A. D. MCCXXIV

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunità Europea
Fondo Sociale Europeo

UNDERSTANDING THE ERROR BEHAVIOR OF COMPLEX CRITICAL SOFTWARE SYSTEMS THROUGH FIELD DATA

RAFFAELE DELLA CORTE

Tesi di Dottorato di Ricerca

(XXVIII Ciclo)

Marzo 2016

Il Tutore

Prof. Marcello Cinque

Il Coordinatore del Dottorato

Prof. Francesco Garofalo

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

via Claudio, 21- I-80125 Napoli - ☎ *[#39]* (0)81 768 3813 - *[#39]* (0)81 768 3816

UNDERSTANDING THE ERROR BEHAVIOR OF COMPLEX CRITICAL SOFTWARE SYSTEMS THROUGH FIELD DATA

By
Raffaele Della Corte

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
"FEDERICO II" UNIVERSITY OF NAPLES
VIA CLAUDIO 21, 80125 – NAPOLI, ITALY
MARCH 2016

“Alla mia principessa ed a tutta la mia famiglia”

Table of Contents

List of Tables

List of Figures

Acknowledgements

Giunto a questo nuovo traguardo della mia carriera universitaria, vorrei ringraziare tutti coloro che mi hanno supportato durante questi tre anni.

Innanzitutto vorrei ringraziare il mio tutor, il prof. Marcello Cinque, per avermi guidato in questo percorso, trasmettendomi sempre quella fiducia e quella calma che lo hanno sempre contraddistinto, e per tutto il supporto fornito in questi anni, soprattutto nei momenti di maggiore difficoltà, che mi ha permesso di giungere fino alla stesura di questa tesi. Vorrei poi ringraziare il prof. Stefano Russo ed il prof. Domenico Cotroneo, per avermi aperto le porte del Mobilab, e per essere stati sempre disponibili quando necessario. Come non ringraziare poi Antonio, instancabile tutor sul campo, nonché fonte di preziosi consigli, che mi ha costantemente e pazientemente seguito durante questi tre anni, spronandomi sempre al miglioramento. Un ringraziamento va anche al prof. Saurabh Bagchi per tutto il supporto durante il mio periodo di formazione presso la Purdue University.

Vorrei poi ringraziare Emilia, la mia principessa, che in questi anni insieme non ha mai smesso di supportarmi e sostenermi, anche quando a separarci fisicamente c'erano migliaia di miglia e sei ore di fuso orario!!! Grazie per essermi sempre accanto, per aver messo da parte le tue paure raggiungendomi per ben due volte dall'altra parte del mondo, per tutti gli indimenticabili momenti trascorsi e che trascorreremo insieme...ma, soprattutto, grazie per qull'immenso Amore che riempie i miei giorni da quasi sette anni e che mi da la forza di superare ogni difficoltà. Sei quanto di più bello la vita mi abbia donato, la mia perfetta metà...TI AMO!!!

Un enorme grazie va poi ai miei genitori senza i quali tutto questo non sarebbe mai stato possibile...grazie per tutti i sacrifici fatti, per aver sempre creduto in me, dandomi sempre la possibilità di fare ci in cui credevo, ma soprattutto grazie per il vostro immancabile affetto, che si è fatto sempre sentire, soprattutto nei miei momenti no e durante le nottate trascorse al computer. Un grazie anche a mia sorella ed a Marco per tutto il loro sostegno in questi anni, per aver tollerato le mie indisponibilità, e per avermi dato l'onore di fargli da testimone! Grazie anche a mia nonna Rosa, che ad ogni visita è in grado sempre di portare allegria lei e il suo "vecchio", ed a mio nonno Vincenzo, che sicuramente sarebbe contento di

vedermi raggiungere anche questo traguardo. Un ringraziamento va anche ai miei suoceri, ed a Enzo, che hanno sempre fatto di tutto per evitare qualsiasi disturbo quando il lavoro chiamava mentre ero da loro, e per tutto il supporto datomi in questi anni, ma soprattutto per l'enorme affetto che mi hanno sempre dimostrato.

Un caloroso grazie va anche ai ragazzi del Mobilab/CINI, Anna, Alma, Ugo, Luca, Fabio, Sal, Francesco, Flavio, Gigi, Ken, Stefano, Roby N., Roby P., Mario, Domenico, Emanuela, Dario, Susy e Stefania, per avermi accolto fin da subito, per le lunghe giornate di lavoro trascorse insieme, ma in particolare per essere degli incredibili compagni di viaggio...vi auguro di cuore tutto il meglio. Non può mancare anche un grazie a Luigi, Fulvia, Salvatore, Mario, Teresa, i piccoli Ciro e Luigi, Davide, Miriam e Giovanni per esserci sempre e comunque anche se non ci vediamo spesso, ma anche a Raffaella, Ciro, Rossella, Maria, Manuela, Emanuele, Sara, Fabio, Chiara ed Antonio per i momenti trascorsi insieme. Grazie anche a Daniele, Jeff e Chris per la loro grande disponibilità durante il mio periodo a Purdue.

Infine, un grazie va anche a colui che da lassù ha sempre fatto si che le cose andassero per il verso giusto, facendo avvertire la sua mano sempre, in ogni momento.

A voi tutti, ed anche a coloro che avrò sicuramente dimenticato di citare,...

GRAZIE DI VERO CUORE!!

Napoli, Italy *Raffaele*
31 Marzo 2016

Introduction

Software systems are the basis for human everyday activities, which are increasingly dependent on software. Software is an integral part of systems we interact with in our daily life raging form small systems for entertainment and domotics, to large systems and infrastructures that provide fundamental services such as telecommunication, transportation, and financial. In particular, software systems play a key role in the context of **critical domains**, supporting crucial activities. For example, ground and air transportation, power supply, nuclear plants, and medical applications strongly rely on software systems: failures affecting these systems can lead to severe consequences, which can be catastrophic in terms of business or, even worse, human losses. Therefore, given the growing dependence on software systems in life- and critical-applications, dependability, i.e., *"the ability of the system to avoid service failures that are more frequent and more severe than is acceptable"* [1], has become among one of the most relevant industry and research concerns in the last decades.

Software faults have been recognized as one of the major cause for system failures [1, 2, 3] since the hardware failure rate has been decreasing over the years [4]. Time and cost constraints, along with technical limitations, often do not allow to fully validate the correctness of the software solely by means of testing [5, 6]; therefore, software might be released with residual faults that activate during operations. According to [1], the activation of a fault generates errors which propagate through the components of the system, possibly leading to a failure. Therefore, in order to produce reliable software, it is important to **understand how errors affect a software system**. For example, analyzing types of

errors that affect the software, the effect these errors may have on it as well as how they propagate through its components, allows both (i) the design of dependability structures and mechanisms, such as *Error Detection Mechanisms* (EDMs) and *Error Recovery Mechanisms* (ERMs), and (ii) their optimal placement in the source code, allowing the improvement of the system reliability.

This is of paramount importance especially in the context of **complex critical software systems**, where the occurrence of a failure can lead to severe consequences. However, the analysis of the error behavior of this kind of system is not trivial. They are often distributed systems based on many interacting heterogeneous components and layers, including Off-The-Shelf (OTS), third party components and legacy systems. In addition, they are expected to satisfy the rules imposed by certification standards, such as the DO-178B [7] in the avionics domain. Often, they include legacy components and/or obsolete kernel versions, which limit the use of cutting-edge technologies and the level of intervention, e.g., in terms of code instrumentation for error analysis. All these aspects, undermine the understanding of the error behavior of complex critical software system.

A well established methodology to evaluate the dependability of operational systems and to identify their dependability bottlenecks is represented by *field failure data analysis* (FFDA), which is based on the monitoring and recording of errors and failures occurred during the operational phase of the system under real workload conditions, i.e., **field data**. Indeed, direct measurement and analysis of natural failures occurring under real workload conditions is among the most accurate ways to assess dependability characteristics [8]. One of the main sources of field data, are monitoring techniques, such as event logging, assertion checking, source code instrumentation. Beside being recommended by several international safety standards and governmental guidelines, e.g., IEC 61508-7 [9], the AUTomotive Open System Architecture (AUTOSAR) through the ISO-26262 [10], and the DoD Guide for achieving Reliability, Availability, and Maintainability (RAM) [11], they are a consolidated

and pervasive practice both within the open-source community and proprietary software systems industry.

The contribution of the thesis is to provide a methodology that allows understanding the error behavior of complex critical software systems by means of field data generated by the monitoring techniques already implemented in the target system. The use of available monitoring techniques allows to overcome the limitations imposed in the context of critical systems, avoiding severe changes in the system, and preserving its functionality and performance.

The methodology is based on fault injection experiments that stimulate the target system with different error conditions. Injection experiments allow to accelerate the collection of error data naturally generated by the monitoring techniques already implemented in the system. The collected data are analyzed in order to characterize the behavior of the system under the occurred software errors. In particular, the dissertation aims to provide answers to the following compelling research questions (RQs):

- **RQ1**: *Is it possible to use monitoring techniques to characterize the error behavior in complex critical software system?* Field data generated by means of monitoring technique contain valuable information about the behavior of the target system at runtime. However, it is not trivial to analyze them in order to obtain valuable information about the error behavior of the system during failing executions, and especially about the propagation of errors. Therefore, there is the need to understand if data provided by monitoring techniques can be leveraged to analyze the error behavior of a software system.

- **RQ2**: *Is it possible to improve the error detection/recovery of a complex critical software system from error data?* The knowledge of the error behavior of the target system and of how the errors propagate through its components are often used in the

literature to identify the locations for EDM and ERM, which allow an improvement of the detection and recovery of errors in the target system. However, there is no prior experience on the use of field data to infer the locations where the placement of EDMs and ERMs might be beneficial for the system, and the type of error/failure they have to cope with.

- **RQ3**: *How do the error and failure reporting ability change between different monitoring techniques implemented in a given system? And what about the dissimilarity of their data?* A number of monitoring techniques can be implemented in a complex critical software system, which can be of different types and consider different error models; therefore, it can be useful to compare the performance exhibited by each technique in order to provide insights to developers to implement better monitoring techniques. There are existing studies that try to address this topic; however, they do not characterize the effectiveness of a monitoring technique with respect to failures and errors.

- **RQ4**: *Is it useful to combine different monitoring techniques implemented in a complex critical software system?* Different monitoring techniques implemented in a software system might expose orthogonal performance, complementing each other in terms of failure reporting and/or error propagation reporting ability. However, the orthogonality of monitoring techniques and the potentiality of their combination in a software system is still unexplored.

In order to provide answers to the before-mentioned research questions, the proposed methodology leverages a set of innovative means defined in this dissertation, i.e., (i) **Error Propagation graphs**, which allow to analyze the error propagation phenomena occurred in the target system and that can be inferred by the collected field data, and a set of metrics composed by (ii) **Error Determination Degree**, which allows gaining insights into the

ability of error notifications of a monitoring technique to suggest either the fault that led to the error, or the failure the error led to in the system, (iii) **Error Propagation Reportability**, which allow understanding the ability of a monitoring technique at reporting the propagation of errors, and (iv) **Data Dissimilarity**, which allows gaining insights into the suitability of the data generated by the monitoring techniques for failure analysis.

The methodology has been experimented on two instances of complex critical software systems in the field of Air Traffic Control (ATC), i.e., a *communication middleware* supporting data exchanging among ATC applications, and an *arrival manager* that is responsible for managing flight arrivals to a given airspace, within an industry-academia collaboration in the context of a national research project[1].

Results show that field data generated by means of monitoring techniques already implemented in a complex critical software system can be leveraged to obtain insights about the error behavior exhibited by the target system, as well as about the potential beneficial locations for EDMs and ERMs. In addition, the proposed methodology also allowed to characterize the effectiveness of the monitoring techniques in terms of failure reporting, error propagation reportability, and data dissimilarity.

The dissertation is organized as follows. *Chapter 1* introduces to the context of complex critical software system and provides basic notions of dependability. Moreover, an overview on the software error analysis is presented, along with its challenges in the context of complex critical software systems.

Chapter 2 describes the field failure data analysis methodology and examines the related literature. A discussion about open issues and challenges about the use of FFDA for analyze the error behavior of complex critical system is also provided.

[1]The systems considered in this dissertation are developed by Finmeccanica, a top leading company in electronic and information solutions for critical systems (www.finmeccanica.com). The evaluation versions of the systems are used as case study to support research activities conducted in the framework of the MINIMINDS PON Project (n. B21C12000710005), funded by the Italian Ministry of Education and Research, and led by the Federico II University of Naples, CINI and Finmeccanica

Chapter 3 describes the proposed methodology, providing details on all its characteristics, such as faultload, workload, experimental procedures and evaluation metrics. Also the description of the proposed metrics is provided.

Chapter 4 provides the description of the target systems and of the target monitoring techniques that are considered in this dissertation, and on which the proposed methodology has been applied. Also details about the conducted experimental campaign, i.e., the workloads, faultloads, labeling and error clustering processes, are provided. Finally, the obtained datasets are detailed at the end of the chapter.

Chapter 5 discuss the results obtained by applying the proposed methodology to the considered monitoring techniques. Their effectiveness has been evaluated by measuring the evaluation metrics of the methodology.

Chapter 6 provides a comparison of the considered monitoring techniques, which has been conducted by comparing the measures obtained from the evaluation metrics of the proposed methodology. Also their combination has been analyzed in order to evaluate the potential benefits that can be achieved by considering multiple techniques at the same time.

Chapter 1

Error Characterization of Complex Critical Software Systems

Characterizing error behavior of a software system is crucial to engineers. Characterization encompasses, for example, errors classification, analysis of error propagation, identification of error-prone components. The knowledge of which types of software errors affect the system, the effect these errors may have on it as well as how they propagate through its components allows both the design of efficient dependability structures and mechanisms, and their placement where they are the most effective, improving the dependability of future system releases. This knowledge is extremely valuable especially in the context of complex critical software system, where the occurrence of a failure has a high cost since it can lead to severe consequences, such as loss of life, damage to the environment or extensive economic losses. This chapter introduces to complex critical software systems, and provides basic notions of dependability that will be used in the rest of the dissertation. Then an introduction to the software error analysis is presented, along with its challenges in the context of complex critical software systems.

1.1 Complex Critical Software Systems

Critical systems are a special class of systems providing functionalities whose malfunction, i.e., *failure*, could result in damage to the equipment, reputational losses or, even worse, in human injury, loss of life, damage to the environment or extensive economic losses [12]. Examples of critical systems include embedded medical systems, flight control systems, automobile control systems, and online money transfer systems. For these systems a high level of *dependability* is essential in order to reduce the risk of failure and the losses that may

result from such a failure. Based on the consequences of a system failure, critical systems can be divided in three different categories:

- *Safety-critical systems*: A system whose failure may result in loss of life or serious environmental damage, such as a control system for a chemical manufacturing plant.

- *Mission-critical systems*: A system whose failure may result in the failure of some goal-directed activity, such as a navigational system for a spacecraft.

- *Business-critical systems*: A system whose failure may result in very high costs for the business using that system, such as the customer accounting system in a bank.

The complexity of critical systems has increased during the years. Many modern critical systems have been built with such complexity that they cannot be based on hardware alone. For example, advanced, aerodynamically unstable, military aircraft require continual software-controlled adjustment of their flight surfaces to ensure that they do not crash. Software is essential in order to cope with this growing complexity, making it possible to manage large numbers of devices, complex control laws and functionality. In addition, it can be cheaper than hardware solutions.

Many critical systems are nowadays based on large and complex software, such as smart grids, air traffic control (ATC) systems. These systems, named **Complex Critical Software Systems**, are in general the result of the integration of many strongly interacting

heterogeneous components and layers, including Off-The-Shelf (OTS) and third party components, such as operating system, communication middleware, database, network protocols, virtual machines, as well as legacy systems[1]. In addition, they are typically distributed systems, composed by several software-intensive systems deployed on many remote nodes communicating on a network.

As critical software systems grow in complexity, interconnectedness, and distribution, the possibilities to incur in a system failure increase. Complex critical software systems have a long lifetime, during which they evolve since they are integrated with other systems, and/or extended to cope with new requirements. The evolution causes a further growth of their complexity, and it often forces the integrated systems to operate beyond the original design conditions. The usage of many heterogeneous components causes complex interdependencies, and introduces sources of non-determinism, that often lead to the activation of subtle *faults*. Such behaviors, due to their complex triggering patterns, typically escape the testing phase. The activation of these faults and the propagation of errors among components can result in failures and system downtime with huge costs [14].

There are many examples of critical software systems which have failed due to software related faults. For instance, the Ariane V launch failure [15], which was due to a fault with software successfully used on earlier version of the launcher, and the loss of the Mars Climate Orbiter [16], which was due to a mismatch between Imperial and SI units. In addition,

[1]Legacy systems are software systems that have been developed in the past, often using older or obsolete technology. The maintenance actions of these systems (e.g., modifications to the source code) are prohibitively costly because (i) the component is written in a programming language which has become obsolete as compared to the rest of the technologies used by the enterprise and/or (ii) the component is not well-documented [13]. They are maintained because it is too risky to replace them.

some studies of anomalies of NASA space missions Voyager and Galileo has revealed that anomalies of the most recent mission are mostly due to software and are fixed by changing in-flight or ground-software systems [17, 18].

The high costs of failure of critical systems imply that they have to be developed so that failures are very rare, but they have also include effective monitoring and recovery mechanisms that are of paramount importance if and when failures occur. In many complex critical systems a problematic issue is that one single source problem often leads to many software errors that are often unrelated to the actual problem. In such a situation, it is of prime importance that the monitoring software gives the user a fast hint about what is really going wrong. Therefore, understand the error behavior of complex critical software systems and the effectiveness of the monitoring techniques they implement are vital issues in these type of systems.

1.2 Basic Dependability Concepts

Dependability has been considered a fundamental attribute since early computer systems. First studies in the context of dependable computing dates back to the 1960s, e.g., [19]. However, the effort on the definition of the basic concepts and terminology for computer systems dependability dates back to 1980, when a joint committee on "Fundamental Concepts and Terminology" was formed by the Technical Committee on Fault-Tolerant Computing of the IEEE Computer Society and the IFIP Working Group 10.4 "Dependable Computing and Fault Tolerance". A synthesis of this work, which represents a milestone in the area

of dependability, was presented at FTCS-15 in 1985 [20], where computer system dependability was defined as *the quality of the delivered service such that reliance can justifiably be placed on this service.* This notion has evolved over the years. A later work [1] defines dependability as the *the ability of the system to avoid service failures that are more frequent and more severe than is acceptable.* The dependability is a composed quality attribute, that encompasses the following subattributes:

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

1.2.1 Threats: Fault, Error, Failure

The causes that lead a system to deliver a service deviating from its function, i.e., an incorrect service, are manifold and can manifest at any phase of its life-cycle. Hardware faults and design errors are an example of the possible sources of failure. These causes are categorized as **failures**, **errors**, and **faults**, and are recognized in the literature as dependability threats [1].

A **failure** is an event that occurs when the delivered service deviates from correct service. for example, a service might fail either because it does not comply with the functional specification, or because this specification did not adequately describe the system function.

A service failure is a transition from correct service to incorrect service. The period of delivery of incorrect service is a service outage, while the transition from incorrect service to correct service is a service recovery or repair. The deviation from correct service may occur in different ways that are called *failure modes*.

An **error** can be defined as the part of the system state that may lead to a its subsequent failure. Precisely, a failure occurs when the error causes the delivered service to deviate from correct service. A **fault** is the adjudged or hypothesized cause of an error. Faults can be either *internal* or *external* of a system.

Failures, errors, and faults are related each other in the form of a chain of threats [1], i.e., the so-called *fault-error-failure* chain. A fault is *active* when it produces an error; otherwise, it is *dormant*. An active fault can be either i) an internal dormant fault that has been activated, or ii) an external fault. A failure occurs when an error is *propagated* to the service interface, leading the service delivered by the system to deviate from correct service. An error that does not lead to a failure is said to be a *latent* error. For example, programming bugs, i.e., faulty instructions in a program (e.g., common programming mistakes, such as missing variable initializations, or poorly-written logical clauses), are dormant fault in the software; they are activated when an appropriate input pattern is fed to the component where the faulty instruction resides, and an error is generated. The error might propagate within the system and affect the delivered service, i.e., a failure has occurred. A failure of a system component causes an internal fault of the system that contains such a component, or causes an external fault for the other system(s) that receive service from the given system.

1.2.2 Means

The need to attain the various attributes of dependability during system operations has lead to the design of a variety of dependability means over the last decades. These means can be grouped into four major categories [1]:

- **Fault prevention** aims to prevent the occurrence or introduction of faults. Fault prevention is enforced during the design phase of a system, and applies both for *software*, e.g., information hiding, modularization, use of strongly-typed programming languages, and *hardware*, e.g., by means of precise design rules.

- **Fault tolerance** aims to avoid service failures in the presence of faults. It takes place during the operational life of the system. Fault tolerance is commonly achieved by means of redundancy, either temporal or spatial. *Temporal redundancy* attempts to re-execute the operation which caused the failure after the system has been restored in an error-free state, while *spatial redundancy* exploits the computation performed by multiple systems replicas. Spatial redundancy relies on the assumption that replicas are not affected by the same faults: this is achieved via design *diversity* [21]. Moreover, both temporal and spatial redundancy adopt error detection and recovery approaches, i.e., once the error is detected, a recovery action is initiated.

- **Fault removal** aims to reduce the number and severity of faults. The removal activity is usually performed during the verification and validation phases of the system development, by means of testing and/or fault injection [22]. During the operational phase, fault removal encompasses corrective and perfective maintenance.

- **Fault forecasting** aims to estimate the current number, the future incidence, and the consequences of faults. Fault forecasting is conducted by performing an evaluation of the system behavior in face of activated faults. Evaluation can be either *qualitative*, which aims at identifying and classifying the failure modes, and *quantitative*, which aims to evaluate in terms of probabilities the extent system attributes are satisfied in terms of probabilities.

1.3 Understanding the Behavior of Software under Error

Software errors represent a major dependability threat for any software systems. As discussed in Section 1.2.1, the activation of a fault lead to a software error, which can propagate through the components of the system, leading to a system failure. Therefore, in order to produce reliable software, it is important to understand how faults and errors may affect a software system. The knowledge of which types of error affect the software, the effect these errors may have on it as well as how they propagate through its components allows both (i) the design of efficient dependability structures and mechanisms, and (ii) their placement where they are the most effective.

The study of the behavior of errors in software systems may be used to find the components which are most exposed to errors and to understand how different components affect each other in the presence of different type of errors. In addition, the analysis of software errors occurred in a system allows also to know where and what type of errors are likely to do the most damage. This information represents a valuable knowledge when deciding where to place an error detection mechanism (EDM), such as assertion, logging instruction,

or an error recovery mechanism (*ERM*), such as wrapper, redundant piece of code. Indeed, two factors that might impact the effectiveness of both EDMs and ERMs are (i) the type of error they have to cope with and (ii) their location.

Several studies have addressed the analysis of the software errors in order to characterize the error behavior of a software systems and/or to identify locations for EDMs and ERMs. For example, in [23] an approach that allows early identification of effective detector locations in dependable software design is presented. The approach leverages *module coupling* to identify potential error detector locations at module-levels for data-value errors. Detailed information are required for each module composing the target system in order to evaluate its coupling, such as input and output data/control parameters, global variables used as data/control, number of modules called/calling. In addition, the approach works for fault-intolerant software, which has to be subsequently enhanced with detectors at specific locations. An open-source flight simulator has been used to validate the approach.

Authors in [24] proposed an analytical approach to estimate the probability of error propagation between components in a software architecture. The approach is based on a proposed metric, named *error propagation probability*, which represents the probability that an error that arises in one component propagates to other components. The evaluation of the metric requires analytical approach that is based on architecture specifications, and uses information that is typically available at an architectural level, such as set of states of a component and set of messages that a component can exchange with another component. The analytical approach has been validated by comparing the analytical results with the ones obtained by an experimental campaign, both conducted on a part of a large command

and control system used in a critical application.

A black-box error analysis technique for Commercial OTS (COTS) system is described in [25]. The proposed technique studies how information flows between software components. The technique forcefully corrupts the information that flows between components and observes what impacts the corruption had, in order to isolate those components that cannot tolerate the failure of other components. A fault injection technique that injected faults into the interfaces between components is used along with a set of monitors that checks the output of each component to evaluate the propagation of error through the system.

In [26] an extension of an existing Bayesian methodology for reliability estimation of component-based software systems is proposed in order to take into account also the propagation of errors. To this aim, an approach to error propagation probability calculation is presented, which has been integrated into the existing Bayesian methodology for reliability prediction. An automated Personnel Access Control System has been used as case study to compare the existing methodology with the existing one. The obtained results indicate that error propagation can make a significant difference in system reliability prediction, especially if components leaking erroneous states are complex and frequently used. However, the approach is based on some assumptions, i.e., existence of information about failure rates for components and connectors; independence of the failures among different components; each component is expected to exhibit the same failure rate whenever it is invoked.

The impact of inter-modular data error propagation is assessed in [27]. Adopting a

white-box approach, the authors characterized the data error propagation process and derived a set of metrics that quantitatively represents the inter-modular software interactions. A real embedded target system has been used to perform fault-injection experiments to obtain experimental values for the proposed metrics. The obtained results showed that the metrics allow to determine candidate module for replication or equip with EDMs and ERMs.

An approach to the analysis of the reliability of a component-based system that takes into account the error propagation probability is proposed in [28]. The modeling approach can be used to drive several tasks, such as: (i) placing error detection and recovery mechanisms, (ii) focusing the design, implementation and selection efforts on critical components, (iii) devising cost-effective testing strategies. In order to generate the model, the approach assumes that the operational profile of each component is known, as well as its internal failure probability, i.e., the probability that, given a correct input, a failure occurs during the execution of the component causing the production of an erroneous output. The approach has been validate on an Automated Teller Machine (ATM) bank system example.

In [29] the authors present an approach based on static software product metrics to identify software modules where the effects of a fault in that module are not observable. Indeed, the conducted study shows that there is an empirical relationship between static software product metrics and propagation of errors. The work used an adventure game, named Nethack, as case study for the proposed approach.

Other studies, such as [30, 31, 32, 33, 34, 35], address the analysis of the software errors by leveraging the information provided by field data, i.e., data generated by the target

system during operational phase. These studies are detailed in the Chapter 2, where an overview on the analysis of field data is proposed.

1.4 Challenges to Software Error Analysis in Complex Critical Software Systems

As discussed in Section 1.3, software errors represent a major dependability threat for any software systems, since they can propagate through the components of the systems and can potentially lead to a system failure if they are not properly managed.

The analysis of software errors occurred in a software system is a valuable process as it allows to understand how and what type of errors may affect the system, the effect these errors may have on it as well as how they propagate through its components. In turn, this knowledge provides valuable insights on where to place *EDMs* or *ERMs*, which allow improving the reliability of the system.

This is of paramount importance especially in the context of complex critical software system, where the occurrence of a failure has a high cost since it can lead to severe consequences. However, the analysis of the error behavior of this kind of system is not trivial. As discussed in Section 1.1, complex critical software system are often based on many strongly interacting heterogeneous components and layers, including OTS and third party components, legacy systems, and are typically distributed, composed by several software-intensive systems deployed on many remote nodes communicating on a network. In addition, they are expected to satisfy the rules imposed by certification standards, such as the DO-178B [7], as well as they might not be built on the top of cutting-edge technologies as they include

legacy and/or obsolete kernel versions, which limit the intervention degree on the system. Moreover, complex critical systems often lack of documentation, since they can be based on OTS and legacy components, which often do not have a specification, or when this is present, it is not precise and complete.

All these aspects, undermine the use of existing approaches since they are not designed to this kind of system and cope with their complexity. For example, the approach in [24] requires information that is typically available at an architectural level, such as set of states of a component and set of messages that a component can exchange with another component, while the ones in [26, 28] require information about failure rates of each components. The approaches proposed in [27, 29] require a white-box view of the system. Differently, the technique proposed in [25] is black-box, but the used fault injection technique requires both to know the undesirable states of each components and a monitor for each components of the system. In addition, these approaches are not applied to complex systems, as reported in Table 2.1 that summarizes the most complex case study used in each before-mentioned study and in the ones that are detailed in Section 2.4.

Therefore, **the contribution of the thesis is to provide a methodology that allows understanding the error behavior of complex critical software systems, going beyond the limits imposed by this type of systems**. To this aim, the methodology leverages the data generated by the target system during operational phase, i.e., *field data*, by means of monitoring techniques, and is based on the fundamental of *Failure Field Data Analysis* (FFDA). An overview of FFDA and field data is provided in Chapter 2, while the methodology is described in Chapter 3.

Chapter 2

Field Failure Data and Software Errors

Field Failure Data Analysis (FFDA in the following) provides information that allows to understand the effect of errors on system behavior. It provides accurate information on the target system, for the elaboration and validation of analytical models, and for the improvement of the development process. The collected data helps to characterize the system under observation. Quantitative analysis of the failure, error and fault types observed in the field yields feedback to the development process and can thus contribute to improving the production process as well as the reliability of the systems. This chapter discusses the principles of the FFDA methodology and the related literature is examined. Finally, a discussion about open issues and challenges about the use of FFDA for analyze the error behavior of complex critical system concludes the chapter.

2.1 Field Failure Data Analysis: definition and goals

Field Failure Data Analysis groups all fault forecasting techniques which are performed in the operational phase of the life time of a software system. This analysis is valuable in a variety of industrial domains, because it allows evaluating and improving dependability characteristics of computer systems. It is based on the monitoring and recording of errors and failures occurred during the operational phase of the system under real workload conditions, i.e., the failing behavior is not forced or induced in the systems by means of fault/error

injection techniques. Filed data contain rich information about the system reliability, providing valuable information on actual error/failure behavior of a software system during the normal system operation: analysis of naturally-occurring failures/errors is among the most accurate ways to achieve insights into the failure/error behavior of the system [8, 36]. The objective of a FFDA campaign mainly concerns the detailed characterization of the actual dependability behavior of the operational system. More in detail, main goals of FFDA studies can be summarized as the following:

- identification of the classes of errors/failures as they manifest in the field, i.e., the actual failure model and error model of an operational system;
- analysis of failure and recovery times statistical distributions;
- correlation between failures and system workload;
- modeling of the failing behavior and recovery mechanisms, if any;
- identification of the root causes of outages, and indication of dependability bottlenecks;

Although FFDA studies are useful for evaluating the real system, they have some drawbacks. For example, they are limited to manifested failures, such as the ones that can be traced. In addition, the particular conditions under which the system is observed can vary in different installations of the system, thus raising doubts on the statistical validity of the results. Noteworthy, the analysis of data collected on a given system is hardly beneficial to the current version of the system. It can be instead useful for the next generations of

systems. In addition, FFDA may require a long period of observation of the target system, especially when the system is robust and failure events are rare. To achieve statistical validity and to shorten the observation period, these studies should be carried out on more than one deployed system, each of them under different environmental conditions.

2.2 FFDA methodology

FFDA studies are usually based on three consecutive steps: (i) collection, concerning the collection of data to analyze from the actual system, (ii) filtering, which consists in the extraction of the information which are useful for the analysis, and (iii) analysis, that is the derivation of the intended results from the manipulated data. Figure 2.1 summarizes the FFDA methodology, highlighting the sequential relationship among its phases. In particular, once a data source has been selected and field data have been collected from a target system, data filtering phase makes it possible to infer failure data from the selected data source. Finally, failure data are analyzed to characterize properties of interest of the system. Details about the best practice on each of these steps are presented are surveyed in the following.

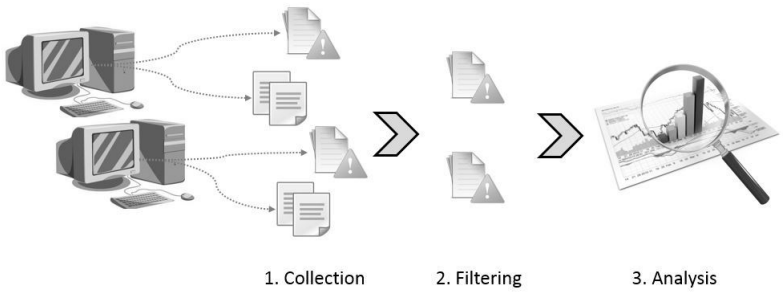

Figure 2.1: The FFDA methodology.

2.2.1 Collection

The **collection** phases of the FFDA methodology allows the gathering of the field data generated from the target system during operational phase. A preliminary study of the system is required for this phase in order to understand what to collect and how to collect it, along with the technique that can be successfully used.

Techniques that are commonly used for the collection phase of FFDA are Failure reports and Monitoring techniques.

Failure reports are generated by human operators, typically users or specialized maintenance staff. A report usually contains information such as the date and hour when the failure has occurred, a description of the observed failing behavior, comments form the operator about the action taken to restore the system, the hardware/software module pinpointed as responsible of the failure, and, if possible, the root cause of the failure. The problem with this technique is that human operators are responsible for the detection of the failure, hence some failure may remain undetected. Moreover, the information contained in the report depends on the experiences and opinions of the operator and can vary from one operator to another. Automated failure report systems have been proposed. An example is represented by the Corporate Error Reporting software proposed by Microsoft. It creates a detailed report every time that an application crashes or hangs, or when the OS crashes. The report contains a snapshot of the state of the system during the crash; including a list that contains the name and timestamp of binaries that were loaded in the memory of computer at the time of crash, as well as a brief stack trace. This information

allows for a quick identification of the routine that caused the failure as well as the reason and cause for the failure.

Monitoring [37, 38] is a well-established practice in software systems because it supports a variety of engineering tasks, such as managing the computing environment, measuring performance indicators, troubleshooting and post-mortem characterization of failures and errors. A monitoring system is defined as a process or a set of possibly distributed processes whose function is the dynamic gathering, interpreting, and acting on information concerning an application as it executes [37]. Therefore, monitoring is a valuable source of field data that analysts can leverage to understand the behavior of a software system at runtime. Beside general-purpose software, monitoring is recommended (if not even mandatory) to comply with a number of international standards and governmental guidelines in **critical software systems**. In fact, monitoring makes it possible to verify whether a critical system is compliant with its expected behavior and preventing catastrophic consequences, such as loss of life and damage to the environment. For example, the IEC 61508-7 [9] suggests the use of "*failure detection by on-line monitoring*", the DoD Guide for achieving Reliability, Availability, and Maintainability (RAM) [11] emphasizes the need to "*monitor system performance to ensure that RAM performance levels meet user needs and constraints*", the AUTomotive Open System Architecture (AUTOSAR), whose safety is enforced through the ISO-26262 standard [10], indicates the use of *execution sequence monitoring* to trace the paths taken by a given program and detect control flow errors.

It is worth noting that monitoring the occurrence of failures plays, and *will play*, a crucial role in critical systems. Testing activities are not able to exhaustively validate a complex

system against every potential fault trigger because of time and budget constraints. A software system is likely to be released with a number of residual software faults. Gaining insights into the reporting ability and limitations of a monitoring technique against failures is crucial to increase the accuracy of runtime data and to infer a number of implications in developing and improving monitoring.

Several **techniques** are currently used to generate monitoring data. Monitoring techniques can be broadly classified into *direct* and *indirect*. **Direct monitoring** techniques actively involve the monitored system in the data gathering phase [39, 40, 41, 42]. Monitoring data are either (i) generated by the target system itself, i.e., *push-based monitoring*, or (ii) obtained by querying the target system, i.e., *pull-based monitoring*. **Indirect monitoring** aims to collect data without relying on the monitored system [43, 44, 45, 46]. The data concerning the system execution are generated at different locations, such as network or operating system, by means of internal or external probes. Examples of indirect monitoring tools are Ganglia [43] and Nagios [44], which monitor the target system by evaluating metrics, such as CPU utilization, system load, used memory, number of running processes.

In the case of **direct monitoring**, the source code of a software system is arranged at development time in a way to generate monitoring data at runtime. For instance, event logging, which belongs to direct monitoring, has been extensively used over the past decades for either post-mortem failure analysis [47, 48, 49], on-line analysis [50, 51], and for characterizing the runtime behavior of industrial and critical systems [52, 53]. On the contrary, **indirect monitoring** relies on a number of probes that generate data without the need for the direct involvement of the monitored software system. For example, operating system

probes can be adopted to measure a number of metrics, such as CPU/memory utilization and system load.

Several direct monitoring techniques have been proposed and used in many domains in the last decades. A substantial body of literature used **event log files** to conduct analysis studies in several application domains, such as networked systems [47], cloud infrastructures [54, 55], web applications and large clusters [49, 50]. The log files are generated at runtime by the monitored software by means of logging instructions in the source code, either using a dedicated library or simple file writing functions. They contains contain valuable information to understand the system dependability behavior. Event logging is a widely consolidated and pervasive practice both within the open-source community and proprietary software systems industry [56, 57, 58, 59, 60]. Studies such as [56, 60] point out that software programs might contain up to one logging instruction every 30 lines of code. Event logs have been successfully adopted also in the context of **critical industrial systems**. For example, in [52] it is proposed a log analysis framework for Mars Science Laboratory flight software [39]. The proposed method is based on the extraction of simple events from application text logs. The extraction is implemented by means of regular expressions and it aims to obtain a well-structured log, which can be analyzed with appropriate patterns to detect failures. In [53] the authors proposed a failure detection and diagnosis framework for component-based distributed embedded systems, such as automotive systems. The framework encompasses a logging layer that allows collecting system events, which are fed to a monitoring and diagnosis layers.

Assertion checking is a direct monitoring technique based on the use of assertions,

i.e., code statements that check invariant properties of a program and produce an alert if one of the properties is violated at runtime [61]. An example is *range checking*, where the assertions perform boundary checks on the values of program variables in order to detect anomalies [62]. In [40] is presented an assertions-based mechanism to detect data errors in Embedded Control Systems. A configuration parameter is defined for each type of monitored signal. An error in a signal is detected as soon as the signal violates the constraints given by the configuration parameter. The mechanism is evaluated by means of an error injection campaign involving an embedded system. Many works use assertions to analyze the runtime behavior of a system. For example, in [63] the authors presented an on-line mechanism to detect software errors during operational execution from data and control-flow viewpoints. In particular, for data viewpoint, the authors used executable assertions defined on functional blocks to detect anomalies on data values. A general-purpose monitoring approach that is implemented for sequential, concurrent, and reactive systems written in Java is presented in [42]. This approach, named *JASS* (Java with ASSertion), includes a pre-compiler for annotated Java programs and an assertion language that support all standard *Design by Contract*[1] assertions that can be introduced in a given program.

Source code instrumentation is based on the insertion of specific instructions into the source code of a given program with the aim of monitoring its behavior. Several works use this approach for failure analysis. For example, [41] presents an approach that uses

[1] An approach widely adopted in the context of critical systems; it allows the specification of assertions in the form of method pre- and post-conditions, class and loop invariants, and further additional checks that can be introduced at several points of the program code.

code instrumentation to detect violations of time constraints in real-time systems. Authors in [64] propose a *rule-based logging approach*. The approach defines a set of rules that establishes how the logging mechanism must be implemented to detect several types of errors, such as errors affecting services and interactions between software modules. Each rule defines the placement of the logging instructions in the code. A monitoring framework, named *LogBus*, has been developed by the authors in [64] based on rule-based logging. In [65] the authors used a software instrumentation package, i.e. DTrace, to perform *Function Boundary Tracing* (FBT), which traces function entry and exit events, in order to monitor the execution of a software system. Another technique based on source code instrumentation is Aspect-Oriented Logging, i.e. a systematic approach to log management based on the *Aspect-Oriented Programming* (AOP) paradigm [66]. In this technique logging can be treated as a system-wide feature orthogonal to other services or to the business logic. For instance, through aspect weaving, a log entry can be systematically produced for each runtime exception, with the aim of supporting transparent exception reporting. Monitoring techniques based on source code instrumentation are also used in the area of runtime verification [67, 68], where a set of *properties* are checked during the execution of the target system. The properties, usually written in a formal specification language, are used to generate a number of monitors; the monitors analyze the events generated by the system at runtime with the aim of verifying whether the properties are met or not. The system is previously instrumented in order to generate the events required by the monitors.

2.2.2 Filtering

Filtering consist in analyzing the collected data for correctness, consistency, and completeness. This concerns the filtering of invalid data and the coalescence of redundant or equivalent data. Indeed, given a large volume of data collected in real systems, a crucial step is inferring the failure data that will be used to perform an FFDA analysis. Filtering encompasses two types of activity, i.e., (i) removing non-useful data, and, more importantly, (ii) coalescing redundant failure data by grouping entries that are related to the manifestation of the same problem. This is especially true when event logs are used. Logs, indeed, contain many information which are not related to failure events. Only a fraction of the entries in the log is useful to conduct the failure analysis: many entries report non-error events and can be excluded from the failure analysis [69]. Filtering non-error events is essentially time-consuming task; however, it does not represent a real problem to failure analysis. It is used to reduce the amount of information to be stored, and to concentrate the attention only on a significant set of data, thus simplifying the analysis process. A manual inspection of the data is valuable to identify the severity of the entries and error-specific keywords. Two basic filtering strategies can be adopted: *blacklist* and *whitelist* strategies. The blacklist can be thought as a list of all the terms that surely identify an event which is not of interest for the analysis. The blacklist filtering discards all those events which description message contains at least one of the blacklist terms. On the contrary, the whitelist is the list of all permitted terms, hence only events which contain these terms are not rejected. Both the approaches can be supported by *de-parameterization* procedures. De-parameterization

replaces variable fields within a text entries (e.g., usernames, IP and memory addresses, folders) with a generic token. For example, the hypothetical entries

```
incoming connection from 225.178.20.37
```

```
incoming connection from 143.195.0.100
```

appear the same once IP addresses are replaced with the `IPAddr` token. De-parameterization reduces the number of distinct messages templates to scrutinize. As shown by [70] around 200 million entries in the log of a supercomputing system were generated by only 1,124 distinct messages.

Once non-error data has been filtered out, it still remains the problem of grouping the error entries representing the manifestation of the same problem. Events which are close in time may be representative of one single failure events. They thus need to be coalesced into one failure event. *Coalescence* techniques can be distinguished into temporal, spatial, and content-based.

Temporal coalescence, or tupling [69], exploits the heuristic of the tuple, i.e., a collection of events which are close in time. The heuristic is based on the observation that two failure events, if related to the same fault activation, are likely to occur near in time. Consequently, if the time distance of the entries is smaller than a predetermined threshold, i.e., the *coalescence window*, they are placed in the same group (called *tuple*). To explain how the tupling scheme works, let X_i be the $i-th$ entry in the log, and $t(X_i)$ the timestamp of the entry X_i. If the condition $t(X_{i+1})t(X_i) < W$ is satisfied (with W denoting the mentioned coalescence window), X_{i+1} is place in the same tuple of X_i. The window size is a crucial parameter which need to be carefully tuned in order to minimize collapses (events

related to two different faults are grouped into the same tuple) and truncations (events related to the same fault are grouped into more than one tuple).

Spatial coalescence is used to relate events which occur close in time but on different nodes of the system under study. It allows to identify failure propagations among nodes, resulting particularly useful when targeting distributed systems. The techniques adopted for spatial coalescence are usually the same as the ones used for temporal coalescence [71, 36]. Finally, *content-based* coalescence groups several events into one event by looking at the specific content of the events into the data. For example, in [72] this technique is adopted to identify machine reboots: when a the system is restarted, a sequence of initialization events is generated by the system. By looking at the specific contents of these events, it is possible to develop proper algorithms to identify machine reboots sequences and group them into one reboot event. Also, content-based coalescence can be used to group events belonging to the same type [73].

2.2.3 Analysis

Collection and filtering make it possible to infer the failure event from the collected data; analysis allows practitioners to achieve meaningful insights from the data. Data analysis step consists in performing statistical analysis on the filtered data to identify trends, to evaluate quantitative measures and to assess fault tolerance and recovery mechanisms.

Error and failure classification is a the first step of the analysis, which aims at categorizing all the observed failures events on the basis of different criteria, e.g., their nature,

severity and originating component. Classification allows pinpoint the most errors/failures-prone components and, in general, the failure modes of the system. Classification results can be used to drive finer-grain analysis. In addition, descriptive statistics can be derived from the data to analyze the location of faults, errors and failures among system components, the time to failure or time to repair distributions, the impact of the workload on the system behavior, the coverage of error detection and recovery mechanisms, etc. Commonly used statistical measures in the analysis include frequency, percentage, and probability distribution [74]. They are often used to quantify the reliability, the availability, and the maintainability.

A substantial body of literature try to conduct the *evaluation and modeling of dependability attributes*. More detailed analysis try to determine the probability distribution of the *time to failure* variable, and, in some cases, of the *time to repair*. This permits to detail the failure model of the system under study. To this aim, the real data are fitted with theoretical, continuous time distributions. The most adopted distributions in this field are the exponential, the *hyper-exponential*, the *lognormal*, and the *weibull*.

The *exponential distribution* was firstly adopted to model the time to failure and time to repair of electronic components. However, it has been often shown that this distribution does not fit real data, especially when the data involves multiple underlying causes or software failures. This is due to the simplistic *memoryless* property of the exponential distribution. Authors in [75] use a hyper-exponential distribution to fit the duration of failures. This type of distribution has been adopted in the mentioned study because the authors observed the existence of multiple predominant failure dynamics in the data: as a result, a two-stage

hyper-exponential model was chosen.

The *lognormal distribution* has been recognized as a proper distribution for software failure rates [76]. Many successful analytical models of software behavior share assumptions that suggest that the distribution of software event rates will asymptotically approach lognormal. The lognormal distribution has its origin in the complexity, that is the depth of conditionals, of software systems and the fact that event rates are determined by an essentially multiplicative process. The central limit theorem links these properties to the lognormal: just as the normal distribution arises when summing many random terms, the lognormal distribution arises when the value of a variable is determined by the multiplication of many random factors. The lognormal distribution has been also used in the context of high-performance computing systems [77].

The *weibull distribution* [78] is probably the most adopted function to model the failure data. widely used parametric family of failure distributions. The reason is that by a proper choice of its shape parameter, an increasing, a decreasing, or a constant failure rate distribution can be obtained. Weibull distributions have been used in many application domains, e.g., [47, 78, 75].

In practice, the modeling of the failure data by means of statistical distribution, is usually supported by goodness-of-fit test procedures, e.g., the Kolmogorov-Smirnov test, to establish whether the chosen distribution is a good model to fit the data.

Other types of analysis are concerned with the *correlation between failure distributions*. The correlation can uncover possible links between failures in different hardware and software modules or in different nodes constituting the system under study. This analysis can

also conduct to the discovery of trends among failure data on event logs. From a theoretical perspective, the trend analysis of event logs is based on the common observation that a module exhibits a period of (potentially) increasing unreliability before final failure. By discovering these unreliability trends, it can be possible to predict the occurrence of certain failures. To this aim, principal component analysis, cluster analysis, and tupling can be adopted [2].

Finally, the analysis activity often conducts to the development of simulation models of the dependability behavior. Models often adopted in the literature are state-machines, fault trees, Markov chains, and Petri nets. The understanding gathered from field data allows to define these models and to populate their parameters with realistic figures, e.g., failure and recovery rates.

2.3 Relevant Applications

FFDA has been used for decades to characterize dependability of operational systems. This section summarizes relevant efforts and reference works in the area of dependability characterization by means of field data: studies have been grouped based on the main analysis objectives they pursue.

2.3.1 Error and Failure Classification

As discussed in Section 2.2.3, the primary task to achieve insights into the meaning of collected failure data is classification. Error and failure classification usually represent the starting point of a FFDA study and have several advantages. For examples, they allow determining the most-predominant failure classes, pinpointing system components that are

prone to generate error/failure data, and support the evaluation of the improvement between subsequent releases of the same product. This information is valuable to conduct quantitative evaluations of the system, and allows a better interpretation of the measurement.

The work in [79] propose an interesting FFDA study on a server machine with the Sun SPARC UNIX OS. Starting from event logs, the work performs a classification of failures and identify the potential trends of errors which lead to failures. Data in the log is classified and categorized to identify error trends leading to failures, and to support MTBF and availability measurements. For examples, authors show that the input-output subsystem is the most error-prone subsystem, and that many network problems observed in the log were not caused by the system under study.

A characterization of operating system reboots of Windows NT and 2K machines is proposed in [80]. The data source adopted in the study was collected over a period of 36 months. The study focuses on unplanned reboots, representing the occurrence of a failure, identified via a content-based coalescence approach. The study demonstrates that the number of failures caused by the operating system itself is smaller in Windows 2K when compared to NT machines; however, the number of failures caused by application code is larger in Windows 2K.

The work in [81] conduct a similar classification study, addressing Windows XP SP1. Th author shows how the percentage of OS failures decreases from 12% for Windows 2K to the 5% of Windows XP, thus demonstrating that system crashes are often due to applications and third party software. In addition, they conduct a detailed classification study to

pinpoint the `.dll` and executable files causing crashes.

Authors in [82] face a rather different application domain. From a classification study of 62 user-visible failures in three large-scale Internet services, i.e., Online, Content, and ReadMostly, they observe that front-ends are a more significant problem than is commonly believed. In particular, operator error and network problems are shown to be leading contributors to user-visible failures.

Understanding the distribution of the failure data among different classes can provide a feedback about the quality of analysis results. In [70] authors demonstrate, in the context of supercomputing system, that the classes of failures that bias the content of the log, i.e., the most entries-prone classes, can distort measurements.

2.3.2 Diagnosis and Correlation of Failures

FFDA analysis allows achieving in-depth understanding of causes and correlation among failures. The use of only measurements-based approach does not allow to obtain this type of evidence. The use of models, and statistic artifacts applied to the data are also required to reach this goal. Works in the area, dating back to the 1980s, demonstrated the existence of a relationship between the failure behavior and the workload run by a system.

In particular, during a performance measurement campaign for a large DEC-1OA time-sharing system, it was found that the simplistic assumption of a constant system failure rate did not agree with measured data [83]. Subsequent research by the same authors [84] involves use of a doubly stochastic Poisson process to model failures. The model relates the instantaneous failure rate of a system resource to the usage of the considered resource.

Moved by this research, the authors in [85] proposed an approach to evaluate the relationship between system load and failure behavior that presumes no model a priori, but rather starts from a substantial body of empirical data. The study was conducted on three IBM 370 mainframes, and both failure data (maintenance failure reports) and performance counters (via a proprietary IBM system) were gathered. A regression analysis of failure and performance data evidenced the strong correlation between failure manifestation and system load.

Several works suggest that failures observed in different components of a computer system are correlated. In [86] authors defined an analysis methodology for event logs of Tandem systems through multivariate techniques, such as factor and cluster analysis. The event logs were gathered from three Tandem systems over a 7 months period. A 2-phase hyperexponential distribution was adopted to model the error temporal behavior, according to the two error behaviors exhibited by the three systems: error bursts and isolated faults. Although the number of errors observed during the system operations was relatively small, authors observed that multiple processes were affected by the same problem, because of the presence of shared resources.

Authors in [87] perform a measurement study to assess the dependability of seven DEC VAX machines. The analysis aimed to estimate the distributions of the Time Between Errors and Time Between Failures, to analyze dependencies between errors and failures. Again, shared resources turned out to be a relevant dependability bottleneck. Moreover, the analysis showed that errors and failures occur in bursts, and that, neglecting failure correlation phenomena can significantly impact the quality of the measures.

The work in [88] evidences the feasibility of on-line diagnosis approaches based on trend analysis and real data. Specifically, it concentrates on the recognition of intermittent failures and defines a methodology to distinguish between transient, permanent, and intermittent failures by looking at the correlation between consecutive failure events. Statistical techniques are used to quantify the strength of the relationship among entries in the log. About 500 groups of failures are identified over a 14 months time span.

2.3.3 Failure Prediction

Analysis of failure data log is the basis also failure prediction studies. Several works have been developing techniques to predict failures, based on the occurrence of specific event patterns in field data. Predicting failures is challenging; however, it allows applying failure avoidance strategies, triggering corrective and recovery actions, reducing the Time To Repair, enhancing system dependability.

In [78] a failure prediction technique, called the *Dispersion Frame Technique* (DFT), is presented. The technique is defined by starting from the statistical characterization of real data observed on a 13 SUN 2/170 nodes, running the VICE file system, over a 22 months period. By gathering data by both event logs (regarded as errors) and from operators failure reports (regarded as failures), authors concentrate of the identification of error trends which lead to failures. The effectiveness of the DFT is shown via direct experiments on actual data. In particular, it is shown that the DFT uses only one fifth of the error log entry points required by statistical methods for failure prediction. Also, the DFT achieves a 93.7% success rate in failure prediction.

Authors in [50] analyze event logs from a 350-node cluster system. Logs encompass reliability, availability and serviceability (RAS) events, and system activity reports collected over one year. Authors observed that data in the log were highly redundant: for this reason, filtering techniques have been applied to model the data into a set of primary and derived variables. The prediction approach, based on a rule-based classification algorithm, was able to identify the occurrence of critical events with up to 70% accuracy.

A deep study on the logs from a production IBM BlueGene/L system has been conducted in [89]. The authors proposes empirical failure prediction methods which can predict around 80% of the memory and network failures, and 47% of the application I/O failures.

2.3.4 Using Field Data to Characterize Security

FFDA campaigns have also been used in the context of security community, where security analysis have been conducted starting from the data collected during the progression of malicious activities and security attacks. several works appeared which attempted to characterize and to model system vulnerabilities and attacks starting from field data.

An outstanding example is represented by the Honeynet project [90]. A honeypot can be defined as a monitored computer environments placed on the Internet with the explicit purpose of being attacked. By placing honeypots on the Internet and by gathering data on the malicious activity affecting them, one can study the characteristics of attacks and system vulnerabilities. As an example, the study in [91] aimed at using data collected by honeypots to validate fault assumptions required when designing intrusion-tolerant systems. Authors set up three machines equipped with different operating systems (Windows NT and

2K, and Red Hat Linux) and collected network-related data (via tcpdump) for four months to analyze the source of attacker and the attacked ports. The work evidenced that, in most cases, attackers know in advance which ports are open on each machine, without performing any port scan. Moreover, there were no substantial differences in the attacks made on different operating systems.

A similar setup has been used by [92]. In this case the testbed was composed by two Windows 2K machines and security data have been collected over a time period of 109 days by means of the Ethereal tool. The objective of the study was to establish the characteristics of the data that allowed separating different classes of attacks. This work, which shows how to use field data to recognize attacks, established that features, such as, number of bytes constituting the attack or mean distribution of the bytes across the packets, are valuable metrics to separate attacks.

The work in [93] exploits data from the Bugtraq database and proposes a classification of vulnerabilities. In particular, vulnerabilities are dominated by five categories: input validation errors (23%), boundary condition errors (21%), design errors (18%), failure to handle exceptional conditions (11%), and access validation errors (10%). The primary reason for the domination of these categories is that they include the most prevalent vulnerabilities, such as buffer overflow and format string vulnerabilities. Starting from this data and helped by code inspections, authors propose finite state machine models for vulnerabilities, which help to better understand their behavior and/or to uncover new ones.

Authors in [94] conduct an in-depth study of the forensic data (e.g., syslog, Intrusion

Detection System (IDS) logs) produced by the machine of a large-scale computing organization. Attack data adopted in the study are collected over a timeframe of 5 years. The analysis aims to achieve insights into the progression of attacks, to pinpoint the type of alerts that are more likely to catch different types of attacks, and to investigate causes of undetected incidents. Analysis results are valuable to model security attributes and to develop monitoring tools.

2.3.5 Monitoring Techniques Characterization

Field data have also been used to evaluate the effectiveness of monitoring techniques, which are one of the main sources of field data as discussed in Section 2.2.1.

In [95] a platform, called **SMock**, for testing and evaluating runtime monitoring tools is presented. The platform allows generating a *Java mock test system*, which is used as a benchmark for the runtime verification tools under test, starting from a specification. The mock system is run under a given monitoring tool. After the execution, SMock generates a report that contains the execution time of the system, the average memory usage and cpu utilization. The analysis of the reports generated by SMock makes it possible to assess the impact and the performance overhead induced by the considered runtime verification tools on the mock system.

In [64] the rule-based logging technique is compared to the traditional logging in two open-source systems. The authors adopt a software fault-injection approach to assess the logging techniques under different failure manifestations. The results, obtained from the analysis of the collected log for the techniques, indicate that rule-based logging significantly

improves the failure detection capability of the traditional logging approach; however, it misses some details that could help to understand failure causes (e.g., a given file could not be opened, a service was invoked with bad parameters).

The work [96] presents an experimental analysis of different monitoring techniques in web-based applications. The work focuses on both direct and indirect monitoring techniques and tools, i.e., Zabbix [97], the log-analyzer Swatch [98], a monitoring module based on aspect-oriented programming and an end-to-end monitoring technique based on JMeter [99]. The experimental study consists in the emulation of anomalous scenarios and the evaluation of coverage, detection latency and overhead of the techniques. The analysis of the obtained file data allows understandig that the AOP based technique is characterized by the maximum failure reporting rate, followed by the end-to-end monitoring technique, Swatch and Zabbix.

It should be noted that no one of the cited works characterize the effectiveness of a monitoring technique with respect to failures and errors in a comprehensive way. Indeed, [95] does not focus on the failure and error reporting ability of the monitoring techniques, [64] is limited to only two monitoring techniques, and [96] presents preliminary measurements conducted considering a small number of scenarios (i.e., order of 10), which involved fairly obvious error manifestations, such as abnormal memory and CPU consumption.

2.4 Related Research and Thesis Contributions

Literature proposing techniques and measurements based on the analysis of field data encompasses different domains, as discussed in the previous Section. Field data have been

used also to characterize the error behavior of software systems and/or identify locations for EDMs and ERMs. In particular, some studies are based on the use of source code instrumentation and monitoring techniques to collect field data in order to analyze how errors manifests in a given system, the effects they may have on the system as well as how they propagate through the software.

For example, a framework for profiling modular software with regard to error propagation and error effect is proposed in [30]. The framework, called *EPIC*, may be used to find the modules and signals which are most exposed to errors in a system and to ascertain how different modules affect each other in the presence of data errors, i.e., errors in variables and signals. EPIC makes use of *variable instrumentation* to trace the values of variable in order to estimate the proposed error permeability, which evaluates the ability of a module to contain errors, and propose the placement of EDM. The proposed framework has been successfully applied to part of an embedded control system used for arresting aircraft on short runways and aircraft carriers.

In [31] authors propose a system, called *Triage*, that automatically performs onsite software failure diagnosis providing a detailed diagnosis report, which includes the failure nature, triggering conditions, related code and variables, the error propagation chain, and potential fixes. The system makes use of kernel-level components and of multiple re-executions of the target software to support failure diagnosis; during each re-execution, detailed information is collected via *dynamic binary instrumentation* in order to conduct the analysis of the occurred failure and its causes. The system has been applied on 9 applications with different complexity, such as TAR, Apache web server, MySQL.

An environment for examining the propagation of errors in software is proposed [32]. The environment, called *PROPANE*, allows the analysis of the propagation of data errors in a single-process software system written in C, identifying the error paths and evaluating the propagation times. With this aim, *PROPANE* makes use of a fault injection mechanism to induce data error in the system as well as of *instrumentation of the variables* of the system to detect the occurrence of errors. Part of an embedded control system used for arresting aircraft has been used to evaluate the effectiveness of the proposed tool.

A method for assessing error data propagation for operating systems is proposed in [33]. The method is focused on the analysis of the behavior of error in device driver, and on the propagation of these errors to applications which makes use of the target device driver. Errors are induced into a device driver by means of fault injection at interface level, while the detection of the propagation of an occurred error to application is obtained by means of *assertions*. A set of metrics are proposed, i.e., Service Error Permeability, OS Service Error Exposure, Driver Error Diffusion, which help to understand if the target driver needs a wrapper. The method has been assessed on Windows CE .Net operating system, which was chosen for its limited complexity.

In [34] an approach to capture the importance of variables in dependable software systems is proposed. The approach is based on a proposed metric, named importance, which captures the impact a given variable has on the dependability of a software system. The evaluation of the metric requires the *instrumentation of the variables* of the system, in order to understand when a variable is corrupted. Based on the proposed metric, the approach allows to provide insights on the design and positioning of EDMs and ERMs in order to

guarantee that critical variables always hold appropriate values. The approach has been assessed on an open-source flight simulator.

A diagnosis tool, called SherLog, is proposed in [35]. The tool analyzes *event logs* generated by a software system during failure executions and its source code to automatically generate control-flow and data-flow information to help engineers diagnose the errors occurred in the system. In particular, SherLog is able to provide the reporting path of an error, using a static analysis of the source code. The tool has been evaluated on 8 different real-world failures from 7 applications, which range between *rmdir* GNU utils to the *Apache* web server.

It is worth noting that the use of these solutions is not trivial in complex critical software systems, which is the focus of this dissertation. As discussed in Section 1.4, complex critical software systems are usually distributed, characterized by multiple levels, and several components, each one with complex interconnections and several variables. In addition, they are expected to satisfy the rules imposed by certification standards, such as the DO-178B [7], as well as they might not be built on the top of cutting-edge technologies as they include legacy and/or obsolete kernel versions, which limit the intervention degree on the system. These aspects undermine the use of existing techniques based on field data in complex critical software systems. For example, the approaches in [30, 32, 34] are based on instrumentation of variables, which might be expensive in a complex critical software system composed by several modules. In particular, the approach in [30] also requires to measure the proposed error permeability for each input of each module, which lead to a low scalability of the approach; while the tool in [32] can be used only for single process software, which limits

Table 2.1: Most complex case study used in each considered work.

work	**Most complex case study (name, type or LOC)**
[31]	*MySQL*
[23, 34]	*FlightGear Flight Simulator - ∼220K LOCs*
[29]	*Nethack - adventure game*
[32, 30, 27]	*Part of an embedded control system for arresting aircraft*
[33]	*Windows CE .Net operating system*
[24]	*Part of a Computer Software Configuration Item*
[25]	no case studies
[26]	*Personnel Access Control System*
[28]	*Automated Teller Machine (ATM) bank system example*
[35]	*Apache web server - ∼317K LOCs*
this study	*Real-world ATC communication middleware - ∼790K LOCs*

its applicability to complex software systems. On the other hand, the system proposed in [31] makes use of kernel-level components and dynamic binary instrumentation, while the one proposed in [33] is conceived for OS device drivers. In addition, the tool in [35] requires static analysis of the source code, which can be either expensive on system composed by a large number of lines of codes, or inapplicable if the source code is not totally available. Noteworthy, these approaches are not applied to complex systems, as reported in Table 2.1 that summarizes the most complex case study used in each before-mentioned study and the ones discussed in Section 1.3, along with the most complex one considered in this dissertation. This is a further evidence of their inapplicability to this type of systems.

Therefore, **the contribution of the thesis is to provide a methodology that allows understanding the error behavior of complex critical software systems by means of field data generated by the monitoring techniques that are already**

implemented in the target system. The use of monitoring techniques already implemented in a software system allows to avoid any changes in the target system, preserving its functionalities and performance. As discussed in Section 2.2.1, monitoring techniques are one of the main source of field data since, beside being recommended by several international safety standards and governmental guidelines, e.g., IEC 61508-7 [9], the AUTomotive Open System Architecture (AUTOSAR) through the ISO-26262 [10], and the DoD Guide for achieving Reliability, Availability, and Maintainability (RAM) [11], they are consolidated and pervasive practice both within the open-source community and proprietary software systems industry. The following challenging questions are related to the former:

- **RQ1**: *Is it possible to use monitoring techniques to characterize the error behavior in complex critical software system?* Field data generated by means of monitoring technique contain valuable information about the behavior of the target system at runtime. However, it is not trivial to analyze them in order to obtain valuable information about the error behavior of the system during failing executions, and especially about the propagation of errors. Therefore, an ad-hoc methodology is required to extract and analyze data provided by a monitoring technique in order to understand what happen in the system in terms of errors when a fault is activated, according to the considered monitoring technique. More in details, the methodology has to leverage error data generated by a monitoring technique to infer (i) the error model considered by this technique, (ii) the error behavior of the system at varying the type of activated fault and the type of failure occurred in the system, according to the inferred error model,

(iii) how errors propagate through the components of the target system, (iv) the failure reporting ability exposed by the considered monitoring technique at varying the type of activated fault and the type of failure occurred in the system.

- **RQ2**: *Is it possible to improve the error detection/recovery of a complex critical software system from error data?* The knowledge of the error behavior of the target system and of how the errors propagate through its components are often used in the literature to identify the locations for EDM and ERM, which allow an improvement of the detection and recovery of errors in the target system. Therefore, the methodology has to allow inferring the locations where the placement of EDM and ERM might be beneficial for the system, and the type of error/failure they have to cope with.

- **RQ3**: *How do the error and failure reporting ability change between different monitoring techniques implemented in a given system? And what about the dissimilarity of their data?* A number of monitoring techniques can be implemented in a complex critical software system, which can be of different types and consider different error models; therefore, it can be useful to compare the performance exhibited by each technique in order to provide insights to developers to implement better monitoring techniques. To this aim, the methodology has to compare the performance exhibited by each monitoring technique in terms of (i) ability to report failure occurred in the target system, (ii) failure coverage by failure and fault type, (ii) ability to report the propagation of errors and (iv) dissimilarity of the data they provide at varying failure manifestation. There are studies that try to address this topic; however, they do not

characterize the effectiveness of a monitoring technique with respect to failures and errors, as discussed in Section 2.3.5.

- **RQ4**: *Is it useful to combine different monitoring techniques implemented in a complex critical software system?* Different monitoring techniques implemented in a software system might expose orthogonal performance, complementing each other in terms of failure reporting and/or error propagation reportability. Therefore, the methodology has to be able to evaluate (i) the orthogonality of the monitoring techniques implemented into the target system and (ii) the potential benefit of their combination.

The following chapters try to answer the above mentioned questions, by describing the proposed methodology as well as the results obtained form its application to two different real-world critical software systems in the field of Air Traffic Control domain. These chapters are the result of a three years experience, and partially extend previously published results, as [100, 101, 102].

Chapter 3

Software Error Analysis: a data-driven methodology

The analysis of errors occurred in a software system is one of the main activity to incorporate dependability structures and mechanisms where they are the most effective. To this aim, know where errors tend to propagate and where errors tend to do the most damage, leading to failure, is of paramount importance. Several solutions are proposed in the literature to conduct error analysis of a software system. However, their application is not trivial in the context of complex critical software systems. In this chapter a methodology to conduct error analysis in the context of complex critical software systems is presented.

The methodology leverages field data generated by means of monitoring techniques (MUT) already implemented in the target system (SUT), in order to (i) understand the error behavior exposed by the target system and to (ii) assess the effectiveness of the monitoring techniques implemented in the system. An automated framework, based on software fault injection experiments, is used to collect field data from the MUTs. The collected data are used to evaluate a number of metrics in order to quantify the ability of MUTs at reporting useful notifications in face of failures, as well as to evaluate the error behavior exhibited by the SUT they allow to infer.

3.1 Introduction

The knowledge on how faults and errors may affect a software system is of paramount importance in order to improve their reliability. In particular, the design of effective error detection mechanisms requires not only knowledge on which types of errors to detect but also the effect these errors may have on the software, i.e., the failures they lead to, as well as how they propagate through the software, i.e., the error propagation. Existing solutions

allow obtaining this knowledge in different ways. However, their application is not trivial in the context of complex critical software systems, as discussed in the Section 2.4, since they are often characterized by multiple and distributed nodes, multiple levels, each one with multiple modules, and also by some constraints in terms of intervention degree and/or performance.

The proposed methodology aims to leverage field data generated by means of monitoring techniques already implemented in the target software system in order to understand the error behavior of the system, avoiding intrusive modifications of its source code to collect useful data to analyze. It should be noted that the higher the effectiveness of the considered monitoring techniques at reporting errors and failures occurred in the target system, the higher the comprehensiveness the methodology is able to provide in terms of error behavior of the system. For this reason, the proposal is also conceived as a methodology to compare the effectiveness of different monitoring techniques implemented in different target systems. To this aim, the methodology leverages information retrieval metrics and other proposed metrics.

3.2 Proposed Methodology

Let the **System Under Test (SUT)** be the target software system, i.e., the software implementing the monitoring techniques to assess. The SUT is exercised with a *faultload* and a *workload*. The **faultload** consists of a set of software faults, which represents common programming mistakes found in real software systems; the **workload** is a typical operational profile for the SUT. The execution of the SUT subjected to the faultload and the workload

allows inducing errors and failures into the SUT. The proposed method makes it possible to measure the extent the direct monitoring techniques implemented by the SUT are able to report the occurrence of the failures induced through the faultload and also to evaluate the error behavior exhibited by the SUT according the considered monitoring techniques. According to this concept, the notion of **Monitoring technique Under Test (MUT)** is introduced.

Considered MUTs are evaluated by using different metrics. *Precision* and *Recall* of a MUT, i.e., the ability of a MUT at generating monitoring data upon the occurrence of a failure, and its *failure coverage*, i.e., the ability of a MUT at reporting different types of failure occurred in the SUT, are measured. In addition, three new metrics are defined:

- **Data Dissimilarity**, which allows gaining insights into the suitability of the data generated by a MUT for manual failure analysis.
- **Error Determination Degree**, which allows gaining insights into the ability of error notifications of a MUT to suggest either the fault that led to the error, or the failure the error led to in the SUT.
- **Error Propagation Reportability**, which allows understanding the ability of the MUT at reporting the propagation of errors in the SUT.

The effectiveness of different MUTs at reporting the set of failures induced in the SUT are compared as well as their effectiveness in terms of *Error Determination Degree* and *Error Propagation Reportability*. The metrics that are adopted in this dissertation are detailed in Section 3.3. All the characteristics of the proposed method, such as faultload, workload

and experimental procedures, are detailed in the following.

3.2.1 Faultload

The faultload consists of a set of **software faults**, i.e., common programming mistakes that can be found in the source code of real-world software systems. The faults adopted in this dissertation belong to the well consolidated **orthogonal defect classification** (ODC) [103]. The fault types proposed by [104] have been considered, which extend the ODC classes for practical injection purposes. The authors in [104] analyzed the fault distributions of a number of software systems and identified a subset of representative fault types observed in the field.

Table 3.1 reports the fault types used in the proposed methodology and the ODC class to which they belong, i.e., algorithm (ALG), assignment (ASG), checking (CHK), interface (INT). Each fault type represents a typical programming mistake, such as *missing variable initialization*, *missing function call*, *wrong values assigned to variable*. According to the estimates in [104], the fault types adopted in this study represent a subset accounting for total around 80% of representative faults found in real-world software systems.

It should be noted that the methodology leverages fault injection means to induce the faultload in the considered SUTs. The rationale behind this choice is to accelerate the collection of failure data generated by the MUTs, avoiding to wait for naturally occurred failures. However, practitioners might apply the proposed methodology also on naturally occurred failure data. Indeed, knowing the root cause of the occurred failure and the failure mode, i.e., the way the SUT fails, each failure manifestation can be categorized according

Table 3.1: Fault types adopted in the methodology. (*ALG*-algorithm, *ASG*-assignment, *CHK*-checking, *INT*-interface)

Fault		**ODC type**
MFC	missing function call	*ALG*
MVIV	missing variable initialization using a value	*ASG*
MVAV	missing variable assignment using a value	*ASG*
MVAE	missing variable assignment with an expression	*ASG*
MIA	missing IF construct around statements	*CHK*
MIFS	missing IF construct plus statements	*ALG*
MIEB	missing IF construct plus statements plus ELSE before statement	*ALG*
MLC	missing AND/OR clause in expression used as branch condition	*CHK*
MLPA	missing small and localized part of the algorithm	*ALG*
WVAV	wrong value assigned to variable	*ASG*
WPFV	wrong variable used in parameter of function call condition	*INT*
WAEP	wrong arithmetic expression in parameter of a function call	*INT*

the considered fault types and failure model, which is detailed in the next section.

3.2.2 Workload and Failure Model

Each fault of the faultload is injected into the SUT: the SUT is exercised with the workload in order to trigger the fault and to induce the occurrence of a failure. The workload is *SUT-dependent* and does not vary across the experiments that emulate the faults.

A general classification to categorize the failures induced in the SUT has been adopted. The classification is based on a reference paper in the area of dependability [1]:

- *CRASH*: abrupt/unexpected termination of the SUT; the pid(s) of the process(es)

encapsulating the software are deallocated by the operating system before the software system is correctly halted.

- *SILENT*: the SUT is up, but no output/functionality is provided within the expected timeout, e.g., the system is hung or no output is generated at all. The expected timeout has to be established by means of fault-free runs of the system before the injection experiments.

- *ERRATIC:* bad output, misconfigurations, exceptional conditions, and errors impacting the system internal components that do not cause *CRASH* or *SILENT* failures.

- *NO FAILURE:* no failure manifestation is noted during the experiment; the injected fault is not activated or it does not cause the failure of SUT.

3.2.3 Experiments procedure

A campaign of experiments is conducted to collect the data generated by the MUTs. For each experiment, one fault belonging to the faultload is induced into the SUT. The SUT is exercised with the workload, and the monitoring data generated by the MUTs are collected.

Figure 3.1 shows the steps of the procedure adopted for each experiment. The execution of the experiments is automated and supervised by a **controller** program, such as indicated by Figure 3.1. The controller injects the faults, starts/stops the SUT, and reboot the machines to ensure the same initial conditions for each experiment (e.g., no zombie processes, unallocated semaphores, and shared memories are left by the previous experiment). The steps are described in the following:

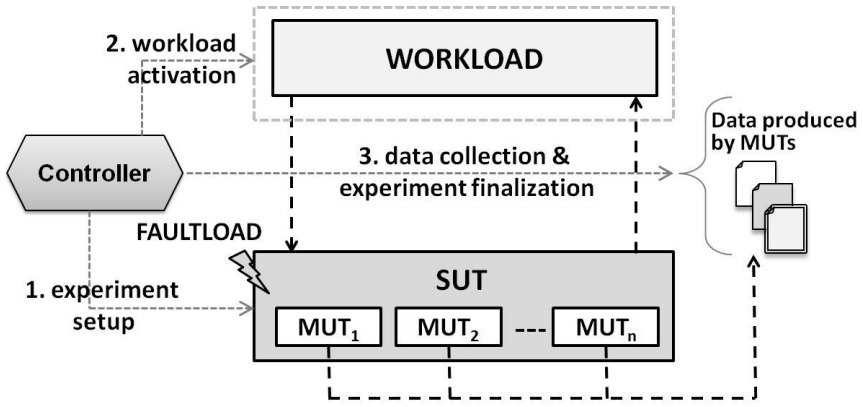

Figure 3.1: Assessment approach.

1. **Experiment setup**. One fault belonging to the faultload is introduced in the SUT. Then, the SUT is started.

2. **Workload activation**. The SUT is exercised with the workload. The workload invokes the SUT with the goal of triggering the fault and activating the MUTs under error conditions.

3. **Data collection and experiment finalization**. The monitoring data generated by a MUT are saved in a file either when (i) the workload completes or (ii) a predefined timeout expires (the timeout is established before the campaign by means of fault-free runs of the SUT). After the files containing the data generated by the MUTs are saved for subsequent analysis, the SUT is restored, the files are cleaned, and the machines are rebooted before the next experiment is performed.

The Controller establishes whether a failure occurred or not upon the completion of the experiment. The Controller analyzes both *operating system*-level data (e.g., pid(s) of the process(es) executing the SUT, core dumps), and *workload*-level data (e.g., the output

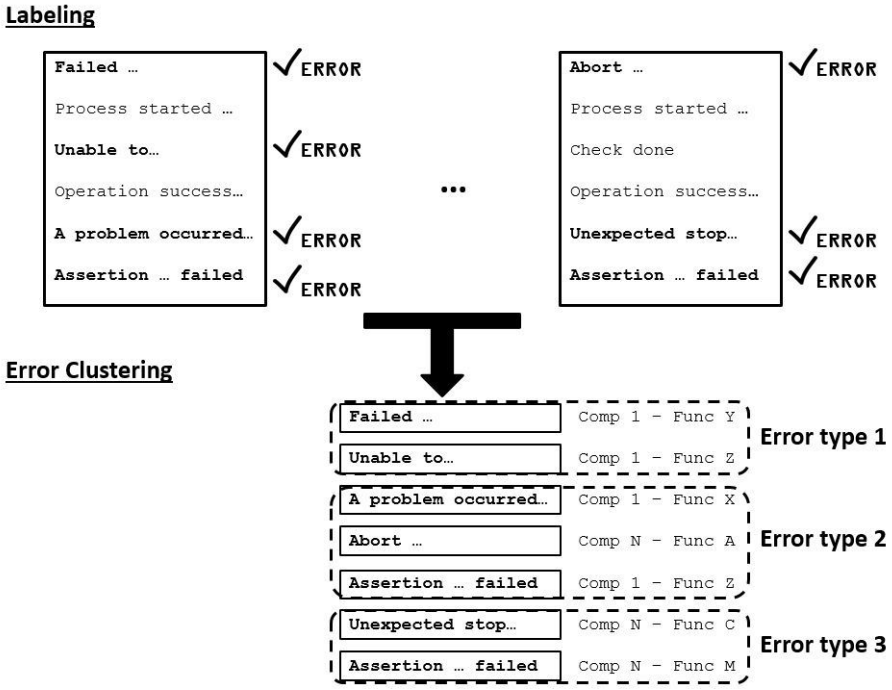

Figure 3.2: Labeling and Error Clustering.

generated by the SUT and the response time) to establish the type of failure out of the adopted failure model. For example, the Controller would label as *SILENT* an experiment where all the OS processes incarnating the SUT are still alive after the completion of the experiment, but no function has been delivered within the expected timeout.

3.2.4 Labeling and Error Clustering

In order to make the collected data suitable for the analysis, the methodology provides two further steps, i.e., *labeling* and *error clustering*, as shown in Figure 3.2.

Labeling aims to label each notification generated by a MUT into *no error*-reporting, i.e., the notification does not report an error, and *error*-reporting, i.e., the notification reports an error; the labeled data allow the evaluation of the metrics provided by the

proposed methodology.

Error clustering aims to classify the types of error reported by each MUT, and to infer the **error model** of the MUT in the related SUT, i.e., the types of error that the MUT is able to report into the related SUT. The clustered error data allow obtaining insights about the error behavior exposed by a SUT according to the considered MUT. Once the error model of a MUT is defined, all its error notifications are labeled with the type of reported error. In addition, this step also includes the labeling of each error notification with its source function and component, i.e., the function that generates the error notification and the component of the SUT the function belong to, respectively, which are provided by the error notification.

3.2.5 Error Propagation Graph

The dataset that contains for each fault injection experiment (i) the number of errors of each type reported by a MUT in each component of the related SUT during the experiment, (ii) the type of injected fault, (iii) the ODC class the fault belongs to, and (iv) the type of failure occurred in the SUT, has to be generated from the dataset obtained during the experimental campaign. For each MUT of a SUT, a dataset of this type is generated in order to evaluate how the errors propagate through the components of the considered SUT.

The information contained in these datasets allows to build non-exhaustive directed graphs, one for each ODC class, that summarizes the error propagation phenomena obtained during the experimental campaign in a SUT, named *Error Propagation graphs*. These graphs have been partially inspired by the ones proposed in [105]. It should be noted that

the error propagation graphs are considered non-exhaustive because they are built based on the errors detected by a MUT, i.e., the errors that led to at least an error notification to be generated by the MUT in the components of the SUT. Therefore, the errors undetected by the MUT cannot be considered. Each directed graph is characterized by three types of node: (i) the *fault type nodes*, i.e., the nodes that represent the fault types that belong to the considered ODC class, (ii) the *component nodes*, i.e., the nodes that represent the components, or groups of components, of the considered SUT, and (iii) the *failure nodes*, i.e., two nodes that represent the occurrence or not of a failure in the considered SUT. Noteworthy, the node that represents the component where the faults have been injected during the experimental campaign, named *faulty component* here, is divided in two nodes:

- **faulty component-IMMEDIATE** that represents the function where a fault has been injected;
- **faulty component-QUICK** that represents the remaining part of the faulty component.

An example of error propagation graph for a specific ODC class and MUT of a SUT is shown in Figure 3.3. The *absolute* and *probability* values in a fault type node indicate the number of fault of this type for which the considered MUT have generated at least an error notification and the probability to have an error notification from the MUT about this type of fault, i.e., the ratio between the before mentioned *absolute* value and the number of faults of the considered ODC class for which the MUT has generated at least an error notification, respectively.

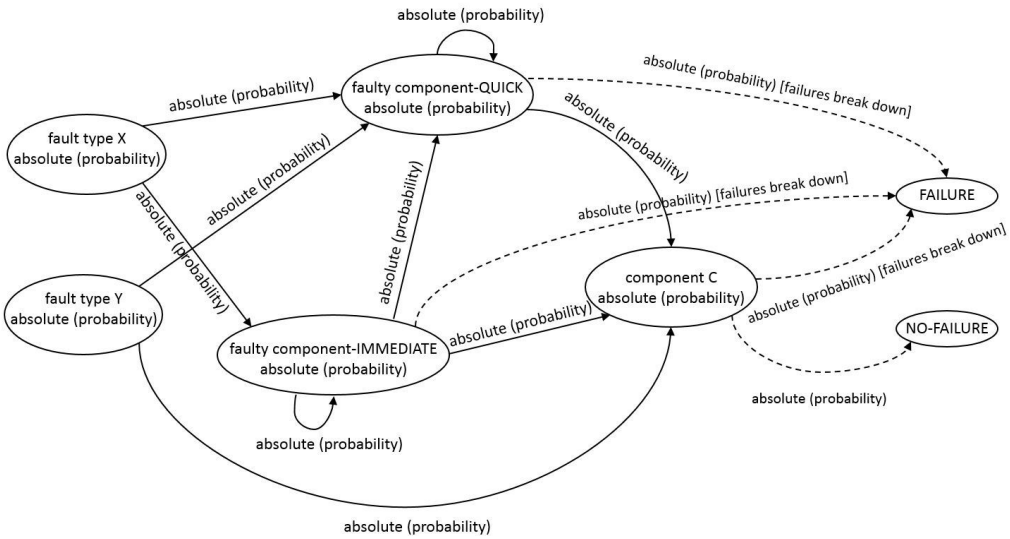

Figure 3.3: Example of directed graph with failure nodes.

Differently, the *absolute* and *probability* values in a component node refer to the errors generated as a consequence of a fault of the considered ODC class and for which the MUT has generated at least an error notification, named *MUT reported errors* here. Precisely, the *absolute* value indicates the number of MUT reported errors that propagate through the related component, i.e., the number of MUT reported errors for which only the component, or almost the component and the faulty component at the same time, has generated at least an error notification; while the *probability* value indicates the probability that a MUT reported error propagates only through the component, or almost through the component and the faulty component at the same time, i.e., at least an error notification is generated for it by the component, or almost by the component and the faulty component at the same time, which is obtained as ratio between the before mentioned *absolute* value and the number of MUT reported errors. In particular, the *absolute* and *probability* values in

the faulty component-IMMEDIATE node indicate the number of MUT reported errors for which at least an error notification has been generated by the function where the fault has been injected, named faulty function, and the probability that at least an error notification is generated for a MUT reported error by the faulty function, which is obtained as ratio between the before mentioned *absolute* value and the number of MUT reported errors. Similarly, the *absolute* and *probability* values in the faulty component-QUICK node indicate the number of MUT reported errors for which at least an error notification has been generated by the other non-faulty functions, i.e., all the function excluding the faulty function, and the probability that at least an error notification is generated for a MUT reported error by one these non-faulty functions, which is obtained as ratio between the before mentioned *absolute* value and the number of MUT reported errors. It should be noted, that in case of component nodes that represent a group of nodes, these nodes only refer to the MUT reported errors that are exclusively reported by them at the same time, or almost by them and the faulty component at the same time.

Regarding the arches, the *absolute* and *probability* values on an arch from a fault type node and a component node indicate the number of fault of this type that led to a MUT reported error notified by the component, i.e., the number of MUT reported errors, generated as consequence of a fault of this type, that propagated through the component, and the probability that a MUT reported errors, generated as consequence of a fault of this type, propagates through the component, which is obtained as ratio between the before mentioned *absolute* value and the *absolute* value of the considered fault type node, respectively. In the same way, the *absolute* value on an arch from two component nodes indicates the number

of MUT reported errors that propagate form the first component to the second one, i.e., the MUT reported errors for which at least an error notification has been generated in both the components at the same time, while *probability* value indicates probability that a MUT reported error propagates from the first component to the second one, i.e., the probability that a MUT reported errors lead to at least an error notification to be generated in both the components at the same time, which is defined as the ratio between the before mentioned *absolute* value and the number of MUT reported errors. Finally, the *absolute* value on an arch from a component node to the *FAILURE* node indicates the number of MUT reported errors, which have been reported by the component, that have led to a failure in the considered SUT (the break down of the failure is reported in the square brackets); while the *probability* value represents the probability that a MUT reported errors, which have been reported by the component, have led to a failure in the considered SUT, which is obtained as ratio between the before mentioned *absolute* value and the *absolute* value related to the node component. In the same way, the *absolute* value on an arch from a component node to the *NO-FAILURE* node indicates the number of MUT reported errors, which have been reported by the component, that have not lead to a failure in the considered SUT; while the *probability* value represents the probability that a MUT reported errors, which have been reported by the component, have not led to a failure in the considered SUT, which is obtained as ratio between the before mentioned *absolute* value and the *absolute* value related to the node component.

It should be noted that the propagation phenomena showed in the error propagation graphs are almost three-level propagation phenomena. The assumption here is that the error

reported in the faulty component are the cause of the errors reported by other components, as well as the error reported by the faulty function is the cause of the error reported by the non-faulty functions of the faulty component. Therefore, for example, if error notifications are generated by the faulty function, by a non-faulty function of the faulty component, and by another component of the considered SUT, e.g., *COMPONENT X*, at the same time, it is possibly to assume that the error has been propagated from the faulty function to the non-faulty function of the faulty component, and from the latter to the other component, than an arch form the *faulty component-IMMEDIATE* node to the *faulty component-QUICK* node can be drown, as well as an arch form the *faulty component-QUICK* to the *COMPONENT X* node.

Differently, nothing can be said about the causality between the notification generated in two non-faulty components since no information are available on how the components interact each other, and which requires a deep knowledge of the system. Therefore, if an error notification is generated in two non-faulty components, no directed arches can be drawn between the components.

Noteworthy, it is also possible to build a graph by considering many MUTs at the same time. In this case, the concept of MUT reported errors is substituted with the *MUTs reported errors*, which represents the errors generated as a consequence of a fault of the considered ODC class and for which the at least one of the considered MUTs has generated at least an error notification. In addition, another node can be also introduced in the error propagation graph, i.e., the *detection node*, which allows to understand what MUTs have reported/detected the errors, as shown in Figure 3.4. The *absolute* values on an arch from a

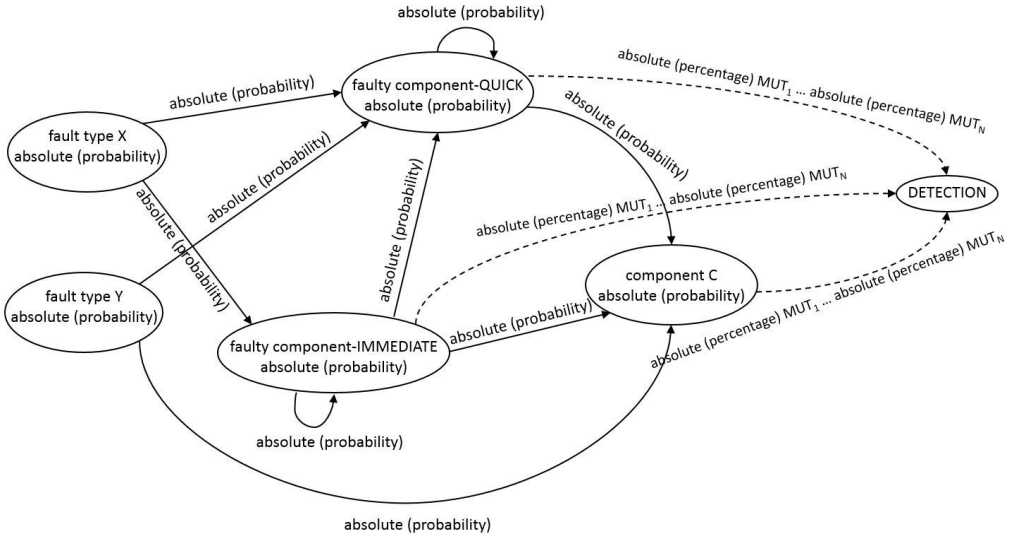

Figure 3.4: Example of directed graph with detection node.

component node to the *DETECTION* node indicates the number of MUTs reported errors, which have been reported by the component, that have been reported by each MUTs; while the *percentage* value represents the same information in percentage terms, which is obtained as ratio between the before mentioned *absolute* value and the *absolute* value related to the node component, multiplied by 100.

3.3 Evaluation Metrics

The output of each experiment consists of the (i) *type of fault* induced in the SUT, (ii) *type of failure* (if any), and (iii) the files containing the *monitoring data* generated by each MUT. Each entry in these files is labeled as *error-* or *no error-* reporting, as well as with the source component and function, and the type of reported error in the entry.

The data collected through the proposed method make it possible to evaluate the MUTs implemented by a system and to quantify their ability at reporting useful notifications in

face of failures, as well as to evaluate the error behavior exhibited by the SUT they allow to infer. The evaluation metrics are presented in the following.

3.3.1 Recall and Precision

The files containing the data generated by a MUT during the injection experiments, are attributed to four disjoint sets, i.e., *true negative* (TN), *true positive* (TP), *false negative* (FN) and *false positive* (FP). For example, the false negative set contains the files of a MUT that do not report any failure notification even if, according to the controller, a failure was actually caused by the injected fault. Similarly, the false positive (FP) set contains the files that report a failure even if no failure occurred during the experiment according to the controller.

Recall (R) and precision (P) of a MUT are computed based on the cardinality of TN, TP, FN and FP. In the context of this study, R measures the probability that a failure is reported by the MUT, i.e., $R = |TP|/(|TP| + |FN|)$; P measures the probability that a file reporting a failure corresponds to an actual failure, i.e., $P = |TP|/(|TP| + |FP|)$.

3.3.2 Failure Coverage

The overall recall of each MUT has to be broken down by failure type, i.e., *CRASH*, *SILENT*, *ERRATIC*, in each SUT in order to evaluate the *failure coverage* of the MUT. The failure coverage of a MUT with respect to a type of failure is the ratio between the number of failures of this type reported by the MUT in the considered SUT, and the total number of failures of the same type observed during the campaign in the SUT. For example, let $|FAILURES_{SUT,X}|$ be the number of failures of type X occurred in the target system

SUT, and $|FAILURES_{MUT,SUT,X}|$ be the number of failures of type X reported by the monitoring technique MUT in the target system SUT, the failure coverage of the monitoring technique MUT in the target system SUT with respect to the failure of type X is:

$FC_{MUT,SUT,X} = |FAILURES_{MUT,SUT,X}|/|FAILURES_{SUT,X}|$.

Failure coverage provides a big-picture of the failure reporting capability of the MUTs.

3.3.3 Error Determination Degree

The dataset that contains for each fault injection experiment (i) the number of errors of each type reported by a MUT of a SUT in the experiment, (ii) the type of injected fault, (iii) the ODC class the fault belongs to, and (iv) the type of failure occurred in the SUT, has to be generated from the dataset obtained during the experimental campaign. For each MUT of a SUT, a dataset of this type is created in order to evaluate the ability of the error notifications generated by a MUT of a SUT to pinpoint the fault type, the ODC class, and the failure type, related to those error notifications. It should be noted that each of these datasets considers only the experiments where the MUT has reported almost one error notifications.

The *Error Determination Degree* (EDD) metric is proposed to evaluate these abilities of a MUT. In particular, the Error Determination Degree of a MUT of a SUT with respect to the fault type, the ODC class, or the failure type, represents the ability of the error notifications generated by the MUT to pinpoint the fault type, the ODC class, or the failure type, respectively, related to those error notifications. Precisely, the EDD of a MUT of a SUT with respect to X, i.e., the fault type, the ODC class, or the failure type, is defined as

the *correct classification rate* of a classifier that has analyzed the dataset generated for the MUT of the SUT by means of a k-fold cross-validation process, and considering the number of errors of each type reported by the MUT as features and X as class to predict. The closer the value to 100.00% the higher is the ability of the MUT to suggest the X related to the error notifications the MUT has generated during the experimental campaign. Noteworthy, the classifier used for the evaluation of EDD, as well as the number of folds to consider for the cross validation, have to be the same for each MUT of each SUT in order to perform a comparison.

3.3.4 Error Propagation Reportability

The *Error Propagation Reportability* (EPR) metric is proposed to evaluate the ability of each MUT of a SUT to report the error propagation phenomena in the related SUT with respect to a specific ODC class. In particular, considering the fault related to the considered ODC class, let $|ENFC|$ be the number of MUT reported errors that have at least an error notification generated in the faulty component of the considered SUT, i.e., the sum of the *absolute* value on the arches from a fault type node to a faulty component node (either $IMMEDIATE$ or $QUICK$) on the error propagation graph, and $|MRE|$ the number of MUT Reported Errors of the considered SUT, i.e., the sum of the *absolute* values of the fault nodes on the graph, the EPR of the monitoring technique MUT of the target system SUT with respect to the ODC class X is defined as:

$$EPR_{MUT,SUT,X} = |ENFC|/|MRE|$$

The closer the value to 1 the higher is the ability of the MUT to properly report the

error propagation phenomena occurred in the SUT during the experimental campaign. Differently, a low value for this MUT suggests that there is the need of add some EDMs into the components of the considered SUT, in order to improve the ability of the MUT at reporting the propagation of errors.

It should be noted that in some cases the assumption that all the errors are expected to generate at least an error notification in the faulty component can be not valid. For example, when a component that reports the error works as a detector of the faulty component. Therefore, the proposed *EPR* metric might provide a not accurate value in this case. However, it can be successfully used to decide where to place EDMs, and, more important, as a comparative metric between MUTs. Indeed, if a MUT reports the error propagation path that is unreported by another MUT (with a lower *EPR*), this suggests that the second one actually exhibited a low ability at reporting error propagation paths.

3.3.5 Orthogonality of the MUTs

Given a SUT, the set of failures of the same type, i.e., *CRASH*, *SILENT*, *ERRATIC*, is broken down into a number of disjoint subsets, which are detailed in the following:

- NONE: the failures reported by no MUT.
- MUT_i^*: the failures reported exclusively by the MUT_i (i=1...n, where n is the total number of MUTs of the SUT, such as depicted by Figure 3.1). Let MUT_i be the set of failures reported by the MUT_i: $\text{MUT}_i^* = MUT_i - \bigcup_{k=1}^{n} MUT_k$, where k≠ i.
- $(\text{MUT}_i \cdot \text{MUT}_j)^*$: the failures reported by both MUT_i and MUT_j but not by any other MUT, i.e., $(\text{MUT}_i \cdot \text{MUT}_j)^* = (MUT_i \cap MUT_j) - \bigcup_{k=1}^{n} MUT_k$, where i,j=1...n, i≠j,

k≠ i, j.

- ALL: the failures that are reported by all the MUTs;

Figure 3.5 shows a graphical representation of the sets in the case of three MUTs. For example, MUT_i=$\text{MUT}_i^* \cup (\text{MUT}_i \cdot \text{MUT}_j)^* \cup (\text{MUT}_i \cdot \text{MUT}_k)^* \cup \text{ALL}$.

For each set the **percentage of reported failures**, i.e., the cardinality of the set divided by the total number of failures in percentage terms, are computed. These measurements provide strong insights into the effectiveness of the MUTs. For example, a large number of failures belonging to the NONE set, suggests the need for improving the detection mechanisms implemented by the SUT. Even more important, the analysis of the sets $(\text{MUT}_i \cdot \text{MUT}_j)^*$ allows understanding if the MUTs can complement at reporting failures.

3.3.6 Dissimilarity of the Monitoring Data

The above described evaluation metrics is complemented with measurements of **dissimilarity** of the data generated by the MUTs during the experiments. The dissimilarity aims to measure the degree of difference among the data produced by a MUT in response to different failures. It should be noted that, in spite of a high recall, a MUT might generate

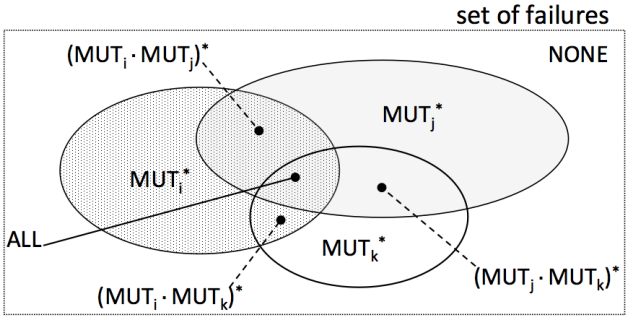

Figure 3.5: Representation of the MUTs comparison approach.

similar notifications under different failures. The data contained in few failure notifications could be more effective when compared to the data that are repeated across many notifications, which indicate generic error reporting. The dissimilarity of the monitoring data provides insights into the suitability of a MUT for manual failure inspection, since dissimilar notifications help to discriminate between the failures occurred in the SUT.

The dissimilarity through the *log.entropy* scheme is measured. **Log.entropy** is a well-established term weighting scheme in the information retrieval domain [106, 107]. Term weighting assesses features, such as *frequency*, *rarity* and *randomness* of textual information across a collection of documents (i.e., the files containing the monitoring data in this study). Log.entropy is used because it allows gaining quantitative insights into unstructured data with no specific assumptions regarding semantics and patterns they might contain. In this respect, log.entropy allows potential analysts to measure the dissimilarity of the notifications even without a deep knowledge of the MUT.

As described in Section 3.2.3, the data generated by the MUTs are saved into distinct files at the end of each experiment. Given a MUT, let D (i.e., the **documents set**) denote set of files produced by the MUT during the experiments where it reported the occurrence of a failure. Term weighting is performed by generating a **term-document** matrix beforehand. The term-document is a $|T| \times |D|$ matrix, where $|T|$ is the total number of distinct terms occurring across the collection of documents in D, and $|D|$ the total number of documents (again, log files in this study). A **term** is a sequence of characters separated by one or more whitespaces. Each element $x_{i,j}$ of the term-document matrix, with $1 \leq i \leq |T|$ and $1 \leq j \leq |D|$, represents the number of times the term i occurs in the document j.

Log.entropy quantifies the importance of a term (i) within each document, and (ii) across all the documents in D. The value of log.entropy$_j$ for a given document j is estimated as follows:

$$log.entropy_j = \sqrt{\sum_{i=1}^{|T|}(e_i \cdot \log_2(1 + x_{i,j}))^2} \tag{3.1}$$

$$e_i = 1 + \frac{1}{\log_2(|D|)} \cdot \sum_{j=1}^{|D|} p_{ij} \log_2(p_{ij}) \tag{3.2}$$

where e_i, with $0 \leq e_i \leq 1$, is computed according to Equation 3.2 (where $p_{i,j} = 1 + x_{i,j} / \sum_{j=1}^{|D|} x_{ij}$), and represents the entropy value of the term i across the documents in the set D. The occurrence of the term i in the document j, i.e., $x_{i,j}$, is scaled by $\log_2$ in the log.entropy technique.

Log.entropy is a positive numeric score computed for each document: the smaller the value of log.entropy, the higher the chance the document contains terms that are strongly repeated in D. As it can be inferred from Equations 3.2, terms that occur regularly across the document set have a small weight. For example, a term occurring the same number of times across all the documents would be weighted 0.

In the context of this study, it would be desirable that each file generated by a MUT exhibited a large log.entropy score, which denotes that the file contains very specific notifications (i.e., not frequently repeated across the documents set) for a given failure, that is, the MUT generates dissimilar data. At the other end of the spectrum, a small log.entropy score denotes generic reporting (similar data).

It should be noted that each of these metrics allows to provide insights related to the research questions addressed in this dissertation. In particular, the analysis of the *Error Propagation Reportability* and of the *Error Determination Degree* allows to understand the suitability of the monitoring techniques for the characterization of the error behavior of the considered system (*RQ1*). The analysis of the *Error Propagation Graphs* allows to infer the potential locations for EDMs and ERMs (*RQ2*). In addition, the study of all the metrics allows to characterize the failure and error reporting ability of the monitoring techniques (*RQ3).* Finally, the study of the MUTs *orthogonality*, and of the EPR and EDD obtained by combining the MUTs, allows to assess if the combination of MUTs can be useful (*RQ4*).

Chapter 4

Target Systems, Techniques, and Datasets

This chapter provides the description of the target systems, i.e., the Systems Under Test (SUTs), and the target monitoring techniques, i.e., the Monitoring techniques Under Test (MUTs), that are considered in this dissertation. The reference SUTs are two real-world critical industrial systems in the Air Traffic Control (ATC) domain, i.e., a communication middleware (SUT_1) and an arrival manager (SUT_2). Both the SUTs implemented three monitoring techniques, i.e., event logging (MUT_1), assertion checking (MUT_2) and rule-based logging (MUT_3), which represent the reference MUTs in this dissertation. Also details about the conducted experimental campaign, i.e., the workloads, faultloads, Labeling and Error Clustering processes, are provided. Finally, the obtained datasets are detailed at the end of the chapter.

4.1 The Reference SUTs

The proposed methodology has been applied on the MUTs implemented by two real-world **critical industrial systems** in the Air Traffic Control (ATC) domain, where monitoring is strongly recommended.

The SUTs are a **communication middleware** (**MW** or $\mathbf{SUT}_1$) and a standalone ATC program called **arrival manager** (**AM** or $\mathbf{SUT}_2$), described in the Sections 4.1.1 and 4.1.2, respectively. A controller has been developed in order to supervise the injection experiments for each SUT. The experimental framework consists of Virtual Machines (VMs), which run

the SUTs. The VMs are hosted on machines equipped with Intel i7-2670QM CPU, 6 GB of RAM, running a Fedora 16 OS installation. Each VM is based on the Red Hat 5 EL OS and it is configured with 4 cores, and with 2GB of RAM.

4.1.1 Communication Middleware

The SUT_1 is a communication middleware for the integration and the interoperability of heterogeneous critical systems, such as ATC and crisis management applications. For example, the middleware is used to integrate flight data processors (FDPs) and controller working positions (CWPs) in the ATC domain. The access to the middleware and its source code has been granted within the MINIMINDS academic-industrial project[1]. Figure 4.1 shows the SUT_1 and the experimental framework deployed to generate the monitoring data. The framework includes the *adapting* layers, which allow legacy applications to invoke the SUT_1 and its services. SUT_1 ensures the communication between legacy applications, according to the *publish-subscribe* paradigm; its source code consists of 796,353 lines of C code.

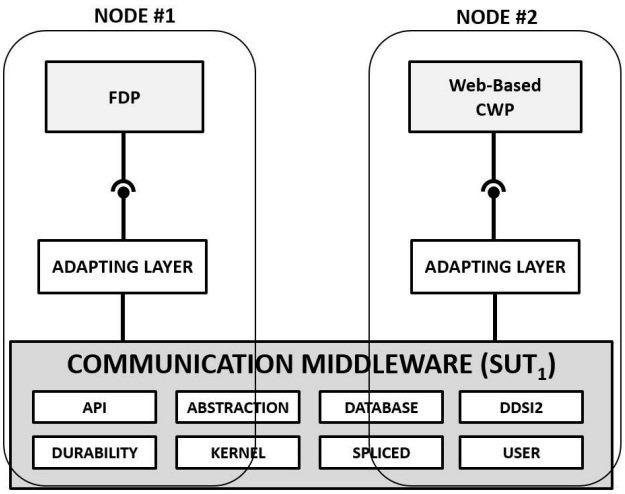

Figure 4.1: SUT_1: Experimental framework.

[1]http://www.cosmiclab.it

In details, the SUT_1 is composed by 8 components, i.e., *abstraction*, which represents the abstraction level between the middleware and the operating system, *api*, which represents the API exposed to applications, *database*, which bridges data from middleware to a DBMS and vice versa, *ddsi2*, which provides QoS-driven real-time networking based on multiple reliable multicast channels, *durability*, which provides fault-tolerant storage for both state data as well as persistent settings, *kernel*, which represents the core of the middleware, *spliced*, which is responsible for creating and initialising the database which is used to manage the middleware data, and *user*, which represents an intermediate level between the api and the kernel module.

4.1.2 Arrival Manager

The SUT_2 is the ATC arrival manager. This system manages the arrivals of flight in a given airspace. The arrival manager implements two tasks, which are represented by Figure 4.2. The **sequencing process** assists a human operator at optimizing the runway capacity; the **metering process** regulates/manages the flow of aircrafts entering the airspace. The arrival manager continuously computes an *arrival sequences list* (ASL) and times for flights based on different parameters, such as the landing rate and the spacing requirements for flights arrivals.

The system is multi-process and multi-thread, and adopts an Oracle Database[2] to store the data. The high-level architecture of the SUT_2 is shown by Figure 4.2. The access to the source code of the system, which is composed by 40,396 lines of C++ code, has been granted in the context of an industrial partnership with the developers of the system.

[2]http://www.oracle.com/

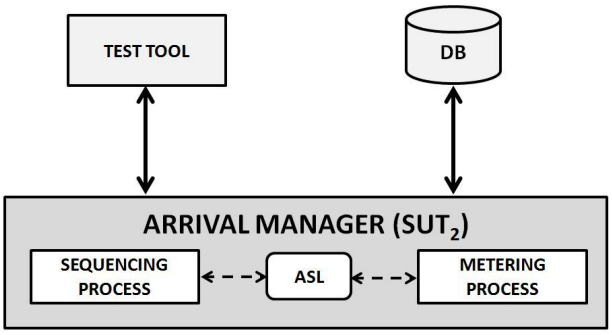

Figure 4.2: SUT_2: Experimental framework.

4.2 Workloads

Controlled testbeds have been setup to exercise both the SUTs. It is worth noting that even if the experiments are conducted in a controlled environment, adopted software emulate a real-world scenario in order to collect representative monitoring data.

The experimental framework for the SUT_1, i.e., the communication middleware, is composed by real-world ATC prototypes, which are developed by the industrial partners of the MINIMINDS project, to collect representative monitoring data. It is worth noting that adopted software (i.e., middleware, adapters and applications) emulate a production ATC installation. The experimental framework includes a number of Off-The-Shelf (OTS) components, such as the JBoss application server[3] and the Hypersonic Database[4]. The **workload** of the SUT_1 is implemented by two ATC legacy applications (shown by Figure 4.1). The applications exchange *flight data* through the communication middleware. The considered applications are (i) a FDP that generates flight data (i.e., the data that describe a flight, such as arrival and departure time, flight trajectory) and publishes the data on

[3]http://www.jboss.org
[4]http://hsqldb.org

the communication middleware, and (ii) a web-based CWP that receives the data from the middleware and displays the flight information on a web console.

The experimental framework for the SUT_2, i.e., the arrival manager, represents a real-world scenario, which has been deployed in conjunction with the developers of the system, to collect representative monitoring data. The **workload** is represented by a test suite that emulates the nominal usage of the system during the operations. The test suite consists of a sequence of test cases that are adopted by the developers to exercise the SUT_2 by means of representative requests. The test cases (that are shown by Figure 4.2, i.e., *Test Tool*) verify the behavior of the system under *insert*- and *delete*-flight orders submitted to the SUT_2.

4.3 The Reference MUTs

Both the considered SUTs natively implement **event logging** (**EL** or $\mathbf{MUT}_1$) and **assertion checking** (**AC** or $\mathbf{MUT}_2$) in order to generate events of interest and failure data during the execution. Furthermore, the code instrumentation technique proposed in [64], called **rule-based logging** (**RB** or $\mathbf{MUT}_3$), has been implemented into both the SUTs before the experimental campaign. Event logging, assertion checking, and source code instrumentation represent widely-established direct monitoring techniques in critical industrial systems.

4.3.1 Event Logging

Log-based techniques consist in collecting and analyzing event log files produced by the system, where available. Event logs contain valuable human-readable information to gain

insights into regular and anomalous system activities. Well known logging frameworks are UNIX Syslog [108] and Microsoft Event Logging [109]. The SUTs implement a variety of built-in mechanisms to generate the event logs.

The **SUT**$_1$ adopts a variety of procedures to generate the events in the log, which are reported by the leftmost column of Table 4.1. Figure 4.3[5] (*line 4*) shows a fragment of logging code that adopts the `OS_REPORT` procedure. The generation of the event is triggered by an `if` statement (line 4): in particular, a warning is reported if no name is specified for a topic.

Table 4.1: Logging procedures implemented by the SUT$_1$.

logging procedure	**occurrences**	**occurrence with error message**
`OS_REPORT`	2,126	331
`printf`	1,713	79
`snprintf`	863	82
`fprintf`	485	21
`DLRL_Except_THROW`	467	189
`sprintf`	277	1
`NN_ERROR`	67	43
`OS_DEBUG`	45	32
`YY_`	35	25
`cfg_error`	35	35
`NN_FATAL`	34	12
`yyerror`	30	30
`gapi_errorReport`	15	15
tot. occurrences	**6,192**	**895**

Total 6,192 logging instructions have been identified in the SUT$_1$, with a **density** 0.78% as shown in TABLE 4.2, i.e., 1 logging instruction every 129 lines of code. The second column of Table 4.1 reports the number of logging instructions by procedure. It is worth

[5]In the Figures showing snippets of code (sample monitoring data), `omitted` has been used in place of the lines of code (notifications), which have not been reported in the dissertation.

```
1            Communication Middleware (v_topic.c, line 480)
2  //omitted
3   if (name == NULL) {
4     OS_REPORT(OS_WARNING, "v_topicNew", 0,
5               "Topic '?' is not created. No name specified (NULL).");
6     return null;
7  }
8  //omitted
```

Figure 4.3: Example of error logging (SUT_1).

```
1            Communication Middleware (spliced.c, line 614)
2  //omitted
3  if (createResult == os_resultSuccess) {
4       os_sharedMemoryRegisterUserProcess(splicedGetDomainName(), info->procId);
5       OS_REPORT_2(OS_INFO, OSRPT_CNTXT_SPLICED,
6       0, "Started service %s with args %s", info-¿name, args);
7  }
8  //omitted
```

Figure 4.4: Example of informational logging (SUT_1).

noting that event logging is also used to report informational events, such as the code snippet in Figure 4.4 that notifies the start of a service of the middleware. Total 895 out of 6,192 logging instructions contain an error message in the SUT_1. The rightmost column of Table 4.1 shows the breakdown of the logging instructions containing an error message by procedure.

Similar considerations apply to the **SUT_2**. Again, it has been observed that the event logging mechanism consists of a variety of procedures. The distribution of the logging procedures is shown in Table 4.3, while Figure 4.5 reports an example of the most recurring

Table 4.2: MUTs density for each case study. EL-error* denotes the percentage of logging instructions containing an error message out of the total of logging instructions.

	SUT_1 (796,353 LOC)	**SUT_2 (40,396 LOC)**
EL	0.78%	1.98%
*EL-error**	*14.45%*	*11.39%*
AC	0.99%	0.18%
RB	0.36%	8.59%

Table 4.3: Logging procedures implemented by the SUT_2.

logging procedure	occurrences	occurrence with error message
DIAG	493	69
Log	115	2
sprintf	71	14
Diagnostic	66	3
TRACE	39	0
printf	15	3
tot. occurrences	**799**	**91**

logging procedure. The SUT_2 contains 799 logging instructions, with a density of 1.98%, i.e., 1 logging instruction every 51 lines of code. Total 91 logging instruction contain an error message, such as indicated by the rightmost column of Table 4.3.

```
1           Arrival Manager (CSeqUtl.cpp, line 306)
2  //omitted
3  if ( 0 == pAerodromes ){
4       sDiag.Format("searchForAerodrome(), aerodrome[%s] : DB CONNECT ERROR ", pAdCode );
5       DIAG(sDiag);
6       //omitted
```

Figure 4.5: Example of logging instruction (SUT_2).

4.3.2 Assertion Checking

Assertion checking is based on the use of **assertions**, i.e., code statements that check invariant properties of a given program and produce an alert if one of the properties is violated at runtime [61]. Examples are *range checking*, where the assertions perform boundary checks on the values of program variables [62], and *specification of function interfaces* [61], where the assertions perform checks on preconditions and postconditions of functions. Most of the assertions implemented by the SUTs belong to the *specification of function interfaces* class.

```
1           Communication Middleware (v_writer.c, line 1670)
2 //omitted
3 v_writer v_writerNew(v_publisher p, const c_char *name, v_topic topic,
4                       v_writerQos qos, c_bool enable){
5     //omitted
6     assert(p != NULL);
7     assert(C_TYPECHECK(p, v_publisher));
8     assert(C_TYPECHECK(topic, v_topic));
9     //omitted
```

Figure 4.6: Example of assert instructions (SUT_1).

```
1           Arrival Manager (STUB_HMI_MTCD.cpp, line 582)
2 //omitted
3 void HMI_MTCD::set_ThdInfo(CDatabase* pUserDB, tThdInfo *pThdInfo)
4 {
5     assert(pUserDB != 0);
6     assert(pThdInfo != 0);
7     //omitted
```

Figure 4.7: Example of assert instructions (SUT_2).

Examples of assertions implemented by the SUT_1 are shown by Figure 4.6. The `assert` instructions (lines 6-8) check the value/type of two variables: they evaluate to TRUE when the internal state of the monitored program is correct, FALSE otherwise. The `assert` statement at line 6 generates a warning if the `p` is `NULL`; lines 7-8 generate an alert if `C_TYPECHECK` returns FALSE.

Examples of assertions implemented by the SUT_2 are shown by the snippets in Figure 4.7. The `assert` instructions (lines 5-6) check the values of two program variables: they evaluate to TRUE when the internal state of the monitored program is correct, FALSE otherwise. For example, the considered `assert` statement generate a warning if the `pUserDB` and `pThdInfo` variables value is equal to 0.

Both the examples taken from SUT_1 and SUT_2 verify a precondition of a function, i.e., `v_writerNew` and `set_ThdInfo`, respectively. The source code of the SUT_1 contains around 7,954 `assert` instructions, with a density of 0.99%, i.e., 1 assertion every 101 lines of code. The source code of the SUT_2 contains 72 `assert` instructions, with a density of 0.18%, i.e.,

1 assertion every 561 lines of code.

4.3.3 Source Code Instrumentation

Source code instrumentation is based on the insertion of specific instructions into the code of a software system with the aim of monitoring its behavior. The technique proposed in [64], i.e., the **rule-based logging**, has been used to instrument both SUT_1 and SUT_2. Rule-based logging consists of a number of *rules*, which drive the placement of the monitoring instructions. For example, the *Service STart* (SST) and *Service ENd* (SEN) rules aim to trace the start and the end of a function; similarly, the *Interaction STart* (IST) and *Interaction ENd* (IEN) rules suggest how to trace the start and the end of a function call.

For example, Figure 4.8 shows a function that has been instrumented with the rule-based logging technique. The function, i.e., `CASDI_t`, belongs to the SUT_2. It can be noted that the SST (line 8) and SEN (line 14) instruction have been introduced to trace the start and the end of the function, while IST (line 10) and IEN (line 12) are placed before and after the invocation of `SendOrdLst`.

The instruction `logAnEvent` consists of three fields that indicate (i) the event introduced

```
1            Arrival Manager (CAsdIt.cpp, line 37)
2  //omitted
3  CASDI_t::CASDI_t(CASFDoc* pDoc) :
4                   m_bMessageInQueue(false), m_nDiagnosticLevel(0),
5                   m_nTimeAssignType(0), m_nTime(0),
6                   m_nYear(0), m_nMounth(0), m_nDay(0)
7  {
8      logAnEvent( SST, CASDI_t, CAsdIt.cpp);
9      //omitted
10     logAnEvent( IST, SendOrdLst, CAsdIt.cpp );
11     SendOrdLst();
12     logAnEvent( IEN, SendOrdLst, CAsdIt.cpp );
13     //omitted
14     logAnEvent( SEN, CASDI_t, CAsdIt.cpp);
15 }
16 //omitted
```

Figure 4.8: Example of rule-based instructions (SUT_2).

in the source code, (ii) the monitored function and (iii) the monitored module. The events generated by the `logAnEvent` instruction are collected by a dedicated monitoring framework, named LogBus [64], which generates error events. For instance, if the SEN (IEN) event is not generated within an expected time since the observation of the corresponding SST (IST), a Service Error (Interaction Error) is generated by the LogBus: the error indicates that a function (or a call to a function) failed to terminate within the expected timeout. The SUT_1 contains 2,895 rule-based logging instructions, with a density of 0.36%, i.e., 1 instruction every 275 lines of code, while the SUT_2 contains around 3,470 rule-based logging instructions, with a density of 8.59%, i.e., 1 instruction every 12 lines of code.

Table 4.2 summarizes the **density** of each MUT in both the SUTs. For example, the density of the logging instructions in the SUT_1 is 0.78%, i.e., $(6,192/796,353) \cdot 100$. The density of assertions and rule-based logging is rather different across the SUTs. It should be noted that for SUT_1 only the kernel component implements the rule-based logging.

4.4 Falutloads

The datasets considered in this dissertation have been obtained by running each SUT under a representative faultload. The faultloads have been generated by the SAFE[6] tool [110]. The tool parses the Abstract Syntax Tree (AST) of the SUT (generated by the compiler frontend) and automatically searches for all the locations in the source code where each fault type reported by Table 3.1 can be injected. As a result, the greater the size and complexity

[6] http://www.critiware.com/safe.html

of the source code, the larger the number of possible injectable faults that are inferred by the tool. The **injection** is accomplished by means of changes of the source code, which emulate the programming mistake. For each fault, SAFE generates a *.patch* file containing the lines of code that will be subtracted and added to the SUT in order to emulate the fault. It should be noted that the faultload has been generated by means of changes of the source code. The rationale behind this choice is the availability of the source code of the SUTs that have been considered in this dissertation. However, the same goal can be reached by using a different tool or apporach, such as *binay-level* [111] and *interface-level* [112] fault injection that allow fault injection also if the source code of the SUTs is not available.

The faultload of the $\mathbf{SUT}_1$ is composed by 12,733 faults. Overall 3,159 faults caused the failure of the SUT_1.

Table 4.4 presents the total number of failures by fault and failure type. For example, the value 185 reported by the cell (*MFC, CRASH*) indicates that 185 algorithm faults, i.e., *ALG*, of type missing function call, i.e., *MFC*, caused a *CRASH* failure. Differently, the value 1,482 reported by the cell (*total, CRASH*) of the *ALG* ODC class indicates that 1,482 algorithm faults caused a *CRASH* failure. A closer look into the data collected by the controller of the experiments revealed that the causes of *ERRATIC* failures can be divided as follows: abnormal termination of one (more) internal service thread(s) of the SUT_1 (39%), misconfigurations and bad setting of the quality of service parameters (18%), inability of the SUT_1 at properly executing all the publish requests of the FDP (14%), interaction issues between the *publishing* and the *core* module of the SUT_1 (12%), data delivery issues (5%), other minor causes (12%). It is worth noting that the dataset generated in the SUT_1 has

Table 4.4: Failures by fault and failure type (SUT_1).

		failure			
fault					
ODC	**type**	*CRASH*	*SILENT*	*ERRATIC*	***tot. faults***
ALG	MFC	185	56	41	282
	MIFS	54	2	13	69
	MIEB	44	6	12	62
	MLPA	1,199	94	156	1,449
total		*1,482*	*158*	*222*	*1,862*
ASG	MVIV	4	0	0	4
	MVAV	23	1	2	26
	MVAE	627	34	57	718
	WVAV	22	0	4	26
total		*676*	*35*	*63*	*774*
CHK	MIA	38	1	9	48
	MLC	4	0	2	6
total		*42*	*1*	*11*	*54*
INT	WPFV	389	9	20	418
	WAEP	50	1	0	51
total		*439*	*10*	*20*	*469*
tot. failures		*2,639*	*204*	*316*	***3,159***

been made publicly available[7].

A faultload of total 6,597 faults has been injected in the $\mathbf{SUT}_2$. Overall 685 injections caused a failure of the SUT_2. Table 4.5 divides the set of failures by fault and failure type. For example, again the value 6 reported in the cell (*MFC, CRASH*) indicates that 6 algorithm faults of type missing function call caused a *CRASH* failure. Differently, the value 68 reported by the cell (*total, CRASH*) of the *ALG* ODC class indicates that 68 algorithm faults caused a *CRASH* failure. *ERRATIC* failures are mainly caused by corruptions of timestamps at determining the arrival time of flights (49%) and database exceptions (45%); other minor causes account for around 6% of *ERRATIC* failures.

[7] http://www.mobilab.unina.it/Datasets.htm

Table 4.5: Failures by fault and failure type (SUT_2).

		failure			
fault					
ODC	**type**	*CRASH*	*SILENT*	*ERRATIC*	***tot. faults***
ALG	MFC	6	6	12	24
	MIFS	0	0	0	0
	MIEB	0	1	0	1
	MLPA	62	199	258	519
total		*68*	*206*	*270*	*544*
ASG	MVIV	0	1	0	1
	MVAV	0	0	0	0
	MVAE	11	63	48	122
	WVAV	0	0	0	0
total		*11*	*64*	*48*	*123*
CHK	MIA	1	3	0	4
	MLC	0	0	0	0
total		*1*	*3*	*0*	*4*
INT	WPFV	2	4	8	14
	WAEP	0	0	0	0
total		*2*	*4*	*8*	*14*
tot. failures		*82*	*277*	*326*	***685***

4.5 Labeling and Error Clustering

The files that contains the data generated by the MUTs during the fault injection experiments have been analyzed through a **post-mortem** inspection in order to conduct the labeling and error clustering of the data. Labeling aims to label each notification generated by a MUT into *no error*-reporting, i.e., the notification does not report an error, and *error*-reporting, i.e., the notification reports an error; the labeled data allow the evaluation of the metrics provided by the proposed methodology that are related to the failure coverage of MUTs. Differently, Error Clustering aims to classify the types of error reported by each MUT, and to infer the error model of the MUT in the related SUT; the clustered error data allow obtaining insights about the error behavior exposed by a SUT according to the

considered MUT.

4.5.1 Labeling

The files that contains the data generated by the MUTs after each fault injection experiment have been labeled into *no error*- and *error*-reporting through a post-mortem inspection. The files have been scrutinized with the aim of pinpointing one or more error notifications generated by the MUTs. The presence of error notifications in a given file, possibly suggests that a failure occurred during the execution of the experiment according to the MUT that generated the file. Figure 4.9 and Figure 4.10 report error notification generated by means of event logging and assertion checking, respectively.

In order to label the event logs, the procedures that have been commonly used by several works in the area, such as [113, 114], have been adopted. First, the content of the event logs collected across all the experiments has been **de-parameterized**, i.e., *variable fields*, such as IP and memory addresses, file system paths and timestamps, has been replaced with a general token (e.g., `IP_ADDRESS`, `PATH`). For example the entries

```
sshd[7654]:  Accepted publickey for rob from 192.168.0.184
```

```
sshd[4154]:  Accepted publickey for lisa from 210.140.12.6
```

share the same information structure, referred here as **statement**, once the variable fields have been replaced with the generalized tokens:

```
sshd[PID]: Accepted publickey for USER from IP_ADDRESS
```

This procedure identifies a small number of statements because most of the entries in the event logs differ because of the variable fields. A manually categorization of each statement

as error and no-error reporting has been performed by inspecting the source code of the SUTs, by analyzing available documentation, and trough direct communication with the developers. A regular expression, which catches only the statements that have been flagged as *error-reporting* after the manual categorization, is applied to the event logs in order to identify the ones reporting an error.

The post-mortem labeling of the files containing the notifications generated by means of assertion checking and rule-based logging required a smaller effort because these techniques

```
1          Communication Middleware
2  Report       : ERROR
3  Date         : Tue Dec 10 18:34:53 2013
4  Description  : Type mismatch: object type...
5    ...is v_cfEle but v_cfAttr was expected
6  Node         : localhost.localdomain
7  Process      : Receiver <26873>
8  Thread       : ddsdeamon 2b05e366c940
9  Internals    : //Database::c_checkType/c_misc.c
10 ================================================
11 Report       : ERROR
12 Date         : Tue Dec 10 18:34:53 2013
13 Description  : Type mismatch: object type...
14  ...is v_cfEle but v_cfAttr was expected
15 Node         : localhost.localdomain
16 Process      : Sender <26874>
17 Thread       : ddsdeamon 2b23441ae940
18 Internals    : //Database::c_checkType/c_misc.c
19
20              Arrival Manager
21 13 23:57:32.219 [ELGT-8] CEligThdHandler:...
22  ...Invalid ETO in points: H=24, M=00, S=00
23 13 23:57:32.219 [ELGT-8] ACDPM_getEtoTime:...
24  ...Invalid ETO in points: H=24, M=00, S=00
25 13 23:57:32.220 [ELGT-8] CEligThdHandler:...
26  ...Invalid ETO in points: H=24, M=00, S=00
27 13 23:57:32.220 [ELGT-8] CEligThdHandler:...
28  ...Invalid ETO in points: H=24, M=00, S=00
```

Figure 4.9: Example of error notifications in the event logs.

```
1          Communication Middleware
2 //code/qentc:1212: proxywriteraddconn:...
3 ...Assertion 'pwr->ctopic != (0)' failed
4
5              Arrival Manager
6 19:50:47.237 TST: WARNING EXCEPTION,...
7 ...Assertion Failed.
8 (  26) B729EFE0: WARN/Assert
9 Module     Procedure    Line Instruction
10 CDB_AM_UKR AM_TST_CMD   26    B729EFE0
```

Figure 4.10: Example of assertions.

are inherently conceived for error reporting. In this respect, all the files containing the notifications generated by the `assert` instruction or by the LogBus monitoring framework have been labeled as *error-reporting*.

It should be noted that the labeling step has been conducted in this study by a post-mortem manual inspection of the obtained data. However, practitioners might apply different approaches in order to reach the same goal.

4.5.2 Error Clustering

The error notifications generated from the reference MUTs during the experimental campaign have been further analyzed in order to infer the error model they consider. For each MUT, the files that contains the error notifications have been scrutinized with the aim of grouping together the error notifications that have common characteristics, such as same message, same semantic, same source module/file, etc. Groups containing error notifications with similar characteristics, here named *clusters*, for a given MUT possibly represent the types of error that the MUT is able to report into the target system, i.e., its error

```
1                          Error notification #1
2 Description: Operation failed, couldn't resolve
3              type "kernelModule v_builtin"
4 Internals: //kernel::v_builtinNew/v_builtin.c
5
6                          Error notification #2
7 Description: Field (null) not found in type d_deleteData_s
8 Internals: //kernel::v_filterNew:/v_filter.c
```

Figure 4.11: Example of error notifications in the event logs of SUT_1

model in the target system. For example, Figure 4.11 reports two different error notifications generated by event logging in the SUT_1 in different fault injection experiments. It should be observed that, despite the error notifications contain different messages, they have quite similar semantic since both refer to a data type problem. Therefore, they can be potentially grouped together in a cluster that represent data type errors. Noteworthy, the type of characteristics to consider in order to create clusters of error notifications of a MUT changes based on the nature of the MUT. For example, the event logging often generates notifications with a very high semantic level, as seen in Figure 4.11. Therefore, the semantic of the notifications can be a valid feature to cluster error notifications of event logging.

Differently, assertion checking, which is based on assertions that check invariant properties of a given program and produce an alert if one of the properties is violated at runtime, generates notifications where only the violated property and the location of the assertion are reported, as seen in Figure 4.10. Therefore, for this MUT the type of violated properties can be a potential feature to group together the notifications. For example, Figure 4.12 reports two different notifications generated by assertion checking in the SUT_1 in different

```
1                          Error notification #1
2 //code/v_networkQueue.c:414:  v_networkQueueTakeFirst...
  ...Assertion 'sample != NULL' failed.
3
4                          Error notification #2
5 //code/v_groupInstance.c:1140:  v_groupInstanceInsert...
  ...Assertion 'message != NULL' failed.
```

Figure 4.12: Example of assertions in SUT_1

fault injection experiments. It should be observed that, despite the notifications have different content, both indicate that the checked variable contains a `NULL` value. Therefore, they can be potentially grouped together in a cluster that represent errors due to `NULL` value.

Finally, the rule-based logging generates error notifications with a very low verbosity level, where only the type of error, e.g., SER, IER, the source function and module are reported. All the three reported information represent potential features for clustering. However, the source module has been considered as feature to generate clusters of error notifications of this MUT since it avoids obtaining both large number of clusters with few occurrences (in the case of source function) and small number of clusters with many occurrences (in the case of error type). For example, Figure 4.13 shows two different error notifications generated by rule-based logging in the SUT_1 in different fault injection experiments. The reported error notifications indicate two different errors raised in two different functions, i.e., `v_kernelNew` and `v_builtinNew`, which belong to the same module of the kernel, i.e., `kernel`, which represents the core module of the kernel component of the SUT_1 (as a reminder, only the kernel of the SUT_1 implements the rule-based logging; therefore the reported modules are the ones that compose the kernel itself, i.e., *Writer*, *DataReader*,

Subscriber, *Publisher*, *Network*, *Topic*, *Group*, *Kernel*, *Message*). Therefore, they can be potentially grouped together in a cluster that represent errors generated in the core component of the kernel of the SUT_1.

Based on the above considerations, the clusters of error notifications have been generated, and their error model in the related SUT has been inferred. It should be noted that the data clustering approach has been applied only to the MUTs implemented by the SUT_1, which is the most complex one (it is a distributed system deployed on two different nodes and it is composed by 796,353 lines of code for each node; differently, the SUT_2 is not a distributed system, and it is composed by a lower number of lines of code, i.e., 40,396). In addition, a high number of experiments in SUT_1 lead to at least one error notification generated by one of the MUTs (2,748 for SUT_1 against the 957 for SUT_2).

The semantic of the message, the type of check and the source kernel module of the notifications are used as feature to group together the error notifications for MUT_1, MUT_2 and MUT_3, respectively. Regular expressions, which catch the error notifications that belongs to each considered cluster, has been applied to the error notifications of each MUT in SUT_1 in order to place each one in the right cluster.

Table 4.6 contains the error model considered by each MUT in the SUT_1, i.e., the set

```
1                          Error notification #1
2 SER v_kernelNew              kernel
3
4                          Error notification #2
5 IER v_builtinNew             kernel
```

Figure 4.13: Example of error notifications of rule-based logging in SUT_1

Table 4.6: Error models considered by MUTs in SUT_1.

cluster	**errors description**	**example of notification**
event logging		
e1-EL	*Memory errors*	`Failed to allocate cache`
e2-EL	*Quality of Service errors*	`Writer not created inconsistent qos`
e3-EL	*Unexpected result errors*	`Operation returned ... but expected ...`
e4-EL	*Data type errors*	`Operation failed, couldn't resolve type ...`
e5-EL	*Main daemon errors*	`Could not claim the DDSdaemon!`
e6-EL	*Consistency errors*	`Illegal contained object`
e7-EL	*Topic errors*	`Failed to produce built-in ... topic`
e8-EL	*Mutex errors*	`Operation failed mutex ... Invalid argument`
e9-EL	*Kernel entities errors*	`Create kernel entity failed`
e10-EL	*Timeout/liveliness errors*	`A fatal error was detected when trying to... ...register the daemon liveliness hbCheck ...`
e11-EL	*Threads progress errors*	`Thread ... failed to make progress`
e12-EL	*Configuration errors*	`Could not initialise configuration`
e13-EL	*Other errors*	`Maximum number of network queues exceeded ...` `Expression ... is not a valid ... statement`
assertion checking		
e1-AC	*Data type errors*	`'(w == c_checkType(w,"v_writer"))' failed`
e2-AC	*Unexpected value errors*	`'c_refCount(found) == 4' failed.`
e3-AC	*Forced assertion execution*	`'(0)' failed.`
e4-AC	`NULL` *value errors*	`'message != NULL' failed`
e5-AC	*Data size errors*	`'c_aSize(msgKList) == c_aSize(instKList)' failed`
rule-base logging		
e1-RB	*Writer module errors*	`IER v_pubGetQosRef writer`
e2-RB	*DataReader module errors*	`IER v_subAddReader datareader`
e3-RB	*Subscriber module errors*	`SER v_subNew subscriber`
e4-RB	*Publisher module errors*	`SER v_pubNew publisher`
e5-RB	*Network module errors*	`IER v_grpNotifyAwareness network`
e6-RB	*Topic module errors*	`IER v_cfEleXPath topic`
e7-RB	*Group module errors*	`SER regInstance group`
e8-RB	*Kernel module errors*	`IER c_free kernel`
e9-RB	*Message module errors*	`SER v_msgQos_new message`

of obtained clusters for each MUT, and an example of error notification included in each cluster.

During the error data clustering process, each error notifications generated by each MUT has been also labeled with its source function and component, i.e., the function that generates the error notification and the component of the SUT_1 the function belong

to, respectively. Indeed, the error notifications generated by each MUT in SUT_1 allow obtaining these information. For example, the `Internals` field in the error notifications of the event logging provides both information, as can be seen in Figure 4.11, where the source function of the first error notification is `v_builtinNew`, while its source component is `kernel`. Similarly, both assertions and rule-base logging provide the source function and component of the error notification by design. For example, in Figure 4.12 the source function and component of the first assert notification are `v_networkQueueTakeFirst` and `v_networkQueue.c`, which is one of the source file of the kernel component, respectively; while in Figure 4.13 the source function and component of the first rule-based logging notification are `v_kernelNew` and `kernel`, respectively.

It should be noted that practitioners might apply different approaches respect than the one described here in order to reach the same goals.

4.5.3 Discussion on the Error Models

Error clustering allowed inferring the error model considered by each MUT in the SUT_1. According to the obtained clusters, each MUT considers a rather different error model in SUT_1, as it can be observed by Table 4.6. More in details, the main differences are:

- MUT_1 mainly considers error types that are related to application logic of the target systems, such as *e2-EL* and *e7-EL* that represent errors on the management of the *Quality of Service* and of the *Topics*, respectively.

- MUT_2 considers error types that are less related to application logic respect than the ones considered by MUT_1. Instead, they are related to the properties that the target

system has to satisfy at runtime, such as *e2-AC* and *e5-AC* that represent errors related to an unexpected value for a system variable/function result, e.g., when the value of a system variable/function result does not satisfy a constraint, and to an unexpected size of a system variable/function result, respectively. Exceptions are the errors included in *e3-AC*, which are explicitly raised by the developers by verifying a property that is always unsatisfied, e.g., when the control flow enters in a known error path.

- The inferred error model for the MUT_3 considers error types that are related to the module of the kernel that reports the error, e.g., *e1-RB* and *e2-RB*, which represent the errors raised by the *Writer* and *DataReader* kernel module, respectively. Indeed, rule-based logging, differently from the other MUTs, aims to detect errors reflecting the adopted system structure in terms of modules, functions/services, and interactions.

4.6 Obtained Datasets

The overall results of the campaigns are summarized from Table 4.7 to Table 4.16.

Table 4.7 and Table 4.8 report the **absolute number** and **percentage (i.e., RF%)** of failures reported by each MUT by ODC fault and failure type in SUT_1 and SUT_2, respectively. For example, the value 210 reported by the cell (*EL, Absolute*) - *CRASH* column and *ALG* row - in Table 4.7 indicates that 210 out of 1,482 *CRASH* failures caused by *ALG* faults (the number of failures by fault type is shown in Table 4.4) were detected by the event logs, i.e., *EL*, of the middleware (SUT_1). On the other hand, the value 38 reported by the cells (*EL, Absolute*) - *CRASH* column and *ALG* row - in Table 4.8, indicates

that the event logs generated by the arrival manager (SUT_2), reported 38 *CRASH* failures caused by *ALG* faults out of 68 (again, the number of failures by fault type for the SUT_2 is shown in Table 4.5). In percentage terms, the event logs generated by the SUT_1 and the SUT_2 reported 14.17% (i.e., $(210/1,482) \cdot 100$) and 55.88% (i.e., $(38/68) \cdot 100$) of *CRASH* failures caused by *ALG* faults, respectively: these values are reported by the *RF%* column of Table 4.7 and Table 4.8, respectively. The rightmost columns of Table 4.7 and Table 4.8 report the total number of activated faults detected by each MUT in the SUT_1 and SUT_2, respectively. Similarly, the bottom rows of Table 4.7 and Table 4.8 aggregate the number and the percentage of reported failures by type.

Table 4.9 and Table 4.10 report the **absolute number** of errors reported by each MUT by fault (*ODC* class and type, according to Table 3.1) and failure type in SUT_1, respectively.

Table 4.7: Absolute number (Absolute) and percentage of reported failures (RF %) by fault and failure type for each MUT of SUT_1.

		failure							
fault	**MUT**	*CRASH*		*SILENT*		*ERRATIC*		***total faults***	
		Absolute	*RF %*	*Absolute*	*RF %*	*Absolute*	*RF %*	***Absolute***	***RF %***
ALG	*EL*	210	14.17	93	58.86	45	20.27	348	18.69
	AC	916	61.80	0	0.00	0	0.00	916	49.19
	RB	987	66.60	115	72.79	16	7.21	1,118	60.04
ASG	*EL*	130	19.23	30	85.71	16	25.40	176	22.74
	AC	396	58.58	0	0.00	0	0.00	396	51.16
	RB	474	70.12	22	62.86	2	3.17	498	64.34
CHK	*EL*	9	21.43	1	100.00	1	9.09	11	20.37
	AC	29	69.05	0	0.00	0	0.00	29	53.70
	RB	20	47.62	0	0.00	0	0.00	20	37.04
INT	*EL*	116	26.42	9	90.00	4	20.00	129	27.51
	AC	263	59.91	0	0.00	0	0.00	263	56.08
	RB	313	71.30	6	60.00	0	0.00	319	68.02
tot. failures	*EL*	465	17.62	133	65.20	66	20.89	**664**	**21.02**
	AC	1,604	60.78	0	0.00	0	0.00	**1,604**	**50.78**
	RB	1,794	67.98	143	70.10	18	5.70	**1,955**	**61.89**

Table 4.8: Absolute number (Absolute) and percentage of reported failures (RF %) by fault and failure type for each MUT of SUT_2.

		failure							
fault	**MUT**	*CRASH*		*SILENT*		*ERRATIC*		***total faults***	
		Absolute	*RF %*	*Absolute*	*RF %*	*Absolute*	*RF %*	***Absolute***	***RF %***
ALG	*EL*	38	55.88	5	2.43	24	8.89	67	12.32
	AC	19	27.90	0	0.00	21	7.78	40	7.35
	RB	23	41.18	199	96.60	169	62.59	396	72.79
ASG	*EL*	1	9.09	1	1.56	1	2.08	3	2.44
	AC	5	45.45	0	0.00	0	0.00	5	4.07
	RB	10	90.91	59	92.19	47	97.92	116	94.31
CHK	*EL*	1	100.00	3	100.00	0	0.00	4	100.00
	AC	0	0.00	0	0.00	0	0.00	0	0.00
	RB	0	0.00	0	0.00	0	0.00	0	0.00
INT	*EL*	2	100.00	1	25.00	4	50.00	7	50.00
	AC	0	0.00	0	0.00	0	0.00	0	0.00
	RB	0	0.00	0	0.00	6	75.00	6	42.86
tot. failures	*EL*	42	51.22	10	3.61	29	8.90	**81**	**11.82**
	AC	24	29.27	0	0.00	21	6.44	**45**	**6.57**
	RB	38	46.34	258	93.14	222	68.10	**518**	**75.62**

For example, the value 68 reported by the cell (*MFC, EL*) - *ALG* row - in Table 4.9 indicates that 69 algorithm faults, i.e., *ALG*, generated by a *missing function call*, i.e., *MFC*, have lead to an error in the SUT_1, which has been reported by the event logging, i.e., *EL*; while the value 369 reported by the cell (*total, EL*) indicates that 369 algorithm faults have lead to an error in the SUT_1, which has been reported by the event logging. On the other hand, the value 1,604 reported by the cells (*CRASH, EL*) in Table 4.10, indicates that the assertions generated by the communication middleware (SUT_1) reported 465 errors that lead to a CRASH failures in the SUT. The bottom rows of Table 4.9 and Table 4.10 aggregate the number of reported errors for each MUT.

Table 4.11, Table 4.12 and Table 4.13 report the **absolute number (i.e., *Abs*)** and **percentage (i.e., %)** of errors reported by MUT_1, MUT_2 and MUT_3, respectively, by

Table 4.9: Absolute number of reported errors by fault type for each MUT of SUT_1.

		error		
fault				
ODC	**type**	*EL*	*AC*	*RB*
ALG	*MFC*	69	160	140
	MIEB	26	29	31
	MIFS	12	44	34
	MLPA	262	684	926
total		369	917	1,131
ASG	*MVAE*	175	364	468
	MVAV	8	14	15
	MVIV	0	3	4
	WVAV	12	16	17
total		195	397	504
CHK	*MIA*	15	27	18
	MLC	2	2	2
total		17	29	20
INT	*WAEP*	11	36	38
	WPFV	122	227	282
total		133	263	320
tot. errors		**714**	**1,606**	**1,975**

Table 4.10: Absolute number of reported errors by failure type for each MUT of SUT_1.

	error		
failure	*EL*	*AC*	*RB*
CRASH	465	1,604	1,794
SILENT	133	0	143
ERRATIC	66	0	18
NO_FAILURE	50	2	20
tot. errors	**714**	**1,606**	**1,975**

fault and error type, i.e., the error clusters identified for each MUT during the error data clustering phase (as a reminder, the error data clustering phase has been conducted only on the SUT_1). For example, the value 2 reported by the cell (*MFC, Abs*) - *e1-EL* column - in Table 4.11 indicates that 2 out of 69 errors detected by the event logs of the middleware (SUT_1), and caused by *ALG* faults of type *MFC* (the number of the detected errors by fault

Table 4.11: Absolute number (Abs) and percentage of reported errors (%) by fault and error type for MUT_1 of SUT_1.

	error																									
fault	*e1-EL*		*e2-EL*		*e3-EL*		*e4-EL*		*e5-EL*		*e6-EL*		*e7-EL*		*e8-EL*		*e9-EL*		*e10-EL*		*e11-EL*		*e12-EL*		*e13-EL*	
	Abs	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*
MFC	2	2.90	1	1.45	1	1.45	10	14.49	27	39.13	1	1.45	2	2.90	13	18.84	26	37.68	0	0.00	11	15.94	1	1.45	0	0.00
MIEB	16	61.54	1	3.85	3	11.54	2	7.69	3	11.54	0	0.00	1	3.85	0	0.00	0	0.00	2	7.69	0	0.00	0	0.00	1	3.85
MIFS	1	8.33	1	8.33	2	16.67	5	41.67	1	8.33	0	0.00	0	0.00	0	0.00	1	8.33	1	8.33	0	0.00	1	8.33	2	16.67
MLPA	28	10.69	18	6.87	30	11.45	83	31.68	51	19.47	9	3.44	7	2.67	9	3.44	51	19.47	7	2.67	28	10.69	4	1.53	16	6.11
total ALG	47	12.24	21	5.69	36	9.76	100	27.10	82	22.22	10	2.71	10	2.71	22	5,69	78	21.14	10	2.71	39	10.57	6	1.63	19	5.15
MVAE	22	12.57	2	1.14	32	18.29	65	37.14	35	20.00	8	4.57	6	3.43	0	0.00	21	12.00	7	4.00	15	8.57	3	1.71	10	5.71
MVAV	1	12.50	5	62.50	3	37.50	2	25.00	0	0.00	3	37.50	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
MVIV	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
WVAV	1	8.33	6	50.00	4	33.33	1	8.33	1	8.33	3	25.00	0	0.00	0	0.00	1	8.33	0	0.00	2	16.67	0	0.00	1	8.33
total ASG	24	12.31	13	6.67	39	20.00	68	34.87	36	18.46	14	7.18	6	3.08	0	0.00	22	11.28	7	3.59	17	8.72	3	1.54	11	5.64
MIA	0	0.00	7	46.67	5	33.33	1	6.67	1	6.67	1	6.67	1	6.67	1	6.67	1	6.67	0	0.00	2	13.33	0	0.00	0	0.00
MLC	0	0.00	0	0.00	1	50.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	2	100.00
total CHK	0	0.00	7	41.18	6	35.29	1	5.88	1	5.88	1	5.88	1	5.88	1	5.88	1	5.88	0	0.00	2	11.76	0	0.00	2	11.76
WAEP	0	0.00	0	0.00	0	0.00	9	81.82	2	18.18	0	0.00	0	0.00	0	0.00	2	18.18	0	0.00	1	9.09	0	0.00	0	0.00
WPFV	2	1.64	12	9.84	41	33.61	46	37.70	12	9.84	12	9.84	15	12.30	1	0.82	22	18.03	3	2.46	0	0.00	1	0.82	26	21.31
total INT	2	1.50	12	9.02	41	30.83	55	41.35	14	10.53	12	9.02	15	11.28	1	0.75	24	18.05	3	2.26	1	0.75	1	0.75	26	19.55
tot. errors	73	10.22	53	7.42	122	17.09	224	31.37	133	18.63	37	5.18	32	4.48	24	3.36	125	17.51	20	2.80	59	8.26	10	1.40	58	8.12

Table 4.12: Absolute number (Abs) and percentage of reported errors (%) by fault and error type for MUT_2 of SUT_1.

	error									
fault	*e1-AC*		*e2-AC*		*e3-AC*		*e4-AC*		*e5-AC*	
	Abs	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*
MFC	4	2.50	107	66.88	9	5.63	36	22.50	4	2.50
MIEB	0	0.00	10	34.48	8	27.59	11	37.93	1	3.45
MIFS	2	4.55	30	68.18	0	0.00	11	25.00	1	2.27
MLPA	90	13.16	336	49.12	67	9.80	147	21.49	48	7.02
total ALG	96	10.47	483	52.67	84	9.16	205	22.36	54	5.89
MVAE	70	19.23	166	45.60	39	10.71	70	19.23	21	5.77
MVAV	0	0.00	4	28.57	1	7.14	7	50.00	2	14.29
MVIV	0	0.00	3	100.00	0	0.00	0	0.00	0	0.00
WVAV	0	0.00	7	43.75	1	6.25	6	37.50	2	12.50
total ASG	70	17.63	180	45.34	41	10.33	83	20.91	25	6.30
MIA	0	0.00	15	55.56	1	3.70	11	40.74	0	0.00
MLC	0	0.00	2	100.00	0	0.00	0	0.00	0	0.00
total CHK	0	0.00	17	58.62	1	3.45	11	37.93	0	0.00
WAEP	16	44.44	12	33.33	5	13.89	3	8.33	0	0.00
WPFV	35	15.42	126	55.51	14	6.17	51	22.47	5	2.20
total INT	51	19.39	138	52.47	19	7.22	54	20.53	5	1.90
tot. errors	217	13.51	818	50.93	145	9.03	353	21.98	84	5.23

type and MUT is shown in Table 4.9) are of type *e1-EL*; while the value 47 reported by the cell (*total ALG, Abs*) - *e1-EL* column - in Table 4.11 indicates that 47 out of 369 errors detected by the event logs of the middleware, and caused by *ALG* faults (again, the number of the detected errors by fault type and MUT is shown in Table 4.9) are of type *e1-EL*. On the other hand, the value 4 reported by the cells (*MFC, Abs*) - *e1-AC* column - in Table 4.12, indicates that 4 out of 160 errors detected by the assertion checking of the middleware, and caused by *ALG* faults of type *MFC* are of type *e1-AC*; while the value 96 reported by the cell (*total ALG, Abs*) - *e1-AC* column - in Table 4.12 indicates that 96 out of 917 errors detected by the assertion checking of the middleware, and caused by *ALG* faults are of type *e1-AC*. Similarly, the value 63 reported by the cells (*MFC, Abs*) - *e1-RB* column -

Table 4.13: Absolute number (Abs) and percentage of reported errors (%) by fault and error type for MUT_3 of SUT_1.

	error																	
fault	*e1-RB*		*e2-RB*		*e3-RB*		*e4-RB*		*e5-RB*		*e6-RB*		*e7-RB*		*e8-RB*		*e9-RB*	
	Abs	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*
MFC	63	45.00	40	28.57	29	20.71	28	20.00	29	20.71	10	7.14	55	39.29	97	69.29	0	0.00
MIEB	9	29.03	11	35.48	5	16.13	5	16.13	0	0.00	0	0.00	9	29.03	26	83.87	0	0.00
MIFS	13	38.24	8	23.53	3	8.82	3	8.82	6	17.65	2	5.88	14	41.18	23	67.65	2	5.88
MLPA	346	37.37	278	30.02	109	11.77	147	15.87	94	10.15	80	8.64	290	31.32	753	81.32	3	0,32
total ALG	431	38.11	337	29.80	146	12.91	183	16.18	129	11.41	92	8.13	368	32.54	899	79.49	5	0.44
MVAE	169	36.11	150	32.05	67	14.32	54	11.54	36	7.69	44	9.40	163	34.83	377	80.56	6	1.28
MVAV	3	20.00	6	40.00	1	6.67	0	0.00	1	6.67	1	6.67	2	13.33	12	80.00	0	0.00
MVIV	2	50.00	1	25.00	1	25.00	1	25.00	0	0.00	0	0.00	3	75.00	4	100.00	0	0.00
WVAV	5	29.41	6	35.29	1	5.88	1	5.88	1	5.88	2	11.76	6	35.29	16	94.12	0	0.00
total ASG	179	35.52	163	32.24	70	13.89	56	11.11	38	7.54	47	9.33	174	34.52	409	81.15	6	1.19
MIA	8	44.44	5	27.78	1	5.56	2	11.11	1	5.56	2	11.11	9	50.00	13	72.22	0	0.00
MLC	0	0.00	1	50.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	2	100.00	0	0.00
total CHK	8	40.00	6	30.00	1	5.00	2	10.00	1	5.00	2	10.00	9	45.00	15	75.00	0	0.00
WAEP	26	68.42	9	23.68	6	15.79	5	13.16	0	0.00	2	5.26	15	39.47	31	81.58	0	0.00
WPFV	62	21.99	127	45.04	32	11.35	22	7.80	11	3.90	38	13.48	71	25.18	219	77.66	0	0.00
total INT	88	27.50	136	42.50	38	11.88	27	8.44	11	3.44	40	12.50	86	26.88	250	78.13	0	0.00
tot. errors	706	35.75	642	32.51	255	12.91	268	13.57	179	9.06	181	9.16	637	32.25	1573	79.65	11	0.56

in Table 4.13, indicates that 63 out of 140 errors detected by the rule-based logging of the middleware, and caused by *ALG* faults of type *MFC* are of type *e1-RB*; while the value 431 reported by the cell (*total ALG, Abs*) - *e1-RB* column - in Table 4.13 indicates that 431 out of 1,131 errors detected by the rule-based logging of the middleware, and caused by *ALG* faults are of type *e1-RB*. In percentage terms, the 2.90% (i.e., $(2/69) \cdot 100$), 2.50% (i.e., $(4/160) \cdot 100$) and 45.00% (i.e., $(63/140) \cdot 100$) of detected errors by MUT_1, MUT_2 and MUT_3, respectively, and caused by ALG faults of type MFC are of type *e1-EL*, *e1-AC* and *e1-RB*, respectively: these values are reported by the *%* column of Table 4.11, Table 4.12 and Table 4.13, respectively. The bottom rows of Table 4.11, Table 4.12 and Table 4.13, aggregate the number and the percentage of reported errors by type.

Table 4.14: Absolute number (Abs) and percentage (%) of reported errors by failure and error type for MUT_1 of SUT_1.

	error																							
failure	*e1-EL*		*e2-EL*		*e3-EL*		*e4-EL*		*e5-EL*		*e6-EL*		*e7-EL*		*e8-EL*		*e9-EL*		*e10-EL*		*e11-EL*		*e12-EL*		*e13-EL*	
	Abs	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*
CRASH	49	10.54	38	8.17	96	20.65	219	47.10	12	2.58	33	7.10	30	6.45	24	5.16	30	6.45	9	1.94	6	1.29	0	0.00	48	10.32
SILENT	3	2.26	2	1.50	2	1.50	0	0.00	114	85.71	4	3.01	1	0.75	0	0.00	92	69.17	7	5.26	22	16.54	0	0.00	1	0.75
ERRATIC	11	16.67	5	7.58	12	18.18	0	0.00	2	3.03	0	0.00	1	1.52	0	0.00	3	4.55	0	0.00	28	42.42	7	10.61	4	6.06
NO_FAILURE	10	20.00	8	16.00	12	24.00	5	10.00	5	10.00	0	0.00	0	0.00	0	0.00	0	0.00	4	8.00	3	6.00	3	6.00	5	10.00
tot. errors	73	10.22	53	7.42	122	17.09	224	31.37	133	18.63	37	5.18	32	4.48	24	3.36	125	17.51	20	2.80	59	8.26	10	1.40	58	8.12

Table 4.15: Absolute number (Abs) and percentage (%) of reported errors by failure and error type for MUT_2 of SUT_1.

	error									
failure	*e1-AC*		*e2-AC*		*e3-AC*		*e4-AC*		*e5-AC*	
	Abs	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*
CRASH	217	13.53	817	50.94	144	8.98	353	22.01	84	5.24
SILENT	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
ERRATIC	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
NO_FAILURE	0	0.00	1	50.00	1	50.00	0	0.00	0	0.00
tot. errors	217	13.51	818	50.93	145	9.03	353	21.98	84	5.23

Table 4.16: Absolute number (Abs) and percentage (%) of reported errors by failure and error type for MUT_3 of SUT_1.

	error															
failure	*e1-RB*		*e2-RB*		*e3-RB*		*e4-RB*		*e5-RB*		*e6-RB*		*e7-RB*		*e8-RB*		*e9-RB*	
	Abs	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*	*Abs*	*%*
CRASH	633	35.28	591	32.94	226	12.60	223	12.43	139	7.75	154	8.58	577	32.16	1457	81.22	10	0.56
SILENT	69	48.25	44	30.77	26	18.18	43	30.07	26	18.18	20	13.99	54	37.76	110	76.92	0	0.00
ERRATIC	2	11.11	5	27.78	3	16.67	1	5.56	12	66.67	1	5.56	4	22.22	1	5.56	0	0.00
NO_FAILURE	2	10.00	2	10.00	0	0.00	1	5.00	2	10.00	6	30.00	2	10.00	5	25.00	1	5.00
tot. errors	706	35.75	642	32.51	255	12.91	268	13.57	179	9.06	181	9.16	637	32.25	1573	79.65	11	0.56

Table 4.14, Table 4.15 and Table 4.16 report the **absolute number (i.e., *Abs*)** and **percentage (i.e., %)** of errors reported by MUT_1, MUT_2 and MUT_3, respectively, by failure and error type. For example, the value 49 reported by the cell (*CRASH, Abs*) - *e1-EL* column - in Table 4.14 indicates that 49 out of 465 errors detected by the event logs of the middleware (SUT_1), which have lead to a *CRASH* failures in the SUT (the number of the detected errors by failure type and MUT is shown in Table 4.10), are of type *e1-EL*. On the other hand, the value 217 reported by the cells (*CRASH, Abs*) - *e1-AC* column - in Table 4.15, indicates that 217 out of 1,604 errors detected by the assertion checking of the middleware, which have lead to a *CRASH* failures in the SUT, are of type

e1-AC. Similarly, the value 633 reported by the cells (*CRASH, Abs*) - *e1-RB* column - in Table 4.16, indicates that 633 out of 1,794 errors detected by the rule-based logging of the middleware, which have lead to a *CRASH* failures in the SUT, are of type *e1-RB*. In percentage terms, the 10.54% (i.e., $(49/465) \cdot 100$), 13.53% (i.e., $(217/1,604) \cdot 100$) and 35.28% (i.e., $(633/1,794) \cdot 100$) of detected errors by MUT_1, MUT_2 and MUT_3, respectively, which have lead to a *CRASH* failures in the SUT, are of type *e1-EL*, *e1-AC* and *e1-RB*, respectively: these values are reported by the *%* column of Table 4.14, Table 4.15 and Table 4.16, respectively. The bottom rows of Table 4.14, Table 4.15 and Table 4.16, aggregate the number and the percentage of reported errors by type.

Chapter 5

Experimental Results: Analysis of the target Techniques

The effectiveness of the considered MUTs, i.e., event logging (EL - MUT_1), assertion checking (AC - MUT_2), rule-based logging (RB - MUT_3), has been evaluated by measuring the evaluation metrics presented in Section 3.3. Precisely, Recall (R), Precision (P), Failure Coverage (FC), Error Determination Degree (EDD) and Error Propagation Reportability (EPR) have been evaluated for each MUT. It should be noted that Recall, Precision and Failure Coverage of each MUT have been evaluated on both the target SUTs considered in this dissertation. Differently, the Error Determination Degree and the Error Propagation Reportability have been evaluated only on the SUT_1, which is the most complex one (again, it is a distributed system deployed on two different nodes and it is composed by 796,353 lines of code for each node; differently, the SUT_2 is not a distributed system, and it is composed by a lower number of lines of code, i.e., 40,396); moreover, a high number of experiments in SUT_1 led to almost one error notification generated by one of the MUTs.

5.1 Event Logging Analysis

The effectiveness of the MUT_1, i.e., event logging, has been evaluated by analyzing the data generated by the MUT during the conducted experimental campaign, which are summarized in the tables described in Section 4.6. The data have allowed the measurement of the metrics defined in the proposed methodology, i.e., Recall (R), Precision (P), Failure Coverage (FC), Error Determination Degree (EDD) and Error Propagation Reportability (EPR). It should be noted that Recall, Precision and Failure Coverage of the MUT have been evaluated on both the considered target SUTs, i.e., the communication middleware (SUT_1) and arrival

manager (SUT_2). Differently, the Error Determination Degree and the Error Propagation Reportability have been evaluated only for the MUTs implemented by the SUT_1.

5.1.1 Recall and Precision

Figure 5.1 and Figure 5.2 show the percentage of reported errors of the MUT_1, i.e., the percentage of experiments where the MUT has generated at least one error notification, with respect to the failure and non-failure experiments, i.e., the experiments where the injected fault led to a failure in the considered SUT or not, respectively, conducted during the experimental campaign for SUT_1 and SUT_2, respectively.

Figure 5.1 shows that MUT_1 has reported at least an error notification for a high percentage of *SILENT* failures occurred in the SUT_1, i.e., 65.20%, while reported at least an error notification for a limited percentage of *CRASH* and *ERRATIC* failures, i.e., 17.62% and 20.89%, respectively. However, most of the errors reported by the MUT_1 led to a *CRASH* failure in the SUT_1. Indeed, *CRASH* failures are the most occurred failures in the SUT_1, i.e., 2,639 out of 3,159 failures occurred in SUT_1 (as reported in Table 4.4), which are followed by the *SILENT* and *ERRATIC* failures that account for 204 and 316, respectively.

Differently, Figure 5.2 shows that MUT_1 has reported at least an error notification for a high percentage of *CRASH* failures occurred in the SUT_2, i.e., 51.22%, while reported at least an error notification for a limited percentage of *SILENT* and *ERRATIC* failures.

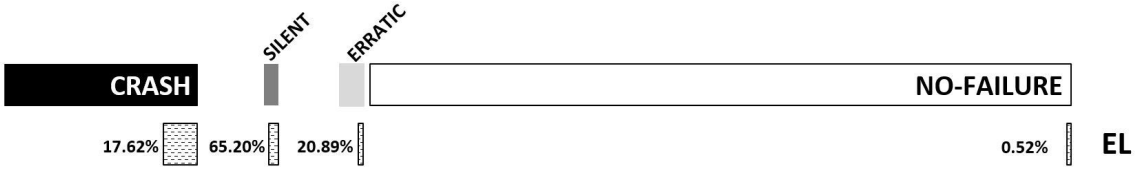

Figure 5.1: Percentage of reported errors of MUT_1 by failure type for SUT_1.

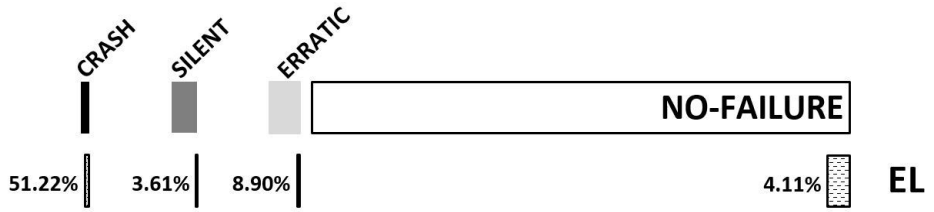

Figure 5.2: Percentage of reported errors of MUT_1 by failure type for SUT_2.

As for SUT_1, most of the errors reported by the MUT_1 in the SUT_2 led to a *CRASH* failure. Indeed, MUT_1 has generated at least an error notification in 42 experiments where a *CRASH* occurred in the SUT_2, against the 10 and 29 experiments where a *SILENT* and an *ERRATIC* occurred, respectively. In addition, Figure 5.1 and Figure 5.2 show also that MUT_1 has generated error notifications also when no failures occurred in the SUTs. These error notifications represent False Positives (FPs) with respect to the failures.

Table 5.1 reports FP, FN, TP, P and R for the MUT_1 for each SUT. It can be noted that the *precision* is very close to 1 for the event logging mechanism implemented by the SUT_1 (i.e., MW-EL): almost all the failures reported by the MUT are actual failures occurred in the communication middleware. Differently, MUT_1 exhibits a low *precision* value in the SUT_2. In fact, event logging generates a relevant number of FPs in the arrival manager (i.e., AM-EL). The number of FNs is 2,495 out of total 3,159 failures in the SUT_1 and 604 out of total 685 failures occurred in the SUT_2. These findings suggest that event logging

Table 5.1: False Positive (`FP`), False Negative (`FN`), True Positive (`TP`), Precision and Recall of MUT_1 for each SUT.

	`FP`	`FN`	`TP`	***Precision***	***Recall***
MW-EL	50	2,495	664	0.930	0.210
AM-EL	243	604	81	0.250	0.118

might miss a relevant number of failures. The rightmost column of Table 5.1 reports the recall of the MUT.

It should be noted that the density of the MUT, which is reported by Table 4.2 (i.e., EL and EL-error), is potentially related to the value of recall/precision. For example, Table 5.1 reports that the recall of MW-EL is bigger than AM-EL: the percentage of error logging instructions out of the total number of logging instructions of MW-EL, i.e., 14.45%, is bigger than AM-EL, i.e., 11.39%. Nevertheless, the high density of the MUT might affect the precision, as it can be inferred from the values of precision of AM-EL reported by Table 5.1. In fact, a large number of logging instructions might increase the probability to generate FPs.

5.1.2 Failure Coverage

The overall recall has been broken down by failure type in each SUT. Given a failure type, the bottom row of Table 4.7 and Table 4.8 show the absolute number and the percentage of failures that have been reported by each MUT in SUT_1 and SUT_2, respectively. For example, EL reports 465 and 45 *CRASH* failures in SUT_1 and SUT_2, respectively (such as shown by the first row of the third cell in the bottom of Table 4.7 and Table 4.8, respectively). These numbers account for total 17.72%, i.e., $(465/2,639) \cdot 100$, and 51.22%, i.e., $(45/82) \cdot 100$, of *CRASH* failures that have been induced in the SUT_1 and SUT_2, respectively.

Figure 5.3 shows the percentage of reported failures of MUT_1 by failure type and SUT. It can be noted that **the reporting ability of the MUT changes significantly across**

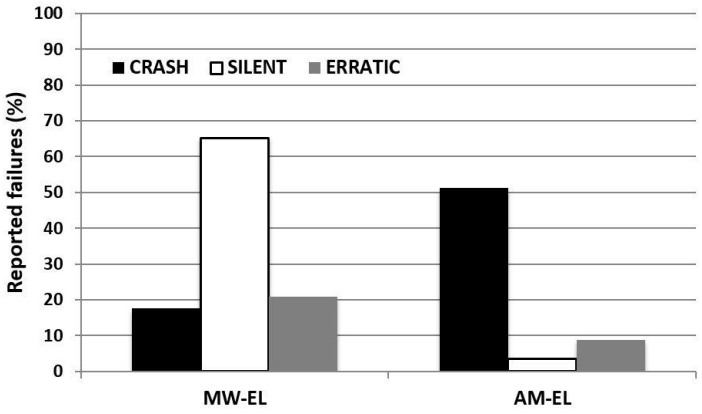

Figure 5.3: Percentage of reported failure of MUT_1 by type and case study.

the SUTs. Moreover, **the MUT might show a different ability at reporting the same type of failure in different SUTs**. In fact, the coverage of the event logs ranges from a minimum of 3.61%, i.e., AM-EL (*SILENT* failures), to a maximum of 65.20%, i.e., MW-EL (*SILENT* failures).

The overall recall has been also broken down by fault type in each SUT. Given a fault type, the rightmost column of Table 4.7 and Table 4.8 shows the absolute number and the percentage of failures that have been reported by each MUT in SUT_1 and SUT_2, respectively. For example, EL reports 348 and 67 activated *ALG* faults in SUT_1 and SUT_2, respectively (such as shown by the first row of the penultimate rightmost cell in Table 4.7 and Table 4.8, respectively). These numbers account for total 18.69% and 12.32% of activated *ALG* faults that have been injected in the SUT_1 and SUT_2, respectively.

Figure 5.4 shows the percentage of activated **faults** that are reported by MUT_1 in both the SUTs. Percentage of reported failures can be observed in the rightmost column of Table 4.7 and Table 4.8 for SUT_1 and SUT_2, respectively. It can be noted that **the reporting ability by fault type of the MUT changes slightly in the SUT**$_1$,i.e., event logging is

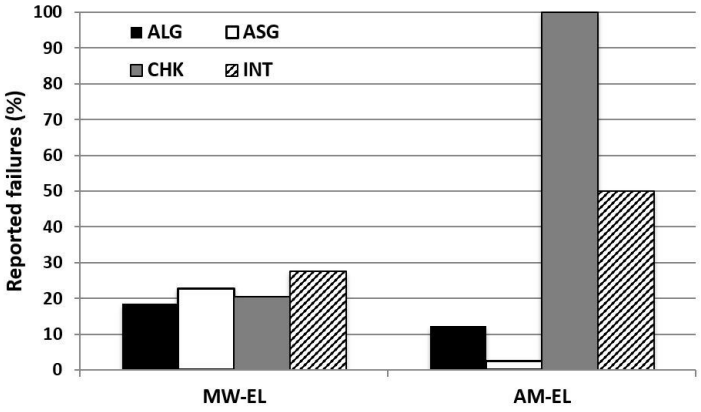

Figure 5.4: Percentage of reported failure of MUT_1 by fault type and case study.

able to detect almost the same percentage of failures irrespectively of the type of injected fault in the SUT_1. Moreover, **the MUT might show a different ability at reporting the same type of fault in different SUTs**. In fact, the reported failure by fault type of the event logs ranges from a minimum of 18.69%, i.e., MW-EL (*ALG* faults), to a maximum of 27.51%, i.e., MW-EL (*INT* faults), in the SUT_1, and from a minimum of 2.44%, i.e., AM-EL (*ALG* faults), to a maximum of 100.00%, , i.e., AM-EL (*CHK* faults), in the SUT_2.

5.1.3 Error Determination Degree

The error behavior inferred by the error notifications of the MUT_1 has been evaluated by considering the error model extracted during the error clustering process, which is described in Section 4.5.2.

Figure 5.5 shows the breakdown of the errors reported by MUT_1 in the SUT_1 by error type, i.e., the error types that belong to the inferred error model reported in Table 4.6, and fault ODC class. Percentage of the errors reported by the considered MUT by error type and ODC class can be observed in the *total ALG*, *total ASG*, *total CHK* and *total*

INT rows of Table 4.11. It can be noted that **the percentage of errors reported by the MUT for each error type changes slightly at varying the ODC class in SUT$_1$**. For example, considering the error type *e4-EL*, i.e., *data type errors*, the MUT has generated at least an error notification of this type in 27.10%, 34.87% and 41.35% of *ALG*, *ASG*, and *INT* experiments, respectively, i.e., experiments where only an *ALG*, an *ASG*, or an *INT* fault has been injected in the SUT. Similarly, the MUT has generated at least an error notification of type *e5-EL*, i.e., error related to the main daemon of the SUT, in 10.53%, 18.46% and 22.22% of *ALG*, *ASG*, and *INT* experiments, respectively. In particular, *ALG* and *ASG* exposed a very similar error behavior. A closer look into the error notifications obtained from these ODC classes and into the source code of the related injected faults allowed to understand that often faults of different classes led to error notifications of the same type, and in some cases exactly the same ones. For example, in some cases both the *elimination of small part of source code*, i.e., *MPLA* faults that

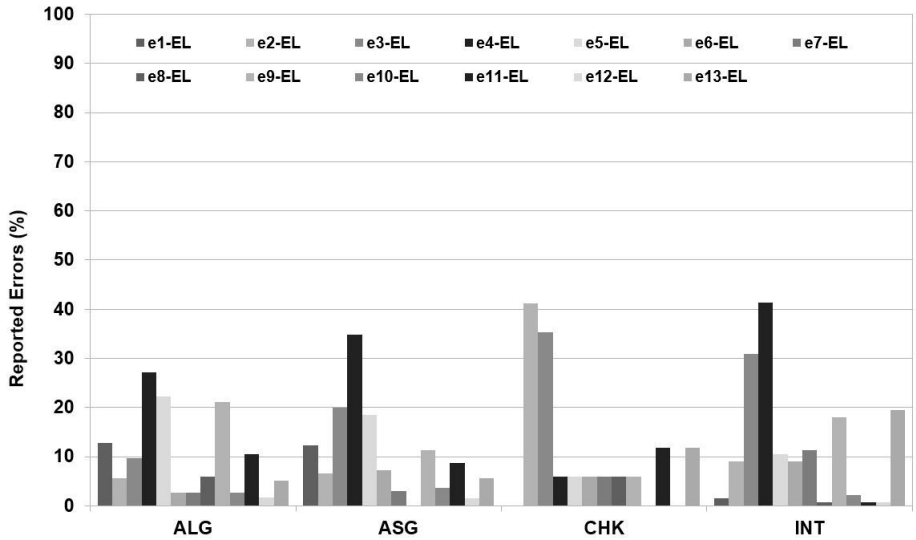

Figure 5.5: Percentage of reported errors by cluster and fault ODC type for MUT$_1$.

belong to the *ALG* class, and the *substitution of variable assignment with an expression*, i.e., *MVAE* faults that belong to the *ASG* class, led a variable to be assigned with a data of an unexpected type, which has been reported by the MUT with the same error notification, i.e., `"Type mismatch: object type is ... but ... was expected"` that belongs to the type *e4-EL*. Moreover, in three of the 4 ODC classes, i.e., *ALG*, *ASG* and *INT*, *e4-EL* errors are the most reported ones by the MUT. It should be noted that *CHK* faults led to a different error behavior with respect to the other fault types. However, a small number of samples have been collected for this type of fault. Therefore, no conclusions can be drawn for *CHK* class.

As discussed in section 3.3.3, an extract of the obtained dataset has been submitted to a classifier in order to measure the *Error Determination Degree* of the event logging with respect to the ODC fault class, i.e., the ability of its error notifications to suggest what is the ODC class of the fault that have led to those error notifications. The extracted dataset contains (i) the ODC class of the injected fault and (ii) the number of error notifications generated by the event logging for each error type of its inferred error model, for each fault injection experiment where at least one error notification has been generated by the MUT. This dataset has been submitted to a *Random Forest* classifier[1] classifier [115]. The numbers of error notifications for each type are used as features of the classification, while the ODC fault class is used as the *class* to predict, i.e., the value the classifier have to predict. A *K-fold cross-validation* has been conducted, with *K=30*. It should be noted that different classifiers and number of folds have been tried in order to choose the best combination;

[1]The parameters of the classifier have been left at default value.

Table 5.2: Prediction results for MUT_1 (k-fold cross-validation: Random Forest and k=30).

class	***correct classification (%)***	***incorrect classification (%)***
ODC fault class	57.10%	42.89%
fault type	43.47%	53.53%
failure type	88.92%	11.08%

the *Random Forest* classifier with the considered number of folds have outperformed all other combinations. However, practitioners might apply different classifier and approaches in order to evaluate the EDD.

The first row of Table 5.2 reports the results of the classification. The considered classifier was able to predict the ODC type of a fault from the error notifications the fault have led to with a not very high accuracy; in fact, the percentage of correct classification, i.e., the considered *Error Determination Degree*, is equal to 57.10%. This result confirm the finding inferred from Figure 5.5. Indeed, the low variability of the error behavior inferred by means of the MUT_1 at varying the ODC class led to the poor prediction performance of the classifier.

The ODC fault class has been broken down by fault type in order to understand the error behavior at varying the fault type. Figure 5.6 shows the percentage of **errors** that are reported by MUT_1 by error and fault type, i.e., the types of fault reported in Table 3.1. Percentage of the errors reported by the considered MUT by error and fault type can be observed in Table 4.11. It can be noted that **the percentage of errors reported by the MUT for each error type changes at varying the fault type in SUT_1**. For example, the error type *e1-EL*, i.e., *memory errors*, has been reported with different percentages at varying the fault type, e.g., at least an error notification of this type has been reported

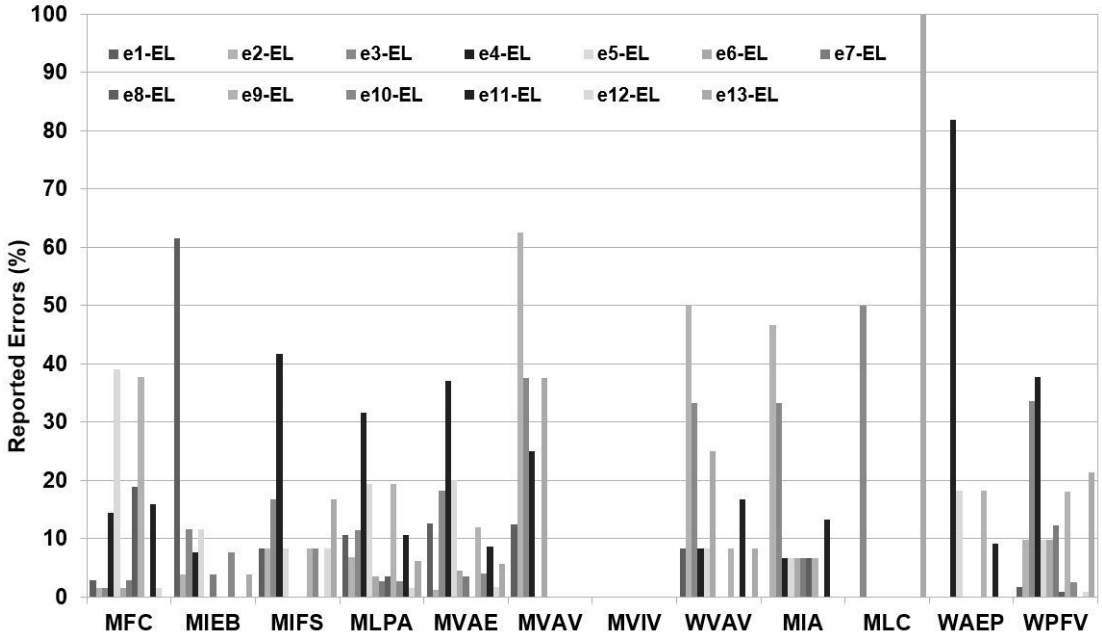

Figure 5.6: Percentage of reported errors by cluster and fault type for MUT_1.

in 1.64% of experiments where a *WPFV* fault has been injected, as well as in 61.54% of experiments where a *MIEB* fault has been injected. However, *MLPA* and *MVAE* faults exposed a very similar error behavior. This finding is strictly related to the one obtained for the ODC classes. Indeed, *MLPA* and *MVAE* are fault types that belongs to *ALG* and *ASG* class, respectively, and they are the ones that occurred more often respect than the other types in the related class, as can be seen in Table 4.9. Therefore, their error behaviors influence the error behavior of the ODC class they belong to. In particular, as previously discussed, they tended to generate the same errors, which are reported by MUT_1 with error notification of the same type. In addition, **given a fault type, in almost all cases the percentage of reported errors strongly changes at varying the error type**. For example, for *MIEB* faults the error type *e1-EL* is reported more than the other types as

well as the error type *e4-EL* for *MVAE* faults. It should be noted that a limited number of samples have been collected for some types of fault, i.e., *MVIV*, *MIFS*, *MVAV*, *WVAV*, *MIA*, *MLC* and *WAEP*, as showed in Table 4.9 (EL column). Therefore, no conclusions can be drawn for these types of fault.

An extract of the obtained dataset has been submitted to the *Random Forest* classifier in order to measure the *Error Determination Degree* of the event logging with respect to the fault type, i.e., the ability of its error notifications to suggest what is the type of the fault that have led to those error notifications. The extracted dataset contains (i) the type of the injected fault and (ii) the number of error notifications generated by the event logging for each error type of its inferred error model, for each fault injection experiment where at least one error notification has been generated by the MUT. The numbers of error notifications for each type are used as features of the classification, while the fault type is udes as *class* to predict. A *K-fold cross-validation* has been conducted, with *K=30*. The second row of Table 5.2 reports the results of the classification. Despite the more variability exposed by the error behavior at varying the fault type respect than the case where the faults are grouped into ODC class, the considered classifier was able to predict the fault type from the error notifications the fault have led to with a not very high accuracy; in fact, the percentage of correct classification, i.e., the considered *Error Determination Degree*, is equal to 43.47%, which is even worse than the one obtained in the case where the ODC class has been considered as class to predict.

Figure 5.7 shows the breakdown of the errors reported by MUT_1 in the SUT_1 by failure type, i.e., the failure that the reported error(s) led to, and error type. Percentage of the

errors reported by the considered MUT by error and failure type can be observed in Table 4.14. It can be noted that **the percentage of errors reported by the MUT for almost all error types strongly changes at varying the failure type in SUT$_1$**. For example, the percentage of experiments where at least an error notification of type *e4-EL* is raised ranges from 0%, for experiments where a SILENT or an ERRATIC failures occurred in the system, to 47.10%, for experiments where a *CRASH* failure occurred in the system. Similarly, the percentage of experiments where at least an error notification of type *e5-EL* ranges from 2.58%, for experiments where a *CRASH* failures occurred in the system, to 85.71%, for experiments where a *SILENT* failure occurred in the system.

An extract of the obtained dataset has been submitted to the *Random Forest* classifier in order to measure the *Error Determination Degree* of the event logging with respect to the failure type, i.e., the ability of its error notifications to suggest what is the type of failure occurred in the system as consequence of the error(s) behind those notifications. The

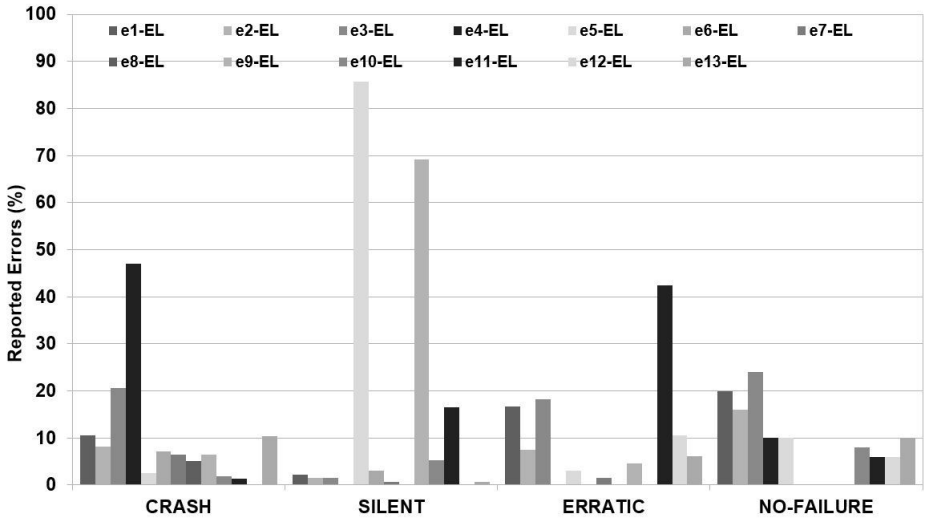

Figure 5.7: Percentage of reported errors by cluster and failure type for MUT$_1$.

extracted dataset contains (i) the failure occurred in the system and (ii) the number of error notifications generated by the event logging for each error type of its inferred error model, for each fault injection experiment where at least one error notification has been generated by the MUT. The numbers of error notifications for each type are used as features of the classification, while the failure type is used as *class* to predict. A *K-fold cross-validation* has been conducted, with *K=30*. The third row of Table 5.2 reports the results of the classification. The classifier was able to predict the failure type from the error notifications with a high accuracy; in fact, the percentage of correct classification, i.e., the considered *Error Determination Degree*, is equal to 88.92%. This result confirms the finding inferred from Figure 5.7. Indeed, the high variability of the error behavior inferred by means of the MUT_1 at varying the failure type led to the good prediction performance of the classifier.

Based on these findings, it should be noted that event logging can potentially allow the determination of the failure type occurred in the communication middleware from the obtained error notifications, while nothing can be said about the ODC class and the type of the fault that has been injected.

5.1.4 Error Propagation Reportability

The propagation paths of the errors raised in SUT_1 during the experimental campaign, as consequences of the injected faults, have been inferred by means of the error notifications generated by MUT_1 in the SUT. As a reminder, the error notifications generated by event logging in the SUT_1 provides the component, and also the function, that has generated the notification. Therefore, the knowledge of the component, and also of the function, where the

faults have been injected (both provided by the tool used for the fault injection), along with the knowledge of the source component and function of each error notification, allowed to build non-exhaustive graphs (according to the methodology proposed in Section 3.3.4) that summarize the error propagation phenomena obtained during the experimental campaign. As a reminder, the obtained directed graphs are considered non-exhaustive because they have been built based on the errors detected by the MUT_1; therefore the errors undetected by the MUT cannot be considered. It should be noted that only the directed graphs that summarize propagation paths of the errors generated as a consequence of *ALG* and *ASG* faults have been generated, since they are the ODC fault classes with the highest number of collected samples.

Figure 5.8 shows the error propagation graph that summarizes the major error propagation paths through the components of the SUT_1, which are generated as a consequence of *ALG* faults. Noteworthy, the number of faults considered in the graph refers to the number of faults that have led to at least an error notification to be raised by event logging. It can be observed that, excepted from the *MIEB* faults, **most of the faults led to errors that have not been reported by the event logging in the *kernel***. For example, 91.30%, 66.66% and 65.27% of *MFC*, *MIFS* and *MLPA* faults, respectively, did not lead to an error notification in the *kernel*. In particular, 33.33%, 31.68% and 41.67% of *MFC*, *MLPA* and *MIFS* faults, respectively, have led to at least an error that has generated at least an error notification in the *database* component of the SUT_1; while 37.68% and 15.27% of *MFC* and *MLPA* faults, respectively, have led to at least an error that has generated at least an error notification in the *api*, *spliced* and *user* components of the SUT_1, at the same time. This

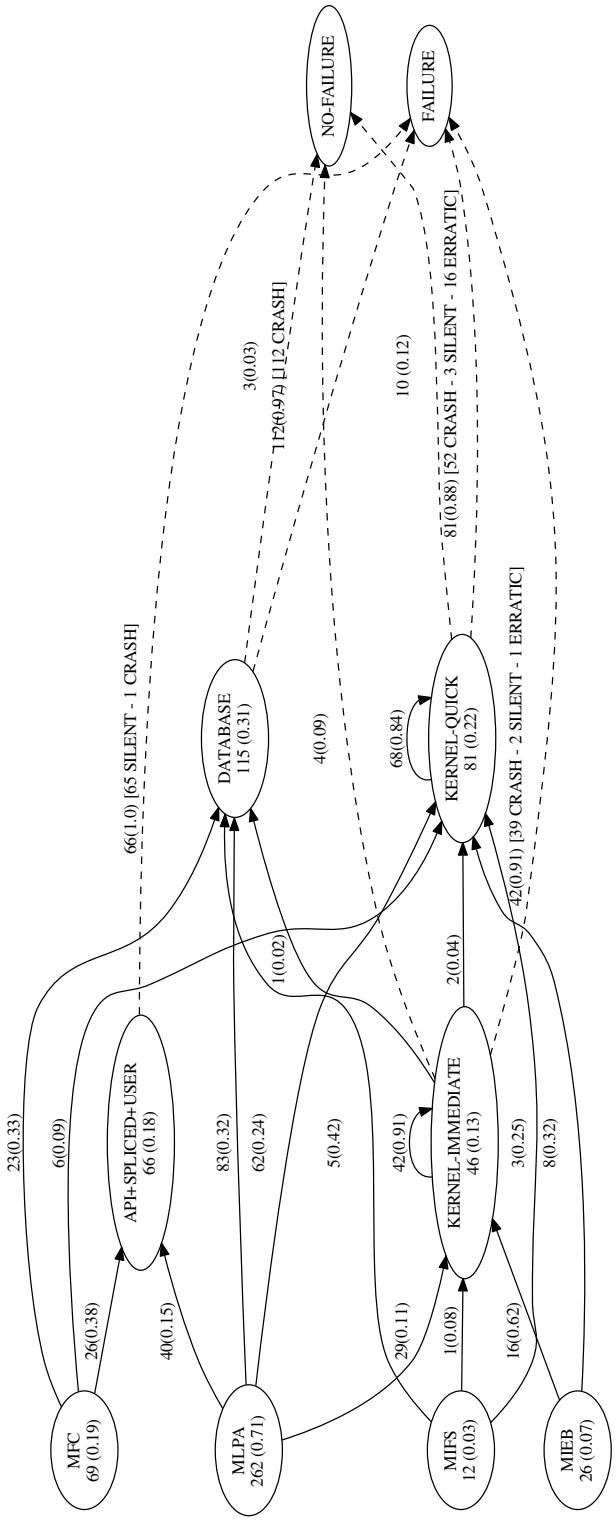

Figure 5.8: Error propagation graph for ALG faults (MUT_1).

finding is also confirmed by the proposed *Error Propagation Reportability*, which exhibits a value of 26.02%, i.e., $(96/369) \cdot 100$ (last cell of first row in Table 5.3), that suggests the low ability of MUT_1 at reporting the propagation of the errors generated as consequence of *ALG* faults. Indeed, the errors are expected to arise in the *kernel* component of the SUT_1 since all the faults have been injected into this component. Therefore, there is the need to improve the error detection ability of the event logging in the *kernel* by adding some *EDMs*, i.e., *Error Detection Mechanisms*, also with the aim to obtain more realistic error propagation paths, which can help practitioners in different analysis, such as root cause analysis, problem determination, etc. A closer look into the error notifications generated by the MUT in the SUT's components can allow to understand the type of error the EDMs have to consider. For example, a closer look into the error notifications generated by the *database* during the experiments, where no other components have reported an error, allowed to understand that most of them belong to the *e4-EL* type. Therefore, a potential EDM into the kernel have to consider this kind of error, e.g., by inserting a number of checks on the type of data that are used at runtime. As a reminder, it should be noted that there is the possibility that a propagation path has not been reported by a MUT because it is not present by design in the target SUT, e.g., when the component that reports the error works as a detector of the component where the fault has been injected. Therefore, the proposed *EPR* metric might provide not accurate value in this case. However, it can be successfully used to decide where to place EDMs, and, more important, as a comparative metric between MUTs, as will be shown in Section 6. Indeed, if a MUT reports the error propagation path that is unreported by another MUT (with a lower *EPR*), this suggests that the second one

Table 5.3: Error Propagation Reportability (EPR) of MUT_1 with respect to *ALG* and *ASG* faults.

ODC	***# experiments with at least an error notification***	***# experiments with at least an error notification generated in the kernel***	***EPR (%)***
ALG	369	96	26.02
ASG	195	82	42.05

actually exhibited a low ability at reporting error propagation paths. Another findings that can be inferred from Figure 5.8 is that **a limited number of errors have been reported in the function where the fault has been injected**, as shown by the low probability of the *kernel-immediate* node, i.e., 0.13. In addition, **almost all the errors reported by event logging in the kernel are not reported in any other components**, as shown by the high probability of the self-loop of the nodes *kernel-immediate* and *kernel-quick*, i.e., 0.91 and 0.84, respectively.

Figure 5.8 also shows that **errors reported by event logging in different system components have led to different failures in the SUT**. For example, in 100.00% of the cases where an error have propagated to the *api*, *spliced* and *user* components of the SUT, at the same time, a *SILENT* occurred in the SUT; while in 97.39% of the cases where an error have propagated to the *database* component a *CRASH* occurred in the SUT. On the other hand, it can be observed that *ERRATIC* failures occurred in the system mainly in the cases where an error propagated to the functions different from the one where the fault has been injected. It should be noted that this information allow to identify potential position for *ERMs*, i.e., *Error Recovery Mechanisms*, that can increase the resiliency of the system. For example, if the practitioners are interested in avoiding *SILENT* failures, probably it can

be useful to insert an *ERM* in the *api*, *spliced* or *user* component, or an *ERM* that checks these components at runtime. A closer look into the error notifications generated by event logging in these components allowed to understand that most of them are of type *e5-EL*, i.e., *main daemon error*; therefore, it could be useful to add an ERM in the system that checks for the availability of the main daemon (i.e., the *spliced* component) and provides a recovery mechanism in case of problem, such as re-execution of the component, execution of a different version of the component, i.e., the same component developed with a given diversity degree. It should be noted that the implementation of ERM is not a trivial task an requires a deep knowledge of the system, especially in complex critical system. Therefore, system developers, which have this knowledge, can simply leverage the results obtained from this analysis to design and implement effective ERM in their system.

Figure 5.9 shows the directed graph that summarizes the major error propagation paths through the components of the SUT_1, which are generated as a consequence of *ASG* faults. Again, the number of faults considered in the graph refers to the number of faults that have led to at least an error notification to be raised by event logging. It can be observed that **more than half of the errors reported by the event logging have not been reported by the *kernel* component** of the SUT_1. In fact, 60.57% of *MVAE* faults, i.e., the most recurrent fault type in the *ASG* class, have led to at last an error that has generated at least an error notification in a component different form the *kernel*. In particular, 34.86% of *MVAE* faults have led to at least an error that has generated at least an error notification in the *database* component of the SUT_1. This finding is also confirmed by the proposed *Error Propagation Reportability*, which exhibits a value of 42.05%, i.e.,

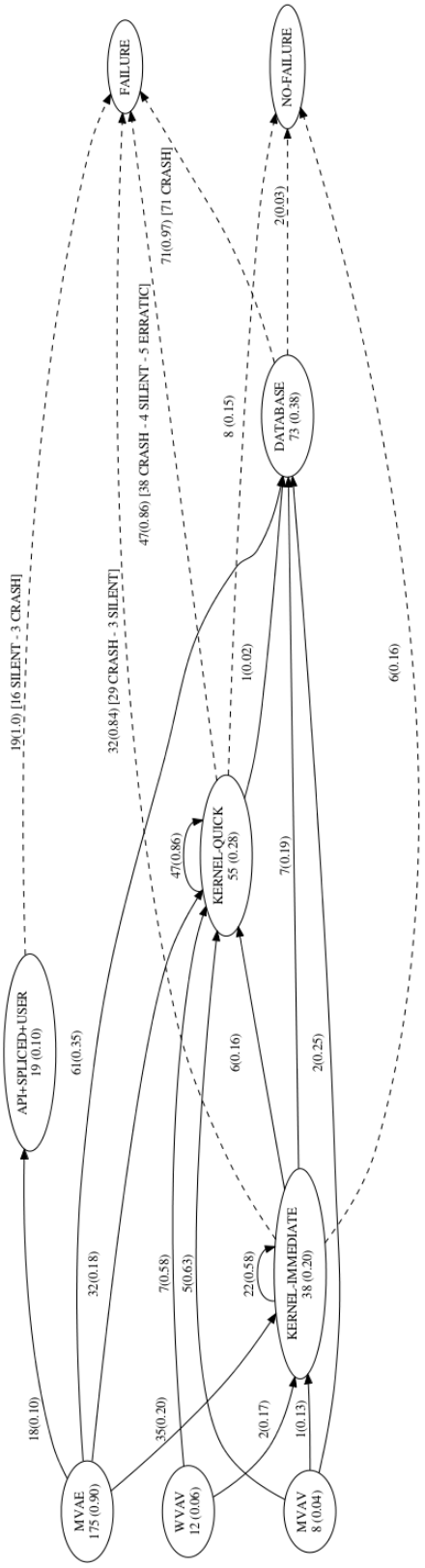

Figure 5.9: Error propagation graph for ASG faults (MUT_1).

$(82/195) \cdot 100$ (last cell of last row in Table 5.3), which means that 42.05% of the error reported by the event logging are not reported by the *kernel* component. Noteworthy, EPR value exhibited by the event logging for *ASG* class is greater than the one obtained for the *ALG*, which means that for *ASG* class this MUT exposed better performance at reporting error propagation. However, also for *ASG* there is the need to improve the error detection ability of the event logging in the *kernel* by adding some *EDMs*. A closer look into the error notifications generated by the MUT in the SUT's components can allow to understand the type of error the EDM have to consider. For example, the error notifications generated by the *database* during the experiments where no other components have reported an error, belong to the *e4-EL* type, as in the case of *ALG* class. This similarity is an expected result since in Section 5.1.3 it has been found that *ALG* and *ASG* exposed a very similar error behavior. In addition, **almost all the errors reported by event logging in the kernel are not reported in any other components**.

Figure 5.9 also shows that also for the *ASG* class the **errors reported by event logging in different system components have led to different failures in the SUT**. For example, in 100.00% of the cases where an error have propagated to the *api*, *spliced* and *user* components of the SUT, at the same time, a *SILENT* occurred in the SUT; while in 97.26% of the cases where an error have propagated to the *database* component a *CRASH* occurred in the SUT. Again, this information allows to identify potential position for *ERMs*. For example, if the practitioners are interested in avoiding *CRASH* failures, probably it can be useful to insert an *ERM* in the *database* component. A closer look into the error notifications generated by event logging in this component allowed to understand

that most of them are of type *e4-EL*; therefore, it could be useful to add an ERM in the *database* component, which try to avoid failures by executing some recovery action when an error data type occurs, such as request again the data or try to continue the execution with default values, etc.

It should be noted that the analysis showed here, and also in the next sections, is intentionally conducted to at an higher level, i.e., at component level. However, practitioners can leverage the proposed methodology also to conduct a more fine-grained analysis, reaching the function level.

The findings obtained from the analysis of event logging provide part of the answer to the **Research Question 1**, i.e., **RQ1**, the **Research Question 2**, i.e., **RQ2**, and the **Research Question 3**, i.e., **RQ3**. In fact, the analysis of the *Error Propagation Reportability* allowed to understand that event logging can be used both to characterize the error behavior of the considered SUT (*RQ1*) and to provide insights about the placement of *EDMs* and *ERMs* (*RQ2*). In addition, this analysis also allows to understand that the Error Propagation Reporting Ability of the event logging changes when different ODC fault class are considered (*RQ3).* Differently, the analysis of *Recall* and *Precision*, of *Failure coverage* and of *Error Determination Degree* exhibited by event logging during the experiments allowed to infer how its Recall and Precision values change between the SUTs, how its failure reporting ability changes at varying the failure and fault type, and also the values exhibited by its EDD with respect to the fault type, ODC class and failure type (*RQ3*).

5.2 Assertion Checking Analysis

The effectiveness of the MUT_2, i.e., assertion checking, has been evaluated by analyzing the data generated by the MUT during the conducted experimental campaign, which are summarized in the tables in Section 4.6. The data have allowed the measurement of the metrics defined in the proposed methodology, i.e., Recall (R), Precision (P), Failure Coverage (FC), Error Determination Degree (EDD) and Error Propagation Reportability (EPR). It should be noted that, as for event logging, Recall, Precision and Failure Coverage of the MUT have been evaluated on both the considered target SUTs. Differently, the Error Determination Degree and the Error Propagation Reportability have been evaluated only for the MUTs implemented by the SUT_1.

5.2.1 Recall and Precision

Figure 5.10 and Figure 5.11 show the percentage of reported errors of the MUT_2, i.e., the percentage of experiments where the MUT has generated at least one error notification, with respect to the failure and non-failure experiments, i.e., the experiments where the injected fault led to a failure in the considered SUT or not, respectively, conducted during the experimental campaign for SUT_1 and SUT_2, respectively.

Figure 5.10 shows that MUT_2 has been able to report only *CRASH* failures occurred

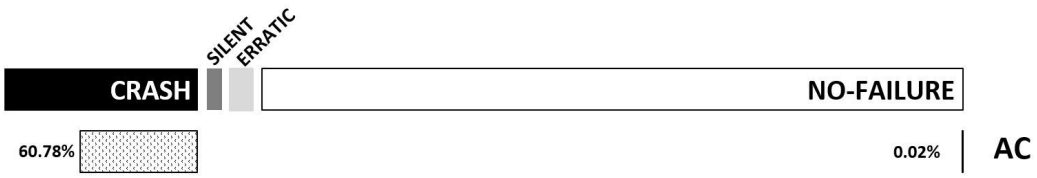

Figure 5.10: Percentage of reported errors of MUT_2 by failure type for SUT_1.

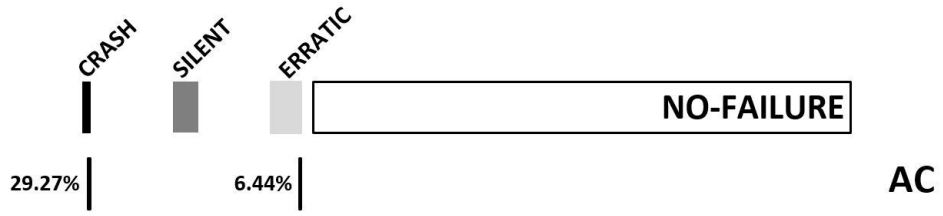

Figure 5.11: Percentage of reported errors of MUT_2 by failure type for SUT_2.

Table 5.4: False Positive (FP), False Negative (FN), True Positive (TP), Precision and Recall of MUT_2 for each SUT.

	FP	FN	TP	***Precision***	***Recall***
MW-AC	2	1,555	1,604	0.999	0.508
AM-AC	0	640	45	1.000	0.066

in the SUT_1, reporting at least an error notification for 60.78% of this kind of failures.

Differently, Figure 5.11 shows that MUT_2 has reported at least an error notification for 29.27% of *CRASH* failures occurred in the SUT_2, while reported at least an error notification for a limited percentage of *ERRATIC* failures, i.e., 6.44%. However, in absolute terms, the numbers of errors reported by the MUT_2 in the SUT_2 are quite similar for both kinds of failure. Indeed, MUT_2 has generated at least an error notification in 21 experiments where an *ERRATIC* occurred in the SUT_2, while has generated at least an error notification in 24 experiments where a *CRASH* occurred. In addition, Figure 5.10 and Figure 5.11 show also that MUT_2 has generated error notifications also when no failures occurred in the SUT_1. These error notifications represent False Positives (FPs) with respect to the failures.

Table 5.4 reports FP, FN, TP, P and R for the MUT_2 for each SUT. It can be noted that the *precision* is very close to 1 in SUT_1 and it is equal to 1 in the SUT_2, therefore, almost all the failures reported by the MUT are actual failures occurred in both the SUTs. The number of FNs is 1,555 out of total 3,159 failures in the SUT_1 and 640 out of total 685

failures occurred in the SUT_2. These findings suggest that assertion checking might miss a relevant number of failures, especially in the SUT_2. The rightmost column of Table 5.4 reports the recall of the MUT.

It should be noted that the density of the MUT, which is reported by Table 4.2 (i.e., AC), is potentially related to the value of recall. For example, the rather different values of recall measured for the MW-AC and AM-AC, are likely caused by the different density of the assertions in the SUTs, i.e., 0.99% and 0.18% out of the total number of assertions placed in the source code of SUT_1 and SUT_2, respectively.

5.2.2 Failure Coverage

The overall recall has been broken down by failure type in each SUT. Figure 5.12 shows the percentage of reported failures of MUT_2 by failure type and SUT. It can be noted that, as for MUT_1, **the reporting ability of the MUT_2 changes significantly across the SUTs**. Moreover, **the MUT might show a different ability at reporting the same type of failure in different SUTs**. In fact, the coverage of assertion checking ranges from a minimum of 0.00%, i.e., MW-AC for *SILENT* and *ERRATIC* failures, to a maximum of 68.78%, i.e., MW-AC for *CRASH* failures, in the SUT_1. Differently, the coverage exposed by this MUT in the SUT_2 ranges from a minimum of 0.00%, i.e., AM-AC for *SILENT* failures, to 29.27%, i.e., AM-AC for *CRASH* failures.

The overall recall has been also broken down by fault type in each SUT. Figure 5.13 shows the percentage of activated **faults** that are reported by MUT_2 in both the SUTs. It can be noted that **the reporting ability by fault type of the MUT changes slightly**

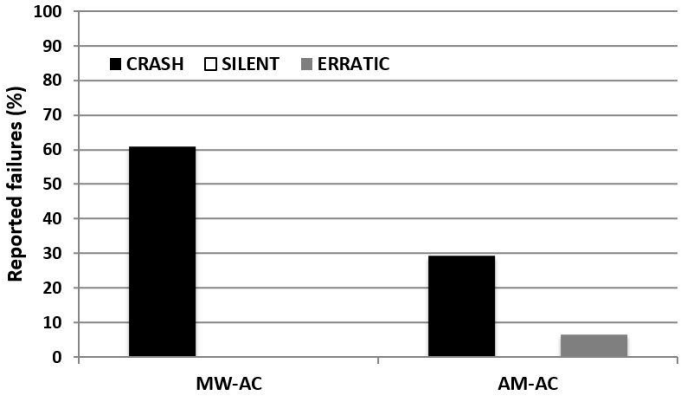

Figure 5.12: Percentage of reported failure of MUT_2 by type and case study.

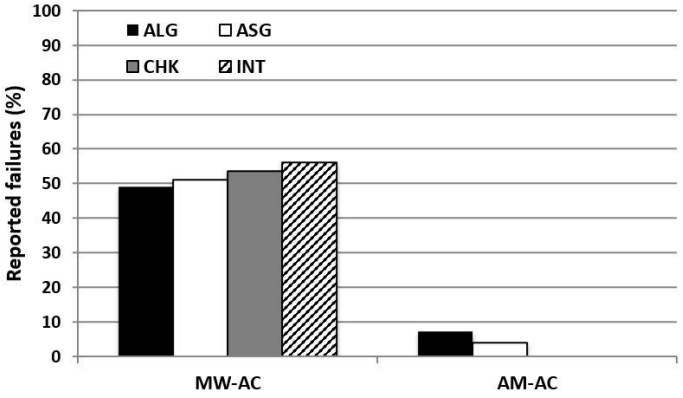

Figure 5.13: Percentage of reported failure of MUT_2 by fault type and case study.

in the SUTs. In fact, results indicate that assertion checking is able to detect almost the same percentage of failures irrespectively of the type of injected fault in both the SUTs. Moreover, **the MUT might show a different ability at reporting the same type of fault in different SUTs**. In fact, the reported failure by fault type of assertion checking ranges from a minimum of 49.19%, i.e., MW-AC (*ALG* faults), to a maximum of 56.08%, i.e., MW-AC (*CHK* faults), in the SUT_1, and from a minimum of 0.00%, i.e., AM-AC (*CHK* and *INT* faults), to a maximum of 7.35%, , i.e., AM-AC (*ALG* faults), in the SUT_2.

5.2.3 Error Determination Degree

The error behavior inferred by the error notifications of the MUT_2 has been evaluated by considering the error model extracted during the error clustering process.

Figure 5.14 shows the breakdown of the errors reported by MUT_2 in the SUT_1 by error type, i.e., the error types that belong to the inferred error model reported in Table 4.6, and fault ODC class. Percentage of the errors reported by the considered MUT by error type and ODC class can be observed in the *total ALG*, *total ASG*, *total CHK* and *total INT* rows of Table 4.12. It can be noted that **the percentage of errors reported by the MUT for each error type changes slightly at varying the ODC class in SUT_1**. For example, considering the error type *e2-AC*, i.e., *unexpected value errors*, the MUT has generated at least an error notification of this type in 52.67%, 45.34%, 58.62% and 52.47% of *ALG*, *ASG*, *CHK* and *INT* experiments, respectively, i.e., experiments where only an *ALG*, an *ASG*, *CHK* or an *INT* fault has been injected in the SUT. Similarly, the MUT

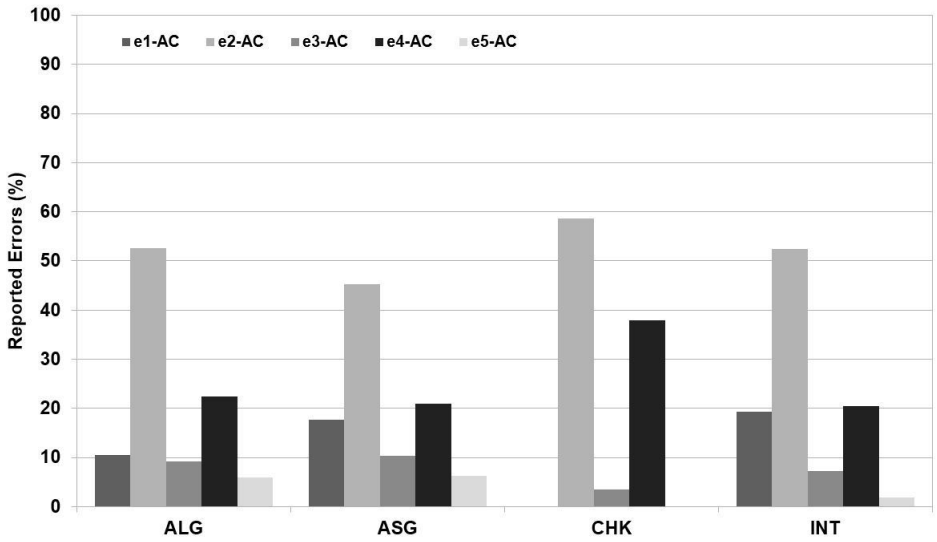

Figure 5.14: Percentage of reported errors by cluster and fault ODC type for MUT_2.

has generated at least an error notification of type *e4-AC*, i.e., `NULL` value errors, in 20.53%, 20.91%, 22.36% and 37.93% of *ALG*, *ASG*, *CHK* and *INT* experiments, respectively. In particular, *ALG*, *ASG* and *INT* exposed a very similar error behavior. A closer look into the error notifications obtained from these ODC classes and the source code of the related injected faults allowed to understand that often faults of different classes led to error notifications of the same type. For example, in some cases the *elimination of small part of source code*, i.e., *MPLA* faults that belong to the *ALG* class, the *substitution of variable assignment with an expression*, i.e., *MVAE* faults that belong to the *ASG* class, and the *wrong use of a variable in a parameter of a function call*, i.e., *WPFV* faults that belong to the *INT* class, led to obtaining a `NULL` value as value returned by a function or as value of a variable checked at runtime, which has been reported by the MUT with the same type of notification, i.e., `"Assertion ‘<<variable name>> != NULL’ failed"` that belongs to the type *e4-AC*. Moreover, in all the ODC classes *e2-AC* errors are the most reported ones by the MUT. It should be noted that *CHK* faults led to a different error behavior with respect to the other fault type. However, a small number of samples have been collected for this type of fault. Therefore, no conclusions can be drawn for *CHK* class.

An extract of the obtained dataset has been submitted to a classifier in order to measure the *Error Determination Degree* of the assertion checking with respect to the ODC fault class, i.e., the ability of its error notifications to suggest what is the ODC class of the fault that have led to those error notifications. The extracted dataset contains (i) the ODC class of the injected fault and (ii) the number of error notifications generated by the assertion checking for each error type of its inferred error model, for each fault injection experiment

Table 5.5: Prediction results for MUT_2 (k-fold cross-validation: Random Forest and k=30).

class	***correct classification (%)***	***incorrect classification (%)***
ODC fault class	56.30%	43.70%
fault type	42.12%	57.88%
failure type	99.87%	0.13%

where at least one error notification has been generated by the MUT. This dataset has been submitted to the *Random Forest* classifier; the numbers of error notifications for each type are used as features of the classification, while the ODC fault class is used as the *class* to predict. A *K-fold cross-validation* has been conducted, with *K=30*. The first row of Table 5.5 reports the results of the classification. The considered classifier was able to predict the ODC type of a fault from the error notifications the fault have led to with a not very high accuracy; in fact, the percentage of correct classification, i.e., the considered *Error Determination Degree*, is equal to 56.30%. This result confirm the finding inferred from Figure 5.14. Indeed, the low variability of the error behavior inferred by means of the MUT_2 at varying the ODC class led to the poor prediction performance of the classifier.

The ODC fault class has been broken down by fault type in order to understand the error behavior at varying the fault types. Figure 5.15 shows the percentage of **errors** that are reported by MUT_2 by error and fault type. Percentage of the errors reported by the considered MUT by error and fault type can be observed in Table 4.12. It can be noted that **the percentage of errors reported by the MUT for each error type changes at varying the fault type in SUT_1**. For example, the error type *e1-AC*, i.e., *data type errors*, has been reported with different percentages at varying the fault type, e.g., at least an error notification of this type has been reported in 2.50% of experiments where

a *MFC* fault has been injected as well as in 44.44% of experiments where a *WAEP* fault has been injected. However, *MLPA*, *MVAE* and *WPFV* faults exposed a very similar error behavior. This finding is strictly related to the one obtained for the ODC classes. Indeed, *MLPA*, *MVAE* and *WPFV* are fault types that belongs to *ALG*, *ASG* and *INT* class, respectively, and they are the ones that occurred more often respect than the other types in the related class, as can be seen in Table 4.9. Therefore, their error behaviors influence the error behavior of the ODC class they belong to. In particular, as previously discussed, they tended to generate the same errors, which are reported by MUT_2 with error notification of the same type. In addition, **given a fault type, in almost all cases the percentage of reported errors strongly changes at varying the error type**. For example, for *MFC* faults the error type *e2-AC* is reported more than the other types as well as the error type

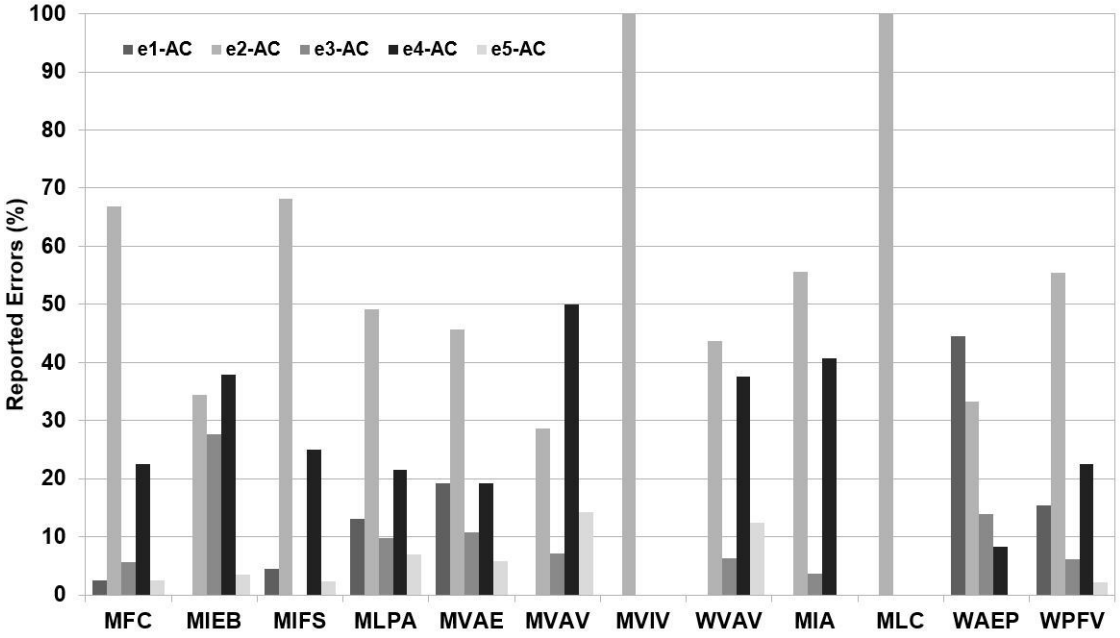

Figure 5.15: Percentage of reported errors by cluster and fault type for MUT_2.

e4-AC for *MIEB* faults. It should be noted that a limited number of samples have been collected for some types of fault, i.e., *MVIV*, *MVAV*, *WVAV*, and *MLC*, as showed in Table 4.9 (AC column). Thus, no conclusions can be drawn for these types of fault.

An extract of the obtained dataset has been submitted to the *Random Forest* classifier in order to measure the *Error Determination Degree* of the assertion checking with respect to the fault type, i.e., the ability of its error notifications to suggest what is the type of the fault that have led to those error notifications. The extracted dataset contains (i) the type of the injected fault and (ii) the number of error notifications generated by the assertion checking for each error type of its inferred error model, for each fault injection experiment where at least one error notification has been generated by the MUT. The numbers of error notifications for each type are used as features of the classification, while the fault type is udes as *class* to predict. A *K-fold cross-validation* has been conducted, with *K=30*. The second row of Table 5.5 reports the results of the classification. Despite the more variability exposed by the error behavior at varying the fault type respect than the case where the faults are grouped into ODC class, the considered classifier was able to predict the fault type from the error notifications the fault have led to with a not very high accuracy; in fact, the percentage of correct classification, i.e., the considered *Error Determination Degree*, is equal to 42.12%, which is even worse than the one obtained in the case where the ODC class has been considered as class to predict.

Figure 5.16 shows the breakdown of the errors reported by MUT_2 in the SUT_1 by failure type, i.e., the failure that the reported error(s) led to, and error type. Percentage of the errors reported by the considered MUT by error and failure type can be observed in Table

4.15. It can be noted that **the percentage of errors reported by the MUT changes at varying the error types in the case of *CRASH* failures in SUT$_1$**, which are the only type of failures reported by the MUT$_2$. For example, the percentage of experiments where at least an error notification is raised ranges from 5.24%, for errors of type *e5-AC*, i.e., *data size error*, to 50.94%, for errors of type *e2-AC*. It should be noted that only error of type *e2-AC* and *e3-AC*, i.e., *forced assertion execution*, led to false positives whit respect to the failures.

An extract of the obtained dataset has been submitted to the *Random Forest* classifier in order to measure the *Error Determination Degree* of the assertion checking with respect to the failure type, i.e., the ability of its error notifications to suggest what is the type of failure occurred in the system as consequence of the error(s) behind those notifications. The extracted dataset contains (i) the failure occurred in the system and (ii) the number of error notifications generated by the assertion checking for each error type of its inferred

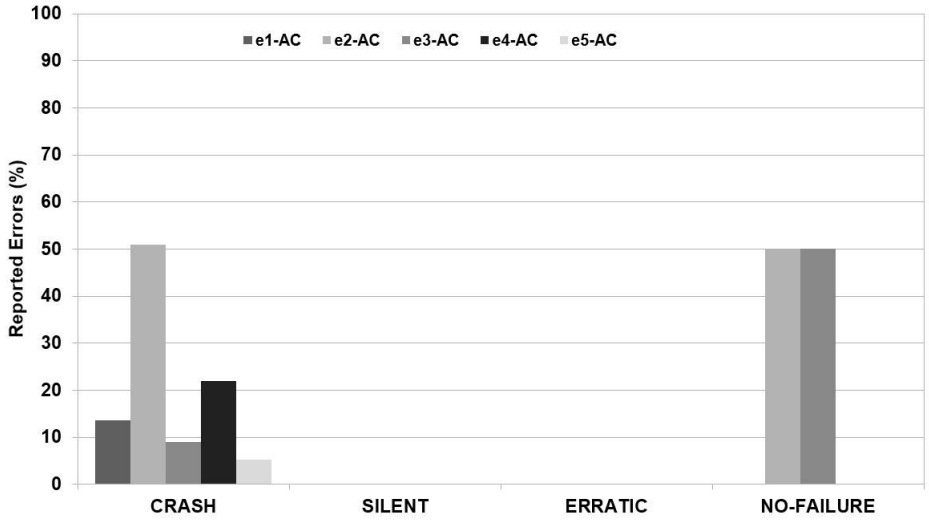

Figure 5.16: Percentage of reported errors by cluster and failure type for MUT$_2$.

error model, for each fault injection experiment where at least one error notification has been generated by the MUT. The numbers of error notifications for each type are used as features of the classification, while the failure type is used as *class* to predict. A *K-fold cross-validation* has been conducted, with *K=30*. The third row of Table 5.5 reports the results of the classification. The classifier was able to predict the failure type from the error notifications with a very high accuracy; in fact, the percentage of correct classification, i.e., the considered *Error Determination Degree*, is equal to 98.87%. This result confirms the finding inferred from Figure 5.16. Indeed, the high variability of the error behavior inferred by means of the MUT_2 at varying the failure type led to the very good prediction performance of the classifier.

Based on these findings, it should be noted that also assertion checking can potentially allow the determination of the failure type occurred in the communication middleware from the obtained error notifications, while nothing can be said about the ODC class and type of the fault that has been injected.

5.2.4 Error Propagation Reportability

The propagation paths of the errors raised in SUT_1 during the experimental campaign, as consequences of the injected faults, have been inferred by means of the error notifications generated by MUT_2 in the SUT. As a reminder, the error notifications generated by assertion checking in the SUT_1 provides the component, and also the function, that has generated the notification. Therefore, the knowledge of the component, and also of the function, where the faults have been injected (both provided by the tool used for the fault injection), along

with the knowledge of the source component and function of each error notification, allowed to build the error propagation graphs that summarize the error propagation phenomena obtained during the experimental campaign.

Figure 5.17 shows the directed graph that summarizes the major error propagation paths through the components of the SUT_1, which are generated as a consequence of *ALG* faults. Noteworthy, the number of faults considered in the graph refers to the number of faults that have led to at least an error notification to be raised by assertion checking. It can be observed **that most of *MFC* and *MLPA* faults led to errors that have not been reported by the assertion checking in the *kernel***. For example, 71.25% and 51.02% of *MFC* and *MLPA* faults, respectively, did not lead to an error notification in the *kernel*. On the other hand, 77.27% and 65.52% of *MIFS* and *MIEB* fault, respectively, led to an error notification in the *kernel*. In particular, 49.38% and 36.55% of *MFC* and *MLPA* faults, respectively, have led to at least an error that has generated at least an error notification in the *database* component of the SUT_1; while 20.63% and 12.28% of *MFC* and *MLPA* faults, respectively, have led to at least an error that has generated at least an error notification in the *ddsi2* component of the SUT_1.

This finding is also confirmed by the proposed *Error Propagation Reportability*, which exhibits a value of 36.64%, i.e., $(336/917) \cdot 100$ (last cell of first row in Table 5.6), that suggests the low ability of MUT_2 at reporting the propagation of the errors generated as consequence of *ALG* faults. This suggest that there is the need to improve the error detection ability of the assertion checking in the *kernel* by adding some *EDMs*. A closer look into the error notifications generated by the MUT in the SUT's components can allow

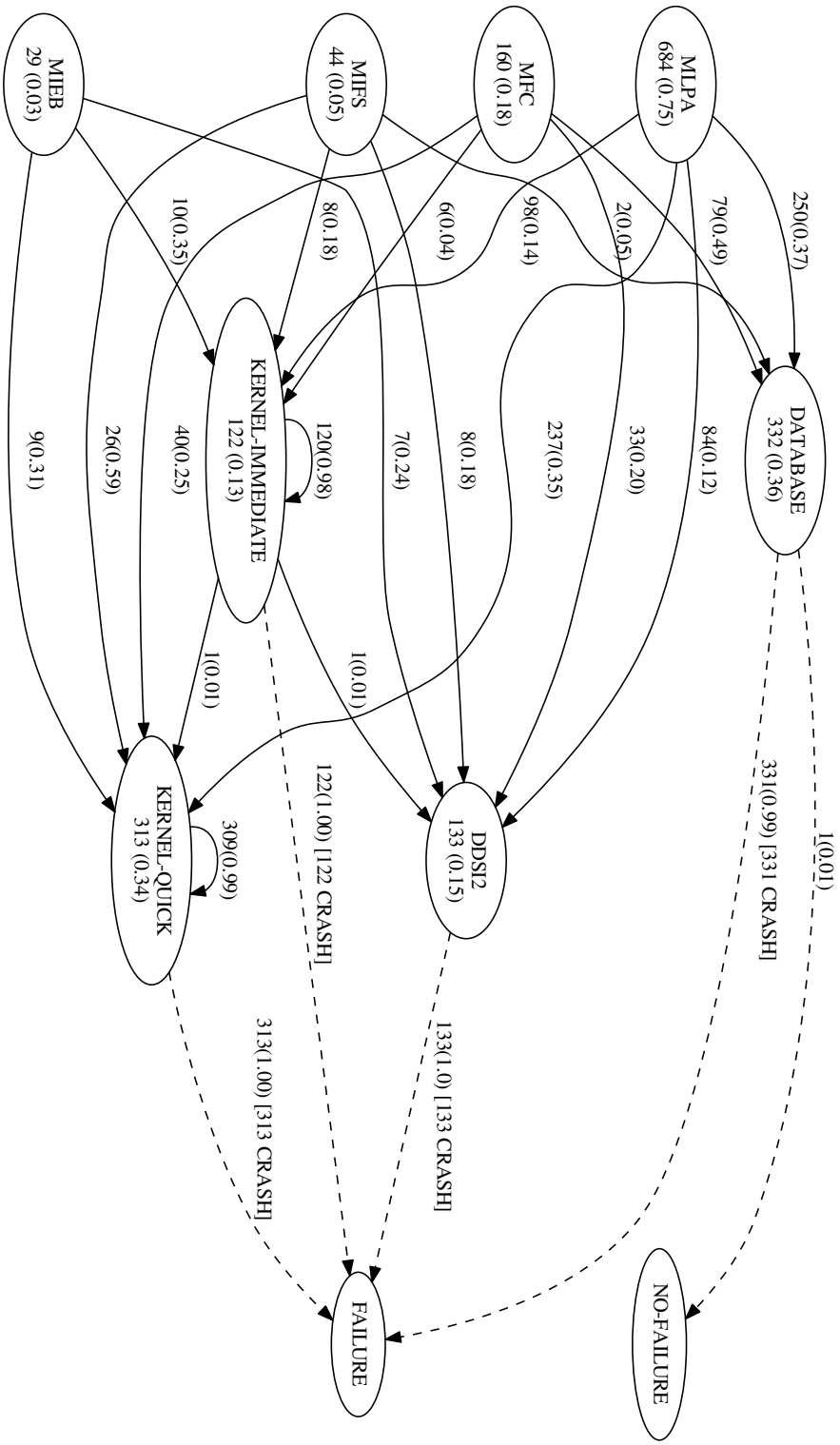

Figure 5.17: Error propagation graph for ALG faults (MUT_2).

Table 5.6: Error Propagation Reportability (EPR) of MUT_2 with respect to *ALG* and *ASG* faults.

ODC	***# experiments with at least an error notification***	***# experiments with at least an error notification generated in the kernel***	***EPR (%)***
ALG	917	336	36.64
ASG	397	250	62.97

to understand the type of error the EDM have to consider. For example, a closer look into the error notifications generated by the *database* during the experiments, where no other components have reported an error, allowed to understand that most of them belong to the *e2-AC* type. Therefore, a potential EDM into the kernel have to consider this kind of error, e.g., by inserting a number of assertions that checks the values returned by functions in order to detect unexpected results. Another findings that can be inferred from Figure 5.17 is that **a limited number of errors have been reported in the function where the fault has been injected**, as shown by the low probability of the *kernel-immediate* node, i.e., 0.13. In addition, **almost all the errors reported by assertion checking in the kernel are not reported in any other components**, as shown by the high probability of the self-loop of the nodes *kernel-immediate* and *kernel-quick*, i.e., 0.98 and 0.99, respectively.

Figure 5.17 also shows that **errors reported by assertion checking in different system components have led to same type of failure in the SUT, i.e., *CRASH* failures**. For example, in all the components that are shown in Figure 5.17 the only type of reported failure is *CRASH*. This is an expected result since, as seen in 5.2.2, assertion checking is able to report only *CRASH* failures in the SUT_1. Based on this findings,

practitioners can find potential positions for *ERMs*. For example, since almost all the errors that propagated to *ddsi2* component of the SUT_1 were not reported by any other component, and all these errors led to a *CRASH* failure, practitioners can decide to put an ERM in this component. A closer look into the error notifications generated by assertion checking in this component allowed to understand that most of them are of type *e4-AC*; therefore, it could be useful to add an ERM in the *ddsi2* component that tries to avoid that `NULL` *value error* can further propagate into the system.

Figure 5.18 shows the directed graph that summarizes the major error propagation paths through the components of the SUT_1, which are generated as a consequence of *ASG* faults. Again, the number of faults considered in the graph refers to the number of faults that have led to at least an error notification to be raised by event logging. It can be observed that **more than half of the errors reported by the assertion checking have been reported by the *kernel* component** of the SUT_1. In fact, 62.91% of *MVAE* faults, i.e., the most recurrent fault type in the *ASG* class, have led to at least an error that has generated at least an error notification in a the *kernel*. On the other hand, 23.35% of *MVAE* faults have led to at least an error that has generated at least an error notification in the *database* component of the SUT_1.

This finding is also confirmed by the proposed *Error Propagation Reportability*, which exhibits a value of 62.97%, i.e., $(250/397) \cdot 100$ (last cell of last row in Table 5.6), which means that 62.97% of the error reported by the assertion checking are reported by the *kernel* component. Noteworthy, EPR value exhibited by the MUT_2 for *ASG* class is greater than the one obtained for the *ALG*, which means that for *ASG* class this MUT exposed better

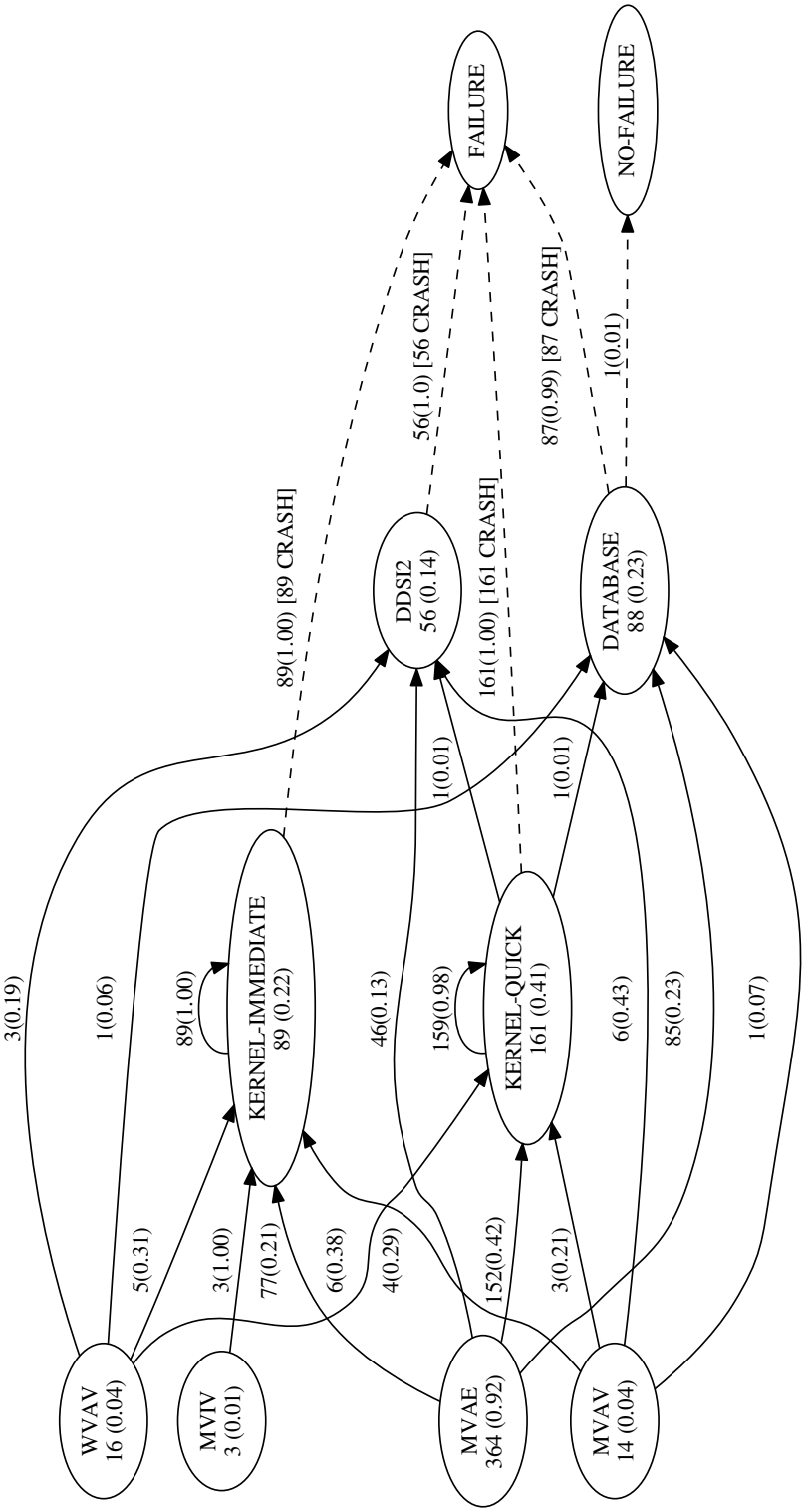

Figure 5.18: Error propagation graph for ASG faults (MUT_2).

performance at reporting error propagation. However, also for *ASG* there is the need to further improve the error detection ability of the assertion checking in the *kernel* by adding some *EDMs*. A closer look into the error notifications generated by the MUT in the SUT's components can allow to understand the type of error the EDM have to consider. For example, the error notifications generated by the *database* during the experiments where no other components have reported an error, belong to the *e4-AC* type, as in the case of *ALG* class. This similarity is an expected result since in Section 5.2.3 has been found that *ALG* and *ASG* exposed a very similar error behavior. In addition, **almost all the errors reported by assertion checking in the kernel are not reported in any other components**, as shown by the high probability of the self-loop of the nodes *kernel-immediate* and *kernel-quick*, i.e., 1.00 and 0.98, respectively.

Figure 5.18 also shows that also for the *ASG* class the **errors reported by assertion checking in different system components have led to same type of failure in the SUT, i.e., *CRASH* failures**, as in the case of *ALG* class. In fact, in all the components that are shown in Figure 5.17 the only type of reported failure is *CRASH*. Again, this is an expected result since, as seen in 5.2.2, assertion checking is able to report only *CRASH* failures in the SUT_1. Regarding the placement of ERMs, same considerations made for *ALG* apply here, since the classes exposed a very similar error behavior.

The findings obtained from the analysis of assertion checking provide part of the answer to the **Research Question 1**, i.e., **RQ1**, the **Research Question 2**, i.e., **RQ2**, and the **Research Question 3**, i.e., **RQ3**. In fact, the analysis of the *Error Propagation*

Reportability allowed to understand that assertion checking can be used both to characterize the error behavior of the considered SUT (*RQ1*) and to provide insights about the placement of *EDMs* and *ERMs* (*RQ2*). In addition, this analysis also allows to understand that the Error Propagation Reporting Ability of the assertion checking changes when different ODC fault class are considered (*RQ3).* Differently, the analysis of *Recall* and *Precision*, of *Failure coverage* and of *Error Determination Degree* exhibited by assertion checking during the experiments allowed to infer how its Recall and Precision values change between the SUTs, how its failure reporting ability changes at varying the failure and fault type, and also the values exhibited by its EDD with respect to the fault type, ODC class and failure type (*RQ3*).

5.3 Rule-Based Logging Analysis

The effectiveness of the MUT_3, i.e., rule-based logging, has been evaluated by analyzing the data generated by the MUT during the conducted experimental campaign, which are summarized in the tables in Section 4.6. The data have allowed the measurement of the metrics defined in the proposed methodology, i.e., Recall (R), Precision (P), Failure Coverage (FC), Error Determination Degree (EDD) and Error Propagation Reportability (EPR). It should be noted that, as for the other MUTs, Recall, Precision and Failure Coverage of the MUT have been evaluated on both the considered target SUTs. Differently, the Error Determination Degree and the Error Propagation Reportability have been evaluated only for the MUTs implemented by the SUT_1.

5.3.1 Recall and Precision

Figure 5.19 and Figure 5.20 show the percentage of reported errors of the MUT_3, i.e., the percentage of experiments where the MUT has generated at least one error notification, with respect to the failure and non-failure experiments, i.e., the experiments where the injected fault led to a failure in the considered SUT or not, respectively, conducted during the experimental campaign for SUT_1 and SUT_2, respectively.

Figure 5.19 shows that MUT_3 has reported at least an error notification for a high percentage of *SILENT* and *CRASH* failures occurred in the SUT_1, i.e., 70.10% and 67.98%, respectively, while reported at least an error notification for a limited percentage of *ERRATIC* failures, i.e., 5.70%. However, most of the errors reported by the MUT_1 are the ones that led to a *CRASH* failure in the SUT_1. Indeed, *CRASH* failures are the most occurred failures in the SUT_1, i.e., 2,639 out of 3,159 failures occurred in SUT_1 (as reported in Table 4.4), which are followed by the *SILENT* and *ERRATIC* failures that account for 204 and 316, respectively.

Differently, Figure 5.20 shows that MUT_3 has reported at least an error notification for a high percentage of *SILENT* and *ERRATIC* failures occurred in the SUT_2, i.e., 93.14% and 68.10%, respectively, while reported at least an error notification for 46.34% of *CRASH* failures. However, in absolute terms, the numbers of errors reported by the MUT_3 in the

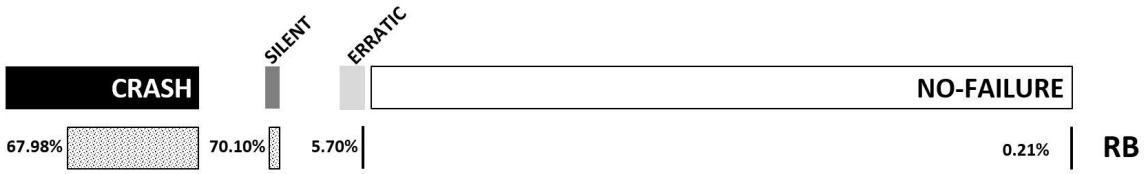

Figure 5.19: Percentage of reported errors of MUT_3 by failure type for SUT_1.

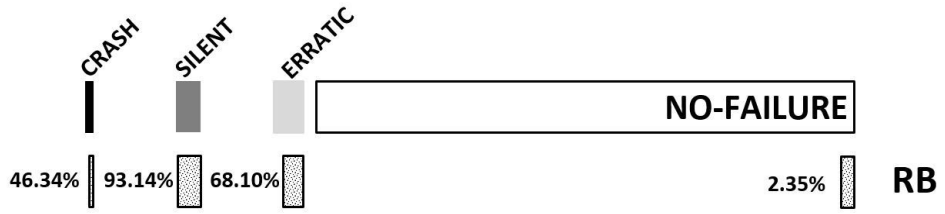

Figure 5.20: Percentage of reported errors of MUT_3 by failure type for SUT_2.

SUT_2 are quite similar for *SILENT* and *ERRATIC* failures. Indeed, MUT_3 has generated at least an error notification in 258 experiments where a *SILENT* occurred in the SUT_2, while has generated at least an error notification in 222 experiments where an *ERRATIC* occurred. In addition, Figure 5.19 and Figure 5.20 show also that MUT_3 has generated error notifications also when no failures occurred in both the SUTs. These error notifications represent False Positives with respect to the failures.

Table 5.7 reports FP, FN, TP, P and R for the MUT_3 for each SUT. It can be noted that the *precision* is very close to 1 in SUT_1 but not in SUT_2. Therefore, almost all the failures reported by the MUT are actual failures occurred in the SUT_1. The number of FNs is 1,204 out of total 3,159 failures in the SUT_1 and 167 out of total 685 failures occurred in the SUT_2. These findings suggest that rule-based logging might miss some failures, especially in the SUT_1. The rightmost column of Table 5.7 reports the recall of the MUT.

It should be noted that the density of the MUT, which is reported by Table 4.2 (i.e.,

Table 5.7: False Positive (`FP`), False Negative (`FN`), True Positive (`TP`), Precision and Recall of MUT_3 for each SUT.

	`FP`	`FN`	`TP`	***Precision***	***Recall***
MW-RB	20	1,204	1,955	0.990	0.619
AM-RB	139	167	518	0.788	0.756

RB), is potentially related to the value of recall. For example, the different values of recall measured for the MW-RB and AM-RB, i.e., 0.619 and 0.765, respectively, are likely caused by the different density of the rule-based logging instructions in the SUTs, i.e., 0.36% and 8.59% out of the total number of rule-based logging instructions placed in the source code of SUT_1 and SUT_2, respectively.

5.3.2 Failure Coverage

The overall recall has been broken down by failure type in each SUT. Figure 5.21 shows the percentage of reported failures of MUT_3 by failure type and SUT. It can be noted that **the reporting ability of the MUT_3 changes significantly across the SUTs**. Moreover, **the MUT might show a different ability at reporting the same type of failure in different SUTs**. In fact, the coverage of rule-based logging ranges from a minimum of 5.70%, i.e., MW-RB (*ERRATIC* failures), to a maximum of 93.14%, i.e., AM-RB (*SILENT* failures).

The overall recall has been also broken down by fault type in each SUT. Figure 5.22

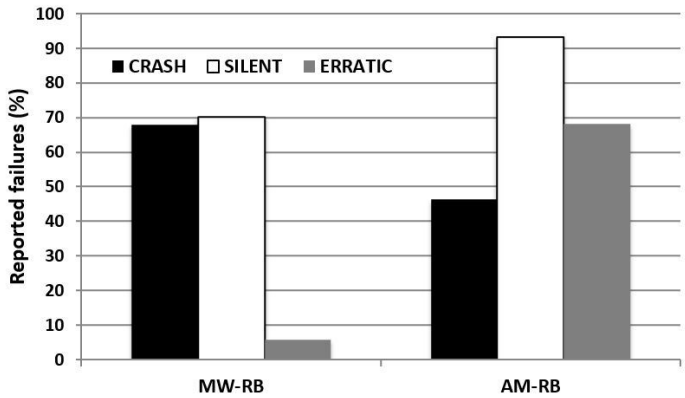

Figure 5.21: Percentage of reported failure of MUT_3 by type and case study.

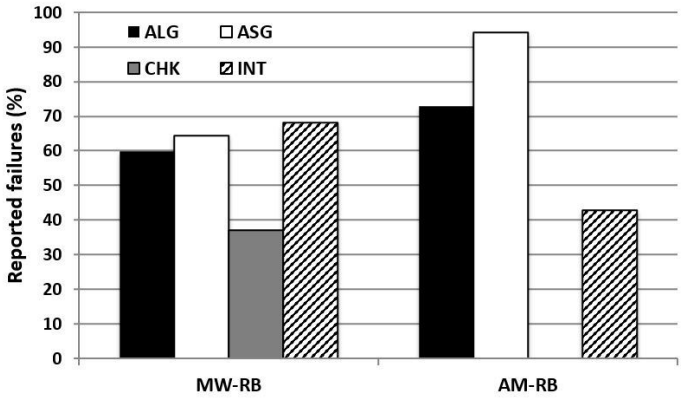

Figure 5.22: Percentage of reported failure of MUT_3 by fault type and case study.

shows the percentage of activated **faults** that are reported by MUT_3 in both the SUTs. It can be noted that **the reporting ability by fault type of the MUT changes slightly across the SUTs**. Moreover, **the MUT might show a different ability at reporting the same type of fault in different SUTs**. In fact, the reported failure by fault type of rule-based logging ranges from a minimum of 37.04%, i.e., MW-RB (*CHK* faults), to a maximum of 68.02%, i.e., MW-RB (*INT* faults), in the SUT_1, and from a minimum of 0.00%, i.e., AM-RB (*CHK* faults), to a maximum of 94.31%, i.e., AM-RB (*ASG* faults), in the SUT_2.

5.3.3 Error Determination Degree

The error behavior inferred by the error notifications of the MUT_3 has been evaluated by considering the error model extracted during the error clustering process.

Figure 5.23 shows the breakdown of the errors reported by MUT_3 in the SUT_1 by error type, i.e., the error types that belong to the inferred error model reported in Table 4.6, and fault ODC class. Percentage of the errors reported by the considered MUT by error type

and ODC class can be observed in the *total ALG*, *total ASG*, *total CHK* and *total INT* rows of Table 4.13. It can be noted that **the percentage of errors reported by the MUT for each error type slightly changes at varying the ODC class in SUT$_1$**. For example, considering the error type *e8-RB*, i.e., *kernel module errors*, the MUT has generated at least an error notification of this type in 79.49%, 81.15%, 75.00% and 78.13% of *ALG*, *ASG*, *CHK* and *INT* experiments, respectively, i.e., experiments where only an *ALG*, an *ASG*, *CHK* or an *INT* fault has been injected in the SUT. Similarly, the MUT has generated at least an error notification of type *e1-RB*, i.e., *writer module errors*, in 38.11%, 35.52%, 40.00% and 27.50% of *ALG*, *ASG*, *CHK* and *INT* experiments, respectively. All the ODC classes exhibited a very similar error behavior. A closer look into the error notifications obtained from these ODC classes and the source code of the related injected faults allowed to understand that often faults of different classes led to error notifications of the same type, that is from the same module of the kernel in the case of rule-based logging.

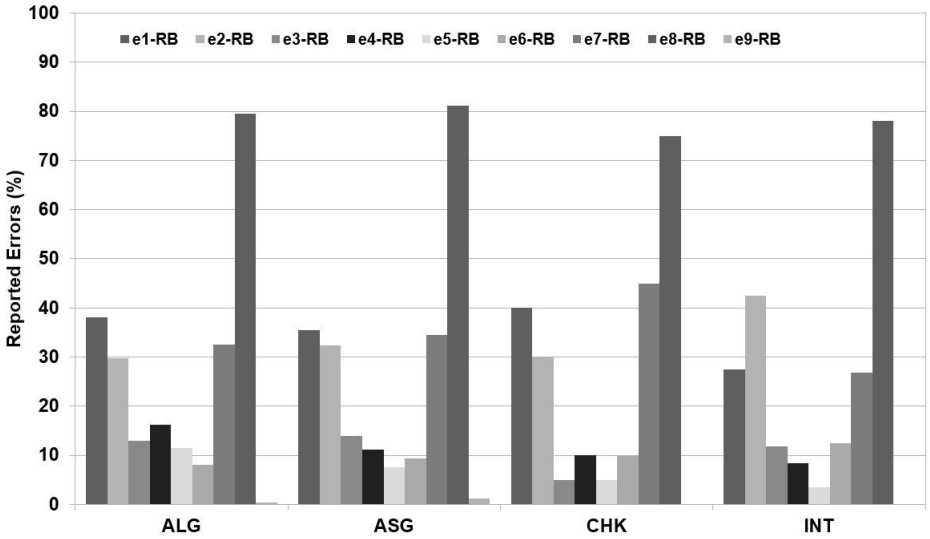

Figure 5.23: Percentage of reported errors by cluster and fault ODC type for MUT$_3$.

Table 5.8: Prediction results for MUT_3 (k-fold cross-validation: Random Forest and k=30).

class	***correct classification (%)***	***incorrect classification (%)***
ODC fault class	54.70%	45.30%
fault type	44.90%	55.10%
failure type	91.97%	8.03%

For example, the analysis of the error notifications revealed that different types of fault, injected in different location of the kernel, generated errors that have propagated often to the core module of the kernel, leading this module to generate an error notification, which is than part of the *e8-RB* cluster. Indeed, it should be noted that *e9-RB* errors are the most reported ones by the MUT in all the ODC classes.

An extract of the obtained dataset has been submitted to a classifier in order to measure the *Error Determination Degree* of the rule-based logging with respect to the ODC fault class, i.e., the ability of its error notifications to suggest what is the ODC class of the fault that have led to those error notifications. The extracted dataset contains (i) the ODC class of the injected fault and (ii) the number of error notifications generated by the rule-based logging for each error type of its inferred error model, for each fault injection experiment where at least one error notification has been generated by the MUT. This dataset has been submitted to the *Random Forest* classifier; the numbers of error notifications for each type are used as features of the classification, while the ODC fault class is used as the *class* to predict. A *K-fold cross-validation* has been conducted, with *K=30*. The first row of Table 5.8 reports the results of the classification. The considered classifier was able to predict the ODC type of a fault from the error notifications the fault have led to with a not very high accuracy; in fact, the percentage of correct classification, i.e., the considered

Error Determination Degree, is equal to 54.70%. This result confirm the finding inferred from Figure 5.23. Indeed, the low variability of the error behavior inferred by means of the MUT_3 at varying the ODC class led to the poor prediction performance of the classifier.

The ODC fault class has been broken down by fault type in order to understand the error behavior at varying the fault types. Figure 5.24 shows the percentage of **errors** that are reported by MUT_3 by error and fault type. Percentage of the errors reported by the considered MUT by error and fault type can be observed in Table 4.13. It can be noted that **the percentage of errors reported by the MUT for each error type slightly changes at varying the fault type in SUT$_1$**. For example, the error type *e3-RB*, i.e., *subscriber module errors*, has been reported with different percentages at varying the fault type, e.g., at least an error notification of this type has been reported in 23.53% of

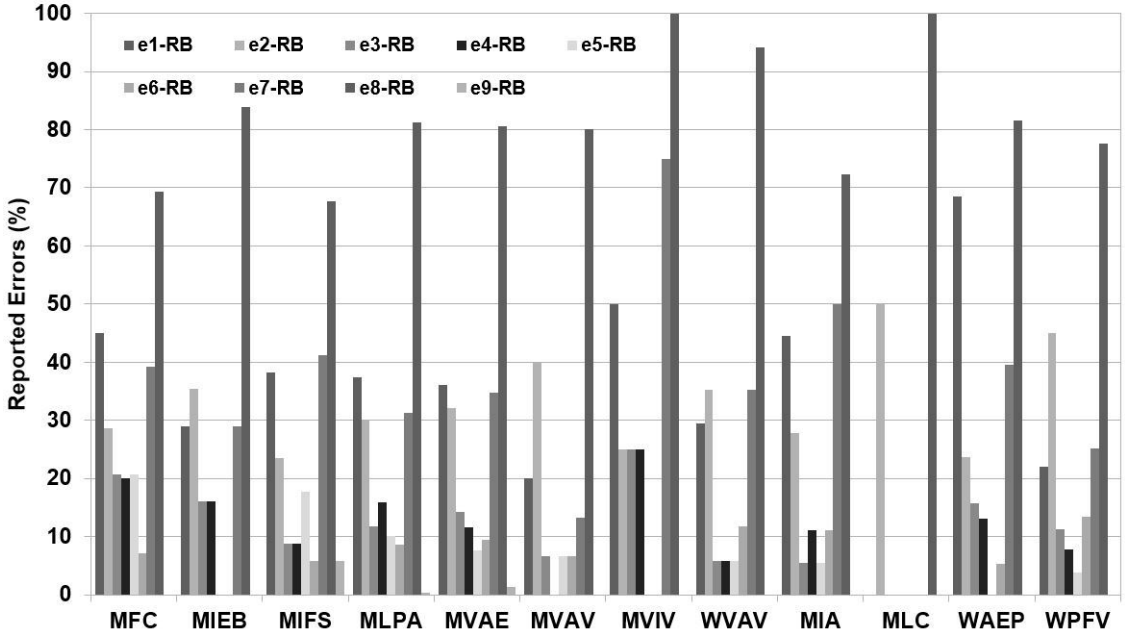

Figure 5.24: Percentage of reported errors by cluster and fault type for MUT_3.

experiments where a *MIFS* fault has been injected as well as in 45.04% of experiments where a *WPFV* fault has been injected. However, many fault types exposed a very similar error behavior. This finding is strictly related to the one obtained for the ODC classes, where almost all the ODC classes exposed a very similar behavior. In addition, **given a fault type, in almost all cases the percentage of reported errors strongly changes at varying the error type**. For example, for *MFC* faults the error type *e6-RB*, i.e., topic module error, is reported in 7.14% of experiments, while the error *e8-RB* is reported in 69.29% of experiments. It should be noted that a limited number of samples have been collected for some types of fault, i.e., *MVIV*, *MVAV*, *WVAV*, *MIA*, and *MLC*, as showed in Table 4.9 (RB column). Thus, no conclusions can be drawn for these types of fault.

An extract of the obtained dataset has been submitted to the *Random Forest* classifier in order to measure the *Error Determination Degree* of the rule-based logging with respect to the fault type, i.e., the ability of its error notifications to suggest what is the type of the fault that have led to those error notifications. The extracted dataset contains (i) the type of the injected fault and (ii) the number of error notifications generated by the rule-based logging for each error type of its inferred error model, for each fault injection experiment where at least one error notification has been generated by the MUT. The numbers of error notifications for each type are used as features of the classification, while the fault type is udes as *class* to predict. A *K-fold cross-validation* has been conducted, with *K=30*. The second row of Table 5.8 reports the results of the classification. The considered classifier was able to predict the fault type from the error notifications the fault have led to with a not very high accuracy; in fact, the percentage of correct classification, i.e., the considered *Error*

Determination Degree, is equal to 44.90%, which is even worse than the one obtained in the case where the ODC class has been considered as class to predict. This result confirm the finding inferred from Figure 5.24. Indeed, the low variability of the error behavior inferred by means of the MUT_3 at varying the fault type led to the poor prediction performance of the classifier.

Figure 5.25 shows the breakdown of the errors reported by MUT_3 in the SUT_1 by failure type, i.e., the failure that the reported error(s) led to, and error type. Percentage of the errors reported by the considered MUT by error and failure type can be observed in Table 4.16. It can be noted that **the percentage of errors reported by the MUT for almost all error types strongly changes at varying the failure type in SUT_1**. For example, the percentage of experiments where at least an error notification of type *e8-RB* is raised ranges from 5.56%, for experiments where an ERRATIC failures occurred in the system, to 81.22%, for experiments where a *CRASH* failure occurred in the system. Similarly, the

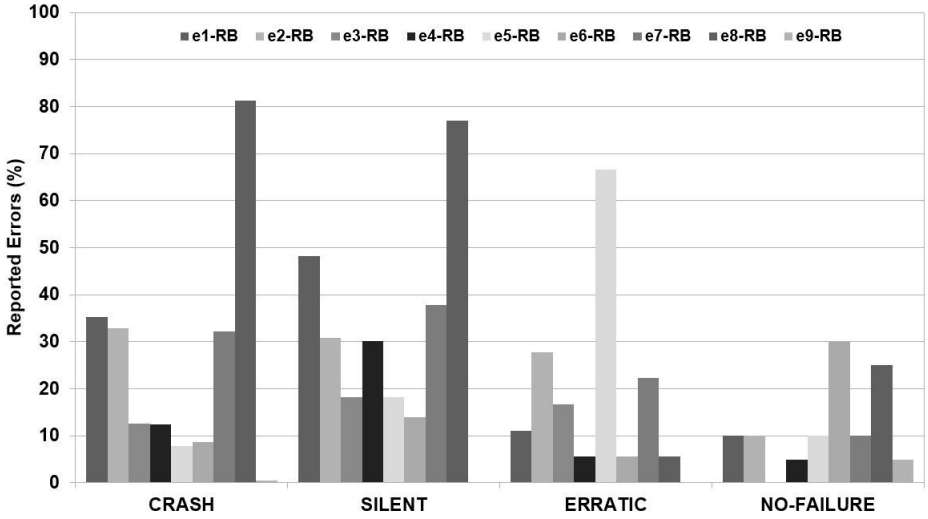

Figure 5.25: Percentage of reported errors by cluster and failure type for MUT_3.

percentage of experiments where at least an error notification of type *e5-RB*, i.e., *network module error* ranges from 7.75%, for experiments where a *CRASH* failures occurred in the system, to 66.67%, for experiments where an *ERRATIC* failure occurred in the system.

An extract of the obtained dataset has been submitted to the *Random Forest* classifier in order to measure the *Error Determination Degree* of the rule-based logging with respect to the failure type, i.e., the ability of its error notifications to suggest what is the type of failure occurred in the system as consequence of the error(s) behind those notifications. The extracted dataset contains (i) the failure occurred in the system and (ii) the number of error notifications generated by the rule-based logging for each error type of its inferred error model, for each fault injection experiment where at least one error notification has been generated by the MUT. The numbers of error notifications for each type are used as features of the classification, while the failure type is used as *class* to predict. A *K-fold cross-validation* has been conducted, with *K=30*. The third row of Table 5.8 reports the results of the classification. The classifier was able to predict the failure type from the error notifications with a high accuracy; in fact, the percentage of correct classification, i.e., the considered *Error Determination Degree*, is equal to 91.97%. This result confirms the finding inferred from Figure 5.25. Indeed, the high variability of the error behavior inferred by means of the MUT_3 at varying the failure type led to the very good prediction performance of the classifier.

Based on these findings, it should be noted that also rule-based logging can potentially allow the determination of the failure type occurred in the communication middleware from the obtained error notifications, while nothing can be said about the ODC class and type

of the fault that has been injected.

5.3.4 Error Propagation Reportability

The propagation paths of the errors raised in SUT_1 during the experimental campaign, as consequences of the injected faults, have been inferred by means of the error notifications generated by MUT_3 in the SUT. As a reminder, the error notifications generated by rule-based logging in the SUT_1 provides the module of the kernel, and also the function, that has generated the notification. Therefore, the knowledge of the component, and also of the function, where the faults have been injected (both provided by the tool used for the fault injection), along with the knowledge of the source component and function of each error notification, allowed to build non-exhaustive graphs that summarize the error propagation phenomena obtained during the experimental campaign. It should be noted that the MUT_3 is implemented only in the kernel component of the SUT_1. Therefore, only the propagation between the node *kernel-immediate* and *kernel-quick* can be inferred.

Figure 5.26 shows the directed graph that summarizes the error propagation paths that can be inferred form the error notifications provided by the MUT_3 in the SUT_1, which are generated as a consequence of *ALG* faults. Noteworthy, the number of faults considered in the graph refers to the number of faults that have led to at least an error notification to be raised by rule-based logging. It can be observed **that most of errors raised the 4 fault type of the *ALG* class have been reported immediately**, i.e., in the same function where the fault has been injected. In fact, 70.82% of the faults considered in the graph have led to an error notification in the same function where they have been injected.

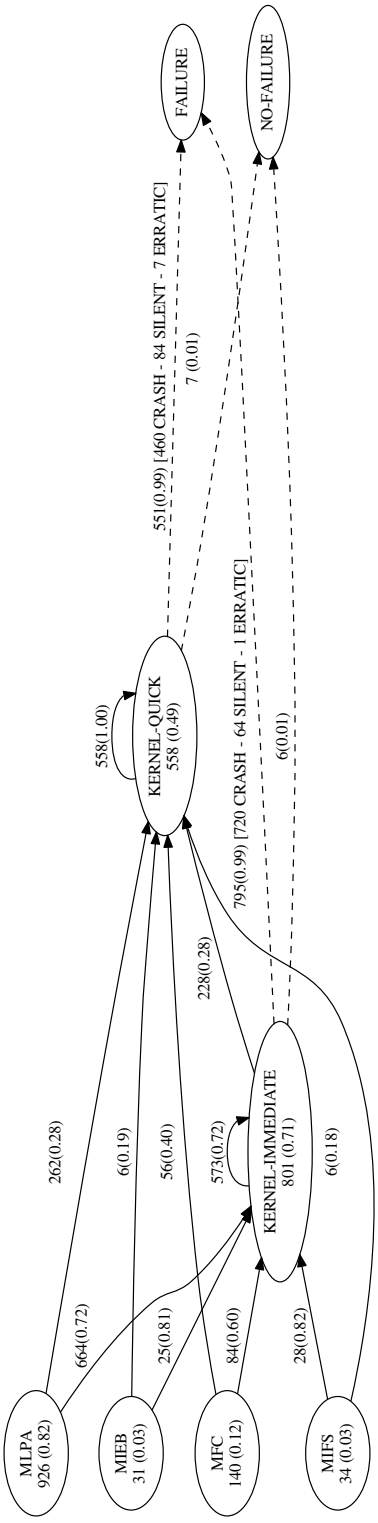

Figure 5.26: Error propagation graph for ALG faults (MUT_3).

In addition, only a small fraction of the errors led to these faults have propagated in the other modules of the kernel, as can be inferred from the high probability of the self loop in the node *kernel-immediate*, i.e., 0.72.

Similar considerations apply also to Figure 5.18, which shows the directed graph that summarizes the error propagation paths that can be inferred form the error notifications provided by the MUT_3 in the SUT_1, which are generated as a consequence of *ASG* faults. Indeed, the Figureshows that 78.37% of the faults considered in the graph have led to an error notification in the same function where they have been injected; while only a small fraction of the errors led to these faults have propagated in the other modules of the kernel, as can be inferred from the high probability of the self loop in the node *kernel-immediate*, i.e., 0.70.

It should be noted that for both the graphs the evaluation of the proposed *Error Propagation Reportability* is not useful since it exposes a value equal to 100.00% in this case (i.e., all the errors notifications of the rule-based logging are generated from the *kernel* component of the SUT_1), which is not realistic because the rule-based logging is not implemented in all the components of the system. However, by considering the obtained perfect value for the EPR and the absence of notifications form other components of the target system, one can infer that there is the need of EDMs in the other components.

Figure 5.26 and Figure 5.27 also show that **errors reported by rule-based logging in the *kernel* of the SUT_1 have led to different failure manifestation in the SUT**. In fact, in both the Figures it can be seen that the errors reported in both the kernel nodes led to all considered types of failure. However, in both the cases, most of the *CRASH*

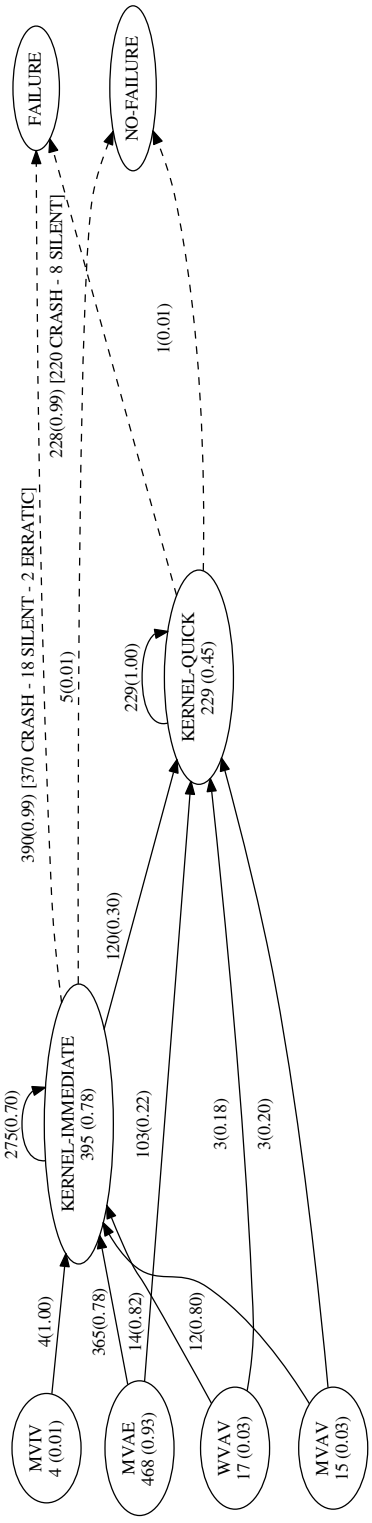

Figure 5.27: Error propagation graph for ASG faults (MUT_3).

failures are reported by the error notifications generated in *kernel-immediate*; while most of the *SILENT* failures are reported by the error notifications generated in *kernel-quick* for the *ALG* class, i.e., Figure 5.26, and in *kernel-immediate* for the *ASG* class, i.e., Figure 5.27.

Based on these findings, practitioners can only define the kernel as potential location for *ERMs*. However, they can obtain more detailed information on where put the ERMs inside the kernel. A closer look into the error notifications generated by rule-based logging allowed to understand what is the type of the most reported error notifications, i.e., the module of the kernel that generates more error notifications with respect the other ones. In particular, it has been found that in both the cases, i.e., *ALG* and *ASG*, most of the error notifications are generated by the core module of the kernel, i.e., the cluster *e8-RB*. Therefore, this module represents a good candidate where to locate an ERM.

It should be noted that the analysis showed here is intentionally conducted to at an higher level, i.e., at component level. However, practitioners can leverage the proposed methodology also to conduct a more fine-grained analysis, reaching the function level. More in details, by leveraging the tracing ability of the rule-based logging, it is possible to recreate the actual error propagation between the modules that compose the kernel. For example, Figure 5.28 shows some of the error propagation paths inside the kernel component, limited to some *MLPA* experiments. It can be noted that it has been possible to recreate the actual error propagation paths between the modules, as shown in the *kernel-QUICK node*, occurred during the considered experiments, allowing to understand how the errors propagated inside the kernel during these experiments. Noteworthy, only the absolute values are

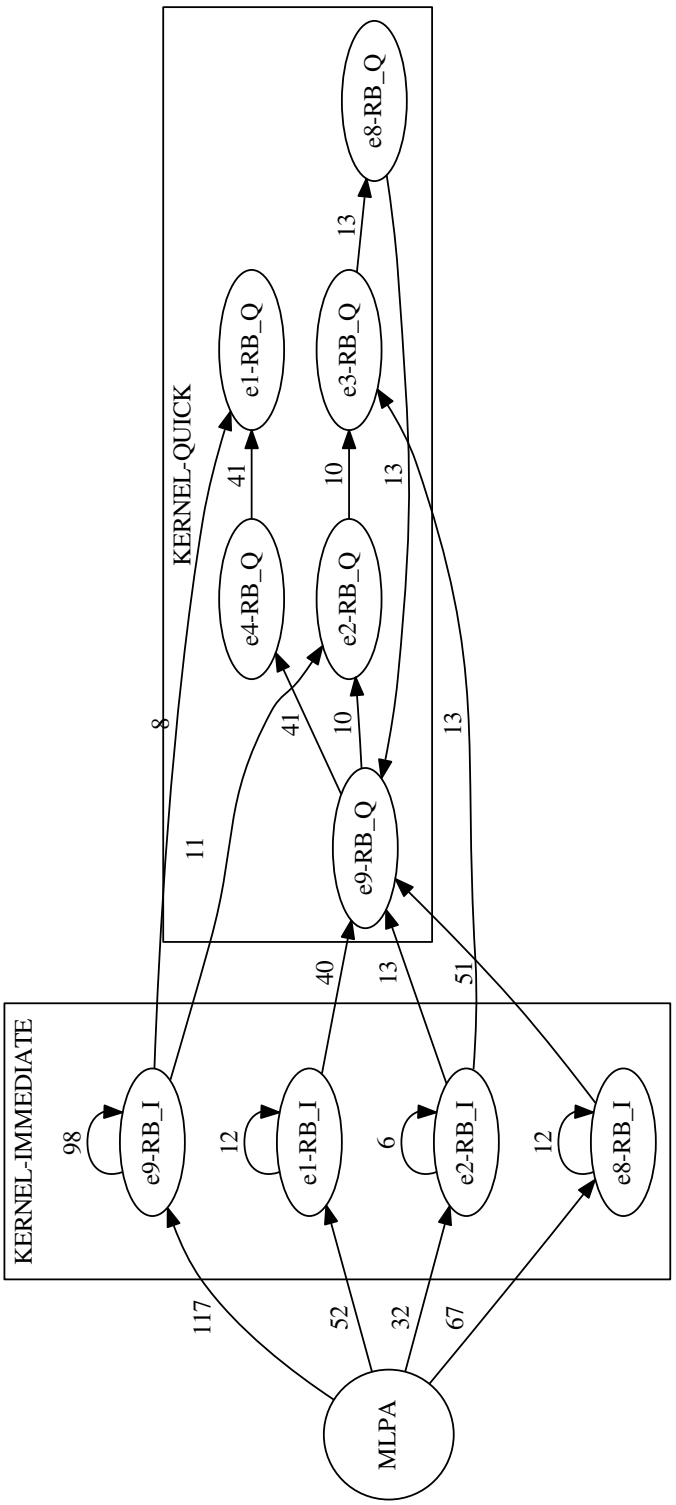

Figure 5.28: Error propagation graph at kernel module-level for some MLPA faults (MUT_3).

shown on the graph due to not comprehensive set of the considered MLPA experiments.

The findings obtained from the analysis of rule-based logging provide part of the answer to the **Research Question 1**, i.e., **RQ1**, the **Research Question 2**, i.e., **RQ2**, and the **Research Question 3**, i.e., **RQ3**. In fact, the analysis of the *Error Propagation Reportability* allowed to understand that rule-based logging can be used to characterize the error behavior of the considered SUT (*RQ1*), as well as that there is the need to improve the rule-based logging in the other components since error notifications are raised only in the faulty component (*RQ2*). Differently, the analysis of *Recall* and *Precision*, of *Failure coverage* and of *Error Determination Degree* exhibited by rule-based logging during the experiments allowed to infer how its Recall and Precision values change between the SUTs, how its failure reporting ability changes at varying the failure and fault type, and also the values exhibited by its EDD with respect to the fault type, ODC class and failure type (*RQ3*).

5.4 Practical Implications and Threats to Validity

The conducted analysis allowed to obtain a number of practical implications to improve the error detection and recovery of the target system. In particular, insights about the placement of *EDM* and *ERM* in the source code of the communication middleware have been provided. For example, the analysis of the *Error Propagation Reportability* of event logging and assertion checking highlighted the need of EDMs in the *kernel* component, which have to consider both the *e4-EL* and *e2-AC* error types, i.e., *Data type errors* and

Unexpected value errors, respectively. In addition, also the need of ERM in the system has been highlighted. Indeed, the analysis of the *Error Propagation Reportability* of event logging revealed that the placement of an ERM, which tries to avoid *e5-EL* errors, i.e., *Main daemon errors*, into the *api*, *spliced* or *user* component can be beneficial to cope with *SILENT* failures, while, in the case of assertion checking, it can be beneficial to insert an ERM, which tries to avoid that *NULL value error*, i.e., *e4-AC*, into the *ddsi2* component can be beneficial to cope with *CRASH* failures.

Regarding the threats to validity, it should be noted that the study relies on the error reported by the considered monitoring techniques, therefore the undetected errors cannot be considered into the analysis. As a result, the propagation graphs have to be considered non-exhaustive since they do not include all the errors occurred into the system, but only the ones reported by the techniques. In addition, there is the possibility that some propagation paths from the faulty component, i.e., where the fault has been injected, to the other ones of the target system are not reported by a monitoring technique because they are not present by design in the target system, e.g., when the component that reports the error works as a detector of the faulty component. Therefore, the proposed *EPR* metric might provide not accurate value in this case. However, it can be successfully used to decide where to place EDMs and ERMs, and, more important, as a comparative metric between MUTs.

Chapter 6

Experimental Results: Comparison of Monitoring Techniques

A comparison of the considered MUTs, i.e., event logging (EL), assertion checking (AC), rule-based logging (RB), has been conducted by comparing the measures obtained from the evaluation metrics presented in Section 3.3. The comparison allowed to understand how the effectiveness of a MUT varies across the different SUTs in terms of failure reporting and dissimilarity of data, as well as to understand how the EDD and EPR change between the MUTs of the SUT_1. Also the combination of different MUTs has been analyzed in order to evaluate the potential benefits that can be achieved by considering multiple MUTs at the same time.

6.1 Comparison of the MUTs

The effectiveness of the MUTs of each SUT have been compared by analyzing the results obtained for each MUT in the analysis described in Chapter 5. The comparison have been conducted by studying the measures obtained for each evaluation metric defined in Section 3.3. In addition, the potential benefits that can be obtained by combining different MUTs have been evaluated in terms of both failure reporting and error propagation reportability.

6.1.1 Recall and Precision

Figure 6.1 and Figure 6.2 show the percentage of reported errors of the MUTs, i.e., the percentage of experiments where the MUTs has generated at least one error notification, with respect to the failure and non-failure experiments, i.e., the experiments where the injected fault led to a failure in the considered SUT or not, respectively, conducted during the experimental campaign for SUT_1 and SUT_2, respectively.

Figure 6.1 shows that MUT_3, i.e., rule-based logging (RB), has been able to report at least an error notification for a percentage of *CRASH* failures occurred in the SUT_1 higher than ones of the other MUTs, i.e., 67.98%, as well as for *SILENT* failures, i.e., 70.10%. Differently, the MUT_1, i.e., event logging (EL), has been able to report at least an error notification for a percentage of *ERRATIC* failures occurred in the SUT_1 higher than ones of the other MUTs, i.e., 20.89%.

Figure 6.2 shows that MUT_3 has reported at least an error notification for a percentage of *SILENT* failures and of *ERRATIC* failures occurred in the SUT_2 higher than the ones of the other MUTs, i.e., 94.13% and 68.10%, respectively. Differently, MUT_1 has reported at least an error notification for a percentage of *CRASH* failures occurred in the SUT_2 higher

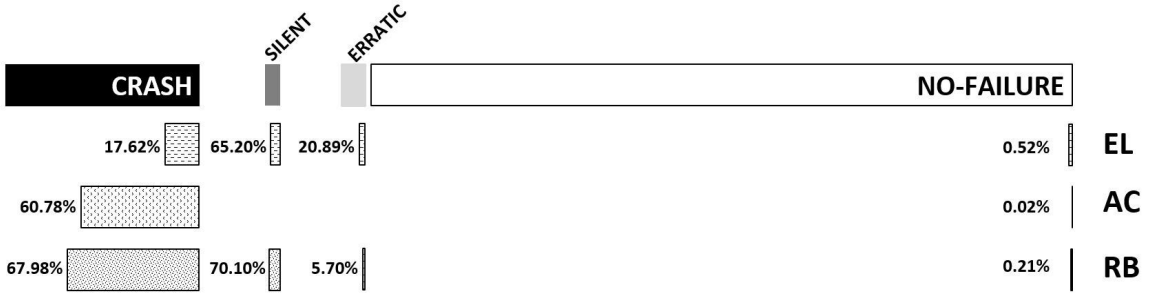

Figure 6.1: Percentage of reported errors of MUTs by failure type for SUT_1.

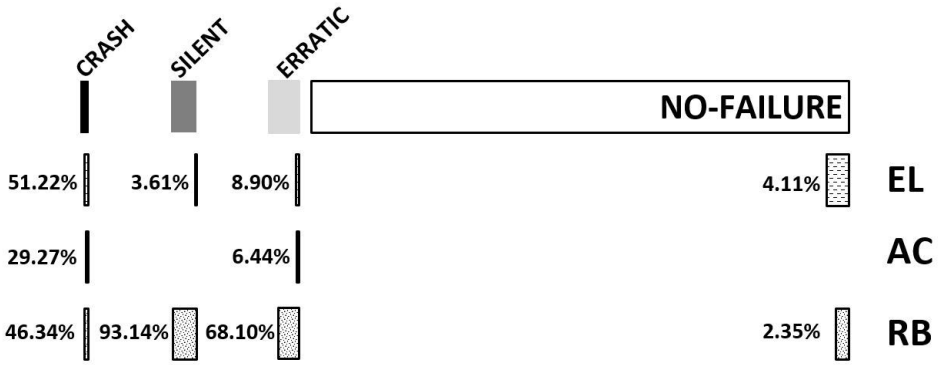

Figure 6.2: Percentage of reported errors of MUTs by failure type for SUT_2.

than the ones of the other MUTs, i.e., 51.22%.

In addition, Figure 6.1 and Figure 6.2 show also that in bothe the SUTs the MUT_1 has generated at least an error notification for a percentage of experiment where no failures occurred in the SUTs higher than the ones of the other MUTs. These error notifications represent False Positives (FPs) in respect to the failures.

Table 6.1 reports FP, FN, TP, P and R for each MUT and each SUT. It can be noted that the *precision* is very close to 1 for all the MUTs implemented by the SUT_1: almost all the failures reported by the MUTs are actual failures occurred in the communication middleware. In the SUT_2, only the assertions exhibit a high *precision* value. In fact, event

Table 6.1: False Positive (`FP`), False Negative (`FN`), True Positive (`TP`), Precision and Recall of each SUT and MUT.

	`FP`	`FN`	`TP`	***Precision***	***Recall***
MW-EL	50	2,495	664	0.930	0.210
MW-AC	2	1,555	1,604	0.999	0.508
MW-RB	20	1,204	1,955	0.990	0.619
AM-EL	243	604	81	0.250	0.118
AM-AC	0	640	45	1.000	0.066
AM-RB	139	167	518	0.788	0.756

(AM-EL) and rule-based (AM-RB) logging generate a relevant number of FPs. The number of FNs in the SUT_1 ranges from 1,204 (i.e., MW-RB) to 2,495 (i.e., MW-EL) out of total 3,159 failures. In the SUT_2 FNs range from 167 (i.e., AM-RB) to 640 (i.e., AM-AC) out of total 685 failures occurred in the system. These findings suggest that a monitoring technique might miss a relevant number of failures. The rightmost column of Table 6.1 reports the recall of the MUTs.

It should be noted that the density of the MUTs, which is reported by Table 4.2, is potentially related to the value of recall/precision. For example, the rather different values of recall measured for the MW-AC and AM-AC, are likely caused by the different density of the assertions in the SUTs, i.e., 0.99% and 0.18%, respectively. The MW-RB detects a smaller number of failures when compared to AM-RB: again, the density of RB in SUT_2 is bigger than the SUT_1, i.e., 8.59% and 0.36%, respectively. Finally, it can be observed that the recall of MW-EL is bigger than AM-EL: the percentage of error logging instructions out of the total number of logging instructions of MW-EL is bigger than AM-EL, i.e, 14.45% and 11.39%, respectively. Nevertheless, the high density of a MUT might affect the precision, as it can be inferred from the values of precision of AM-EL and AM-RB reported by Table 6.1. In fact, a large number of monitoring instructions might increase the probability to generate FPs.

These findings provide part of the answer to the **Research Question 3, i.e., RQ3**. In fact, Figure 6.1 and Figure 6.2 show that different MUTs of the same SUT exhibited a very different ability at reporting failures occurred in the SUT. Different MUTs of the same SUT exhibited also different values of Recall and Precision, as shown in Table 6.1, which

also shows that the values of Recall and Precision exposed by the MUTs changes at varying the SUT.

6.1.2 Failure Coverage

As discussed in Chapter 5, the overall recall has been broken down by failure type in each SUT. Given a failure type, the bottom row of Table 4.7 and Table 4.8 show the absolute number and the percentage of failures that have been reported by each MUT in SUT_1 and SUT_2, respectively.

Figure 6.3 shows the percentage of reported failures by type, MUT and SUT. It can be noted that **the reporting ability of the MUTs changes significantly across the SUTs**. Moreover, **a MUT might show a different ability at reporting the same type of failure in different SUTs**. For example, the coverage of the event logs ranges from a minimum of 3.61%, i.e., AM-EL (*SILENT* failures) to a maximum of 65.20%, i.e., MW-EL (*SILENT* failures). Rule-based logging achieves the maximum failure coverage observed in this study. Assertions are able to detect almost only *CRASH* failures, as it can

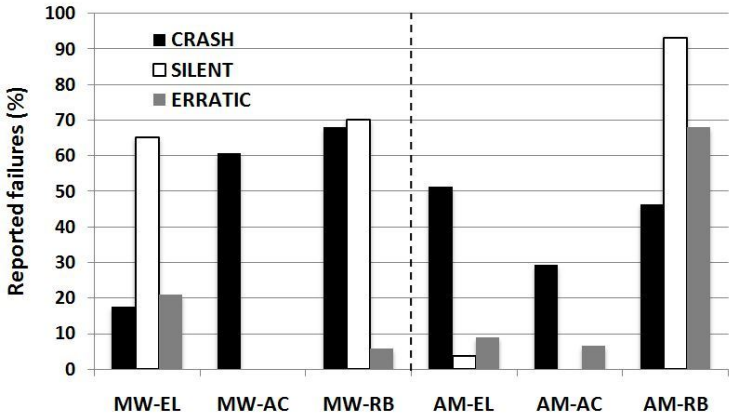

Figure 6.3: Percentage of reported failure by type, technique and case study.

be inferred by Figure 6.3.

The overall recall has been also broken down by fault type in each SUT, as already discussed in Chapter 5. Given a fault type, the rightmost column of Table 4.7 and Table 4.8 shows the absolute number and the percentage of failures that have been reported by each MUT in SUT_1 and SUT_2, respectively.

Figure 6.4 shows the percentage of activated **faults** that are reported by the MUTs in both the SUTs. Percentage of reported failures can be observed in the rightmost column of Table 4.7 and Table 4.8 for SUT_1 and SUT_2, respectively. It can be observed that **the reporting ability of the MUTs by fault type changes significantly at varying the SUT**. In fact, the percentages range between a minimum of 18.69%, for MW-EL (*ALG* faults), and a maximum of 68.02%, for MW-RB (*INT* faults), in SUT_1; while they range between a minimum of 0.00%, for AM-AC (*CHK* and *INT* faults) and AM-RB (*CHK* faults), in SUT_2. Moreover, **a MUT might show a different ability at reporting the same type of activated fault in different SUTs**.

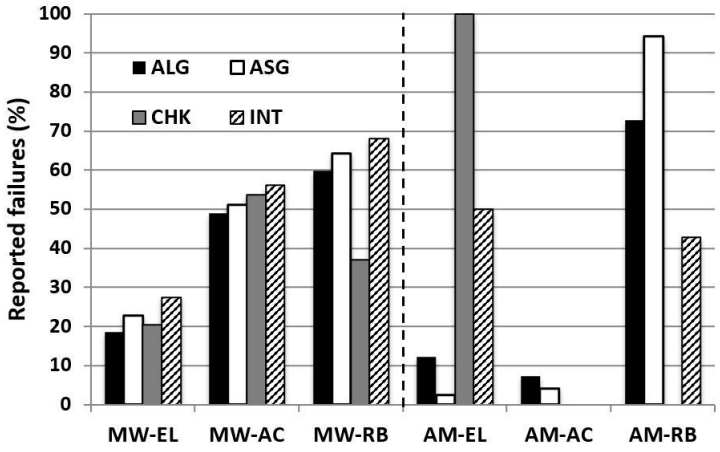

Figure 6.4: Percentage of reported failure of the MUTs by fault type and case study.

These findings provide further insights related to the **Research Question 3, i.e., RQ3**. In fact, Figure 6.3 and Figure 6.4 show how the failure reporting ability of the MUTs changes at varying the failure type and the fault type into the two SUTs, respectively.

6.1.3 Orthogonality of the MUTs

Given a SUT, the set of failures of the same type (i.e., *CRASH*, *SILENT*, *ERRATIC*) is broken down into 8 disjoint subsets, according to the notation described by Section 3.3.5: NONE (i.e., the failures reported by no MUT), EL*, AC* and RB* (i.e., the failures reported exclusively by one MUT), (EL·AC)*, (EL·RB)*, (AC·RB)* (i.e., the failures reported exclusively by two out of the three MUTs), and ALL (i.e., the failures reported by all the MUTs). The reader might refer to Figure 3.5 for the graphical representation of the sets, where MUT_i=EL, MUT_j=AC and MUT_k=RB.

Figure 6.5 shows that the MUTs are strongly orthogonal in SUT_1. As a result, combining the data produced by different MUTs might be useful to improve the detection of different types of failures. For example, both event and rule-based logging, i.e., (EL·RB)* in Figure

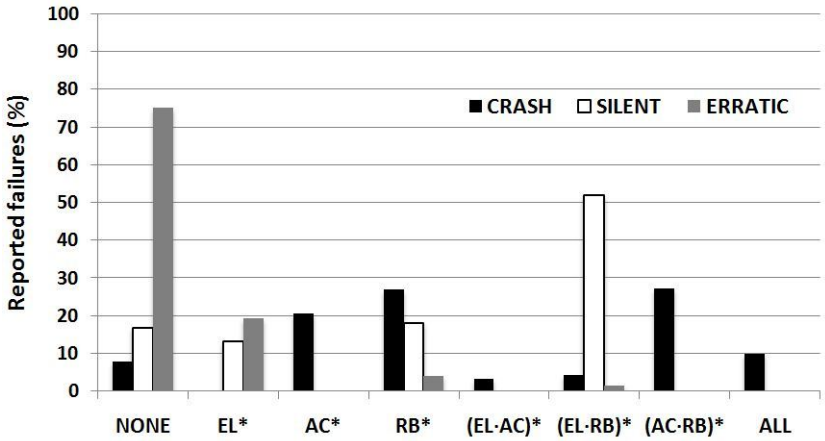

Figure 6.5: Orthogonality of the MUTs (SUT_1).

6.5, report 51.96% of *SILENT* failures; moreover, each EL and RB detects additional 13.24% and 18.14% of *SILENT* failures, respectively. AC alone does not give any contribution at detecting *SILENT* failures. Similarly, both assertion checking and rule-based logging, i.e., (AC·RB)* in Figure 6.5, report 27.09% of *CRASH* failures; moreover, AC and RB report further 20.58% and 26.83% of *CRASH* failures, respectively. Noteworthy, no MUT is able to report around 75% of *ERRATIC* failures, which go undetected, as it can be inferred by the NONE category.

Figure 6.6 shows that the considered MUTs are orthogonal also in the SUT_2. For example, event and rule-based logging, i.e., (EL·RB)* in Figure 6.6, report only 2.44% of *CRASH* failures; however, EL and RB report additional 42.68% and 15.85% of *CRASH* failures, respectively. This finding indicates that EL and RB report disjoints subsets of *CRASH* failures. It can be noted that AC alone does not give a significant contribution at detecting *CRASH* failures. For example, 21.95% of *CRASH* failures reported by means of AC are also reported by RB. More important, RB is able to report the most part of *SILENT* and *ERRATIC* failures in the SUT_2.

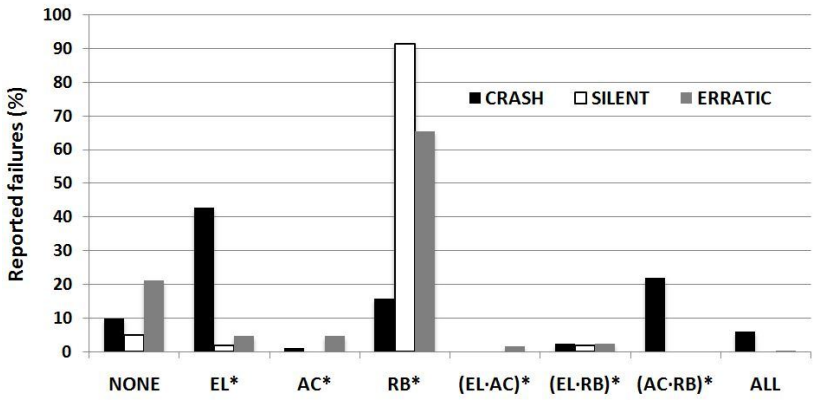

Figure 6.6: Orthogonality of the MUTs (SUT_2).

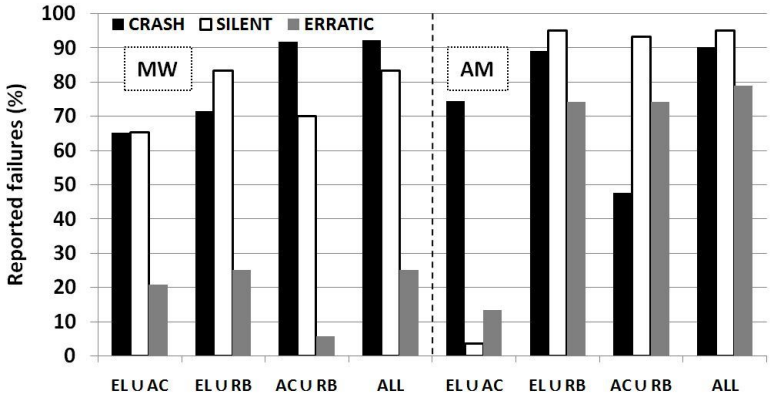

Figure 6.7: Percentage of reported failure by different techniques combination.

Different monitoring techniques can be combined to increase the failure coverage of a given SUT (for instance, by redirecting the notifications generated by different MUTs to the same log file). Figure 6.7 shows the percentage of reported failures that can be achieved by combining different MUTs by failure type and SUT. For example, the combination of event and rule-based logging, i.e., $EL \cup RB$, makes it possible to report 83.33% of *SILENT* failures in the SUT_1. This value is bigger than 70.10%, i.e., the percentage of *SILENT* failures reported by RB in the SUT_1 (RB is the MUT with the maximum *SILENT* reporting ability in the SUT_1).

Differently from the SUT_1, combining EL and RB is beneficial to improve the detection of *CRASH* in the SUT_2. $EL \cup RB$ reports 89.02% of *CRASH* failures, which is higher than the 51.22% exhibited by EL (i.e., the MUT with the best *CRASH* reporting ability in the SUT_2). In both the SUTs the combination of all the MUTs (i.e., ALL in Figure 6.7) exhibits the maximum coverage in each failure type.

The obtained findings provide further insights related to the **Research Question 4, i.e., RQ4**. Indeed, the results obtained from the analysis of the orthogonality of the

considered MUTs allows understanding that the MUTs are orthogonal in both the SUTs, which allows to obtain an improvement of the failure coverage in each SUT, as seen in Figure 6.7.

6.1.4 Dissimilarity of the Monitoring Data

For each SUT three document sets are established based on the data collected during the campaigns. In the SUT_1, the sets contain 664, 1,604 and 1,955 files for EL, AC, and RB, respectively; while in the SUT_2, they contain 81, 45 and 518 files for EL, AC, and RB. The sets are composed by the files generated under the reported failures by each MUT/SUT. The total number of reported failures by MUT is in the rightmost cell of the bottom row of Table 4.7 and Table 4.8 for SUT_1 and SUT_2, respectively.

Figure 6.8 and Figure 6.9 compare the CDFs obtained for the considered MUTs in the SUT_1 and the SUT_2. Given a log.entropy value taken from the x-axis, i.e., l, the y-axis reports the probability that the log.entropy of the documents generated by a given MUT

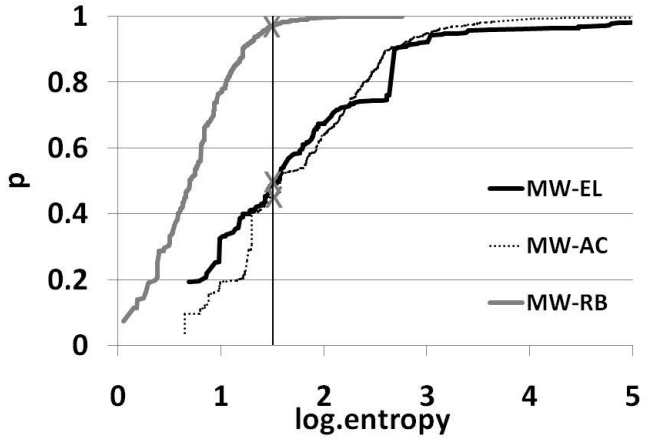

Figure 6.8: CDFs of log.entropy estimated for MUTs in SUT_1.

is $\leq l$ (or, equivalently the percentage of documents whose entropy is $\leq l$). For example, the point marked with a $\times$ (MW-RB series) in Figure 6.8 indicates that around 97% of RB documents produced by the SUT_1 under failures have entropy ≤ 1.5.

The log.entropy of the notifications produced by RB under failures is smaller than EL and AC, with around 3% and 2% of documents exhibiting a log.entropy value > 1.5 in both SUT_1 and SUT_2, respectively. It is worth noting that, while RB is the MUT covering the largest number of failures in both the SUTs, the dissimilarity of the notifications it generates across different failures is smaller than the other MUTs.

A closer look into the data allowed gaining insights into the causes of the small dissimilarity obtained by the monitoring data of RB. It has been noted that a small number of notifications occurs across many documents. Figure 6.10 shows the top 5 recurring generated by RB in both the SUTs. For example, in the SUT_1 the `SER, v_kernelNew, kernel` notification, which notifies that the function `v_kernelNew` did not terminate, was found in 1,325 out of 1,955 documents, i.e., the 67.7%. In the SUT_2 the `CER, root` notification,

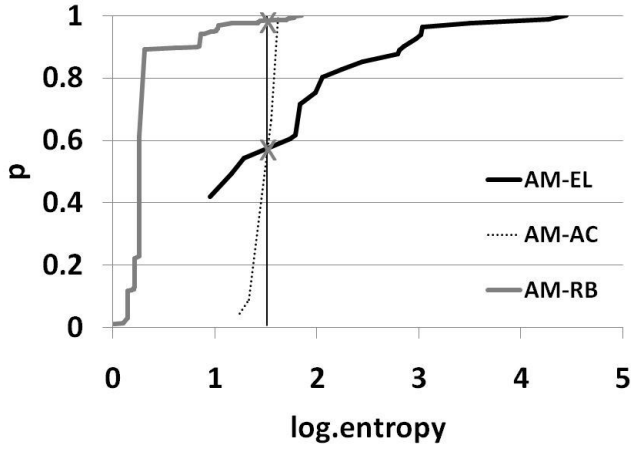

Figure 6.9: CDFs of log.entropy estimated for MUTs in SUT_2.

which notifies the unexpected crash of the `root` software module of AM, was found in 486 out of 518 documents, i.e., the 93.82%. While these notifications ensure high failure coverage, they become less useful for troubleshooting because RB frequently generates them.

The top 2 recurring notifications of EL are reported in Figure 6.11. The `Description Type mismatch` notification of the SUT_1 was found in 110 out of 664 documents, i.e., the 16.6%. In the SUT_2 the `Descr Not Implemented` notification was found in 35 out of 81 documents, i.e., 43.2%. The event logs generated under different failures are rather dissimilar in both the SUTs. This dissimilarity allows EL to achieve log.entropy values

Communication Middleware (1,955 documents)		
1,325	SER v_kernelNew	kernel
648	IER v_builtinNew	kernel
506	SER v_dataReaderNew	reader
353	SER v_writerWrite	writer
291	SER v_writerNew	writer
Arrival Manager (518 documents)		
486	CER service	root
273	SER StateEventTrans	Main
268	SER StartThd	Main
265	SER RunInit	EligThd
263	IER read	Main

Figure 6.10: Top 5 recurring notifications in RB.

Communication Middleware (664 documents)	
110	Description Type mismatch object...
	...type is OBJ but OBJ was expected
94	Description Could not claim DDSdeamon
Arrival Manager (81 documents)	
35	Descr Not ImplementedNUM NUM
34	NUM NUM NUEXCEPTION Code ENUMh from...
	...Task Search in Table PidNUM NUM

Figure 6.11: Top 2 recurring notifications in EL.

```
    Communcation Middleware (1,604 documents)
162 Sender codecsyncc122 cmutexUnlock...
    ...Assertion osthdIdToInt mtx owner...
    ...osthIdToInt osthdIdSelf failed
159 Receiver codecsyncc122 cmutexUnlock...
    ...Assertion osthdIdToInt mtx owner...
    ...osthdIdToInt osthdIdSelf failed

        Arrival Manager (45 documents)
45  TIMESTAMP TST WARNING EXCEPTION...
    ...Assertion Failed
    Module Procedure Line Instruction
 2  MAIN   MAINTEST  2446 B7A81588
```

Figure 6.12: Top 2 recurring notifications in AC.

greater than or equal to AC and RB logging in both the SUTs (i.e., around 50% and 40% of documents exhibiting a value > 1.5 for SUT_1 and SUT_2, respectively). The experiments suggest that **the monitoring data generated by means of event logging might be more suitable for manual failure analysis with respect to the other MUTs**.

The dissimilarity of the monitoring data obtained by EL is very close to AC in the SUT_1: in both cases, log.entropy is > 1.5 for around 50% of documents. Examples of assertions occurred in the SUT_1 are shown by Figure 6.12, which report the top 2 recurring assert notifications in both the SUTs.

Differently from SUT_1, in the SUT_2 EL and AC have different dissimilarity values. The log.entropy of AC varies in a small range around 1.5,as it can be noted from Figure 6.9. There is a lack of differentiation in the notification generated by AC in SUT_2. For example, the top recurring notification `TIMESTAMP TST WARNING EXCEPTION` (shown by Figure 6.12) was found in 45 out of 45 documents, i.e., the 100.0%, while the second top recurring notification `MAIN MAINTEST 2446 B7A81588` was found in 2 out of 45 documents, i.e., the

4.4%. The lack of differentiation leads to very similar log.entropy values in the monitoring data generated by AC.

The obtained findings provide further insights related to the **Research Question 3, i.e., RQ3**. Indeed, the results obtained from the dissimilarity analysis shows that the monitoring data generated by means of event logging might be more suitable for manual failure analysis with respect to the other MUTs, since event logging generate data rather dissimilar under different failures.

6.1.5 Error Determination Degree

The *Error Determination Degrees*, i.e., EDD with respect to the ODC fault class, the fault type, and the failure type, exhibited by each MUT of the SUT_1 have been compared. In addition, the EDDs have been also measured for the combination of all the MUTs of the SUT_1, i.e., the ones obtained by considering all the MUTs at the same time. Table 6.2 contains the values of the before mentioned EDDs.

It can be noted that all the MUTs of the SUT exhibited an EDD lower than 60% with respect to the fault type and ODC fault class, which means that **no MUT of the SUT_1 allows to infer either the fault type or the ODC class of the fault that have led to the error from its error notifications**. On the other hand, all the MUTs of the SUT exhibited an EDD greater than 88% with respect to the failure type, which means that **each MUT of the SUT_1 allows to infer the type of the failure occurred in the SUT from its error notifications**.

Similar considerations apply to the the combination of the MUTs. Indeed, the EDD

Table 6.2: Error Determination Degrees exhibited by each MUT of SUT_1 and by their combination.

EDD w.r.t.	***EDD (%)***			
	MW-EL	MW-AC	MW-RB	all MUTs
ODC fault class	57.10	56.30	54.70	58.98
fault type	43.47	42.12	44.90	47.08
failure type	88.92	99.87	91.97	96.36

with respect to both the ODC fault class and the fault type exhibited a value lower than 60%. However, an improvement can be observed in both the cases; in fact, the EDDs of the combination of all the MUTs outperforms all other in the case of ODC fault class and fault type, exhibiting a value of 58.98% and of 47.08%, respectively. Differently, the combination of the MUTs does not allow to obtain the highest EDD with respect to the failure type. Indeed, the combination exhibited a value of 96.36% that is lower than the 99.87% of the MW-AC. However, it should be noted that assertion checking was able to detect only *CRASH* failures in the SUT_1, as discussed in Section 5.2.2. Therefore, in absolute terms, the value exhibited by the combination of the MUTs can be considered as an improvement of the EDD with respect to the failure type. Both the findings suggest that the combination of all the MUTs of the SUT_1 can be useful to obtain a little improvement in terms of EDDs.

The obtained findings provide further insights related to both the **Research Question 3, i.e., RQ3** and the **Research Question 4, i.e., RQ4**. Indeed, Table 6.2 allows understanding that the MUTs implemented in SUT_1 exhibited very similar ability to determine the type/ODC class of the fault that have led to an error reported by the MUT as well as the type of the failure that the reported error have led to. On the other hand, Table 6.2 allows also understanding that a combination of the MUTs implemented in the SUT_1 can

be useful to improve the EDDs.

6.1.6 Error Propagation Reportability

The *Error Propagation Reportability* values exhibited by each MUT of the SUT_1 with respect to the *ALG* ODC class, i.e., the one for which the higher number of samples has been obtained, have been compared. In addition, the EPR with respect to the same ODC class has been also measured for the combination of all the MUTs of the SUT_1, i.e., the one obtained considering all the MUTs at the same time. Table 6.3 contains the values of the before mentioned EPRs. Noteworthy, the best highest EPR have been exposed by the rule-based logging. However, as discussed in Section 5.3.4, this value is not realistic because the rule-based logging is not implemented in all the components of the system, but only in the kernel component; thus, it can be excluded from the comparison. Based on this consideration, the best EPR is exhibited by the combination of the MUTs, i.e, 79.78%, which significantly outperformed both MW-EL and MW-AC, i.e., 26.01% and 36.64%, respectively. Therefore, **the combination of the MUTs of the SUT_1 allow obtaining an improvement in terms of Error Reporting Ability**.

The EPR of the combination of the MUTs of the SUT_1 has been evaluated by analyzing

Table 6.3: Error Propagation Reportability (EPR) of each MUTs of SUT_1, and of their combination, with respect to the *ALG* faults.

MUT	***EPR***$_{ALG}$ ***(%)***
MW-EL	26.01
MW-AC	36.64
MW-RB	100.00
all MUTs	79.78

the related directed graph. Figure 6.13 shows the obtained graph, which summarizes the major error propagation paths through the components of the SUT_1 that which are generated as a consequence of ALG faults. The number of faults considered in the graph refers to the number of faults that have led to at least an error notification to be raised by at least one of the considered MUTs. The analysis of the graph shows that an improvement has been obtained by combining the MUTs. For example, it can be noted that, differently form the information inferred form the graph generated considering only the event logging, Figure 5.8, all the errors that led to some error notifications in the *api*, *spliced* and *user* components of the SUT_1 at the same time, i.e., *api+spliced+user* node in the graph, are propagated through the *kernel* component. Therefore, the assumption that the errors are expected to arise in the *kernel* component of the SUT_1 since all the faults have been injected into this component is correct for these components.

Similar considerations apply to the *ddsi2* and *database* components. Indeed, it can be noted that, differently form the information inferred form the graph generated considering only the assertion checking, Figure 5.17, where only the 0.8% of the errors that led to at least an error notification in the *ddsi2* component have been reported as propagated through the *kernel component*, the combination of the MUTs reported that 24.78% of those error have been propagated through the *kernel* component. Similarly, more than the half of the errors that led to at least an error notification in the *database* component have propagated through the *kernel* component, differently from the information inferred form the graph generated considering only the event logging and only the assertion checking. Therefore, the assumption is correct also for these components. It should be noted that

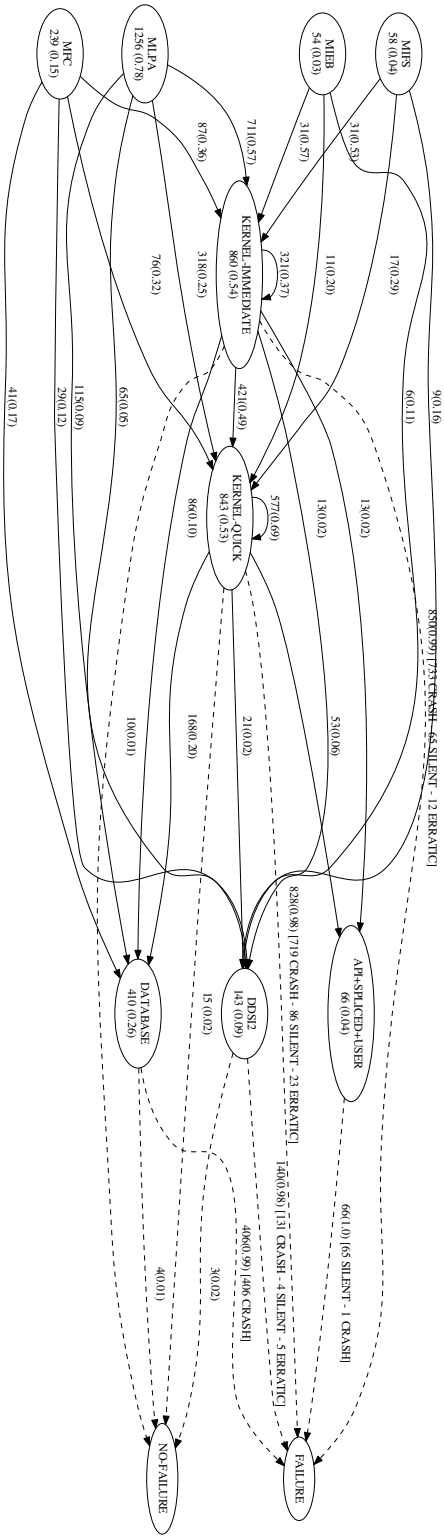

Figure 6.13: Error propagation graph for ALG faults (combination of MUTs of SUT_1).

this improvement is substantially due to the presence of the rule-based logging, which reports only errors occurred in the *kernel*. Indeed, as it can be inferred form Figure 6.14, where the detection node is shown, the number of errors reported by rule-base logging in the considered components is almost equal to the number of error that are reported as propagated from the kernel.

However, the presence of errors for both the components that generated an error notifications without leave any traces in the kernel component, suggest that there is the need of further EDMs in the SUT_1 in order to improve the reporting of errors, as well as of the error propagation. A closer look into the obtained error notifications in these cases allowed to understand that most of the errors that are reported by the *database* and not from *kernel* are of the type *e2-AC*, i.e., *unexpected value errors*; while most of the errors that are reported by the *ddsi2* and not from *kernel* are of the type *e4-AC*, i.e., *NULL value errors*. Therefore, the EDMs to insert into the system have to cope to this kinds of errors. It should be noted that both the types belong to the error model of the assertion checking. Indeed, most of the error notifications raised in both the components have been raised by means of assertions, as can be inferred into the Figure 6.14, where the detection node is shown. In fact, 94% and 92% of error notifications reported in the *database* and *ddsi2*, respectively, are raised by means of assertions.

Finally, Figure 6.13 allows also inferring that there is the need of ERMs in the system since almost all the errors reported by the MUTs led to a failure in the systems. For example, the error notifications reported by the *api*, *spliced* and *user* components at the same time led to almost *SILENT* failures, while the ones reported by the *database* led to

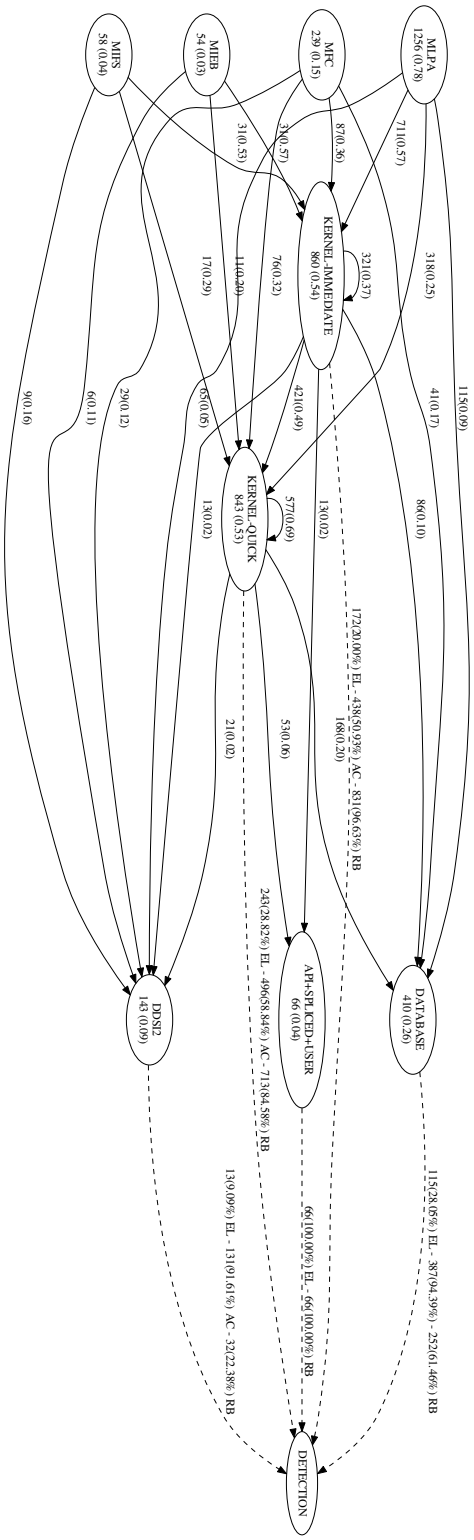

Figure 6.14: Error propagation graph for ALG faults with detection node (combination of MUTs of SUT_1).

only *CRASH* failures; therefore, ERMs can be added in these components to cope with these kinds of failure.

The obtained findings provide further insights related to all the **Research Questions** addressed in this dissertation. Indeed, Figure 6.13 allows understanding that it is possible to use monitoring techniques to characterize the error behavior of a complex critical software system (**RQ1**), even if in a non-exhaustive way. As a reminder, only the errors reported by the MUTs can be considered in the analysis. In addition, there is the possibility that some propagation paths from the faulty component to the other ones are not reported by a MUT because they are not present by design in the target SUT, e.g., when the component that reports the error works as a detector of the faulty component, i.e., where the fault has been injected. Therefore, the proposed *EPR* metric might provide not accurate value in this case. Moreover, the analysis of the graph also allows to understand that the information obtained from the error data collected by the MUTs can be leveraged to improve the error detection/recovery of the MUTs of the SUT (**RQ2**), by providing insight about the placement of EDMs/ERMs. Differently, both Figure 6.13 and Table 6.3 highlight how the error and error propagation reporting ability changes between the MUTs (**RQ3**), even if limited to the only *ALG* class, as well as that the combination of the MUTs can be useful (**RQ4**) to improve the EPR, obtaining better insight about how the error propagates int the SUT.

6.2 Practical Implications and Threats to Validity

The content of the monitoring data collected during the experiments has been further investigated to gain insights into the motivations underlying the coverage results. The conducted investigation aims to identify, if any, specific characteristics of the SUTs that motivate the results and provide practical implications to improve the MUTs.

A key finding of this study suggests that the effectiveness of a MUT is strongly affected by the SUT and type of failure. It has been noted that MW-EL and AM-EL report 17.62% and 51.22% of *CRASH* failures, respectively; on the contrary MW-EL outperforms MW-AC under *SILENT* failures, i.e., 65.20% and 3.61% of reported *SILENT* failures, respectively. A closer look into the monitoring data generated by EL in the SUTs allowed understanding the motivations of such a difference.

It has been noted that 34 out of 45 *CRASH* failures induced in the SUT_2 have been notified by the example pattern in Figure 6.15, where `TIMESTAMP` and `PROCESS_PID` represent the timestamp of the log entry and the PID of the mentioned process, respectively. The analysis of the source code of the SUT_2 indicates that the pattern in Figure 6.15 is generated by a *supervisor* software module of the arrival manager. The **supervisor** module is able to manage the errors raised by its supervised processes (i.e., sequencing and metering processes, shown in Figure 4.2) by executing different actions based on the given error. If no actions can be executed, such as the case of a *CRASH* failure, the supervisor raises an exception and appends an entry to the event log.

Differently, it has been noted that 92 out of 133 *SILENT* failures occurred in the SUT_1

were notified by the example pattern in Figure 6.15. The pattern indicates the occurrence of some problems affecting the main daemon of the middleware, i.e., *DDSdaemon*. By means of repeated runs of the middleware, it has been observed that during most of the *SILENT* failure manifestations the DDSdaemon has been not responsive. Again, the manual inspection of the source code allowed understanding that the pattern in Figure 6.15 is generated by a **diagnostic method** that performs heartbeat checks on the DDSdaemon. The method allows detection anomalies occurring under *SILENT* failures.

The above presented examples indicate that **architectural features of a system are potentially beneficial to event logging**. The implementation of the supervisor module in the SUT_2 allowed AM-EL to report more *CRASH* failures than MW-EL. The heartbeat checks on the main internal daemon of the SUT_1 allowed MW-EL to better report *SILENT* failures when compared to AM-EL. The inclusion of diagnostic supports in the system architecture is beneficial to improve the reporting capability of event logs.

Arrival Manager - *CRASH* failures

```
TIMESTAMP EXCEPTION Code E829h from...
...Task Search in Table  Pid PROCESS_PID
```

Communication Middleware - *SILENT* failures

```
//omitted
Description: An error occurring during...
...exithandling. Unable to determine the...
...presence of application participants.
The DDSdaemon service object was NULL.
//omitted
Description: Could not claim DDSdaemon.
```

Figure 6.15: Top recurring EL pattern of *CRASH* failures in SUT_2 and *SILENT* failures in SUT_1.

Beside architectural considerations, it has been observed that **the placement of monitoring instructions can affect the reporting ability of a MUT**. In the SUT_1 17.62% and 60.78% of *CRASH* failures are reported by EL and AC. The inspection of the source code of the SUT_1 revealed that assertions are often placed **before** the logging instructions. It is worth noting that the assertions implemented in the SUT_1 belong to the *specification of function interface* class (as discussed in Section 4.3): in this respect, they are likely placed at the beginning of functions. The triggering of an assertion might suppress the notifications generated by the logging instructions placed later in the source code of a function.

Figure 6.16 shows an example of AC and EL in the SUT_1. It can be note that the assertion (line 6) is placed before the logging instructions (lines 9-10), which is activated upon the same condition (line 8) of the assertion, i.e., if `_this` is `NULL`. In the case `_this` is `NULL`, the assertion is triggered and the logging instruction is not executed. This placement of the assertions motivates to the small *CRASH* reporting ability shown by EL in the SUT_1. Both AC and EL have been enabled in the conducted experiments because one of the goals of this dissertation is to assess the combined used of different MUTs

As a practical programming implication, developers should place the logging instructions before the assertions triggered by the same conditions. It is worth noting that placement of the assertions does not impact RB: the error detection mechanism of RB relies on the lack of expected events (e.g., the end of a function call) rather than on the execution of a notification instruction. Moreover, the experiments suggest that the monitoring instructions should be placed along the paths where the errors are more likely to manifest. The log.entropy

```
1           Communication Middleware (v_spliced.c, line 729)
2  //omitted
3  static void notifyCoherentReaders(v_kernel _this,
4                                    v_dataReaderSample rSample){
5      //omitted
6      assert(_this != NULL);
7      //omitted
8      if (_this == NULL) {
9          OS_REPORT(OS_WARNING, "v_spliced::notifyCoherentReaders", 0,
10                 "Received illegal '_this' reference to kernel.");
11         return;
12     }
13     //omitted
```

Figure 6.16: Example of assert and logging instructions (SUT_1).

analysis discussed in Section 6.1.4 indicates that only a limited number of monitoring instructions is frequently activated under different failures. In this respect, the knowledge of the system architecture (in terms of software modules and interactions among them) is potentially useful to infer strategic source code locations, which are more suitable to contain the monitoring instructions.

Regarding the threats to validity, it should be noted that the analysis has been conducted on two datasets generated by the execution of controlled experiments aimed to assess recall, precision and the three proposed metrics of the considered monitoring techniques (MUTs) in two target systems (SUTs), under given workload and faultload. The discussed findings might be subject to both construct and internal validity threats. In fact, special care must be taken to reproduce realistic operation scenarios and to exercise the MUTs with representative workload and faultload. The adoption of a scenario that is far from the real application of the SUT or the execution of simplified workload and faultload represent potential threats that practitioners replicating this type of analysis should be aware of.

In this work, special attention has been devoted to exercise both the SUTs according to representative operations. Both MW and AM are deployed according to the settings

provided by the industrial developers and are fed with realistic input data. For instance, in the case of MW the workload consist of real flight data items produced by two legacy ATC applications and consumed by a real instance of the web-based CWP, which is usually deployed at customers' premises. In the case of AM, the system is exercised with the same test suite used by its developers to emulate the nominal usage of the system at the production site.

Regarding the faultload, the study relies on faults belonging to the well consolidated **orthogonal defect classification** (ODC) [103] and the fault types that account for around 80% of representative faults found in real-world software systems have been selected, according to the estimates in [104]. In addition, the faultload has been generated by using the SAFE tool [110], which automatically searches for all the locations in the source code where each of the considered fault types can be injected. As a result, a large number of representative faults in all the possible code locations has been considered.

As a side effect, given the size of the faultload, some faults could be never activated during the experiments (e.g., the fault is injected in a piece of code which is not exercised by the workload, or it causes an error that is tolerated by the SUT). For instance, in the case of MW, even if the faultload is composed by 12,733 faults, only 3,159 fault injections resulted in a failure. Practitioners should be aware of this issue when replicating this type of analysis by introducing faults in a large number of code locations.

Nevertheless, even if faults are exhaustively placed in all possible code locations, still for some of the fault/failure combinations it might happen to have a small number of samples. As an example, the (*CHK*, *SILENT*) combination yields 1 failure in the MW case study.

In these cases conclusions cannot be drawn.

Finally, as for external validity, results observed on two case studies are not statistically generalizable. However, the reported findings, which are strongly supported by data, are still useful to get an overall understanding on the type of characterization that can be performed with the proposed method. Results show how the proposed method can be used to understand the error behavior of complex critical software systems and to assess the monitoring techniques of a given system, to uncover their limitations and to infer insight useful to improve them accordingly. In addition, due to the lack of studies on monitoring techniques of proprietary industrial systems in literature, the reported results can be extremely relevant to reliability engineers and practitioners in spite of the limitations in terms of external validity.

Conclusion

The thesis addressed issues and challenges concerning the use of field data for the analysis of the error behavior of complex critical software systems. Field data contain rich information about the system reliability, providing valuable information on actual error/failure behavior of a software system during the normal system operation. The thesis discussed a substantial body of literature using field data, which highlighted that the analysis of field failure data is useful in a variety of application domains, encompassing error and failure classification, evaluation of dependability attributes, diagnosis and correlation of failures, failure prediction. Field data are also used to understand the error behavior of software systems as well as to characterize monitoring techniques, which are one of the main source of field data. Nevertheless, the analysis of literature also revealed that the application of existing techniques for the analysis of error behavior of software system is not trivial in the context of complex critical software systems, as well as that there are no past experiences on the characterization of the effectiveness of monitoring techniques with respect to failures and errors.

The work proposed a methodology that leverages field data generated by means of monitoring techniques already implemented in the target complex critical software system

in order to understand the error behavior of the system, avoiding intrusive modifications of its source code for the collection of useful data to analyze. Moreover, the proposal is also conceived as a methodology to compare the effectiveness of different monitoring techniques implemented in different target systems. To this aim, the methodology leverages information retrieval metrics and proposed metrics, i.e., Error Determination Degree, Error Propagation Reportability, and Dissimilarity of monitoring data.

The proposed methodology has been applied in two real-world critical industrial software systems in the context of Air Traffic Control domain, i.e., the communication middleware and the arrival manager. The method leverages and analyzes the data reported by monitoring techniques implemented by the mentioned systems, i.e., event logging, assertion checking and source code instrumentation.

The obtained results reveal that field data generated by means of monitoring techniques can be leveraged to understand the error behavior of complex critical software systems, allowing to infer the types of error that affect the system, the effect they have on the system and, more important, how they propagate through the components of the system. The results show that these abilities change in different monitoring techniques, as well as that the reporting ability of the considered techniques changes across the two systems and the failure types. Moreover, the methodology suggests that the considered techniques are strongly orthogonal in the considered case studies: different monitoring techniques can be combined to increase the failure coverage and the error propagation reportability, obtaining more detailed information about the propagation of errors. The analysis also indicates that in the considered systems the monitoring data generated by means of event logging are

more suitable for failure analysis purposes with respect to the other techniques.

Finally, a closer investigation of the error notifications collected during the experimental campaigns revealed that the failure reporting ability of the considered monitoring techniques is impacted by a variety of features, such as architecture of the system and placement of the monitoring instructions. This finding has a number of practical implications that the developer should consider in order to implement better monitoring techniques, and also to place EDM and ERM where they can be more effective.

The general achievements have been summarized in the following, referring to the research questions of this work:

- **RQ1**: *Is it possible to use monitoring techniques to characterize the error behavior in complex critical software system?* The use of the proposed methodology to the monitoring techniques implemented in the communication middleware, along with their comparison, allowed understanding that it is possible to use the data they provide to characterize the error behavior of the system. In particular, the proposed methodology allowed to infer the types of error that affect the system, the effect they have on the system and, more important, how they propagate through the components of the system, according to the considered monitoring technique, even if in a non-exhaustive way as discussed in the Section 5.4.

- **RQ2**: *Is it possible to improve the error detection/recovery of a complex critical software system from error data?* The analysis of the error propagation graphs created using the monitoring data of the communication middleware allowed inferring insights

about potential locations where EDMs and ERMs might be beneficial for the system, as well as the type of error/failure they have to cope with.

- **RQ3**: *How do the error and failure reporting ability change between different monitoring techniques implemented in a given system? And what about the dissimilarity of their data?* Both the analysis of each monitoring techniques implemented in the two considered target systems, and their comparison, allowed to understand that different monitoring techniques of the same target system exhibited a very different ability at reporting occurred failures. In particular, the failure reporting ability of the techniques changed at varying the failure type and the fault type into the two target systems. In addition, the results obtained from the dissimilarity analysis showed that the monitoring data generated by different monitoring techniques exposed different level of dissimilarity, also in different target systems. More in details, the analysis showed that monitoring data generated by means of event logging might be more suitable for manual failure analysis with respect to the other techniques, since event logging generate data rather dissimilar under different failures. Finally, Error Determination Degree analysis allowed understanding that the techniques implemented in the communication middleware exhibited very low ability to determine the type-/ODC class of the fault that have led to an error reported by the techniques, while they exhibited very high ability to determine the type of the failure that the reported error have led to. The Error Propagation Reportability analysis highlighted instead how the error and error propagation reporting ability changes between the considered

monitoring techniques in the communication middleware.

- **RQ4**: *Is it useful to combine different monitoring techniques implemented in a complex critical software system?* The results obtained from the analysis of the orthogonality of the considered monitoring techniques allowed understanding that the techniques are orthogonal in both the considered systems, which allowed to obtain an improvement of the failure coverage in each SUT, as highlighted by the results obtained from their combination. In addition, the combination of the considered monitoring techniques allowed also the improvement at reporting the propagation of errors in the communication middleware, as showed by the results of the analysis on the Error Propagation Reportability.

The proposed methodology allowed to answer to the research questions of this study, showing that the field data generated by means of monitoring techniques can be leveraged for error analysis in complex critical software systems. In particular, useful insights have been obtained about the propagation of errors through the components of a target system, and on where the placement of ERM and EDM can be potential beneficial in order to improve the reliability of the system. Practical implications on how improve the monitoring techniques implemented by the target system have been also provided.

Future work will be devoted to extend this work on different paths. First, the insights obtained from this study will be provided to developers of the considered systems in order to improve their monitoring techniques, as well as to introduce potential beneficial EDMs and

ERMs. Second, the methodology will be applied to other software systems, either critical or not, in order to further validate the proposed approach. In addition, a study on the so-called *indirect monitoring techniques* has been started in order to understand if data generated by operating system-level probes are suitable for error analysis purposes. Finally, also a study on the suitability of monitoring techniques for the detection of anomalies occurred as a consequence of security attacks has been started.

Bibliography

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004.

[2] D. P. Siewiorek, R. Chillarege, and Z. T. Kalbarczyk. Reflections on Industry Trends and Experimental Research in Dependability. *IEEE Transactions on Dependable and Secure Computing*, 1:109–127, April 2004.

[3] J. Gray. Why do computers stop and what can be done about it. In *Symp. on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[4] L. Spainhower and T. A. Gregg. Ibm s/390 parallel enterprise server g5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5.6):863–873, Sept 1999.

[5] Elaine J Weyuker. Testing component-based software: A cautionary tale. *IEEE software*, 15(5):54, 1998.

[6] R Moraes, Joao Duraes, Ricardo Barbosa, Eliane Martins, and Henrique Madeira. Experimental risk assessment and comparison using software fault injection. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 512–521. IEEE, 2007.

[7] Radio Technical Commission for Aeronautics. RTCA DO-178B, software considerations in airborne systems and equipment certification. Technical report, 1992.

[8] R. K. Iyer, Z. Kalbarczyk, and M. Kalyanakrishnan. Measurement-Based Analysis of Networked System Availability. *Performance Evaluation: Origins and Directions*, pages 161–199, 2000.

[9] The International Electrotechnical Commission. IEC 61508: Functional safety of electrical/-electronic/programmable electronic safety-related systems - part 7. Technical report, 1998.

[10] International Organization for Standardization. Product development: software level. ISO/DIS 26262-6. Technical report, 2009.

[11] Department of Defense - United States of America. Dod guide for achieving reliability, availability, and maintainability. Technical report, August 2005.

[12] Ian Sommerville. *Software engineering.* Pearson, 9th ed edition, 2011.

[13] A. Bondavalli, S. Chiaradonna, D. Cotroneo, and L. Romano. Effective fault treatment for improving the dependability of cots and legacy-based applications. *IEEE Transactions on Dependable and Secure Computing*, 1(4):223–237, Oct 2004.

[14] U.S.C.P.S.O. Task. *Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations*. 2011.

[15] European Space Agency. *Ariane 501 Inquiry Board Report (July 1996)*. 1996.

[16] National Aeronautics and Space Administration. *Mars Climate Orbiter Mishap Investigation Report, Washington, DC (November 1999)*. 1999.

[17] Nelson Green, Alan Hoffman, Timothy Schow, and Henry Garrett. Anomaly trends for robotic missions to mars: implications for mission reliability. In *44th AIAA Aerospace Sciences Meeting and Exhibit. Reno, NV*, 2006.

[18] Alan R Hoffman, Nelson H Green, and Henry B Garrett. Assessment of in-flight anomalies of long life outer planet missions. In *Environmental Testing for Space Programmes*, volume 558, pages 43–50, 2004.

[19] Algirdas Avižienis. Design of fault-tolerant computers. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), pages 733–743, New York, NY, USA, 1967. ACM.

[20] Jean-Claude Laprie. Dependable computing and fault tolerance: Concepts and terminology. *Proc. of the 15th IEEE International Symposium on Fault-Tolerant Computing (FTCS-15)*, 1985.

[21] A. Avizienis and J. P. J. Kelly. Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80, Aug 1984.

[22] D. Avresky, J. Arlat, J. C. Laprie, and Y. Crouzet. Fault injection for formal testing of fault tolerance. *IEEE Transactions on Reliability*, 45(3):443–455, Sep 1996.

[23] A. Jhumka and M. Leeke. The early identification of detector locations in dependable software. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 40–49, Nov 2011.

[24] W. Abdelmoez, D. M. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H. H. Ammar, B. Yu, and A. Mili. Error propagation in software architectures. In *Software Metrics, 2004. Proceedings. 10th International Symposium on*, pages 384–393, Sept 2004.

[25] J. Voas. Error propagation analysis for cots systems. *Computing Control Engineering Journal*, 8(6):269–272, Dec 1997.

[26] P. Popic, D. Desovski, W. Abdelmoez, and B. Cukic. Error propagation in the reliability analysis of component based systems. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10 pp.–62, Nov 2005.

[27] A. Jhumka, M. Hiller, and N. Suri. Assessing inter-modular error propagation in distributed software. In *Reliable Distributed Systems, 2001. Proceedings. 20th IEEE Symposium on*, pages 152–161, 2001.

[28] Vittorio Cortellessa and Vincenzo Grassi. *Component-Based Software Engineering: 10th International Symposium, CBSE 2007, Medford, MA, USA, July 9-11, 2007. Proceedings*, chapter A Modeling Approach to Analyze the Impact of Error Propagation on Reliability of Component-Based Systems, pages 140–156. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[29] T. M. Khoshgoftaar, E. B. Allen, Wai Hong Tang, C. C. Michael, and J. M. Voas. Identifying modules which do not propagate errors. In *Application-Specific Systems and Software Engineering and Technology, 1999. ASSET '99. Proceedings. 1999 IEEE Symposium on*, pages 185–193, 1999.

[30] M. Hiller, A. Jhumka, and Neeraj Suri. Epic: profiling the propagation and effect of data errors in software. *IEEE Transactions on Computers*, 53(5):512–530, May 2004.

[31] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: Diagnosing production run failures at the user's site. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 131–144, New York, NY, USA, 2007. ACM.

[32] Martin Hiller, Arshad Jhumka, and Neeraj Suri. Propane: An environment for examining the propagation of errors in software. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 81–85, New York, NY, USA, 2002. ACM.

[33] A. Johansson and N. Suri. Error propagation profiling of operating systems. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 86–95, June 2005.

[34] M. Leeke and A. Jhumka. Towards understanding the importance of variables in dependable software. In *Dependable Computing Conference (EDCC), 2010 European*, pages 85–94, April 2010.

[35] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. *SIGARCH Comput. Archit. News*, 38(1):143–154, March 2010.

[36] A. J. Oliner and J. Stearley. What Supercomputers Say: A Study of Five System Logs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 575–584. IEEE Computer Society, 2007.

[37] B.A. Schroeder. On-line monitoring: a tutorial. *Computer*, 28(6):72–78, Jun 1995.

[38] N. Delgado, A.Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *Software Engineering, IEEE Transactions on*, 30(12):859–872, Dec 2004.

[39] Mars Science Laboratory. http://mars.jpl.nasa.gov/msl.

[40] M. Hiller. Executable assertions for detecting data errors in embedded control systems. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 24–33, 2000.

[41] A.K. Mok and L. Guangtian. Efficient run-time monitoring of timing constraints. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 252–262, June 1997.

[42] D. Bartetzko, C. Fischer, M. Mller, and H. Wehrheim. Jass java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103 – 117, 2001. RV'2001, Runtime Verification.

[43] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[44] Nagios. http://www.nagios.org/.

[45] L. Wang, Z. Kalbarczyk, W. Gu, and R.K. Iyer. Reliability microkernel: Providing application-aware reliability in the os. *Reliability, IEEE Transactions on*, 56(4):597–614, Dec 2007.

[46] G. Khanna, P. Varadharajan, and S. Bagchi. Automated online monitoring of distributed applications through external monitors. *Dependable and Secure Computing, IEEE Transactions on*, 3(2):115–129, Apr. '06.

[47] M. Kalyanakrishnam, Z. Kalbarczyk, and R. K. Iyer. Failure Data Analysis of a LAN of Windows NT based Computers. In *Proceedings of the International Symposium on Reliable Distributed Systems (SRDS)*, pages 178–187. IEEE Computer Society, October 1999.

[48] C. Simache and M. Kaâniche. Availability Assessment of SunOS/Solaris Unix Systems Based on syslogd and wtmpx Log Files: A Case Study. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 49–56. IEEE Computer Society, 2005.

[49] J. Tian, S. Rudraraju, and Zhao Li. Evaluating web software reliability based on workload and failure data extracted from server logs. *Software Engineering, IEEE Transactions on*, 30(11):754–769, Nov 2004.

[50] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of the 9th ACM SIGKDD Int'l Conference on Knowledge Discovery and Data Mining*, pages 426–435, New York, NY, USA, 2003. ACM.

[51] F. Salfner and M. Malek. Using hidden semi-markov models for effective online failure prediction. In *Reliable Distributed Systems, 26th IEEE International Symposium on*, pages 161–174, Oct '07.

[52] H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal Analysis of Log Files. *Journal of Aerospace Computing, Information and Communication*, 7(11):365–390, 2010.

[53] A. Bauer, M. Leucker, and C. Schallhart. Runtime Reflection: Dynamic model-based analyis of component-based distributed embedded systems. In *Modellierung von Automotive Systems*, 2006.

[54] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, G. Goel, S. Sarkar, and R. Ganesan. Characterization of operational failures from a business data processing saas platform. In *The 36th Int'l Conference on Software Engineering*, ICSE, pages 195–204, New York, NY, USA, 2014. ACM.

[55] D. Cotroneo, A. Paudice, and A. Pecchia. Automated root cause identification of security alerts: Evaluation in a saas cloud. *Future Generation Computer Systems*, 56:375 – 387, 2016.

[56] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 102–112, Piscataway, NJ, USA, 2012. IEEE Press.

[57] A. Pecchia and S. Russo. Detection of software failures through event logs: An experimental study. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 31–40, Nov 2012.

[58] Q. Fu, J. Zhu, W. Hu, J. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? An empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 24–33, New York, NY, USA, 2014. ACM.

[59] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to log: Helping developers make informed logging decisions. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 415–425, May 2015.

[60] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo. Industry practices and event logging: Assessment of a critical software development process. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 169–178, Piscataway, NJ, USA, 2015. IEEE Press.

[61] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21, 1995.

[62] R. Venkatasubramanian, J.P. Hayes, and B.T. Murray. Low-cost on-line fault detection using control flow assertions. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, pages 137–143, July 2003.

[63] C. Rabejac, J.-P. Blanquart, and J.-P. Queille. Executable assertions and timed traces for on-line software error detection. In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, pages 138–147, Jun 1996.

[64] M. Cinque, D. Cotroneo, and A. Pecchia. Event logs for the analysis of software failures: A rule-based approach. *Software Engineering, IEEE Transactions on*, 39(6):806–821, June 2013.

[65] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 2–2, '04.

[66] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, October '01.

[67] M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[68] Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. *Engineering Dependable Software Systems*, 34:141–175, '12.

[69] J. P. Hansen and D. P. Siewiorek. Models for Time Coalescence in Event Logs. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*, pages 221–227. IEEE Computer Society, 1992.

[70] A. Pecchia, D. Cotroneo, Z. Kalbarczyk, and R. K. Iyer. Improving log-based field failure data analysis of multi-node computing systems. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 97–108, June 2011.

[71] Yinglung Liang, Yanyong Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo. Bluegene/l failure analysis and prediction models. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 425–434, June 2006.

[72] C. Simache and M. Kaaniche. Measurement-based availability analysis of unix systems in a distributed environment. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 346–355, Nov 2001.

[73] M. F. Buckley and D. P. Siewiorek. A comparative analysis of event tupling schemes. In *Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, FTCS '96, pages 294–, Washington, DC, USA, 1996. IEEE Computer Society.

[74] DBench project. http://webhost.laas.fr/tsf/dbench/.

[75] S. M. Matz, L. G. Votta, and M. Malkawi. Analysis of failure and recovery rates in a wireless telecommunications system. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 687–693, 2002.

[76] R. Mullen. The lognormal distribution of software failure rates: Application to software reliability growth modeling. In *Proceedings of the The Ninth International Symposium on Software Reliability Engineering*, ISSRE '98, pages 134–, Washington, DC, USA, 1998. IEEE Computer Society.

[77] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, Oct 2010.

[78] T. T. Y. Lin and D. P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability*, 39(4):419–432, Oct 1990.

[79] Ronjeet Lal and Gwan Choi. Error and failure analysis of a unix server. In *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, pages 232–239. IEEE, 1998.

[80] C. Simache, M. Kaaniche, and A. Saidane. Event log based dependability analysis of windows nt and 2k systems. In *Dependable Computing, 2002. Proceedings. 2002 Pacific Rim International Symposium on*, pages 311–315, Dec 2002.

[81] A. Ganapathi and D. Patterson. Crash data collection: a windows case study. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 280–285, June 2005.

[82] David Oppenheimer and David A Patterson. Studying and using failure data from large-scale internet services. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 255–258. ACM, 2002.

[83] Xavier Castillo and Daniel P Siewiorek. A performance-reliability model for computing systems. Technical report, DTIC Document, 1980.

[84] Xavier Castillo. Workload, performance, and reliability of digital computing systems. Technical report, DTIC Document, 1980.

[85] Ravishankar K Iyer, Edward J McCluskey, et al. A statistical failure/load relationship: Results of a multicomputer study. *Computers, IEEE Transactions on*, 100(7):697–706, 1982.

[86] I. Lee, R. K. Iyer, and D. Tang. Error/failure analysis using event logs from fault tolerant systems. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 10–17, June 1991.

[87] D. Tang and R. K. Iyer. Dependability measurement and modeling of a multicomputer system. *IEEE Transactions on Computers*, 42(1):62–75, Jan 1993.

[88] R. K. Iyer, L. T. Young, and P. V. K. Iyer. Automatic recognition of intermittent failures: an experimental study of field data. *IEEE Transactions on Computers*, 39(4):525–537, Apr 1990.

[89] Y. Liang, Y. Zhang, A. Sivasubramaniam, R. K. Sahoo, J. Moreira, and M. Gupta. Filtering failure logs for a bluegene/l prototype. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 476–485, June 2005.

[90] R. A. Maxion and F. E. Feather. A case study of ethernet anomalies in a distributed computing environment. *IEEE Transactions on Reliability*, 39(4):433–443, Oct 1990.

[91] M. Dacier, F. Pouget, and H. Debar. Honeypots: practical means to validate malicious fault assumptions. In *Dependable Computing, 2004. Proceedings. 10th IEEE Pacific Rim International Symposium on*, pages 383–388, March 2004.

[92] M. Cukier, R. Berthier, S. Panjwani, and S. Tan. A statistical analysis of attack data to separate attacks. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 383–392, June 2006.

[93] Shuo Chen, Zbigniew Kalbarczyk, Jun Xu, and Ravishankar K Iyer. A data-driven finite state machine model for analyzing security vulnerabilities. In *null*, page 605. IEEE, 2003.

[94] A. Sharma, Z. Kalbarczyk, J. Barlow, and R. Iyer. Analysis of security data from a large computing organization. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 506–517, June 2011.

[95] C. Colombo, R. Mizzi, and G. Pace. SMock A Test Platform for Monitoring Tools. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, volume 8174 of *Lecture Notes in Computer Science*, pages 352–357. Springer Berlin Heidelberg, 2013.

[96] J.P. Magalhaes and L.M. Silva. Anomaly detection techniques for web-based applications: An experimental study. In *Network Computing and Applications, 11th IEEE Int'l Symposium on*, pages 181–190, Aug. '12.

[97] Zabbix. http://www.zabbix.com/.

[98] S. E. Hansen and E. T. Atkins. Automated system monitoring and notification with swatch. In *Proceedings of the 7th USENIX Conference on System Administration*, LISA '93, pages 145–152, Berkeley, CA, USA, 1993. USENIX Association.

[99] Apache JMeter. http://jmeter.apache.org/.

[100] M. Cinque, D. Cotroneo, R. Della Corte, and A. Pecchia. What logs should you look at when an application fails? insights from an industrial case study. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 690–695, June 2014.

[101] M. Cinque, D. Cotroneo, R. Della Corte, and A. Pecchia. Assessing direct monitoring techniques to analyze failures of critical industrial systems. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 212–222, Nov 2014.

[102] M. Cinque, D. Cotroneo, R. Della Corte, and A. Pecchia. Characterizing direct monitoring techniques in software systems. *to appear in IEEE Transaction on Reliability.*

[103] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.-Y. Wong. Orthogonal Defect Classification-A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering*, 18:943–956, 1992.

[104] J.A. Duraes and H.S. Madeira. Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.

[105] Michael R Lyu et al. *Handbook of software reliability engineering*, volume 222. IEEE computer society press CA, 1996.

[106] M. W. Berry, Z. Drmac, and E. R. Jessup. Matrices, vector spaces, and information retrieval. volume 41, pages 335–362. Society for Industrial and Applied Mathematics, June 1999.

[107] A. Pecchia, D. Cotroneo, R. Ganesan, and S. Sarkar. Filtering security alerts for the analysis of a production saas cloud. In *Utility and Cloud Computing, IEEE/ACM 7th Int'l Conference on*, pages 233–241, Dec '14.

[108] C. Lonvick. The bsd syslog protocol. *Request for Comments 3164, The Internet Society, Network Working Group, RFC3164*, August 2001.

[109] J. D. Murray. *Windows NT event logging - help for developers, system administrators, and security administrators.* O'Reilly, 1998.

[110] R. Natella, D. Cotroneo, J.A. Duraes, and H.S. Madeira. On fault representativeness of software fault injection. *Software Engineering, IEEE Transactions on*, 39(1):80–96, Jan 2013.

[111] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa. Experimental analysis of binary-level software fault injection in complex software. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 162–172, May 2012.

[112] Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Comput. Surv.*, 48(3):44:1–44:55, February 2016.

[113] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *Proc. Int'l Conference on Dependable Systems and Networks (DSN 2008)*, Anchorage, Alaska, June 2008.

[114] J. Stearley and A. J. Oliner. Bad words: Finding faults in spirit's syslogs. In *Proc. Int'l Symposium on Cluster Computing and the Grid (CCGRID 2008)*, pages 765–770.

[115] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32.