Technische Universität München, 2005. (Cited on page 15.)

[8] 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 Trans. Softw. Eng.*, 16(2):166–182, 1990. (Cited on pages iii, 27, 29, 30, and 40.)

[9] J. Arlat, K. Kanoun, Y. Crouzet, H. Madeira, M. Dal Cin, P. Gil, and D. Costa. DBench - description of the selected enabling technologies, June 2002. (Cited on page 17.)

[10] J. Arlat and R. Moraes. Collecting, analyzing and archiving results from fault injection experiments. In *Proceedings of the 5th Latin-American Symposium on Dependable Computing (LADC '11)*, pages 100–105, April 2011. (Cited on pages iii, 27, 29, and 42.)

[11] C. Aurrecoechea, A. T. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems*, 6:138–151, 1998. (Cited on page 12.)

[12] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, 2004. (Cited on pages iii, xix, 3, 4, 5, 9, 10, and 95.)

[13] X. Bai, D. Xu, and G. Dai. Dynamic reconfigurable testing of service-oriented architecture. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01 (COMPSAC '07)*, pages 368–378, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 99.)

[14] A. L. Baker, J. M. Bieman, N. Fenton, D. A. Gustafson, A. Melton, and R. Whitty. A philosophy for software measurement. *J. Syst. Softw.*, 12(3):277–281, 1990. (Cited on page 32.)

[15] M. Balakrishnan, K. Birman, and A. Phanishayee. PLATO: Predictive latency-aware total ordering. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, pages 175–188, Washington, DC, USA, 2006. IEEE Computer Society. (Cited on page 40.)

[16] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL processes. In *Service-Oriented Computing (ICSOC '05)*, volume 3826 of *Lecture Notes in Computer Science*, pages 269–282. Springer, 2005. (Cited on pages vi, 99, and 102.)

[17] J. Barnes and et al. Characterization of frequency stability. *IEEE Trans. on Instrumentation and Measurement*, IM-20(2):105–120, May 1971. (Cited on page 77.)

[18] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000. (Cited on page 41.)

[19] A. Bondavalli, F. Brancati, and A. Ceccarelli. Safe estimation of time uncertainty of local clocks. In *Proceedings of the International Symposium on Precision Clock Synchronization for Measurement, Control and Communication (ISPCS '09)*, pages 1 –6, October 2009. (Cited on pages 73, 74, and 76.)

[20] A. Bondavalli, F. Brancati, A. Ceccarelli, and L. Falai. An experimental framework for the analysis and validation of software clocks. In *Proceedings*

of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS '09), pages 69–81, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on pages 73 and 74.)

[21] A. Bondavalli, F. Brancati, A. Ceccarelli, L. Falai, and M. Vadursi. Resilient estimation of synchronization uncertainty through software clocks. *submitted to IEEE Trans. on Computers*, 2011. (Cited on pages 70, 75, 84, and 90.)

[22] A. Bondavalli, F. Brancati, A. Ceccarelli, and M. Vadursi. Experimental validation of a synchronization uncertainty-aware software clock. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS '10)*, pages 245–254, 2010. (Cited on pages 73 and 74.)

[23] A. Bondavalli, A. Ceccarelli, and L. Falai. Assuring resilient time synchronization. In *Proceedings of the 27th IEEE Symposium on Reliable Distributed Systems (SRDS '08)*, pages 3 –12, Washington, DC, USA, 2008. IEEE Computer Society. (Cited on pages v, 73, 74, and 75.)

[24] A. Bondavalli, A. Ceccarelli, L. Falai, and M. Vadursi. Foundations of measurement theory applied to the evaluation of dependability attributes. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, pages 522–533, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on pages iv, 32, 36, and 63.)

[25] A. Bondavalli, A. Ceccarelli, L. Falai, and M. Vadursi. Towards making nekostat a proper measurement tool for the validation of distributed systems. In *Proceedings of the 8th International Symposium on Autonomous Decentralized Systems (ISADS '07)*, March 2007. (Cited on page 91.)

[26] A. Bondavalli, A. Ceccarelli, L. Falai, and M. Vadursi. A new approach and a related tool for dependability measurements on distributed systems. *IEEE Trans. on Instrumentation and Measurement*, 59(4):820–831, 2010. (Cited on pages iv, 23, and 91.)

[27] A. Bondavalli, A. Ceccarelli, F. Gogaj, A. Seminatore, and M. Vadursi. Localization errors of low-cost GPS devices in railway worksite-like scenarios. In *Proceedings of the IEEE International Workshop on Measurements and Networking Proceedings (M&N '11)*, pages 6–11, October 2011. (Cited on pages 127 and 129.)

[28] A. Bondavalli, A. Ceccarelli, F. Gogaj, M. Vadursi, and A. Seminatore. Experimental assessment of low-cost gps-based localization in railway worksite-like scenarios. *submitted to Special Issue on IEEE Trans. on Instrumentation and Measurements*, 2012. (Cited on pages 127 and 129.)

[29] A. Bondavalli, A. Ceccarelli, J. Gronbaek, D. Iovino, L. Karna, S. Klapka, T.K. Madsen, M. Magyar, I. Majzik, and A. Salzo. Design and evaluation of a safe Driver Machine Interface. *IJPE*, 4(2):153–166, 2009. (Cited on pages 139 and 140.)

[30] A. Bondavalli and L. Falai. RODS: General framework for rigorous observation of distributed systems. In *DSN 2008 Workshop on Resilience Assessment and Dependability Benchmarking (RADB '08)*, June 2008. (Cited on pages xix, 27, and 28.)

[31] G. J. Brahnmath, R. R. Raje, A. M. Olson, M. Auguston, B. R. Bryant, and C. C. Burt. A quality of service catalog for software components. Technical report, Purde University, 2002. (Cited on page 55.)

[32] S. Brocklehurst and B. Littlewood. New ways to get accurate reliability measures. *IEEE Softw.*, 9:34–42, July 1992. (Cited on page 32.)

[33] M.F. Buckley and D.P. Siewiorek. Vax/vms event monitoring and analysis. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25), Digest of Papers*, pages 414–423, June 1995. (Cited on page 16.)

[34] J. Carreira, H. Madeira, and J. Silva. Xception: Software fault injection and monitoring in processor functional units. In *Pre-prints 5th Int. Working Conf. on Dependable Computing for Critical Applications (DCCA-5)*, pages 135–149, 1995. (Cited on pages 16, 39, 40, and 57.)

[35] J. Case, M. Fedor, M. Schoffstall, and J. Davin. *Simple Network Management Protocol (SNMP), RFC 1157*, 1990. (Cited on page 12.)

[36] A. Ceccarelli, A. Bondavalli, and D. Iovino. Trustworthy evaluation of a safe driver machine interface through software-implemented fault injection. In *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '09)*, pages 234–241, 2009. (Cited on page 139.)

[37] A. Ceccarelli, J. Gronbaek, L. Montecchi, H.-P. Schwefel, and A. Bondavalli. Towards a framework for self-adaptive reliable network services in highly-uncertain environments. In *Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW '10)*, pages 184 –193, May 2010. (Cited on page 98.)

[38] A. Ceccarelli, M. Vieira, and A. Bondavalli. A service discovery approach for testing dynamic SOAs. In *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW '11)*, pages 133–142, March 2011. (Cited on pages 100, 104, and 108.)

[39] A. Ceccarelli, M. Vieira, and A. Bondavalli. A testing service for lifelong validation of dynamic SOA. In *Proceedings of the IEEE International High Assurance Systems Engineering Symposium (HASE '11)*, pages 1–8, 2011. (Cited on page 100.)

[40] A. Ceccarelli, L. Vinerbi, L. Falai, and A. Bondavalli. RACME: A framework to support V&V and certification. In *Proceedings of the 5th Latin-American Symposium on Dependable Computing (LADC '11)*, pages 116 –125, April 2011. (Cited on pages vi, 69, 113, and 116.)

[41] CENELEC. *EN 50126 - Railway applications - the specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)*, 2000. (Cited on pages 113 and 127.)

[42] CENELEC. *EN 50128 - Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems*, 2001. (Cited on pages 58, 124, 127, and 141.)

[43] CENELEC. *EN 50129 - Railway applications - Communication, signalling and processing systems - Safety related electronic systems for signalling*, 2003. (Cited on pages 127 and 141.)

[44] R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders. Loki: A state-driven fault injector for distributed systems. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (DSN '00)*, pages 237–242, Washington, DC, USA, 2000. IEEE Computer Society. (Cited on pages 16, 38, and 39.)

[45] M. Dal Cin, K. Kanoun, K. Buchacker, L. L. Zuinga, R. Lindstrom, A. Johanson, H. Madeira, V. Sieh, and N. Suri. DBench - workload and faultload selection, June 2002. (Cited on pages 55 and 61.)

[46] M. Cinque, D. Cotroneo, C. Di Martino, S. Russo, and A. Testa. Avr-inject: A tool for injecting faults in wireless sensor nodes. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS '09)*, pages 1–8, May 2009. (Cited on pages 39, 40, and 41.)

[47] T. Cockram and B. Lockwood. Electronic safety case: Challenges and opportunities. Praxis Critical Systems Limited, 2003. (Cited on pages 115 and 116.)

[48] Bojan Cukic. Guest editor's introduction: The promise of public software engineering data repositories. *IEEE Softw.*, 22:20–22, November 2005. (Cited on pages iv, 19, and 44.)

[49] M. Cukier, R. Chandra, D. Henke, J. Pistole, and W. H. Sanders. Fault injection based on a partial view of the global state of a distributed system.

In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS '99)*, pages 168–177, Washington, DC, USA, 1999. IEEE Computer Society. (Cited on page 38.)

[50] DACS. Data and Analysis Center for Software, http://iac.dtic.mil/fact_sheets/dacs.pdf [online]. (Cited on page 44.)

[51] S. Dawson and F. Jahanian. Probing and fault injection of protocol implementations. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 351–359, 1995. (Cited on page 39.)

[52] S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proceedings of the Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, pages 404–414, Washington, DC, USA, 1996. IEEE Computer Society. (Cited on pages 39 and 57.)

[53] DBench - Dependability Benchmarking, IST-2000-25425 project, 2000. (Cited on page 19.)

[54] G. De Marco, M. Ikeda, T. Yang, and L. Barolli. Experimental performance evaluation of a pro-active ad-hoc routing protocol in out- and indoor scenarios. In *Proceedings of the 21st International Conference on Advanced Networking and Applications (AINA '07)*, pages 7–14, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 38.)

[55] C. Delerce-Mauris, L. Palacin, S. Martarello, S. Mosser, and M. Blay-Fornarino. Plateforme SEDUITE: une approche SOA de la diffusion d'informations. Technical report, Sophia Antipolis, I3S CNRS, University of Nice, 2009. (Cited on page 105.)

[56] W. M. P. Van der Aalst and A. H. M. ter Hofstede. YAWL: yet another workflow language. *Inf. Syst.*, 30:245–275, June 2005. (Cited on page 110.)

[57] DO-178B. *Software considerations in airborne systems and equipment certification*, 1992. (Cited on pages 58 and 113.)

[58] P. Donzelli, M. Zelkowitz, V. Basili, D. Allard, and K. N. Meyer. Evaluating COTS component dependability in context. *IEEE Softw.*, 22(4):46–53, 2005. (Cited on page 41.)

[59] Error, fault, and failure data collection and analysis, http://hissa.nist.gov/project/eff.html [online]. (Cited on page 44.)

[60] P. Waller et al. In-orbit performance assessment of Giove clocks. In *40th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 69–72, 2008. (Cited on page 73.)

[61] J.-C. Fabre, F. Salles, M. R. Moreno, and J. Arlat. Assessment of COTS microkernels by fault injection. In *Proceedings of the Conference on Dependable Computing for Critical Applications (DCCA '99)*, pages 25–44, Washington, DC, USA, 1999. IEEE Computer Society. (Cited on pages iii and 39.)

[62] L. Falai. *Observing, Monitoring and Evaluating Distributed Systems*. PhD thesis, University of Florence, 2008. (Cited on pages 27 and 55.)

[63] L. Falai, A. Bondavalli, and F. Di Giandomenico. Quantitative evaluation of distributed algorithms using the Neko framework: The NekoStat extension. In *Proceedings of the 2nd Latin-American Symposium on Dependable Computing (LADC '05)*, pages 35–51, 2005. (Cited on pages 40, 91, and 92.)

[64] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Trans. Softw. Eng.*, 20:199–206, March 1994. (Cited on page 32.)

[65] N. Fenton and S. L. Pfleeger. *Software metrics (2nd ed.): a rigorous and practical approach*. PWS Publishing Co., Boston, MA, USA, 1997. (Cited on page 32.)

[66] N. E. Fenton and M. Neil. Software metrics: roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, pages 357–370, New York, NY, USA, 2000. ACM. (Cited on pages 27 and 31.)

[67] C. Fetzer and F. Cristian. Fortress: A system to support fail-aware real-time applications. In *Proceedings of the IEEE National Aerospace and Electronics Conference (NAEC '97)*, pages 690–697. San Francisco, December 1997. (Cited on page 38.)

[68] Garmin 18 LVC, data sheet [online] http://www.garmin.com. (Cited on page 131.)

[69] GLOBALSAT ND 100s, data sheet [online] http://www.usglobalsat.com. (Cited on page 131.)

[70] M. Greiler, H.-G. Gross, and K. A. Nasr. Runtime integration and testing for highly dynamic service oriented ICT solutions – an industry challenges report. In *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC-PART '09)*, pages 51–55, Washington, DC, USA, 2009. IEEE Computer Society. (Cited on pages vi and 99.)

[71] J. R. Groff and P. N. Weinberg. *SQL: The Complete Reference*. McGraw-Hill Professional, 1999. (Cited on pages 67 and 84.)

[72] J. Gronbaek, H.P. Schwefel, A. Ceccarelli, and A. Bondavalli. Improving robustness of network fault diagnosis to uncertainty in observations. In *Proceedings of the 9th IEEE International Symposium on Network Computing and Applications (NCA '10)*, pages 229–232, July 2010. (Cited on page 98.)

[73] D. Gunter and B. Tierney. Netlogger: a toolkit for distributed system performance tuning and debugging. In *Proceedings of the IFIP/IEEE 8th International Symposium on Integrated Network Management*, pages 97–100, March 2003. (Cited on pages iii, 27, 28, 60, and 81.)

[74] M. J. Hadley. *Web Application Description Language (WADL)*. Sun Microsystems, Inc., Mountain View, CA, USA, 2006. (Cited on page 110.)

[75] R. Hatch, T. Sharpe, and P. Galyean Meeting. A global, high-accuracy, differential GPS system. In *Proceedings of the 2003 National Technical Meeting of the Institute of Navigation*, pages 562 – 573, 2003. (Cited on page 129.)

[76] M. Hiller, A. Jhumka, and N. Suri. An approach for analysing the propagation of data errors in software. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)*, pages 161–170, July 2001. (Cited on page 62.)

[77] Y. Hoffner. Monitoring in distributed systems. Technical report, ANSA Phase III, 1994. (Cited on pages 25 and 62.)

[78] B. Hofmann-Wellenhof, H. Lichtenegger, and E. Wasle. *GNSS - Global Navigation Satellite Systems - GPS, GLONASS, Galileo, and more*. Springer, 2008. (Cited on page 137.)

[79] M. Horauer. *Clock synchronization in distributed systems - Architecture and Evaluation of Ethernet-based Network Interfaces with support for precision clock synchronization*. PhD thesis, Vienna University of Technology, 2004. (Cited on pages 57 and 73.)

[80] D. W. Hosmer and S. Lemeshow. *Applied Logistic Regression*. Wiley Series in Probability and Statistics. Wiley & Sons, second edition, 2000. (Cited on page 30.)

[81] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997. (Cited on pages 15, 18, 55, 60, and 61.)

[82] IBM Rational Software, http://www.ibm.com [online]. (Cited on pages 114 and 115.)

[83] *ICDCS 2010 Workshops - Second Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems*. IEEE Computer Society, 2010. (Cited on page 42.)

[84] International Electrotechnical Commission (IEC). *IEC 61508 Functional safety of electrical / electronic / programmable electronic safety-related systems*, 2000. (Cited on pages 49 and 113.)

[85] IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990. (Cited on page 18.)

[86] IEEE. IEEE standard for a software quality metrics methodology. *IEEE Std 1061-1992*, 1993. (Cited on pages 27 and 30.)

[87] IEEE. IEEE trial-use standard for artificial intelligence exchange and service tie to all test environments (ai-estate): Data and knowledge specification. *IEEE Std 1232.1-1997*, 1997. (Cited on pages 27 and 31.)

[88] IEEE. IEEE standard for software test documentation. *IEEE Std 829-1998*, 1998. (Cited on pages 49 and 68.)

[89] IEEE. IEEE trial-use standard for artificial intelligence exchange and service tie to all test environments (ai-estate): Service specification. *IEEE Std 1232.2-1998*, 1998. (Cited on pages 27 and 31.)

[90] N. Ignat, B. Nicolescu, Y. Savaria, and G. Nicolescu. Soft-error classification and impact analysis on real-time operating systems. In *Proceedings of the conference on Design, automation and test in Europe (DATE '06)*, pages 182–187, Belgium, 2006. European Design and Automation Association. (Cited on page 41.)

[91] Y. Jiang, C.-K. Tham, and C.-C. Ko. Challenges and approaches in providing QoS monitoring. *Int. J. Netw. Manag.*, 10:323–334, November 2000. (Cited on pages 10 and 12.)

[92] A. Johansson, N. Suri, and B. Murphy. On the selection of error model(s) for OS robustness evaluation. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, pages 502–511, June 2007. (Cited on page 16.)

[93] A. M. Johnson and M. Malek. Survey of software tools for evaluating reliability, availability, and serviceability. *ACM Comput. Surv.*, 20:227–269, December 1988. (Cited on page 119.)

[94] Joint Committee for Guides in Metrology (JCGM). *Evaluation of measurement data - Supplement 1 to the Guide to the expression of uncertainty in measurement - Propagation of distributions using a Monte Carlo method*, 2004. (Cited on pages 20, 21, and 63.)

[95] Joint Committee for Guides in Metrology (JCGM). *Evaluation of measurement data - Guide to the expressionof uncertainty in measurement*, 2008. (Cited on pages 20, 21, 22, 38, 63, 84, and 147.)

[96] Joint Committee for Guides in Metrology (JCGM). *ISO international vocabulary of basic and general terms in metrology (VIM)*, third edition, 2008. (Cited on pages 20, 63, and 64.)

[97] Joint Committee for Guides in Metrology (JCGM). *Evaluation of measurement data - Supplement 2 to the Guide to the expression of uncertainty in measurement - Extension to any number of output quantities*, 2011. (Cited on pages 20, 21, and 63.)

[98] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5:121–150, March 1987. (Cited on page 25.)

[99] A. Kalakech, T. Jarboui, J. A., Y. Crouzet, and K. Kanoun. Benchmarking operating system dependability: Windows 2000 as a case study. In *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '04)*, pages 261–270, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on page 38.)

[100] N.A. Kanawati, G.A. Kanawati, and J.A. Abraham. Dependability evaluation using hybrid fault/error injection. In *Proceedings of the International Computer Performance and Dependability Symposium*, pages 224–233, April 1995. (Cited on page 16.)

[101] C. Kaner and W. P. Bond. Software engineering metrics: What do they measure and how do we know? In *Proceedings of the 10th IEEE International Software Metrics Symposium (METRICS '04)*, 2004. (Cited on pages 27 and 31.)

[102] K. Kanoun, Y. Crouzet, J. Arlat, and H. Madeira. DBench - measurements, June 2002. (Cited on page 55.)

[103] K. Kanoun, H. Madeira, and J. Arlat. A framework for dependability benchmarking. In *Supplement of the 2002 Int. Conf. on Dependable Systems and Networks (DSN '02)*, pages 12–15, 2002. (Cited on pages 19 and 20.)

[104] 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 Trans. on Software Engineering*, 19(11):1105–1118, November 1993. (Cited on page 16.)

[105] W.-L. Kao and R.K. Iyer. Define: a distributed fault injection and monitoring environment. In *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 252–259, June 1994. (Cited on page 16.)

[106] E. D. Kaplan and C. J. Hegarty (Eds.). *Understanding GPS: principles and applications*. Artech House, Boston, MA, second edition, 2006. (Cited on pages 19, 129, and 131.)

[107] T. P. Kelly. Can process-based and product-based approaches to software safety certification be reconciled? In Felix Redmill and Tom Anderson,

editors, *Improvements in System Safety*, pages 3–12. Springer London, 2008. (Cited on page 113.)

[108] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003. (Cited on page 14.)

[109] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin/Heidelberg, 1997. (Cited on page 102.)

[110] R. Kimball, M. Ross, and W. Thornthwaite. *The Data Warehouse Lifecycle Toolkit*. J. Wiley & Sons, Inc, 2008. (Cited on pages 42, 57, 65, and 67.)

[111] P. Koopman and J. DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Trans. Softw. Eng.*, 26:837–848, September 2000. (Cited on pages iii and 18.)

[112] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982. (Cited on page 6.)

[113] H. Landau, X. Chen, S. Klose, R. Leandro, and U. Vollath. Trimble's RTK and DGPS solutions in comparison with precise point positioning. In M. Sideris and F. Sansa, editors, *Observing our Changing Earth*, volume 133 of *International Association of Geodesy Symposia*, pages 709–718. Springer Berlin Heidelberg, 2008. (Cited on page 129.)

[114] F. Lange, R. Kroeger, and M. Gergeleit. Jewel: design and implementation of a distributed measurement system. *IEEE Trans. on Parallel and Distributed Systems*, 3(6):657–671, November 1992. (Cited on page 16.)

[115] N. Laranjeiro, S. Canelas, and M. Vieira. wsrbench: An on-line tool for robustness benchmarking. In *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 2*, pages 187–194, Washington, DC, USA, 2008. IEEE Computer Society. (Cited on pages 103, 105, 108, and 109.)

[116] A. Lester. *Project Planning and Control*. Elsevier, fourth edition, 2003. (Cited on page 52.)

[117] R. Lewis. Safety case development as an information modelling problem. In Chris Dale and Tom Anderson, editors, *Safety-Critical Systems: Problems, Process and Practice*, pages 183–193. Springer London, 2009. (Cited on pages 115 and 116.)

[118] C. Liao, M. Martonosi, and D. W. Clark. Experience with an adaptive globally-synchronizing clock algorithm. In *Proceedings of the 11th annual ACM symposium on Parallel algorithms and architectures (SPAA '99)*, pages 106–114, New York, NY, USA, 1999. ACM. (Cited on page 73.)

[119] J. Liberty. *Programming C#*. O'Reilly & Associates, Inc., second edition, 2002. (Cited on page 121.)

[120] P. Lollini and A. Bondavalli (Editors). HIDENETS - Deliverable 4.1: Evaluation methodologies, techniques and tools, 2006. (Cited on pages 10, 17, and 26.)

[121] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. (Cited on page 43.)

[122] H. Madeira, J. P. Costa, and M. Vieira. The OLAP and data warehousing approaches for analysis and sharing of results from dependability evaluation experiments. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN '03)*, pages 86–91, 2003. (Cited on pages iv, 42, 43, and 57.)

[123] H. Madeira, K. Kanoun, J. Arlat, D. Costa, Y. Crouzet, M. Dal Cin, P. Gil, N. Suri, and H. Madeira. Towards a framework for dependability benchmarking. In *Proceedings of the 4th European Dependable Computing Conference on Dependable Computing (EDCC-4)*, 2002. (Cited on page 20.)

[124] H. Madeira, K. Kanoun, J. Arlat, Y. Crouzet, A. Johansson, and R. Lindström. DBench - preliminary dependability benchmark framework, August 2001. (Cited on page 52.)

[125] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors - a survey. *IEEE Trans. Comput.*, 37:160–174, February 1988. (Cited on page 141.)

[126] M. Mansouri-Samani and M. Sloman. Monitoring distributed systems. *Network, IEEE*, 7(6):20–30, November 1993. (Cited on pages 11 and 60.)

[127] M. Mansouri-Samani and M. Sloman. *Monitoring distributed systems*, pages 303–347. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994. (Cited on page 25.)

[128] Microtec. *VRTXsa Real-Time Kernel - Programmer's Guide and Reference*, 1997. (Cited on page 141.)

[129] D.L. Mills. Internet time synchronization: the Network Time Protocol. *IEEE Trans. Communications 39, 10*, pages 1482–1493, 1991. (Cited on pages 63, 74, and 78.)

[130] D. C. Montgomery. *Design and analysis of experiments*. John Wiley & Sons, 2008. (Cited on page iii.)

[131] R. Moraes, J. Duraes, R. Barbosa, E. Martins, and H. Madeira. Experimental risk assessment and comparison using software fault injection. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, pages 512–521, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on pages iii, 27, 29, and 30.)

[132] Y. Morales and T. Tsubouchi. DGPS, RTK-GPS and StarFire DGPS performance under tree shading environments. In *Proceedings of the IEEE International Conference on Integration Technology (ICIT '07)*, pages 519–524, March 2007. (Cited on page 129.)

[133] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *Proceeding of the 17th international conference on World Wide Web (WWW '08)*, pages 815–824, New York, NY, USA, 2008. ACM. (Cited on page 102.)

[134] M. A. Munawar and P. A. S. Ward. Adaptive monitoring in enterprise software systems. In *SIGMETRICS 2006 Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, 2006. (Cited on page 14.)

[135] B. Murphy. Automating software failure reporting. *Queue*, 2:42–48, November 2004. (Cited on page 15.)

[136] National Marine Electronics Association version 4.0. *NMEA 0183 - The standard for interfacing marine electronics*, 2008. (Cited on page 132.)

[137] Modeling Navy and Simulation Office (NMSO). *Modeling and Simulation Verification, Validation, and Accreditation Implementation Handbook Volume I + VV&A Framework*, 2004. (Cited on page 114.)

[138] OASIS. *Web Services Business Process Execution Language (WSBPEL) Version 2.0*, 2007. (Cited on page 100.)

[139] Object Management Group (OMG). *OMG System Modeling Language (OMG SysML) v1.2*, 2010. (Cited on page 117.)

[140] Object Management Group (OMG). *Software Assurance Evidence Metamodel (SAEM) v1.0*, 2011. (Cited on page 115.)

[141] D. Odijk, J. Traugott, G. Sachs, O. Montenbruck, and C. Tiberius. Two precision GPS approaches applied to kinematic raw measurements of miniaturized L1 receivers. In *ION*, 2007. (Cited on pages 129 and 132.)

[142] K. F. O'Donoghue and T. R. Plunkett. Development and validation of network clock measurement techniques. In *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS '96)*, pages 65–68, Washington, DC, USA, 1996. IEEE Computer Society. (Cited on page 73.)

[143] Department of Defence (DoD). VV&A Documentation Tool (DVDT), http://dvdt.nmso.navy.mil/ [online]. (Cited on page 114.)

[144] Department of Defence (DoD). *Directive 8100.1 - Global Information Grid (GIG) Overarching Policy*, 2002. (Cited on page 114.)

[145] D. L. Palumbo. The derivation and experimental verification of clock synchronization theory. *IEEE Trans. Comput.*, 43:676–686, June 1994. (Cited on page 73.)

[146] R. K. Panesar-Walawege, M. Sabetzadeh, L. Briand, and T. Coq. Characterizing the chain of evidence for software safety cases: A conceptual model based on the IEC 61508 standard. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST '10)*, pages 335–344, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on pages 113 and 115.)

[147] R. K. Panesar-Walawege, M. Sabetzadeh, L. Briand, and T. Coq. A model-driven engineering approach to support the verification of compliance to safety standards. In *Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering (ISSRE '11)*, 2011. (Cited on page 115.)

[148] M. Papazoglou and W.-J. van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16:389–415, 2007. (Cited on pages 99 and 110.)

[149] A. Pecchia, D. Cotroneo, Z. Kalbarczyk, and R.K. Iyer. Improving log-based field failure data analysis of multi-node computing systems. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems Networks (DSN '11)*, pages 97–108, June 2011. (Cited on page 40.)

[150] A. Pnueli. The temporal logic of programs. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:46–57, 1977. (Cited on page 14.)

[151] R. Prasad, C. Dovrolis, M. Murray, and K. Claffy. Bandwidth estimation: metrics, measurement techniques, and tools. *Network, IEEE*, 17(6):27–35, 2003. (Cited on page 95.)

[152] M. Z. Rela, H. Madeira, and J. G. Silva. Experimental evaluation of the fail-silent behaviour in programs with consistency checks. In *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing (FTCS*

'96), pages 394–403, Washington, DC, USA, 1996. IEEE Computer Society. (Cited on pages 109 and 141.)

[153] A. Robbins. *Effective AWK Programming: Text Processing and Pattern Matching*. O'Reilly, 2001. (Cited on page 84.)

[154] M. Rodriguez, A. Albinet, and J. Arlat. MAFALDA-RT: A tool for dependability assessment of real-time systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN '02)*, pages 267–272, Washington, DC, USA, 2002. IEEE Computer Society. (Cited on pages 16 and 39.)

[155] SAFEDMI - Safe Driver Machine Interface for ERTMS automatic train control - Project Contract FP6-IST-2006-031413. (Cited on pages 127 and 139.)

[156] SAFEDMI. Deliverable D4.1 - quantitative evaluation methodology, 2008. (Cited on pages 141 and 146.)

[157] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 8(4):10–17, August 2001. (Cited on page iii.)

[158] K. Scarfone and P. Mell. *Guide to Intrusion Detection and Prevention Systems (IDPS) - Recommendations of the National Institute of Standards and Technology*. NIST (National Institute of Standards and Technology), February 2007. (Cited on page 13.)

[159] B. Schroeder and G. Gibson. The computer failure data repository (CFDR): collecting, sharing and analyzing failure data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*, New York, NY, USA, 2006. ACM. (Cited on page 44.)

[160] W. Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Syst.*, 7:129–157, September 1994. (Cited on page 60.)

[161] A. Seminatore, L. Ghelardoni, A. Ceccarelli, L. Falai, M. Schultheis, and B. Malinowsky. Alarp (a railway automatic track warning system based on distributed personal mobile terminals). In *submitted to Transport Research Arena (TRA)*, 2012. (Cited on page 130.)

[162] Y. Shafranovich. *RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files*, 2005. (Cited on page 67.)

[163] J. Sheppard and M. Kaufman. IEEE 1232 and P1522 standards. In *AUTOTESTCON Proceedings, 2000 IEEE*, pages 388 – 397, 2000. (Cited on pages 27, 31, and 32.)

[164] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems: design and evaluation*. Digital Press, Newton, MA, USA, second edition, 1992. (Cited on pages 10, 33, 52, and 141.)

[165] D. Skarin, R. Barbosa, and J. Karlsson. Comparing and validating measurements of dependability attributes. In *Proceedings of the 8th European Dependable Computing Conference (EDCC-8)*, pages 3–12, Washington, DC, USA, 2010. IEEE Computer Society. (Cited on pages 38, 40, 41, and 63.)

[166] *SRDS 2008 Workshops - First Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems, Napoli, Italy*, 2008. (Cited on page 42.)

[167] D. Srivastava, L. Golab, R. Greer, T. Johnson, J. Seidel, V. Shkapenyuk, O. Spatscheck, and J. Yates. Enabling real time data analysis. *Proc. VLDB Endow.*, 3:1–2, September 2010. (Cited on page 62.)

[168] Streambase, http://www.streambase.com/ [online]. (Cited on page 43.)

[169] Stress tool, http://weather.ou.edu/ apw/projects/stress/ [online]. (Cited on page 89.)

[170] L. Strigini, N. Neves, M. Raynal, M. Harrison, M. Kaaniche, and F. von Henke. RESIST - Deliverable 1.2: Resilience-building technologies: State of knowledge, September 2006. (Cited on pages 17 and 52.)

[171] P. Stuckmann and R. Zimmermann. European research on future internet design. *Wireless Communications, IEEE*, 16(5):14 –22, October 2009. (Cited on page iii.)

[172] Inc Sun Microsystems. *Sun GlassFish Enterprise Server v2.1.1 Reference Manual*, 2009. (Cited on page 105.)

[173] Ian Thomas and Brian A. Nejmeh. Definitions of tool integration for environments. *IEEE Softw.*, 9:29–35, March 1992. (Cited on page 114.)

[174] Trimble R7 GNSS Base Station, data sheet [online] http://www.trimble.com/. (Cited on page 132.)

[175] Trimble R8 Rover, data sheet [online] http://www.trimble.com. (Cited on page 132.)

[176] K. S. Trivedi, B. R. Haverkort, A. Rindos, and V. Mainkar. Techniques and tools for reliability and performance evaluation: problems and perspectives. In G. Haring and G. Kotsis, editors, *Proceedings of the 7th international conference on Computer performance evaluation: modelling techniques and tools*, volume 794 of *Lecture Notes in Computer Science*, pages 1–24, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc. (Cited on page 119.)

[177] T. K. Tsai and R. K. Iyer. Measuring fault tolerance with the ftape fault injection tool. In *Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (MMB '95)*, pages 26–40, London, UK, 1995. Springer-Verlag. (Cited on pages 38, 39, and 40.)

[178] T. K. Tsai, R. K. Iyer, and D. Jewitt. An approach towards benchmarking of fault-tolerant commercial systems. In *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, pages 314–323, Washington, DC, USA, 1996. IEEE Computer Society. (Cited on page 40.)

[179] P. Urban, A. Schiper, and X. Defago. Neko: A single environment to simulate and prototype distributed algorithms. In *Proceedings of the 15th International Conference on Information Networking (ICOIN '01)*, pages 503–511, Washington, DC, USA, 2001. IEEE Computer Society. (Cited on pages xix, 40, 91, and 92.)

[180] A. van Moorsel et al. AMBER - Deliverable 2.2: State of the art, 2009. (Cited on pages 11, 12, 14, 17, 52, and 62.)

[181] D. Veitch, S. Babu, and A. Pàsztor. Robust synchronization of software clocks across the internet. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement (IMC '04)*, pages 219–232, New York, NY, USA, 2004. ACM. (Cited on page 73.)

[182] P. Veríssimo and L. Rodriguez. *Distributed Systems for System Architects*. Kluwer Academic, 2001. (Cited on pages 33 and 74.)

[183] M. Vieira, N. Laranjeiro, and H. Madeira. Assessing robustness of web-services infrastructures. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, pages 131–136, June 2007. (Cited on pages 16 and 18.)

[184] M. Vieira, N. Mendes, J. Duraes, and H. Madeira. The AMBER data repository. In *DSN Workshop RADB 2008*, June 2008. (Cited on page 19.)

[185] A. I. Wasserman. Tool integration in software engineering environments. In F. Long, editor, *Software engineering environments*, Lecture Notes in Computer Science, pages 137–149, New York, NY, USA, 1990. Springer Berlin / Heidelberg. (Cited on page 114.)

[186] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *Software, IET*, 1(5):172–179, October 2007. (Cited on pages 11 and 60.)

[187] *Web Network Management Protocol (WNMP), http://barracudaserver.com/ba/doc/en/C/wnmp.html [online]*, 2011. (Cited on page 14.)

[188] R. Wolfinger, D. Dhungana, H. Prähofer, and H. Mössenböck. A component plug-in architecture for the .NET platform. In D. Lightfoot and C. Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 287–305. Springer Berlin/Heidelberg, 2006. (Cited on page 119.)

[189] P. Yuste, D. de Andres, L. Lemus, J. J. Serrano, and P. Gil. Inerte: integrated NExus-based real-time fault injection tool for embedded systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN '03)*, page 669, June 2003. (Cited on page 16.)

[190] Z. Zheng and M.R. Lyu. Ws-dream: A distributed reliability assessment mechanism for web services. In *Proceedings of the 2008 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '08)*, pages 392–397, June 2008. (Cited on page 16.)