Empirical Software Engineering (2020) 25:2450–2484
https://doi.org/10.1007/s10664-020-09801-2

An empirical analysis of error propagation in critical software systems

Marcello Cinque[1] · Raffaele Della Corte[1] · Antonio Pecchia[1]

Published online: 13 March 2020

Abstract
Error propagation analysis is a consolidated practice to gain insights into error modes and effects that pertain to the activation of faults in software systems. A variety of approaches, such as architecture-based, source code instrumentation and variable tracing, have been proposed so far to address software error propagation analysis. Although valuable, existing approaches entail a substantial degree of system internals' knowledge, visibility and code manipulation that is not well-suited for real-life production environments. This paper proposes an empirical analysis of error propagation. We specifically address the challenges in using fault data and error events in the logs, which are a convenient byproduct of the system's execution. The approach puts forth the construction of error reporting graphs. We apply the approach to 2,042 failure data points from two real-world critical systems from the Air Traffic Control domain by a top industry provider. The approach contributes to develop a deep understanding on error modes and propagation paths, which can be leveraged by practitioners to make informed decisions on the placement of error detection mechanisms.

Keywords Error analysis · Error propagation · Critical systems · Monitoring

1 Introduction

Error propagation analysis is a consolidated practice to gain insights into the dependability of software systems. It allows to infer error modes, intermediate paths and effects

Communicated by: Natalia Juristo

✉ Raffaele Della Corte
raffaele.dellacorte2@unina.it

Marcello Cinque
macinque@unina.it

Antonio Pecchia
antonio.pecchia@unina.it

[1] Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione (DIETI), Università degli Studi di Napoli Federico II, via Claudio 21, 80125, Napoli, Italy

that pertain to the activation of faults.[1] Analysis of propagation allows assessing the error behavior of a system, inferring error-prone components, and establishing *where-what* type of errors are likely to cause system-wide failures (Avizienis et al. 2004). This is of utmost importance to support practitioners in making **informed decisions** for designing and placing *error detection mechanisms* (EDMs) and *error recovery mechanisms* (ERMs) (Arora and Kulkarni 1998). To this aim, many existing approaches rely on quite *convoluted* data sources that entail a substantial degree of system internals' knowledge and source code visibility. For example, Jhumka and Leeke (2011), Abdelmoez et al. (2004), Popic et al. (2005), Cortellessa and Grassi (2007), and Voas (1997) require operation details, such as *states* and *failure rates*, for each system component, Hiller et al. (2004), Hiller et al. (2002a), Leeke and Jhumka (2010), and Michael and Jones (1997) leverage data obtained by instrumenting variables, while Tucek et al. (2007) uses dynamic binary instrumentation.

The application of these approaches is far from being seamless when the systems under analysis allow a limited degree of intervention and/or provide a limited view of system internals. This is a common scenario in legacy and OTS-based systems, critical software systems and production environments. Moreover, we observe that above-mentioned literature on error propagation –although valuable– falls short when it comes to adopt *log files*, which are used to collect error events by built-in detection mechanisms, such as event logging and assertion checking.

Log files –or simply *logs*– are a byproduct of the system execution and contain text messages on regular and error events encountered by a system under real workloads (Li et al. 2018; Kabinna et al. 2018). Current systems ubiquitously emit log files. Analysis of logs is well-consolidated for troubleshooting field failures (Kalyanakrishnam et al. 1999; Tian et al. 2004; Chuah et al. 2015; Russo et al. 2015); when needed, analysis is accompanied by a debugging phase that typically leads to the *identification* and *fix* of the fault that caused the failure. Examples of works leveraging log files are Yuan et al. (2010) and Lyu et al. (1996). The approach in Yuan et al. (2010) uses logs to assess *control*- and *data*-flow; however, it requires static code analysis and is not originally conceived for error propagation. In Lyu et al. (1996) debug data are leveraged to build error propagation graphs; however, the approach neglects error messages generated by the system under analysis, which prevents to obtain runtime information about the propagation of errors through system components.

This paper proposes an empirical analysis of error propagation. Analysis is based on logs, which are naturally emitted by a system. We do not address resource consumption metrics, such as CPU, memory and network usage, which are out the scope of this work. We face the research challenge of obtaining insights into error modes and their propagation by means of logs.

We analyze faults and error events in the logs related to 2,042 failures of two real-world mission critical software systems: a **middleware** for data distribution and a **standalone application** for the management of flights and runaway control both used in the Air Traffic Control (ATC) domain by a top leading industry provider in electronic and information solutions for critical systems.[2] We put forth an analysis approach based on the construction of **error reporting graphs**. The paper addresses the formalization of **error modes** from data and proposes a set of novel metrics, such the **error propagation reportability** (EPR), which

[1]In this study, we follow the notion that a software fault is a development fault originated during the coding phase. Faults can be activated by the computation process or environmental conditions and cause errors. An error is the part of the total state of the system that may lead to its subsequent service failure. A failure occurs when the delivered service deviates from correct service (Avizienis et al. 2004).

[2]The evaluation version of these systems, testing applications and workloads are provided by the industry partner within the *MINIMINDS* Project (n. B21C12000710005).

are computed from the graphs to quantify the error behavior of the system under assessment. We rely on a representation of the input data that allows *decoupling* our approach from the data sources, which makes many steps of the approach potentially automated.

The approach contributed to develop a deep understanding on error modes, propagation paths and capabilities of the error reporting mechanisms, which provided actionable insights to the industry provider for improving error detection. The *key findings* of our empirical study are:

- *With respect to the error modes –and their granularity– adopted in this study, different fault types lead to a small subset of error modes*, which mainly concern *type* and *value* of variables. For example, *data type* errors and *unexpected value* errors are the most reported for event logging and assertion checking, respectively; this finding is consistent with the previous literature, such as (Leeke and Jhumka 2010).
- *Early error propagation steps are mostly silent*. We observe that a software component affected by a fault might report *no* error notifications. For example, in our setup, the logs emitted by the component containing an *algorithm fault* report an error only in the 33.87% and 19.20% of cases, in the two target systems.
- *Although missed by the component originating the fault, errors might still be reported by other components along the propagation path*. For example, logs of the *database* component –belonging to the ATC middleware– report 23 out 69 *missing function* faults undetected by the originating faulty component elsewhere. A similar finding is noted for assertion checking and logs generated from the arrival manager stand-alone application.
- *Latest error propagation steps determine the type of failure that will be encountered by the system.* Our study reveals a strong relation between the last components reporting errors and the type of failure observed. This provides insights on the errors that should be handled to avoid failures.
- *The analysis of graphs guides the improvement of the error detection mechanism of a complex software system, and allows to quantify the extent of the improvement itself.* By analyzing the graphs, practitioners can identify the components where to place more EDMs; experiments on the ATC middleware after the placement of new EDMs highlight an improvement, in terms of the error propagation reportability for *algorithmic faults*, from 33.87% up to 94.50%.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 presents the systems under assessment and the datasets. Section 4 describes the proposed approach and the metrics. Section 5 discusses the error modes inferred from data, while Section 6 presents the insights achieved from error reporting graphs. Threats to validity are discussed in Section 7, while Section 8 concludes the paper.

2 Related Work

We position our research with respect to existing work on software error propagation analysis, distinguishing them in architecture- and metrics-based approaches, and code instrumentation.

2.1 Architecture- and Metrics-Based Approaches

Several approaches that address software error propagation require a substantial degree of knowledge on *system internals*, such as architectural dependencies between system components and evaluation of software metrics; moreover, some of the approaches are accompanied with static analysis of the source code.

The approach presented in Jhumka and Leeke (2011) leverages *module coupling* to identify potential data-value error detector locations at module-level. Coupling is evaluated using information about modules, e.g., input and output data/control parameters, data/control global variables, number of called/calling modules. The approach is used in an open-source flight simulator. Authors in Abdelmoez et al. (2004) propose a static analytical approach that leverages architecture specifications to estimate the probability of error propagation in a software architecture. The approach is based on a metric, named *error propagation probability*. Evaluation requires architectural-level data, such as *states* of components and *messages* they can exchange. A similar approach is adopted in Filieri et al. (2010) to component-based systems. Authors propose a methodology, based on a probabilistic model, to analyze the reliability of the system starting from failure modes and failure probabilities of its components. The approach requires detailed architectural information on how the components are assembled, in terms of input/output ports and their connections. The model includes the formalization of probabilistic error propagation among components' ports.

In Popic et al. (2005) an existing Bayesian methodology is extended for reliability prediction of component-based software systems for error propagation. The methodology leverages the error propagation probability metric and it requires the knowledge of *failure rates* of components, under the assumption of failure independency; each component is assumed to exhibit the same failure rate. Similarly, the work in Cortellessa and Grassi (2007) leverages the error propagation probability and requires detailed information on each component, such as, unconditional and conditional (e.g., subject to a given correct input) *failure probability* and *operational profile*. The method has been shown to be beneficial for the placement of error detection and recovery mechanisms.

The impact of inter-modular data error propagation is assessed in Jhumka et al. (2001). The work characterizes data error propagation and derives a set of metrics that quantify inter-modular interactions. Results indicate that the metrics allow to identify candidate modules to be equipped with detection/recovery mechanisms. In Khoshgoftaar et al. (1999) it is presented an approach to identify software modules that do not propagate data errors. The work demonstrates –through experimentation on the Nethack adventure game– that static software metrics are good predictors for the identification of such modules, avoiding the evaluation of their error propagation probability.

The approach proposed in Voas (1997) studies information flows between components of a system. The approach is based on the corruption of the information flowing through components and the observation of its impact during execution, in order to isolate the components that cannot tolerate failures of the other ones. *SherLog* (Yuan et al. 2010) is a diagnosis tool that analyzes the source code and *event logs* it generates at runtime during the occurrence of failures, to automatically provide control-flow and data-flow information.

An approach to evaluate error propagation from *debug data* is presented in Lyu et al. (1996). The approach allows building error propagation graphs from the reports generated by analysts after the occurrence of failures. The graphs provide information about the fault causing the failure, the type of the first error, the error propagation mode and how the error has been detected.

Our work aims to overcome the drawbacks that threaten the application of architecture- and metrics-based approaches in real-life production environments. For example, with respect to Jhumka and Leeke (2011), Abdelmoez et al. (2004), Popic et al. (2005), Cortellessa and Grassi (2007), Voas (1997), and Filieri et al. (2010) our proposal does not require detailed information on software components (such as, input and output data/control parameters, undesirable states, failure rates), which are difficult to retrieve especially when the system is based on OTS and legacy components. The work in Zheng and Lyu (2010) questions the use of such component-level models for reliability prediction, especially when applied on web services, due to the lack of detailed system information and network unpredictability. Similarly to our work, they propose to collect real data about the failures affecting the system, even if for a different purpose (reliability prediction). But, differently from us, they rely on a users-based collaborative framework to collect past failure data from past experiences with the web services to be composed. With respect to Jhumka and Leeke (2011), Jhumka et al. (2001), and Khoshgoftaar et al. (1999) our approach is not limited to data errors and it does not require to monitor the output of each component (Voas 1997). In addition, with respect to Yuan et al. (2010), our approach does not require static analysis of the source code, which can be either expensive in a large system or inapplicable when the code is not available. Finally, with respect to Lyu et al. (1996) our proposal leverages error messages naturally emitted from the target system to perform error propagation analysis, which allows obtaining valuable information about the propagation of errors through the system components.

2.2 Code Instrumentation Approaches

Code instrumentation approaches capitalize on monitoring code (either at source code or binary level) to generate error traces upon fault activation (Lattner and Adve 2004; Cinque et al. 2013).

EPIC (Hiller et al. 2004) is a framework based on *variable instrumentation* to trace the value of variables in order to estimate an *error permeability* metric (which evaluates the ability of a module to contain errors) and to place EDMs. *PROPANE* (Hiller et al. 2002a) analyzes the propagation of data errors in single-process C software systems, and identifies error paths and propagation frequency. *PROPANE* is based on a fault injection approach to induce data errors in the system and *variable instrumentation* to detect errors. In Cinque et al. (2013) is proposed a set of *logging rules* for the placement of log statements in the source code, in order to generate error traces upon the activation of software faults.

The work (Johansson and Suri 2005) proposes an approach for the analysis of errors in Windows CE .Net device drivers, to study how errors propagate to applications. Data errors are induced by means of fault injection at interface level, while propagation is analyzed by instrumenting the code with *assertions*. A set of metrics is proposed to evaluate if the target driver needs a wrapper to handle the errors.

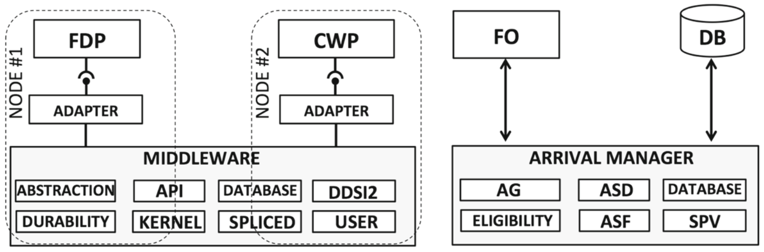
The work (Leeke and Jhumka 2010) introduces the *importance* metric to measure the impact a given variable has on the dependability of a software system. The evaluation of the metric requires to instrument the variables in order to understand when a variable is corrupted. The approach provides insights on the design and positioning of error detection and recovery mechanisms. An open-source flight simulator has been used to assess the proposal. In Tucek et al. (2007) authors propose a system, called *Triage*, that automatically performs onsite software failure diagnosis. The system makes use of both kernel-level components and multiple re-executions of the target software to support failure diagnosis; during each re-execution, detailed data are collected via *dynamic binary instrumentation* to conduct the analysis of occurred failure and its causes.

An approach for error propagation analysis using invariants is presented in Chan et al. (2017). The approach, named *IPA* (Invariant Propagation Analysis), automatically derives invariants for multithreaded programs by instrumenting the source code at function entry and exit points. The approach has been evaluated with different fault types across six programs through fault injection experiments. An error propagation study for MPI applications is presented in Calhoun et al. (2017). The paper investigates how *Silent Data Corruption* due to soft errors propagates through HPC applications. An LLVM-based tool is developed to instrument MPI applications in order to inject faults and track error propagation at *instruction* and *application variable* level. The tool has been applied to three HPC applications.

Differently from these studies, we propose to capitalize on the data already produced by the system, such as log files produced by event logging and/or assertions already available in the source code. The idea is also to receive feedback on how these error reporting techniques work and if/how they should be improved for better error propagation analysis. On the opposite, the use of code instrumentation approaches is not straightforward in production environments, for systems adopting OTS, or when there is limited knowledge on system internals. For example, the approaches (Hiller et al. 2004; 2002a; Leeke and Jhumka 2010; Calhoun et al. 2017; Chan et al. 2017) require instrumenting variables or function entry/exit points, which might be expensive in a complex software system encompassing several components and even not applicable if the source code is not available. In addition, the approach in Hiller et al. (2004) requires measuring the error permeability for each input of each module, leading to a low scalability of the approach; while the tool (Hiller et al. 2002a) addresses only single process software. The system proposed in Tucek et al. (2007) uses kernel-level components and dynamic binary instrumentation, which is not allowed in critical production environments (e.g., mission critical systems) with stringent constraints imposed by certification standards and the use of obsolete kernel versions. Finally, the approaches (Hiller et al. 2004; 2002a; Johansson and Suri 2005) only address data errors, while those presented in Johansson and Suri (2005) and Calhoun et al. (2017) are conceived only for OS device drivers and MPI applications, respectively.

3 Systems and Datasets

Datasets available in this study consist of faults and error events that pertain to total 2,042 distinct failures of two systems. We analyze a middleware and a standalone application – named *arrival manager*– both used by the industry provider in the critical domain of the Air Traffic Control (ATC). In the following we present systems, testing applications and error detection mechanisms beforehand; then, we describe how faults and error events are arranged into tabular **failure data instances** for propagation analysis.



(a) Middleware (MW). (b) Arrival Manager (AM).

Fig. 1 Overview of the systems

3.1 Description of the Systems

Middleware (MW) The middleware assessed in this study is an OMG-compliant data distribution service (DDS) layer among heterogeneous ATC applications. It provides a message-oriented application programming interface (API), which is based on the publish-subscribe paradigm and topics. Figure 1a shows a typical deployment of the middleware by the industry provider, where a *flight data processor* (FDP) and a *controller working position* (CWP), i.e., two ATC applications, generate the messages exchanged through the middleware. The source code of the middleware consists of 796,353 lines of C code, organized into 8 components,[3] depicted in Fig. 1a:

– *abstraction*: level between middleware / operating system;
– *api*: API provided to applications;
– *database*: bridges data to a DB and vice versa;
– *ddsi2*: provides QoS-driven real-time networking based on multiple reliable multicast channels;
– *durability*: implements fault-tolerant storage for both state data and persistent settings;
– *kernel*: the core of the middleware;
– *spliced*: it is responsible for creating and initializing the database used to manage the middleware data;
– *user*: intermediate level between *api* and *kernel* components.

FDP and CWP are testing applications provided by the industry partner and serve as load *generators* to exercise the middleware. FDP and CWP implement a workload, i.e., the library of inputs that a generator submits to the target system (Hsueh et al. 1997), consisting of messages that are published under certain topics. Messages and topics reflect the nominal usage profile of the middleware by ATC operators. The leftmost column of Table 1 shows the top 10 invoked functions of the middleware over a sample of 15,409 invocations of 118 distinct functions. It is worth noting that the usage profile exercises the principal entities of the OMG DDS model,[4] such as Data Reader/Writer, Publisher/Subscriber and Topic.

Arrival Manager (AM) This a standalone ATC application, which is intended to assist human operators in optimizing the runway capacity and regulating the flow of aircrafts entering a given airspace. AM is fundamentally different from the middleware described above and is maintained by a different development team. AM continuously computes an *optimal* list of flight arrivals based on different parameters, such as the landing rate and spacing requirements. The application consists of 40,396 lines of C++ code. A high-level view of AM is given in Fig. 1b, which is characterized by 6 *components*:

– *AG (Arrival Generator)*: computes the arrival list and timing of flights based on the landing rate, spacing and other parameters;
– *ASD (Aircraft Situational Display)*: manages the position and flight data, e.g., location, altitude, airspeed, of aircrafts;
– *Database*: it is responsible for the interaction with a DB;

[3]Consistently with the software engineering terminology, we mean by **component** a software unit encompassing a cohesive subset of functionality provided by a given system; a **subcomponent** is a subset of functionality within the component (Lau and Wang 2007).

[4]https://www.dds-foundation.org/what-is-dds-3/

Table 1 Top 10 invoked functions by system

Middleware (MW)		Arrival Manager (AM)	
(%)	function	(%)	function
7.85	`v_groupUpdatePurgeList`	19.16	`assign`
7.08	`v_topicMessageNew`	9.77	`OnTimer`
6.94	`v_dataReaderEntryWrite`	9.64	`OnTimerStateEventTransition`
6.94	`v_messageQos_isReaderCompatible`	6.78	`MBXReceivePacket`
5.96	`v_checkMaxSamplesWarningLevel`	5.45	`SeqItem`
5.43	`v_objectNew`	5.32	`freeLists`
5.04	`v_messageQos_getTransportPriority`	4.82	`EndWork`
5.04	`instanceTakeSamples`	4.82	`StartWork`
3.74	`v_messageQos_getLifespanPeriod`	4.82	`GetNodeRole`
3.47	`v_dataReaderInstanceWrite`	4.82	`GetApplicationStatus`

– *Eligibility*: at any time provides the list of flights within the *eligibility* horizon (i.e., close enough to be handled by the AMG component);
– *ASF (Aircraft Surveillance Function)*: determines the position of aircrafts;
– *SPV (Supervisor)*: supervises OS processes that underlie the execution of the AM.

Again, the system is provided by the industry partner along with a testing application called *flight orders* (FO in Fig. 1b). The workload implemented by FO is a sequence of requests that consist *insert* and *delete* flights, which emulate aircrafts entering/leaving an airspace. Requests reflect the nominal usage profile of the AM in production. The rightmost column of Table 1 shows top 10 invoked functions of AM over a sample of 18,520 invocations of 202 distinct functions. All the components in Fig. 1b are represented within the functions.

3.2 Error Detection

The systems assessed in this study *natively* implement **event logging** (EL) mechanisms to detect errors. EL consists of dedicated instructions that are inserted by developers during the coding phase with the aim of reporting error events at runtime upon certain conditions. Figure 2 (lines 4-7) shows a snippet of EL. The code produces an error event whenever the variable `newQos` equals to `NULL`, which was judged to be an error symptom by developers at coding time. The source code of both MW and AM contains a large number of logging points to catch potential error conditions. Regarding MW, we also consider error detection by means of **assertion checking**[5] (AC). Figure 2 (lines 16-18) shows a snippet of AC code from MW, where it is checked value and type of some parameters passed to the function `v_writerNew`.

Runtime events produced by EL and AC are typically stored into files –also known as *logs*– for post-mortem analysis. Logs are a byproduct of the system's execution. In the systems assessed in this study, events produced by both EL and AC are written in the logs along with some context fields, such as **name of file** and **function**, which contain the logging/assertion

[5] An assertion checks invariant properties holding in correct executions; an alert is generated if an invariant is violated at runtime (Rosenblum 1995).

```
 1                    Event logging (v_topic.c, line 488)
 2
 3 newQos = v_topicQosNew(kernel,qos);
 4 if (newQos == NULL) {
 5      OS_REPORT_1(OS_ERROR, "v_topicNew", 0,
 6              "Topic '%s' is not created: inconsistent qos",
 7              name);
 8     return NULL;
 9 }
10
11                 Assertion checking (v_writer.c, line 1670)
12
13 v_writer v_writerNew(v_publisher p, const c_char *name,
14          v_topic topic, v_writerQos qos, c_bool enable){
15     //...
16     assert(p != NULL);
17     assert(C_TYPECHECK(p, v_publisher));
18     assert(C_TYPECHECK(topic, v_topic));
```

Fig. 2 Snippets of event logging (lines 4-5) and assertion checking (lines 14-16) code

instructions that generated the event. The following lines show concrete examples of error events found in the logs of the middleware, i.e., EL (lines 1-4) and AC (lines 5-6).

```
1 v_topicNew v_topic.c  Description: Topic 'DCPSParticipant' is
2     not created: inconsistent qos
3 u_serviceFree u_service.c  Description: Operation u_serviceDeinit
4     failed Service=0x67934e0, result=U_RESULT_PRECONDITION_NOT_MET.
5 v_writer.c 2031  v_writerFree  Assertion '(w == c_checkType(w,
6     "v_writer"))' failed.
```

It is worth noting that lines 1-2 are produced by the instruction in Fig. 2 (line 5) at runtime; all the lines are accompanied by names of files (e.g., `v_topic.c`, `u_service.c`) and functions (e.g., `v_topicNew` and `u_serviceFree`).

3.3 Datasets

Data used in this study were collected with a campaign of experiments performed in a controlled monitoring setup (Cinque et al. 2016). Given a system under test –either MW or AM– each experiment consisted in (i) injecting a **software fault** into the system, (ii) exercising the system by means of the testing applications described in Section 3.1, (iii) observing/classifying the consequent failure, and (iv) storing the errors reported in the logs by either EL or AC. Noteworthy, a fault injection experiment does not necessarily cause a failure; moreover, a failure might go *unreported*, i.e., logs contain no error events at all by either EL or AC.

Fault types used in the experiments follow the ODC classification proposed in Chillarege et al. (1992) and subsequent refinement by Duraes and Madeira (2006), which are widely-accepted by the software engineering community. Table 2 summarizes the types pertinent to our study and mapping to corresponding ODC class.

Failure types denote the nature of the deviation with respect to the correct service expected by the system under test. Types are based on the well-established taxonomy in Avizienis et al. (2004):

– *CRASH*: abrupt termination of the system;
– *SILENT*: the system is up, but no output/functionality is provided within an expected timeout;

Table 2 Fault types used in the experiments (*ALG*-algorithm, *ASG*-assignment, *CHK*-checking, *INT*-interface)

Type		ODC
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	*INT*
WAEP	wrong arithmetic expression in parameter of a function call	*INT*

– *ERRATIC*: bad output, exceptions, and other malfunctions that do not cause *CRASH* or *SILENT*.

In this study we consider the failures that are reported by either EL or AC with at least one error event in the logs. Table 3 shows the total number of failures that meet this criteria after the fault injection campaigns. For each system we provide the breakdown of the total failures by fault type and detection technique. For example, the value 69 in the cell (*MFC, EL-MW*) indicates that 69 failures caused by an *MFC* fault injected in the middleware (MW), are reported by at least one error event generated by EL. Overall, failures are grouped into three datasets, which are denoted by EL-MW, AC-MW and EL-AM hereinafter. The bottom row of Table 3 shows the cardinality of the datasets. For example, EL-MW is the set of failures of MW that are reported by EL. Noteworthy, 346 failures of the MW are reported by both EL and AC and –in turn– counted twice in EL-MW and AC-MW; as such, the datasets account for total 2,042 distinct failures.

3.4 Notion of Failure Data Instance

For each failure in the datasets, the corresponding fault and error events are arranged in a more convenient *table* format for the purposes of this study. The table will be referred as **failure data instance** hereinafter. In this study, tables are crated by means of *bash scripts*, which normalize faults and error events available across various files and formats produced after the controlled injection campaigns. Table 4 shows the general format of a failure data instance, which is populated with real data from the middleware system. The table consists of two sections. The former, i.e., **Debug data** (D), includes location/type of the fault leading to the failure, and consequent failure type; the latter, i.e., **error events** (E), are the lines in the logs by either EL or AC generated upon the occurrence of the failure. We regard the former section of the table as "debug data" because it is intended to contain the outcome of a typical debugging process. While in this study it is populated with faults, locations

Table 3 Total number of failure data instances by fault type and reporting mechanism (EL, AC)

		MW		AM
ODC	fault type	EL-MW	AC-MW	EL-AM
ALG	MFC	69	160	9
	MIEB	26	29	0
	MIFS	12	44	0
	MLPA	262	684	57
ALG total		**369**	**917**	**66**
ASG	MVAE	175	364	1
	MVAV	8	14	0
	MVIV	0	3	0
	WVAV	12	16	0
ASG total		**195**	**397**	**1**
CHK	MIA	15	27	1
	MLC	2	2	0
CHK total		**17**	**29**	**1**
INT	WAEP	11	36	0
	WPFV	122	227	0
INT total		**133**	**263**	**1**
total		**714**	**1,606**	**68**

and failures obtained by means of controlled injections, debug data can be typically found within bug trackers in response to field failures, as discussed later on in this section.

As it can be noted from Table 4, the failure data instance is characterized by component/ subcomponent of fault and error events. In the context of the systems in hand, components are listed in Section 3.1. We establish the set of components by analyzing the software documentation and direct discussion with the industry partner. For each component the industry

Table 4 Example of failure data instance from the EL-MW dataset

Debug data (D)				
	Component (C)	*Subcomponent (SC)*	*Fault type (f)*	*Failure type (F)*
	`kernel`	`v_networkReaderNew`	`WPFV`	`SILENT`
Error events (E)				
ID	*Component (C)*	*Subcomponent (SC)*	*Text Message (M)*	
E[0]	`kernel`	`v_networkReaderNew`	`NetworkReader not created inconsistent qos`	
E[1]	`kernel`	`v_readerQosNew`	`ReaderQos not created inconsistent qos`	
E[2]	`ddsi2`	`main`	`Creation of NetworkReader failed`	
E[3]	`user`	`u_networkReaderNew`	`Create kernel entity failed. For NetworkReader <networkReader>`	

partner shared the list of corresponding source code files. As such, we could correctly map faults and error events to the originating component based on the knowledge of the source file. Example of components are *kernel* and *ddsi2* in Table 4; functions belonging to each component, such as *v_networkReaderNew*, *v_readerQosNew* are regarded here as *subcomponents*. In Table 4, a *wrong variable used in parameter of function call* (WPFV fault type - f), located in the *v_networkReaderNew* subcomponent (SC) of the *kernel* component (C), causes a *silent* failure (failure type - F). Error events are reported by four functions, i.e., *v_networkReaderNew*, *v_readerQosNew*, *main* and *u_networkReaderNew* –belonging to the kernel, ddsi2 and user *component*, respectively– as shown by the bottom rows of Table 4.

The *failure data instance* provides a representation that aims to **decouple** our analysis approach from the data that can be encountered in practice. For example, in our study faults follow the ODC types and failures are based on Avizienis et al. (2004). However, the analysis approach does not depend on the naming scheme of faults-failures. Similarly, the definitions of *component* and *subcomponent* can be adapted to different systems. Moreover, as it will be clarified in Section 4, our analysis approach can be used –although at *coarser* grain– even if failure data instances miss some features, such as, fault-failure type or the distinction between component/subcomponent.

While our empirical analysis relies on failures collected in a controlled setup, we would like to point out that debug data can be inferred from bug reports and patches –usually available in bug trackers– created in response to field failures. Let us discuss a real-life motivating example of bug report from the Tomcat[6] server. The report clearly states the faulty component and function, i.e., Catalina and WsRemoteEndpointImplBase, respectively; moreover, it is accompanied by the error events observed in the field. By looking at the description provided by the user, which states "*the browser waits for a response forever*" it can be assumed that a `SILENT` failure occurred. Finally, the analysis of the patch released –consisting of several additions and fixes of the code– makes it possible to state that the fault was a MLPA.

4 Proposed Approach

Our analysis approach infers a representation, namely, **error reporting graph**, of the errors leading from faults in a given component to failures. The representation is based on *directed graphs*. Graphs have been already used in the context of error propagation analysis, e.g., Lyu et al. (1996), Jhumka et al. (2001), and Hiller et al. (2004), because they can be easily understood by practitioners.

Let FDI denote a set of failure data instances where the originating component of the fault is the same. We use an iterative approach. For each instance in FDI we obtain one **reporting path**, beforehand; the reporting path is merged with the error reporting graph. Steps of the analysis are summarized by Algorithm 1, which highlights input-output of each step. We discuss the steps in the following by means of the illustrative example of failure data instance shown in Table 4.

4.1 Construction of the Reporting Table

A failure data instance is processed in order to generate a data structure called **reporting table** as per Algorithm 1 (line 3). The reporting table enriches each error event that accompanies the failure data instance with two attributes: (i) *reporting stage* and (ii)

[6]https://bz.apache.org/bugzilla/show_bug.cgi?id=54711#attach_30061

error mode. An example of reporting table is given in Table 5. Let us describe the attributes in the following.

Algorithm 1 Steps underlying our analysis approach.

```
input  : set of failure data instances (FDI)
output: error reporting graph (ERG)
1  $ERG \longleftarrow \emptyset$;
2  for $fdi \in FDI$ do
3      $reportingTable \longleftarrow buildReportingTable(fdi)$;
4      $reportingPath \longleftarrow buildPathFromTable(fdi, reportingTable)$;
5      $ERG \longleftarrow graphUpdate(ERG, reportingPath)$;
6  end
```

4.1.1 Reporting Stage

The **reporting stage** indicates the spatial *closeness* of the error event with respect to the location of the fault. We use an *object-like* notation to obtain component (C) and subcomponent (SC) of faults and error events. Let us denote by D the *debug data* section of the failure data instance: as such, D.C is the component of the fault, e.g., *kernel* in Table 4. Similarly, let us denote error events by E[i], with 0≤i≤(N-1); for example, E[0].C in Table 4 returns *kernel*, while E[1].SC is *v_readerQosNew*. For a given error event E[i], the reporting stage is obtained automatically and assumes one of the following values, which are adapted from Lyu et al. (1996):

- **immediate (I)**: the subcomponent that reports the error is also the location of the fault, i.e., (D.C==E[i].C) && (D.SC==E[i].SC);
- **quick (Q)**: the subcomponent that reports the error is not the location of the fault, although it belongs to the same component, i.e., (D.C==E[i].C) && (D.SC!=E[i].SC);
- **last (L)**: the subcomponent that reports the error is not the location of the fault and belongs to a different component, i.e., (D.C!=E[i].C) && (D.SC!=E[i].SC).

In case failure data do not come in the granularity of component-subcomponent, our approach is applied without distinguishing between immediate and quick stages. The third column of Table 5 shows the reporting stage of the events in Table 4 according to the rules mentioned above. For example, E[1] is assigned "quick" because E[1].SC is *v_readerQosNew* and D.SC is *v_networkReaderNew*, and thus different; however, E[1].C and D.C are both *kernel*.

Table 5 Reporting table corresponding to the events shown in Table 4

	Comp.	Reporting stage	Error mode	ID
E[0]	kernel	Immediate (I)	quality of service	e2
E[1]	kernel	Quick (Q)	quality of service	e2
E[2]	ddsi2	Last (L)	unexpected result	e3
E[3]	user	Last (L)	kernel entities	e9

4.1.2 Error Mode

The **error mode** is a short description of the *mode* of the error. The mode is established manually *only* at the first occurrence of an error event; automatically, for future occurrences of the same event. The process is illustrated by Fig. 3.

Manual Inspection We scrutinize the event in order to gain insights into the cause of the error. Analysis is supplemented with the software documentation, on-line forum searches, and source code inspection. In the context of our data, E[0] in Table 4 is a "*quality of service*" error, while E[2] denotes an "*unexpected result*" error, which represent examples of error modes. Assigning a mode to events –in order to create a dictionary/taxonomy– is a well-known practice of log analysis. It is a cognitive process and requires a trade-off. For example, if a mode is too generic, its resolution might be small for subsequent analysis. As such, we took a balanced approach by avoiding both overgeneralization and excessive fragmentation.

Once the event is scrutinized, we i) extract the **template** of the event, where the variable parts of the event are replaced with a generic wildcard, ii) formalize a regular expression to match future occurrences of the same template, and iii) assign it to an error mode ei. For example, the template of the text message in E[0] is `* not created inconsistent qos` where the token "*NetworkReader*" is replaced with *.

Error Model Base It contains the results of the manual inspections, i.e., templates and error modes, as shown in Fig. 3. Extracting templates from text logs is a common step in field data studies (Makanju et al. 2012). It should be noted that, in spite of the potentially large number of events, the number of *unique* templates is significantly lower, and thus addressable by human experts. In our study, out of total 63,546 error events reported by event logging we identified 258 unique templates. Templates are grouped by error mode because different templates might account for the same error mode. Table 6 shows some of the templates for the mode *memory error*, which have been extracted by means of event logging in the middleware system; all the templates in Table 6 are related to memory allocation and de-allocation issues.

Automatic Analysis An error event is checked against the templates (and regular expressions) of the error model base, beforehand. If the check is fruitful, the event is automatically

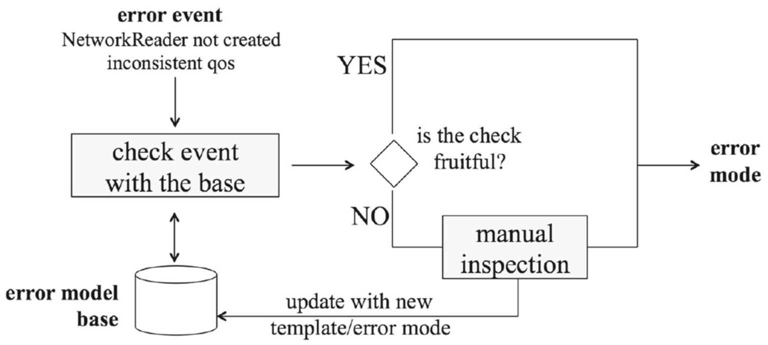

error event
NetworkReader not created inconsistent qos
check event with the base
is the check fruitful?
YES
NO
manual inspection
error mode
error model base
update with new template/error mode

Fig. 3 Assignment of modes to error events

Table 6 Examples of templates assigned to the *memory error* mode – event logging, middleware system

ID	error mode
e1	memory error
`Illegal size * specified`	
`Failed to allocate *`	
`Unable to allocate *`	
`* not created out of memory`	
`Destroy of the shared memory failed`	

marked with its corresponding error mode; if not, manual inspection takes places as discussed above and the base is updated with a new template/error mode. For example, E[1] in Table 4 –reporting an inconsistent qos– would be resolved automatically because the same mode is encountered in E[0].

Table 5 shows the reporting table corresponding to the failure data instance in Table 4. The rightmost columns denote the error modes.

4.2 Construction of the Reporting Path

The reporting table is automatically transformed in a **reporting path**, which is the second step of the approach as in Algorithm 1 (line 4). For any input failure data instance and corresponding reporting table, the reporting path can assume one out of the seven configurations shown in Fig. 4, according to the following rules:

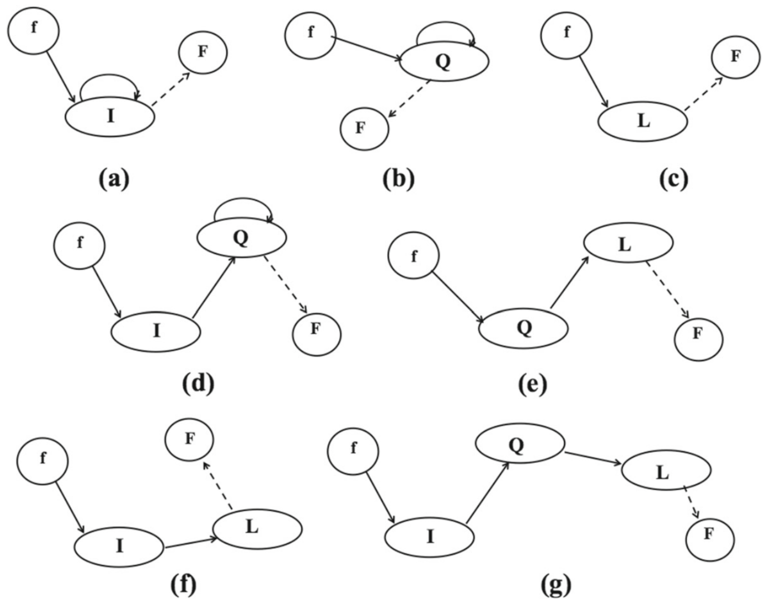


Fig. 4 Configurations of a reporting path

- R1: fault type (f) and failure type (F) are the first and last node of the path, respectively, as for all configurations in Fig. 4.
- R2: immediate (I), quick (Q) and last (L) nodes are drawn if there is at least an error event in the table with *Immediate*, *Quick* and *Last* as reporting stage, respectively.
- R3: if I exists, f is connected to I, as in configurations (a), (d), (f) and (g).
- R4: if Q exists, it is the destination node of an arc starting from either i) f, if I does not exist in the table, as in configurations (b) and (e) or ii) I, otherwise, as in configurations (d) and (g).
- R5: if L exists, it is the destination node of an arc starting from i) f, if both I and Q do not exist in the table, as in configuration (c); ii) I, if I exists and Q does not, as in configuration (f); iii) Q, otherwise, as in configurations (e) and (g). R5 places L as far as possible from f because if either I or Q exist, it means that there exists at least one error event closer to f.
- R6: F is the destination node of an arc from i) I, if I exists and both Q and L do not, as in configuration (a); ii) Q, if Q exists and L does not, as in configurations (b) and (d); iii) L, otherwise, as in (c), (e), (f) and (g).
- R7: if L does not exist –as in configurations (a), (b) and (d)– a self-loop is drawn on the node that is directly connected to the failure type node (F). A self loop indicates that errors are reported only by the component affected by the fault.

Noteworthy, no combination in Fig. 4 encompasses arcs from the fault type node (f) to the failure type node (F) because failure data instances contain at least one error event; in consequence, there will always exist one node among I, Q and L, by construction. Let us provide a concrete example with the data in Tables 4 and 5, which lead to the reporting path in Fig. 5.

Nodes Fault (WPFV) and failure (SILENT) type are the first and last node of the path (R1). Since all stages occur in Table 5, immediate (I-KERNEL), quick (Q-KERNEL) and last (L-DDSI2-USER) nodes are drawn (R2). The names of I, Q and L nodes are obtained by concatenating I-, Q- and L- with the name of the component; if more than one component is labelled as *Last*, their names –without repetitions– are concatenated with L-, e.g., L-DDSI2-USER in Fig. 5. Reporting stages are annotated with tables that contain error modes, as in Fig. 5.

Arcs Figure 5 contains four arcs: (i) from the *fault type* to *immediate* node – R3; (ii) from the *immediate* to the *quick* node – R4; (iii) from the *quick* to *last* node – R5; (iv) from the

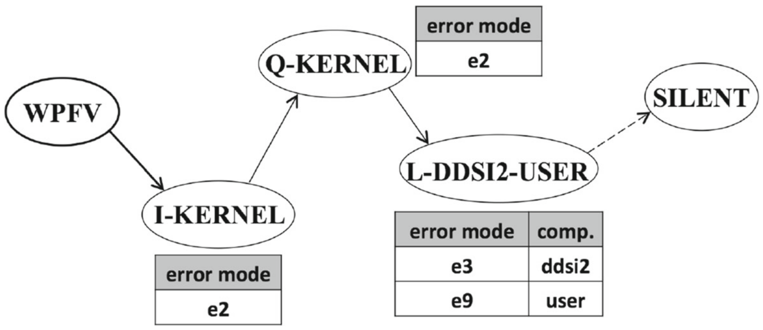


Fig. 5 Reporting path from Tables 4 and 5

last to the *failure type* node – R6. Noteworthy, the path reaches the last reporting stage; therefore, *no* self-loops are drawn on the immediate and quick nodes – R7.

4.3 Graph Update

The **graph update** step updates the current error reporting graph with an individual reporting path as per Algorithm 1 (line 5). Update consists of the **graph union operation** (Bondy et al. 1976). Let $P=(V_P, E_P)$ be the path to be inserted in the graph $G=(V_G, E_G)$, where V and E are set nodes and arcs. The union of P with G is $P \cup G = (V_P \cup V_G, E_P \cup E_G)$. In consequence, the resulting graph encompasses nodes and arcs of both P and G with no repetitions.

Figure 6 shows a general error reporting graph. The graph indicates both multiplicity (M) and Error Propagation Probability (EPP) of each node/arc. M is the number of reporting paths that contain that node/arc; EPPs are discussed in Section 4.4. For better readability, the graph encompasses one *FAILURE* node, while the failure types are shown on the arcs connected to the *FAILURE* node (*failures breakdown* in Fig. 6).

Overall, the graph provides insights into the spatial closeness of the errors with respect to faults and reporting stages (immediate, quick, last). It helps to understand how individual faults (e.g., of type X or Y) impact the system up to system-wide failures. Moreover, the graph allows inferring those cases where errors are reported only in the last stage, hence suggesting actionable improvements in terms of new EDMs; arcs to the *FAILURE* node (and associated error modes) provide indications for error handling.

4.4 Metrics Computation

We propose a set of metrics to accompany a graph: i) *Error Propagation Probabilities*, and ii) *Error Propagation Reportability*.

Error Propagation Probabilities (EPPs). EPP are computed for nodes and arcs. The EPP of a node is the ratio between the multiplicity of the node and the number of failure data instances used to obtain the graph; the EPP of an arc is computed as the ratio between the multiplicity of the arc and the multiplicity of its originating node. The interpretation of EPP –based on the specific node/arc– is given in Table 7.

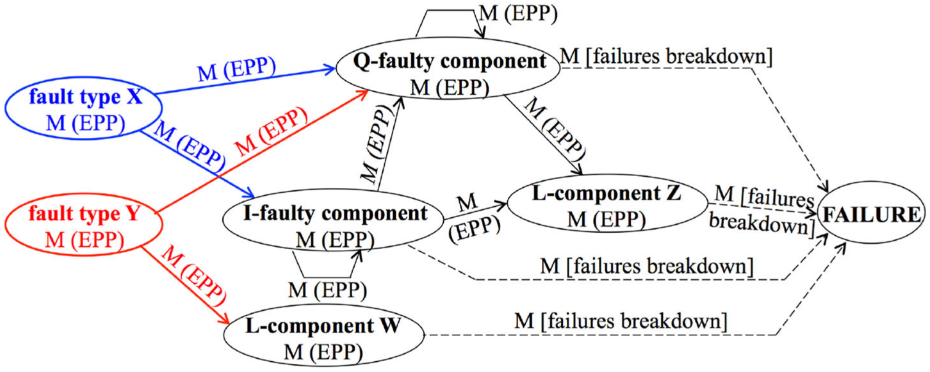


Fig. 6 Example of error reporting graph

Table 7 Meaning of error propagation probabilities

Node probability		
Node type	**Meaning**	
f	the probability that an activated fault is of type f	
I, Q, L	the probability that an error reporting component is the component the node refers to	
Arc probability		
Src node type	**Dest node type**	**Meaning**
f	I, Q, L	the probability that an error caused by an activated fault of type f is reported by the component the destination node refers to
I, Q, L	I, Q, L	the probability that an occurred error is reported by both source and destination components
I, Q, L	F	the probability that an occurred error, reported by the component, lead to a failure of type F
I, Q self-loop		the probability that an occurred error, reported by the component, is not further reported

Error Propagation Reportability (EPR). As mentioned above, the error reporting graph pertains to propagation of faults originated by a given component C. ERP quantifies the ability of the component at catching error propagation. Let i) REC (Reported Errors by the Component) be the sum of the multiplicity of the arcs from a *fault type* to either I or Q nodes (i.e., the cases where the component C reports at least an error event), and ii) RE be the sum of the multiplicities of the *fault type* nodes in the graph. The EPR for the component C is:

$$EPR_C = \frac{REC}{RE} \cdot 100$$

where EPR is in [0, 100]%. The closer EPR to 100% the higher the ability of C at catching error propagation. A low value of EPR indicates the need for improving error reporting mechanisms implemented by C.

5 Error Model

We analyze the error model obtained by applying our analysis approach to the three datasets of failures.

5.1 Event Logging - Middleware Dataset (EL-MW)

EL-MW consists of 714 failures as shown in Table 3. The error model is shown by Table 8, where a short *ID* is assigned to each mode and used hereinafter to refer that mode; an *error event* for each mode is shown for the sake of clarity.

We observe that **event logging** encompasses many errors concerning the high-level business logic and configuration of the application, such as e2-EL-MW, i.e., "*Quality of Service error*", e7-EL-MW, i.e., "*Topic error*", and e12-EL-MW, i.e., "*Configuration error*". Moreover, a relevant number of errors pertain to interactions with OS facilities (e.g., mutex and thread), such as e8-EL-MW and e11-EL-MW.

We closely look at the data to gain insights into the most-likely error modes and their potential relationships with the fault types.

Tables 9 and 10 show the **absolute number** (Abs) and **percentage** (%) of reported errors by fault type and error mode. For example, the value 2 in the column e1-EL-MW, (*MFC, Abs*) cell of Table 9, indicates that 2 failures of the *MFC* fault type caused at least one error belonging to e1-EL-MW; this is the 2.90% –(*MFC,* %) cell– of the total 69 failure data instances where the *MFC* type led to a detection by EL (*MFC, EL-MW* cell of Table 3).

Tables 9 and 10 also show the data aggregated by ODC class ("total" rows highlighted by the grey color). Figure 7 shows the percentages of the ODC classes in Tables 9 and 10

Table 8 Error model of event logging - middleware (EL-MW)

ID	error mode	example of error event
e1-EL-MW	*Memory error*	`Failed to allocate cache.`
e2-EL-MW	*Quality of Service error*	`Writer not created inconsistent qos.`
e3-EL-MW	*Unexpected result error*	`Operation returned 0x0 but expected 0x428fa30.`
e4-EL-MW	*Data type error*	`Operation failed, couldn't resolve type "kernelModule v_builtin".`
e5-EL-MW	*Main daemon error*	`Could not claim the DDSdaemon!`
e6-EL-MW	*Consistency error*	`Illegal contained object (Receiver <7199>).`
e7-EL-MW	*Topic error*	`Failed to produce built-in ParticipantInfo topic.`
e8-EL-MW	*Mutex error*	`Operation failed mutex 0xc8025af8, result = Invalid argument.`
e9-EL-MW	*Kernel entities error*	`Create kernel entity failed.`
e10-EL-MW	*Timeout/liveliness error*	`A fatal error was detected when trying to register the DDSdaemon liveliness hbCheck lease to the liveliness lease manager of the kernel. The result code was 5.`
e11-EL-MW	*Threads progress error*	`Thread main failed to make progress.`
e12-EL-MW	*Configuration error*	`Could not initialise configuration.`
e13-EL-MW	*Other error*	`Maximum number of network queues (42) exceeded, queue not created. Expression=0xbe1a680 is not a valid select statement.`

Table 9 EL-MW: absolute number (Abs) and percentage of reported errors (%) by fault and error mode - from *e1-EL-MW* to *e7-EL-MW*

	error mode													
fault	e1-EL-MW		e2-EL-MW		e3-EL-MW		e4-EL-MW		e5-EL-MW		e6-EL-MW		e7-EL-MW	
	Abs	%	Ab	%	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
MIEB	16	61.54	1	3.85	3	11.54	2	7.69	3	11.54	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
MLPA	28	10.69	18	6.87	30	11.45	83	31.68	51	19.47	9	3.44	7	2.67
total ALG	47	12.74	21	5.69	36	9.76	100	27.10	82	22.22	10	2.71	10	2.71
MVAE	22	12.57	2	1.14	32	18.29	65	37.14	35	20.00	8	4.57	6	3.43
MVAV	1	12.50	5	62.50	3	37.50	2	25.00	0	0.00	3	37.50	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
WVAV	1	8.33	6	50.00	4	33.33	1	8.33	1	8.33	3	25.00	0	0.00
total ASG	24	12.31	13	6.67	39	20.00	68	34.87	36	18.46	14	7.18	6	3.08
MIA	0	0.00	7	46.67	5	33.33	1	6.67	1	6.67	1	6.67	1	6.67
MLC	0	0.00	0	0.00	1	50.00	0	0.00	0	0.00	0	0.00	0	0.00
total CHK	0	0.00	7	41.18	6	35.29	1	5.88	1	5.88	1	5.88	1	5.88
WAEP	0	0.00	0	0.00	0	0.00	9	81.82	2	18.18	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
total INT	2	1.50	12	9.02	41	30.83	55	41.35	14	10.53	12	9.02	15	11.28

by error mode. For example, the (*ALG*, e1-EL-MW) bar in Fig. 7 corresponds to 12.74% of the cell (total *ALG*,*e1-EL-MW %*) in Table 9. It can be noted that the distribution of the error modes is similar across the ODC classes. On average, e3-EL-MW, e4-EL-MW and e5-EL-MW are the most likely modes regardless the fault. For example, the mode e4-EL-MW –denoting the "data type" error– is observed in 27.10%, 34.87% and 41.35% of the instances where the failure is caused by *ALG*, *ASG*, or *INT* faults, respectively.

5.2 Assertion Checking - Middleware Dataset (AC-MW)

A similar analysis is done for the AC-MW dataset, i.e., failures reported by assertions in the middleware system. Table 11 shows the error model. It can be noted that, differently from event logging, errors detected by **assertion checking** pertain to foundational correctness properties –e.g., data type/size, not NULL variables– rather than the overall business logic.

Such as for event logging, we show the **absolute number** (Abs) and **percentage** (%) of reported errors by fault type and error mode for the AC-MW dataset in Table 12. Figure 8 plots the percentages of each mode cumulated by ODC type. Again, we observe the predominance of certain error modes. In this case, the most likely modes are e2-AC-MW and e4-AC-MW for all the ODC types. The top frequent mode -i.e., e2-AC-MW denoting "Unexpected value" errors- occurs in 52.67%, 45.34%, 58.62% and 52.47% of the failures caused by *ALG*, *ASG*, *CHK* and *INT* faults.

Table 10 EL-MW: absolute number (Abs) and percentage of reported errors (%) by fault and error mode - from *e8-EL-MW* to *e13-EL-MW*

fault	error mode											
	e8-EL-MW		e9-EL-MW		e10-EL-MW		e11-EL-MW		e12-EL-MW		e13-EL-MW	
	Abs	%	Abs	%	Abs	%	Abs	%	Abs	%	Abs	%
MFC	13	18.84	26	37.68	0	0.00	11	15.94	1	1.45	0	0.00
MIEB	0	0.00	0	0.00	2	7.69	0	0.00	0	0.00	1	3.85
MIFS	0	0.00	1	8.33	1	8.33	0	0.00	1	8.33	2	16.67
MLPA	9	3.44	51	19.47	7	2.67	28	10.69	4	1.53	16	6.11
total ALG	22	5,69	78	21.14	10	2.71	39	10.57	6	1.63	19	5.15
MVAE	0	0.00	21	12.00	7	4.00	15	8.57	3	1.71	10	5.71
MVAV	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
WVAV	0	0.00	1	8.33	0	0.00	2	16.67	0	0.00	1	8.33
total ASG	0	0.00	22	11.28	7	3.59	17	8.72	3	1.54	11	5.64
MIA	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	0	0.00	0	0.00	0	0.00	2	100.00
total CHK	1	5.88	1	5.88	0	0.00	2	11.76	0	0.00	2	11.76
WAEP	0	0.00	2	18.18	0	0.00	1	9.09	0	0.00	0	0.00
WPFV	1	0.82	22	18.03	3	2.46	0	0.00	1	0.82	26	21.31
total INT	1	0.75	24	18.05	3	2.26	1	0.75	1	0.75	26	19.55

5.3 Event Logging - Arrival Manager Dataset (EL-AM)

We discuss the **error model** obtained by analyzing EL-AM, i.e., the dataset of failures reported by the event logging mechanism of the arrival manager system. Table 13 shows the error model and an *error event* for each mode. Similar to EL-MW in Section 5.1, some

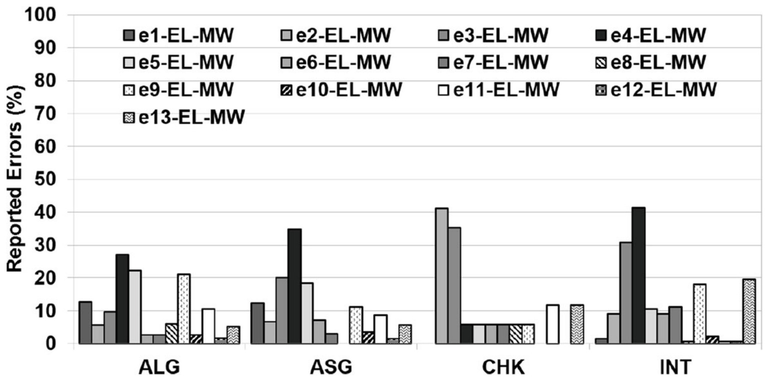


Fig. 7 EL-MW: percentage of reported errors by mode and ODC fault type

Table 11 Error model of assertion checking - middleware (AC-MW)

ID	error mode	example of error event
e1-AC-MW	*Data type error*	`'(w == c_checkType(w,"v_writer"))' failed.`
e2-AC-MW	*Unexpected value error*	`'c_refCount(found) == 4' failed.`
e3-AC-MW	*Forced assertion execution*	`'(0)' failed.`
e4-AC-MW	`NULL` *value error*	`'message != NULL' failed.`
e5-AC-MW	*Data size error*	`'c_aSize(msgKList) == c_aSize(instKList)' failed.`

errors pertain to the high-level business logic of the application, such as e2-EL-AM, i.e., "*Data format error*", and e3-EL-AM, i.e., "*Query error*".

Table 14 shows the **absolute number** (Abs) and **percentage** (%) of reported errors by fault type and error mode. For example, such as described for the other datasets, the value 1 in the column e2-EL-AM, (*MFC, Abs*) cell, indicates that 1 failures of the *MFC* fault type caused at least one error belonging to e2-EL-AM; this is the 11.11% –(*MFC, %*) cell– of the total 9 failure data instances where the *MFC* type led to a detection by EL (*MFC, EL-AM* cell of Table 3).

Percentages of ODC class by error mode –highlighted in Table 14– are plotted in Fig. 9. Such as for the previous datasets, we observe that two error modes are predominant, i.e., e2-EL-AM and e3-EL-AM; noteworthy, e2-EL-AM pertains to data-related errors.

Table 12 AC-MW: absolute number (Abs) and percentage of reported errors (%) by fault and error mode

	error mode									
fault	e1-AC-MW		e2-AC-MW		e3-AC-MW		e4-AC-MW		e5-AC-MW	
	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

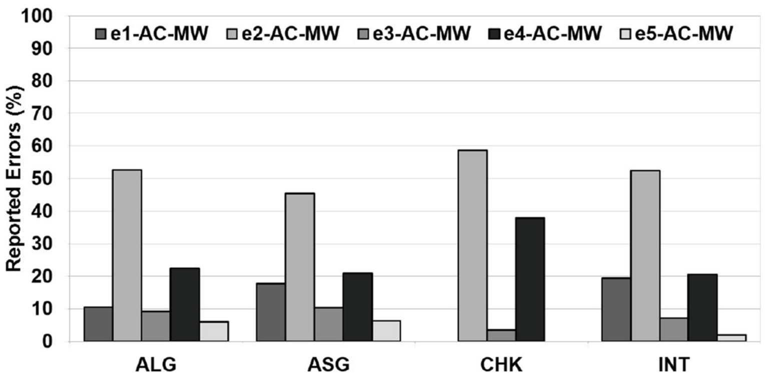


Fig. 8 AC-MW: percentage of reported errors by mode and ODC fault type

5.4 Final Remarks on the Error Models

With respect to the error modes adopted in this study, it can be reasonably stated that different fault types concentrate in a small subset of error modes. Interestingly, these modes concern *type* and *value* of variables expected during execution. This finding is consistent with the literature that highlights the importance of variables and data-error analysis in engineering dependable software (Leeke and Jhumka 2010; Hiller et al. 2002b; Jhumka et al. 2001; Jhumka and Leeke 2011; Hiller et al. 2004; Johansson and Suri 2005; Pattabiraman et al. 2011). Noteworthy, this finding is obtained here on the top of data from logs naturally emitted by the target systems, rather than substantial instrumentation approaches, which makes our approach potentially applicable to a wider class of systems.

6 Propagation Analysis

In this section we discuss error reporting graphs and computation of the metrics by means of **case studies** encompassing different detection techniques (EL and AC) and different systems (MW and AM).

We start with the analysis of EL on MW (Case Study 1 in Section 6.1), which addresses the EL-MW dataset by building the reporting graph, computing the metrics, and inferring *the paths leading to failures*, that are then useful to get insight about the errors that should be handled to avoid failures.

Table 13 Error model of event logging - arrival manager (EL-AM)

ID	error mode	example of error event
e1-EL-AM	*Memory error*	`Cannot remove dev shm SPVsharedmemory.`
e2-EL-AM	*Data error*	`Invalid ETO in points.`
e3-EL-AM	*Query error*	`SQL command not properly ended.`
e4-EL-AM	*Other error*	`Could not create pthread.`
		`Undefined Status INITIAL STATUS Disconnected.`

Table 14 EL-AM: absolute number (Abs) and percentage of reported errors (%) by fault and error mode

fault	error mode							
	e1-EL-AM		e2-EL-AM		e3-EL-AM		e4-EL-AM	
	Abs	%	Abs	%	Abs	%	Abs	%
MFC	0	0.00	1	11.11	7	77.78	1	11.11
MIEB	0	0.00	0	0.00	0	0.00	0	0.00
MIFS	0	0.00	0	0.00	0	0.00	0	0.00
MLPA	2	3.51	14	24.56	38	66.67	4	7.02
total ALG	2	3.03	15	22.73	45	68.18	5	7.58
MVAE	0	0.00	1	100.00	0	0.00	0	0.00
MVAV	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
WVAV	0	0.00	0	0.00	0	0.00	0	0.00
total ASG	0	0.00	1	100.00	0	0.00	0	0.00
MIA	0	0.00	0	0.00	1	100.00	0	0.00
MLC	0	0.00	0	0.00	0	0.00	0	0.00
total CHK	0	0.00	0	0.00	1	100.00	0	0.00
WAEP	0	0.00	0	0.00	0	0.00	0	0.00
WPFV	0	0.00	0	0.00	0	0.00	0	0.00
total INT	0	0.00	0	0.00	0	0.00	0	0.00

We try to generalize the findings obtained by replicating the analysis on the same system but with a different detection technique (i.e, AC-MW dataset, Case Study 2 in Section 6.2) and on a different system with the same technique (i.e., EL-AM dataset, Case Study 3 in Section 6.3). Finally, Section 6.4, shows how the insights inferred from the graphs can bee used **to improve** the detection mechanism; to this aim we use the EL-MW dataset and borrow additional considerations for the other cases as well.

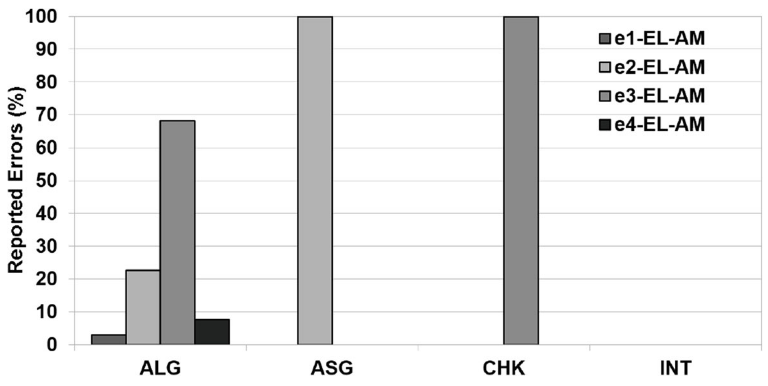


Fig. 9 EL-AM: percentage of reported errors by mode and ODC fault type

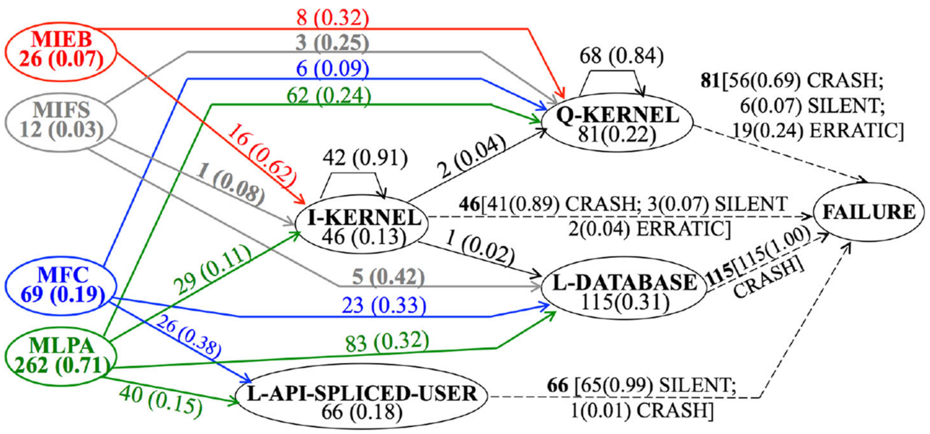


Fig. 10 Error reporting graph for EL on MW - ALG faults

In the following we focus on the graphs obtained for the most recurring faults in our data, i.e., ALG and ASG faults for MW and ALG faults for AM (as highlighted by Table 3), and provide summary results for the other ODC classes.

6.1 Case Study 1: Analysis of EL-MW

The major error propagation paths inferred from event logging for failures caused by *ALG* and *ASG* faults in the MW are shown in Figs. 10 and 11, respectively. By *major* we mean paths involving I/Q/L nodes with at least a multiplicity of 10.

It can be noted that **many errors are not reported by the immediate and quick components** (i.e., *kernel* in our data). For example, from Fig. 10 we can notice that (i) 6 (i.e., MFC→Q-KERNEL) out of 69 *MFC* faults, (ii) 4 (i.e., MIFS→I-KERNEL *plus* MIFS→Q-KERNEL) out of 12 *MIFS* faults, and (iii) 91 (i.e., MLPA→I-KERNEL *plus* MLPA→Q-KERNEL) out of 262 *MLPA* faults led to error events by the *kernel* component (either immediate or quick); *error propagation probabilities* (EPPs) of immediate and quick nodes are 0.13 and 0.22, respectively. Similarly considerations apply to Fig. 11, where the immediate and quick components exhibited an EPP of 0.18 and 0.27, respectively. This

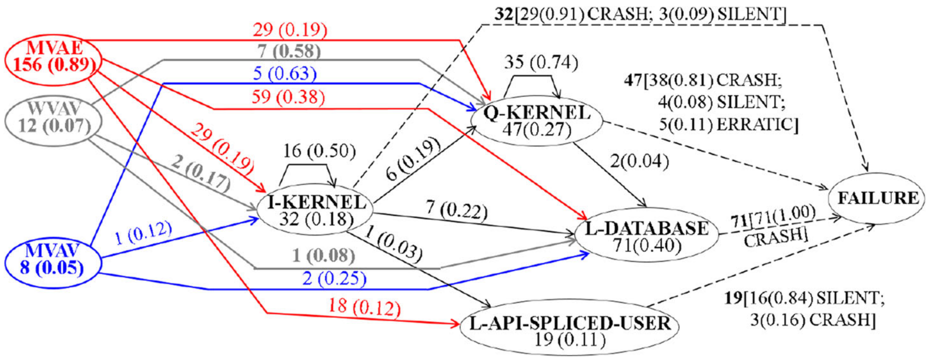


Fig. 11 Error reporting graph for EL on MW - ASG faults

Table 15 Error Propagation Reportability (EPR) of EL on MW with respect to the ODC class

ODC class	Reported errors	Errors reported by the faulty comp.	EPR (%)
Event Logging (EL)			
ALG	369	125	33.87
ASG	195	82	42.05
CHK	17	7	41.18
INT	133	70	52.63

reflects in a low *error propagation reportability* (EPR), which is 33.87% and 42.05% for *ALG* and *ASG*, respectively, as shown in Table 15, where results for all ODC classes are summarized.

Reporting graphs make it possible to infer that the **latest error propagation steps determine the type of failure encountered by the system**, which allows to provide indications on the errors that should be handled in the system to avoid their propagation into failures. For example, Figs. 10 and 11 show that in 65 (i.e., L-API-SPLICED-USER→ FAILURE in Fig. 10) out of 66 data instances (EPP of 0.99) and in 16 (i.e., L-API-SPLICED-USER→ FAILURE in Fig. 11) out of 19 data instances (EPP of 0.84), respectively, where errors propagated to the *api*, *spliced* and *user* components, a *SILENT* failure occurred in the system; similarly, in 115 (i.e., L-DATABASE→ FAILURE in Fig. 10) out 115 cases (EPP of 1.00) and in 71 (i.e., L-DATABASE→ FAILURE in Fig. 11) out 71 cases (EPP of 1.00), for Figs. 10 and 11 respectively, where an error reached the *database*, a *CRASH* occurred. We also noted that *ERRATIC* failures occurred mainly when errors propagated to the non-faulty sub-components of the *kernel*, i.e., the *Q-KERNEL* node of the graphs.

From these indications, we learn that *api*, *spliced* or *user* components are good candidates to handle errors to mitigate *SILENT* failures; similarly, *database* might help to face *CRASH* failures. More in detail, the analysis of the error events generated by EL in the *api*, *spliced* and *user* components pointed out that most of reported errors belong to **e5-EL-MW** (*main daemon error* in Table 8); therefore, to mitigate *SILENT* failures, these components should check the availability of the main daemon of the middleware and, when needed, attempt a reboot of the daemon. On the other hand, a closer look into the error events generated in the *database* component highlighted that reported errors belong to **e4-EL-MW** (*data type error* in Table 8); therefore, requesting again the data or trying to continue the execution with default values, can be useful to avoid *CRASH* failures or go towards a graceful stop.

6.2 Case Study 2: Analysis of AC-MW

In this section we repeat the analysis on the same system as in previous section, but focusing on a different detection technique, namely assertion checking (AC).

Figures 12 and 13 show the major error propagation paths inferred by assertion checking for failures caused by *ALG* and *ASG* faults, respectively, in the MW system. From the graph in Fig. 12 we can observe that: (i) 335 (i.e., MLPA→I-KERNEL *plus* MLPA→Q-KERNEL) out of 684 *MLPA* faults and (ii) 46 (i.e., MFC→I-KERNEL *plus* MFC→Q-KERNEL) out of 160 *MFC* faults led to an error event by the *kernel* component, which reflects into EPP values of 0.13 and 0.34 for immediate and quick nodes, respectively.

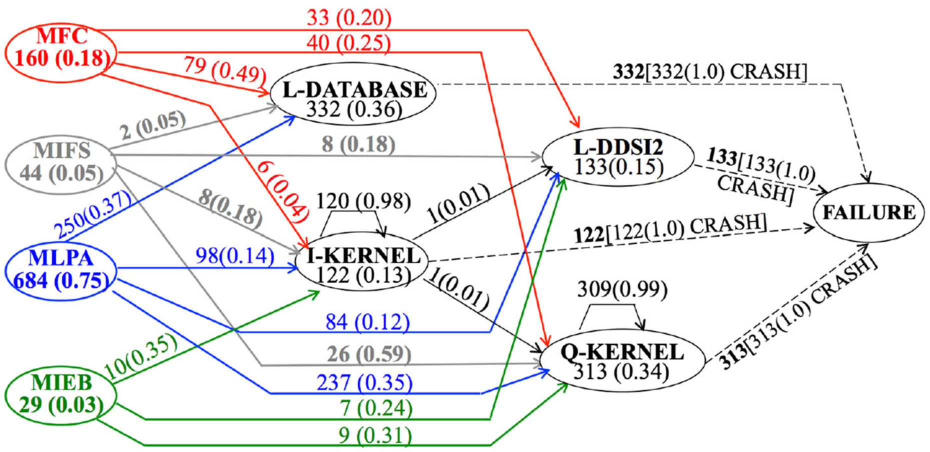


Fig. 12 Error reporting graph for AC on MW - ALG faults

On the other hand, Fig. 13 shows that 229 (i.e., MVAE→I-KERNEL *plus* MVAE→Q-KERNEL) out of 364 of MVAE faults –the most recurrent fault type for the *ASG* class– caused failures that are detected by the *kernel* component. This translates into EPP values of 0.22 and 0.41 for the immediate and quick nodes, respectively.

Similarly for EL, many errors are not reported by immediate and quick components. In other terms, early error propagation steps are mostly silent, **regardless of the detection technique**.

Table 16 summarizes the results, in terms of EPR, for all ODC classes. The maximum EPR is obtained for *ASG* faults, i.e., 62.97%, which means that **more than half of the errors have been reported by assertions located in the *kernel* component**.

Overall –by comparing Tables 15 and 16– it can be stated that in the MW system assertion checking provides better detection with respect to event logging; however, the EPRs in both tables still highlight that none of the techniques has a strong ability at reporting error propagation. In general, *ASG* and *INT* faults, which underlie variables-related problems, have better chances to be detected with respect algorithmic faults (*ALG*).

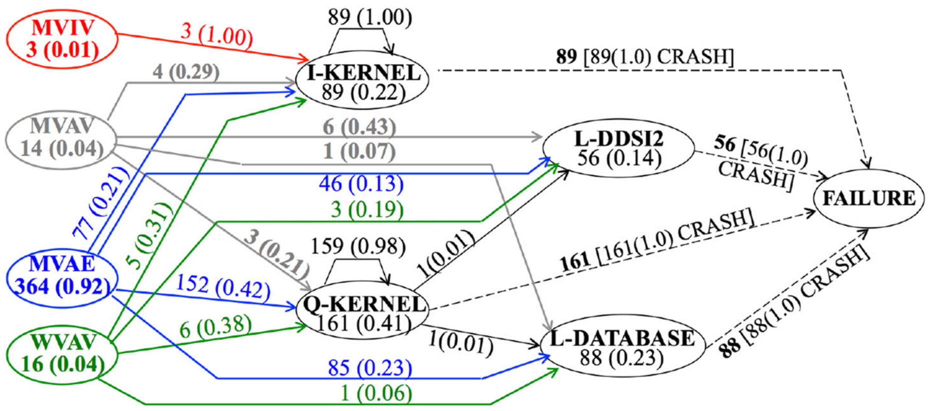


Fig. 13 Error reporting graph for AC on MW - ASG faults

Table 16 Error Propagation Reportability (EPR) of AC on MW with respect to the ODC class

ODC class	Reported errors	Errors reported by the faulty comp.	EPR (%)
Assertion Checking (AC)			
ALG	917	434	47.32
ASG	397	250	62.97
CHK	29	16	55.17
INT	263	119	45.25
total	1,606	819	51.00

Reporting graphs achieved with AC can be useful to infer information on the paths leading to failures, as done with EL. In particular, both Figs. 12 and 13 show that in almost all the cases where errors have propagated to either the *database* or *ddsi2* component a *CRASH* occurred in the system. The analysis of the error events generated by assertion checking in both the components allowed understanding that most of the reported errors belong to **e2-AC-MW** (*unexpected value error*) and **e4-AC-MW** (`NULL` *value error*) –according to the error model in Table 11– for *database* and *ddsi2*, respectively; again, it could be useful attempting to avoid further propagation of value errors, either unexpected or `NULL`.

6.3 Case Study 3: Analysis of EL-AM

In this section we repeat the analysis by focusing on the same detection technique of Case Study 1, namely event logging, but on the AM system. Figure 14 depicts the reporting graph. It can be noted that **many errors are not reported by the immediate and quick components** (i.e., *database* in our data). Once again, we note that early error propagation steps are mostly silent and missed by EL.

For example, Fig. 14 shows that (i) 3 (i.e., MFC→Q-DATABASE) out of 9 *MFC* faults, and (ii) 10 (i.e., MLPA→Q-DATABASE) out of 57 *MLPA* faults led to error events by the *kernel* component (either immediate or quick); the EPP of the quick node is 0.20. Noteworthy, there is no immediate node in the graph because no errors are reported by the faulty

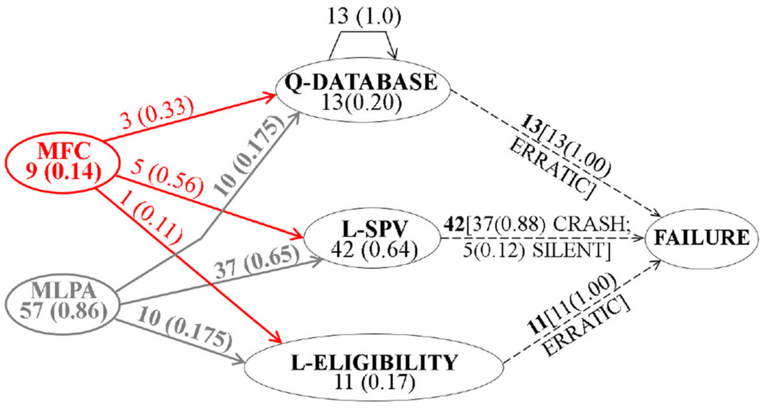


Fig. 14 Error reporting graph for EL on AM - ALG faults

Table 17 Error Propagation Reportability (EPR) of EL on AM with respect to the ALG ODC class

ODC class	Reported errors	Errors reported by the faulty comp.	EPR (%)
Event Logging (EL)			
ALG	66	13	19.70

subcomponent in AM. This reflects in the low EPR, which is 19.70% for *ALG*, as reported in Table 17.

From the graph we can note that in 37 (i.e., L-SPV→FAILURE) out of 42 data instances (EPP of 0.88) where errors propagated to the *spv* component, a *CRASH* failure occurred in the system; similarly, in all the cases where an error reached either the *database* component (i.e., Q-DATABASE→FAILURE) or the *eligibility* component (i.e., L-ELIGIBILITY→FAILURE), an *ERRATIC* failure occurred. These indications allow identifying the components where to handle given types of errors, similarly to what observed in the previous two case studies. For example, the *spv* might handle errors to mitigate *CRASH* failures; similarly, both *database* and *eligibility* components might help to face *ERRATIC* failures.

The analysis of the error events generated by EL in the *spv* component pointed out that most of reported errors belong to **e3-EL-AM** (*query error* in Table 13); therefore, those components should cope with managing exceptions related to the execution of queries. On the other hand, a closer look into the error events generated in the *eligibility* component highlighted that reported errors belong to **e2-EL-AM** (*data format error* in Table 13): recovering from data format errors can useful to avoid *ERRATIC* failures in AM.

In summary, in all the datasets we were able to apply the proposed approach to build error reporting graphs, regardless of the detection technique and of the target system. Graphs are then a useful instrument to quantify reporting performance – in terms of the proposed EPR and EPPs metrics – to spot reporting inefficiencies, to identify errors to be be handled with the aim of avoiding failures, and to improve the reporting mechanism, as discussed in next section.

6.4 Improvement of the Detection Mechanism

As observed for all the case studies, propagation graphs reveal that **errors undetected by the immediate/quick component, might still be reported by other components along the propagation path**, which allows understating how to improve the reporting mechanism.

With respect to the first case study, Figs. 10 and 11 show that many faults lead to failures reported only by late components, such as *database*, *api*, *spliced* and *user*, without involving the *kernel*. For instance, in Fig. 10: (i) 23 (i.e., MFC→L-DATABASE) out of 69 *MFC* faults, (ii) 83 (i.e., MLPA→L-DATABASE) out of 262 *MLPA* faults, and (iii) 5 (i.e., MIFS→L-DATABASE) out of 12 *MIFS* faults (with EPP values of 0.33, 0.32 and 0.42, respectively) led to failures reported only by the *database* component; similarly, 26 (i.e., MFC→L-API-SPLICED-USER) out of 69 *MFC* faults (EPP of 0.38) and 40 (i.e., MLPA→L-API-SPLICED-USER) out of 262 *MLPA* faults (EPP of 0.15) caused failures reported only by the *api*, *spliced* and *user* components. On the other hand, Fig. 11 shows that 59 (i.e., MVAE→L-DATABASE) out of 156 *MVAE* faults (with EPP value of 0.38) led to failures reported only by the *database* component.

Following this finding, we placed additional EDMs in the *kernel* component. To this objective we use the **rule-based logging** approach (Cinque et al. 2013), which consists into placing start-end events at the begin-end of functions. Rule-based logging aims to detect errors preventing the completion of invoked functions and dirty function returns. The technique is applied to the source code of the files belonging to the kernel component, which leads to place detectors into 118 functions.

We analyze the logs generated by MW –now equipped with additional EDMs– under controlled injection experiments and obtain the graph in Fig. 15 for ALG faults. Differently from the original graph in Fig. 10, we observe that:

- most of the errors are reported by the kernel component, either immediate (EPP of 0.67) or quick (EPP of 0.50);
- all the errors reported by the L-API-SPLICED-USER node are reported also by the kernel component, i.e., 66 (53 Q-KERNEL→L-API-SPLICED-USER *plus* 13 I-KERNEL→L-API-SPLICED-USER) out 66 cases, with no arcs connecting faults with the L-API-SPLICED-USER node;
- most of the errors reported by the database component, i.e., 86 (45 Q-KERNEL→L-DATABASE *plus* 41 I-KERNEL→L-DATABASE) out of 115 cases, are now reported also by the kernel, either immediate or quick.

Improvements reflects into the EPR, which increases from 33.87% (as reported in the ALG row of Table 15) **to 94.50%; hence the new placement allows to significantly improve the reporting ability of the kernel component**. Moreover, it is important to note that **the new EDMs also allow to report previously undetected errors**. In fact, errors caused by 11 MIEB, 24 MIFS, 90 MFC and 725 MLPA faults, undetected by the original built-in EL, are now reported by the kernel component, as it can be noted by comparing the multiplicity of the fault type nodes in Figs. 10 and 15.

Similar insights are obtained for the other case studies. Starting from assertion checking, Fig. 12 reveals that 250 (i.e., MLPA→L-DATABASE) out of 684 *MLPA* faults and 79 (i.e., MFC→L-DATABASE) out of 160 *MFC* faults (with EPP of 0.37 and 0.49, respectively) led to failures reported only by the *database*; similarly, 84 (i.e., MLPA→L-DDSI2) out of 684 *MLPA* faults and 33 (i.e., MFC→L-DDSI2) out of 160 *MFC* faults (with EPP of 0.12 and 0.20, respectively) caused failures reported only by the *ddsi2* component. Moreover, Fig. 13 shows that 85 (i.e., MVAE→L-DATABASE) and 46 (i.e., MVAE→L-DDSI2) out of 364 *MVAE* faults (with EPP of 0.23 and 0.13, respectively) led to failures reported only by the

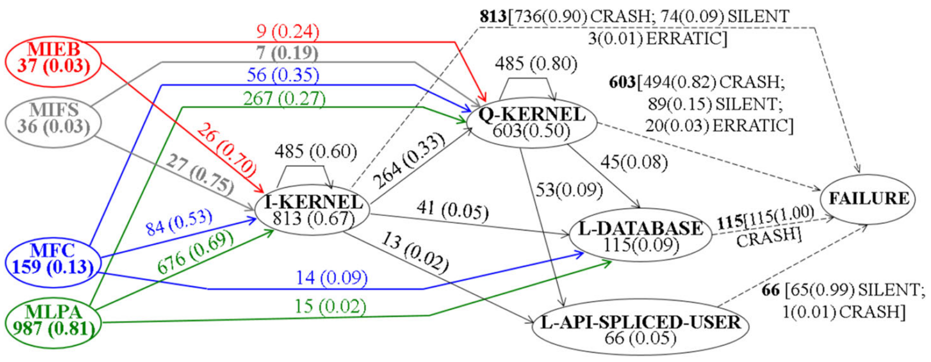


Fig. 15 Error reporting graph for EL on MW after improvement

database and *ddsi2* components, respectively. A closer look into the error events generated by assertion checking of the *database* and *ddsi2*, points out that most of the reported errors belong to the **e2-AC-MW** (*unexpected value error*) and **e4-AC-MW** (`NULL` *value error*) types, respectively. As such, further EDMs can be placed in the *kernel* subcomponents to reveal bad or `NULL` values exchanged with *database* or *ddsi2*.

Concerning the case of the AM, Fig. 14 shows that 5 (i.e., MFC→L-SPV) out of 9 *MFC* faults, and 37 (i.e., MLPA→L-SPV) out of 57 *MLPA* faults (with EPP values of 0.56 and 0.65, respectively) led to failures reported only by the *SPV* component; similarly, 10 (i.e., MLPA→L-ELIGIBILITY) out of 57 *MLPA* faults (EPP of 0.175) caused failures reported only by the *eligibility*. These data suggest the placement of EDMs inside the *database* component. Similarly, a closer look into the instances containing only errors generated by the *SPV* component, highlights that most of the reported errors belong to **e1-EL-AM** (*memory error*) and **e3-EL-AM** (*query error*), which further confirms the need for addressing the *database* in improving error detection.

7 Threats to Validity

We discuss the validity of the study based on the most relevant aspects listed in Wohlin et al. (2000).

Construct Validity The threat relates to the choice of the datasets for the evaluation. We face it by collecting realistic failure data instances from two different real-world software systems from an industrial partner. Failure data instances have been collected in our previous large-scale study on software systems monitoring (Cinque et al. 2016) by running controlled experiments, which aimed to elicit error events under different fault and failure conditions. The reference system has been exercised with testing applications provided by the industry partner to exercise the system under realistic operation scenarios. Injected faults are based on the well consolidated ODC scheme (Chillarege et al. 1992) and on fault types accounting for around 80% of representative faults found in real-world software systems, according to the estimates in Duraes and Madeira (2006).

Internal Validity might be threatened by the analysis of relationships between errors and their granularity. As for any log analysis study, creating a taxonomy/dictionary of events is a cognitive process and requires a trade-off. For example, if an error type is too generic, the resolution might be too small for subsequent analysis. To mitigate the threat, we took a balanced approach in order to avoid both overgeneralizing and excessive fragmentation, also considering error modes typically found in other studies. In addition, we adopted a mixture of diverse faults-failures and error reporting mechanisms. We used error events produced by two error reporting mechanisms, i.e., event logging and assertion checking, under different faults and failures conditions and from two different systems to show the effectiveness of the approach for error propagation analysis through error data logged by the system. Noteworthy, our approach analyzes errors raised by the activation of individual faults, because –in case of coincidental activation of multiple faults– it would be hard to discriminate which fault caused certain errors. The key findings of the study are consistent across the mechanisms and target systems, providing a reasonable level of confidence on the analysis.

External and Conclusion Validity Regarding the possibility to apply the approach on other systems, we provide a concrete examples with two unrelated systems developed by independent teams. We are confident that the details provided should reasonably support the replication and generalization of the steps composing the approach. The reported findings, which are strongly supported by data, are useful to get an overall understanding on the insights that can be obtained; however, they are not intended to establish a general approach based on two systems. Results show how the proposal can be used to understand error modes, propagation paths and capabilities of error reporting mechanisms and to infer useful insights to improve them accordingly.

8 Conclusion

This paper proposed an empirical study on error propagation. The study leverages an approach based on the concept of error reporting graphs and novel metrics, i.e., Error Propagation Probabilities and Error Propagation Reportability. The approach has been used with 2,042 failure data instances from two real-world systems from an industry partners, encompassing logs and error events generated by two error reporting mechanisms.

The use of the approach provided a deep understanding on error modes, propagation paths and capabilities of the error reporting mechanisms implemented by the systems in hand. For example, the study highlighted that a component affected by a fault is likely to report no error notifications, regardless of the error reporting mechanism. On the other hand, the obtained results pointed out that errors missed by a faulty component might still be reported along the propagation path; the analysis of those errors provided insights about the improvement of error reporting mechanisms and the placement of new EDMs. Finally, the study revealed that latest error propagation steps determine the type of failure encountered by the system, which provided useful indications for making informed decisions on the errors that should be handled to avoid system failures.

Acknowledgements This work has been partially supported by the Italian Ministry of Education, University and Research under the MINIMINDS PON Project (n. B21C12000710005) and by Programme STAR, funded by UniNA and Compagnia di San Paolo under project “Towards Cognitive Security Information and Event Management” (COSIEM).

References

Abdelmoez W, Nassar DM, Shereshevsky M, Gradetsky N, Gunnalan R, Ammar HH, Yu B, Mili A (2004) Error propagation in software architectures. In: 10th international symposium on software metrics, 2004. Proceedings., pages 384–393. https://doi.org/10.1109/METRIC.2004.1357923

Avizienis A, Laprie J-C, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1):11–33. ISSN 1545-5971. https://doi.org/10.1109/TDSC.2004.2

Arora A, Kulkarni SS (1998) Detectors and correctors: a theory of fault-tolerance components. In: Proceedings. 18th international conference on distributed computing systems (Cat. No.98CB36183), pp 436–443. https://doi.org/10.1109/ICDCS.1998.679772

Bondy JA, Murty USR et al (1976) Graph theory with applications, vol 290. Citeseer

Calhoun J, Snir M, Olson LN, Gropp WD (2017) Towards a more complete understanding of SDC propagation. In: Proceedings of the 26th international symposium on high-performance parallel and distributed computing, HPDC ’17. ACM, New York, pp 131–142. ISBN 978-1-4503-4699-3. https://doi.org/10.1145/3078597.3078617

Cinque M, Cotroneo D, Pecchia A (2013) Event logs for the analysis of software failures: a rule-based approach. IEEE Trans Softw Eng 39(6):806–821. ISSN 0098-5589. https://doi.org/10.1109/TSE.2012.67

Cinque M, Cotroneo D, Della Corte R, Pecchia A (2016) Characterizing direct monitoring techniques in software systems. IEEE Transactions on Reliability 65(4):1665–1681. ISSN 0018-9529. https://doi.org/10.1109/TR.2016.2570564

Chan A, Winter S, Saissi H, Pattabiraman K, Suri N (2017) IPA: Error propagation analysis of multi-threaded programs using likely invariants. In: IEEE international conference on software testing, verification and validation (ICST), pp 184–195. https://doi.org/10.1109/ICST.2017.24

Chillarege R, Bhandari IS, Chaar JK, Halliday MJ, Moebus DS, Ray BK, Wong M-Y (1992) Orthogonal defect classification-a concept for in-process measurements. IEEE Trans Softw Eng 18:943–956. ISSN 0098-5589. http://doi.ieeecomputersociety.org/10.1109/32.177364

Chuah E, Jhumka A, Browne JC, Barth B, Narasimhamurthy S (2015) Insights into the diagnosis of system failures from cluster message logs. In: 11th European dependable computing conference (EDCC), pp 225–232. https://doi.org/10.1109/EDCC.2015.19

Cortellessa V, Grassi V (2007) 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. ISBN 978-3-540-73551-9. https://doi.org/10.1007/978-3-540-73551-9_10

Duraes JA, Madeira HS (2006) Emulation of software faults: a field data study and a practical approach. IEEE Trans Softw Eng 32(11):849–867. ISSN 0098-5589. https://doi.org/10.1109/TSE.2006.113

Filieri A, Ghezzi C, Grassi V, Mirandola R (2010) Reliability analysis of component-based systems with multiple failure modes. In: Grunske L, Reussner R, Plasil F (eds) Component-Based Software Engineering. Springer, Berlin, pp 1–20

Hiller M, Jhumka A, Suri N (2002a) 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, pp 81–85, New York, NY, USA. ACM. ISBN 1-58113-562-9. https://doi.org/10.1145/566172.566184

Hiller M, Jhumka A, Suri N (2002b) On the placement of software mechanisms for detection of data errors. In: IEEE international conference on dependable systems and networks, pp 135–144. https://doi.org/10.1109/DSN.2002.1028894

Hiller M, Jhumka A, Suri N (2004) Epic: profiling the propagation and effect of data errors in software. IEEE Transactions on Computers 53(5):512–530. ISSN 0018-9340. https://doi.org/10.1109/TC.2004.1275294

Hsueh MC, Tsai TK, Iyer R (1997) Fault injection techniques and tools. IEEE Computer 30(4):75–82

Jhumka A, Hiller M, Suri N (2001) Assessing inter-modular error propagation in distributed software. In: 20th IEEE symposium on reliable distributed systems, 2001. Proceedings, pp 152–161. https://doi.org/10.1109/RELDIS.2001.969769

Jhumka A, Leeke M (2011) The early identification of detector locations in dependable software. In: IEEE 22nd international symposium on software reliability engineering, pp 40–49. https://doi.org/10.1109/ISSRE.2011.34

Johansson A, Suri N (2005) Error propagation profiling of operating systems. In: International conference on dependable systems and networks, 2005. DSN 2005. Proceedings. pp 86–95. https://doi.org/10.1109/DSN.2005.45

Kabinna S, Bezemer C-P, Shang W, Syer MD, Hassan AE (2018) Examining the stability of logging statements. Empirical Software Engineering 23(1):290–333. ISSN 1573-7616. https://doi.org/10.1007/s10664-017-9518-0

Kalyanakrishnam M, Kalbarczyk Z, Iyer R (1999) Failure data analysis of a LAN of windows NT based computers. In: Proceedings of the international symposium on reliable distributed systems (SRDS). IEEE Computer Society, pp 178–187

Khoshgoftaar TM, Allen EB, Tang WH, Michael CC, Voas JM (1999) Identifying modules which do not propagate errors. In: IEEE symposium on application-specific systems and software engineering and technology (ASSET), pp 185–193. https://doi.org/10.1109/ASSET.1999.756768

Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on code generation and optimization: feedback-directed and runtime optimization, CGO '04. IEEE Computer Society, Washington, pp 75–. ISBN 0-7695-2102-9. http://dl.acm.org/citation.cfm?id=977395.977673

Lau KK, Wang Z (2007) Software component models. IEEE Trans Softw Eng 33(10):709–724. ISSN 0098-5589. https://doi.org/10.1109/TSE.2007.70726

Leeke M, Jhumka A (2010) Towards understanding the importance of variables in dependable software. In: 2010 European Dependable Computing Conference (EDCC), pp 85–94. https://doi.org/10.1109/EDCC.2010.20

Li H, Chen T-H(Peter), Shang W, Hassan AE (2018) Studying software logging using topic models. Empirical Software Engineering 23(5):2655–2694. ISSN 1573-7616. https://doi.org/10.1007/s10664-018-9595-8

Lyu MR et al (1996) Handbook of software reliability engineering, vol 222. IEEE Computer Society Press, CA

Michael CC, Jones RC (1997) On the uniformity of error propagation in software. In: Proceedings of the 12th annual conference on computer assurance, 1997. COMPASS '97. Are we making progress towards computer assurance?, pp 68–76. https://doi.org/10.1109/CMPASS.1997.613237

Makanju A, Zincir-Heywood AN, Milios EE (2012) A lightweight algorithm for message type extraction in system application logs. IEEE Trans Knowledge Data Eng 24(11):1921–1936. ISSN 1041-4347. https://doi.org/10.1109/TKDE.2011.138

Pattabiraman K, Saggese GP, Chen D, Kalbarczyk Z, Iyer R (2011) Automated derivation of application-specific error detectors using dynamic analysis. IEEE Transactions on Dependable and Secure Computing 8(5):640–655. ISSN 1545-5971. https://doi.org/10.1109/TDSC.2010.19

Popic P, Desovski D, Abdelmoez W, Cukic B (2005) Error propagation in the reliability analysis of component based systems. In: 16th IEEE international symposium on software reliability engineering, 2005. ISSRE 2005, pp 10–62. https://doi.org/10.1109/ISSRE.2005.18

Rosenblum DS (1995) A practical approach to programming with assertions. IEEE Trans Softw Eng, p 21

Russo B, Succi G, Pedrycz W (2015) Mining system logs to learn error predictors: a case study of a telemetry system. Empirical Software Engineering 20(4):879–927. ISSN 1573-7616. https://doi.org/10.1007/s10664-014-9303-2

Tian J, Rudraraju S, Li Z (2004) Evaluating web software reliability based on workload and failure data extracted from server logs. IEEE Trans Soft Eng 30(11):754–769. ISSN 0098-5589. https://doi.org/10.1109/TSE.2004.87

Tucek J, Lu S, Huang C, Xanthos S, Zhou Y (2007) Triage: Diagnosing production run failures at the user's site. In: Proceedings of Twenty-first ACM SIGOPS symposium on operating systems principles, SOSP '07, pp 131–144, New York, NY, USA. ACM. ISBN 978-1-59593-591-5. https://doi.org/10.1145/1294261.1294275

Voas J (1997) Error propagation analysis for cots systems. Computing Control Engineering Journal 8(6):269–272. ISSN 0956-3385. https://doi.org/10.1049/cce:19970607

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in software engineering: an introduction. Kluwer Academic Publishers, Norwell. ISBN 0-7923-8682-5

Yuan D, Mai H, Xiong W, Tan L, Zhou Y, Pasupathy S (2010) Sherlog: Error diagnosis by connecting clues from run-time logs. SIGARCH Comput Archit News 38(1):143–154. ISSN 0163-5964. https://doi.org/10.1145/1735970.1736038

Zheng Z, Lyu MR (2010) Collaborative reliability prediction of service-oriented systems. In: 2010 ACM/IEEE 32nd international conference on software engineering, vol 1, pp 35–44. https://doi.org/10.1145/1806799.1806809

Marcello Cinque graduated with honors from University of Naples, Italy, in 2003, where he received the PhD degree in computer engineering in 2006. Currently, he is Assistant Professor at the Department of Electrical Engineering and Information Technology of the University of Naples Federico II. Cinque is chair and/or TPC member of several technical conferences and workshops on dependable systems, including IEEE DSN, EDCC, and DEPEND. His interests include dependability assessment of critical systems and log-based failure analysis.

Raffaele Della Corte received the B.S., M.S. and Ph.D. degrees in computer engineering from the Federico II University of Naples, Italy in 2009, 2012 and 2016, respectively. He is currently a Postdoctoral Researcher at the Department of Electrical Engineering and Information Technologies, Federico II University of Naples. His research interests include data-driven failure analysis, on-line monitoring of software systems and security. Dr. Della Corte serves as reviewer in several dependability conferences and he is involved in industrial projects developing techniques for the analysis and monitoring of critical software systems.

Antonio Pecchia received the B.S. (2005), M.S. (2008) and Ph.D. (2011) in Computer Engineering from the Federico II University of Naples, where he is now an Assistant Professor. He is a co-founder of the Critiware spin-off company (www.critiware.com). He serves as TPC member and reviewer in conferences and workshops on software engineering and dependability, such as ISSRE, EDCC and LADC. His research interests include data analytics, log analysis, empirical software engineering, dependable and secure distributed systems. He is the Editor in Chief of the International Journal of Open Source Software and Processes and Associate Editor of IEEE Access. He is a member of the IEEE.