The Journal of Systems and Software 113 (2016) 27–43

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation

Domenico Cotroneo [a], Roberto Pietrantuono [a,*], Stefano Russo [a], Kishor Trivedi [b]

[a] Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione, Università di Napoli Federico II Via Claudio 21, 80125 Naples, Italy
[b] Duke High Availability Assurance Lab, ECE Department, Duke University, 27708 Durham, NC, USA

ARTICLE INFO

Article history:
Received 29 October 2014
Revised 9 October 2015
Accepted 11 November 2015
Available online 2 December 2015

Keywords:
Bug manifestation
Failure
Bug characterization

ABSTRACT

The impact of software bugs on today's system failures is of primary concern. Many bugs are detected and removed during testing, while others do not show up easily at development time and manifest themselves only as operational failures. Besides the importance of understanding the bug features from the programmer perspective (i.e., what is wrong in the code), a key role in counteracting bugs is played by the chain that from the bug activation leads to failure.

This article investigates the characteristics of the bug manifestation process. Through an extensive empirical study, a set of failure-exposing conditions is first identified as bug manifestation characteristics; 666 bug reports from two applications are then analyzed with respect to these characteristics under several perspectives. Findings highlight: (i) the main occurrence patterns of bug triggering conditions in the selected case studies and the role played by the workload, the application and the environment where it runs; (ii) how such conditions evolve over time; (iii) how they relate to bug exposure and fixing difficulty; (iv) how they impact the user. Results provide a fine-grain characterization of bug manifestation that is expected to increase the perceived importance of this dimension in testing, debugging, and fault tolerance strategies.

1. Introduction

Understanding software bugs is of paramount importance to improve software quality and development processes. Researchers, across years, analyzed bugs from different viewpoints to improve the knowledge about their characteristics. Regardless of the semantics of the error committed by a developer, a fundamental aspect in bug comprehension is related to the process by which a bug manifests itself as a failure. Indeed, while static properties of a bug (e.g., its type, or origin) are related to how a bug is introduced in the code, there are different causes for a bug provoking a failure. Expectedly, many bugs systematically cause the same failure on a given (sequence of) input(s). Conversely, there is a non-negligible set of bugs that cause a failure depending on the state of the execution environment, appearing as non-deterministic or transient, in which the failure does not occur unless the environment is in a certain state (Gray, 1985; Grottke and Trivedi, 2007). The latter category contains bugs that likely escaped testing, since their exposure may be a rare event, and different V&V techniques (e.g., static analysis) or runtime fault tolerance are the means to cope with them. In general, the effectiveness of fault detection and tolerance strategies are strictly tied to how bugs manifest themselves.

In the past, some broad classifications took into account the properties of bugs related to the reproducibility of the failures they cause. Gray (1985) distinguished Bohrbugs and Heisenbugs depending on whether the failure caused by the bug is systematically reproducible or not (also called *hard* and *soft* failures, respectively). Later, Trivedi and Grottke define Mandelbug in lieu of Heisenbug (Grottke and Trivedi, 2007), considering the complexity of the bug-failure process in terms of macro-conditions required for a bug to cause a failure (i.e., influence of the execution environment, timing or ordering of inputs or operations, time lag between bug activation and failure occurrence). Several researchers conducted empirical studies and indirectly highlighted the importance of distinguishing the environment as collateral cause of bug exposure (Chandra and Chen, 2000; Lee and Iyer, 1995; Grottke et al., 2010). Others have been highlighting some factors of the execution environment, such as memory (Sullivan and Chillarege, 1991), concurrency (Lu et al., 2008), or resource management (Cotroneo et al., 2013b), as relevant failure causes, and motivated subsequent research on developing proper countermeasures (e.g., static and dynamic analysis tools). A few papers clearly distinguish the need for studying some characteristic of the bug manifesta-

* Corresponding author. Tel.: +39 081676770.
E-mail addresses: cotroneo@unina.it (D. Cotroneo), roberto.pietrantuono@unina.it (R. Pietrantuono), stefano.russo@unina.it (S. Russo), ktrivedi@duke.edu (K. Trivedi).

http://dx.doi.org/10.1016/j.jss.2015.11.021
0164-1212/

Table 1
Summary of findings by type of analysis.

Triggering conditions analysis	
#1	Most of the reported bugs (80.71%) needs only workload conditions to surface, with no environmental condition required
Workload (WL) triggering conditions	
#2	For 57.87% of bug reports, the bug manifestation requires a specific *request type* For 35.04%, a further additional condition is required
#3	For 35.22% of bug reports, the bug manifestation requires a *requests sequence* For 24.78%, a further additional condition is required
#4	For 50.62% of bug reports, the bug manifestation requires a second WL condition, related to the *input type*, to the *application configuration*, or to the *input value*
#5	For 11.68% of bug reports, the bug manifestation requires a specific *input value* (e.g., boundary values); for 4.78%, it requires a specific *range of values*; for 3.54%, it requires a *class of values* with a property in common
#6	For 10.79% of bug reports, the bug manifestation requires a specific *request type* together with a specific *application configuration*; for 9.02%, it requires a specific *request type* together with a specific *input value*; for 7.08%, it requires a specific *request type* together with a specific *input type*
Environment triggering conditions	
#7	63.33% of bug reports with execution environment triggers are caused by "indirect" environmental conditions
#8	For 7.25% of bug reports, the bug manifestation requires a *concurrency* condition
#9	For 6.19% of bug reports, the bug manifestation is related to *memory management*
Workload and environment triggering conditions	
#10	For 76.15% of *environment-dependent* bug reports, the bug manifestation requires, besides the environmental condition, a sequence of request as workload condition
#11	For 7.26% of bug reports, the bug manifestation requires a *transient* environmental conditions
#12	For 21.24% of bug reports, the bug manifestation requires a specific *environment configuration*
#13	When a specific *environment configuration* is required, the necessary factor is: a specific OS (55.83%); specific system-level or application-level software (24.17 %), a specific hardware/network configuration (20.00%)
Temporal analysis	
#14	*Environment-dependent* bug reports tend to start appearing later and have a slower increasing rate than *workload-dependent* ones
#15	The ratio of environment-dependent over workload-dependent bugs in MySQL is lower in the first 4 years than in the period 4–8 years after the release time
Complexity analysis	
#16	For 45.66% of bug reports, the bug manifestation requires at least two conditions to surface, more often two *workload* conditions (37.87%)
#17	Bugs requiring one triggering condition (25.49%) are less common than bugs requiring two conditions (45.66%)
#18	For 7.61% of bug reports, the bug manifestation requires 4 conditions together to surface; for 1.06%, it requires 5 conditions
#19	We cannot state that there is a relation between the bug manifestation, expressed by triggers, and the time to fix a bug
Impact analysis	
#20	The manifestation of most of total bugs ended up in an incorrect response provided to the user (62.12%), or in a crash of the application (26.90%). The remaining bugs resulted in performance issues (5.84%) or omission failures (3.72%)
#21	The *failure mode* is affected (p-*Value* < 0.01) by the type of bug, being *environment-* or *workload-dependent*
#22	The relative proportions of high and low severity bugs are influenced by the bug type, with the percentage of environment-dependent bugs being more relevant for the high severity class (p-*Value* < 0.02).

tion, such as the number of inputs required for a bug to surface, case framed within a wider context (e.g., concurrency bugs Lu et al., 2008, or server bugs Sahoo et al., 2010). Despite the merit of these studies in pointing out the need for examining the entire bug-failure chain, none of them sets the goal of systematically investigating the characteristics of the bug manifestation process. In this work, we present the results of a comprehensive empirical study whose aim is to examine the fine-grain conditions that make a bug cause a failure. A set of bugs are analyzed in terms of "*triggers*" – that is, of necessary failure-exposing conditions that make a bug surface. These are related both to the input workload and to the environment necessary for a bug exposure. By abstracting factors that potentially affect the bug activation and/or propagation process, a set of conditions are identified as triggers, and used to categorize the bug manifestation characteristics. On a set of 666 bug reports taken from the Apache Web Server and the MySQL DBMS, we first analyzed the occurred *patterns of bug triggers*, to figure out the most common factors exposing bugs in the considered case studies. Then, we have investigated the relation of such triggers with complexity in terms of bug exposure difficulty and fixing time, to see if they are related only to the detection or also to the bug fixing process. Their *evolution over time* is also investigated, so as to detect possible differences in the way such triggers appear over time (e.g., if triggers related to the environment differ from triggers related only to the workload in terms of occurrence time). Finally, the relation of triggers with the *impact* of the failure they cause is studied, so as to assess if the exposure characteristics of a bug are also related to the end-user perception of the caused failure. The study provides a set of findings, referred to the selected case studies, summarized in Table 1, which highlight: (i) the impact of each workload condition on bug surfacing, both when it is a *necessary* condition and when it is a *necessary and sufficient* condition; (ii) the additional impact caused by the environment conditions, and which factor of the environment is more relevant; (iii) the impact of the combination of a number of conditions together; (iv) the occurrence pattern of different triggering conditions at operational time; and (v) the relation of workload and environment bug triggers with the failure modes and the perceived severity. Far from claiming the generality of what we observed, the study serves to point out the many differences among bugs in terms of manifestation characteristics. We believe this can foster further investigation along such an important dimension, contributing to figure out how to exploit the knowledge of bug-failure chain for improving development processes. In the following, we first survey past studies on bug characteristics in the literature (Section 2); in Section 3, we present the study methodology; Sections 4, 5, 6, and 7 discuss, respectively, the results of the *bug trigger analysis, temporal analysis, complexity analysis*, and *impact analysis*; Section 8

discusses the limitation of the empirical study; Section 9 concludes the paper.

2. Studies on bug characterization

Researchers have spent much effort to study characteristics of software bugs. On the one hand, they proposed classifications meant to define general characteristics of defects (i.e., the IEEE Std. 1044 scheme I. S., 2010, the Hewlett-Packard (HP) scheme Grady, 1992, and the Orthogonal Defect Classification (ODC) Chillarege et al., 1992). On the other hand, they conducted several empirical studies by analyzing problem reports, and outlined bugs' characteristics from observed data. Along the former trend, the ODC is the closest one to our work. It classifies defects by several attributes, the most important ones being: the *defect type*, which captures the semantic of the fix made by the programmer, and the *defect trigger*, which relates the defect to the V&V activities that made it surface. From the bug manifestation perspective, ODC triggers are interesting in that they are directly related to bug exposure activities. Several empirical studies analyzed ODC triggers in case studies for V&V improvement (Chillarege and Bassin, 1995; Chillarege and Prasad, 2002; Cotroneo et al., 2013d). By contrast, our paper examines the conditions necessary for the bug to surface with a different meaning: while ODC refers to high-level V&V activities (e.g., test sequencing, inspection, black box test), that, in the considered circumstance, made the defect surface, we look for the fine-grained inputs and environmental conditions that are necessarily required for the manifestation, independent of any specific V&V activity.

Regarding empirical studies, there are several papers analyzing the characteristics of bugs, errors, and failures. These refer either to specific classes of systems or to classes of problems (e.g., concurrency). Many of them classify problems by some features that can be related to the bug manifestation dimension, different from study to study, and adopting different terminologies. Sullivan and Chillarege (1991) studied the causes and triggers of software defects in operating systems, distinguishing "regular" from "overlay" (i.e., memory-related) defects. Lee and Iyer (1993), 1995) conducted a field failures analysis of Tandem GUARDIAN operating system, grouping failure causes into computation and data error, missing operations, code update, microcode problems, and unexpected situations (e.g., race/timing problems, unexpected error). The work by Gray (1985) provided a more general distinction between (i) "solid" or "hard" faults, for which failures are easily reproducible (named *Bohrbugs*, alluding to Bohr's simple atom model); and (ii) "elusive" or "soft" faults, for which failure occurrence is not systematically reproducible (named *Heisenbugs*, referring to Heisenberg's uncertainty principle). Later, Grottke and Trivedi (2005), 2007) introduced *Mandelbugs* as the complementary antonym of Bohrbug, considered as those bugs whose activation/propagation appears as non-deterministic or chaotic because of environmental conditions or of time lags between the bug activation and the failure manifestation. Although the coarse distinction Bohr- *vs.* Mandel/Heisenbug is insufficient to account for the many factors involved in the bug-failure chain, its practical importance is to explicitly highlight the relevant differences that may exist in the bug manifestation. Several field studies have found that a significant fraction of problems is due to defects whose features are attributable to Mandelbugs: besides the above-mentioned ones (i.e., Sullivan and Chillarege, 1991; Lee and Iyer, 1993, 1995), several other studies found that *Mandelbug-like* faults account for the 20–40% of the total defects (e.g., Chillarege, 2011; Trivedi et al., 2011; Cotroneo et al., 2013a; Cavezza et al., 2014), even in critical systems (Grottke et al., 2010). In Cotroneo et al. (2013b), Bovenzi et al. (2012) and Cotroneo et al. (2010) we focused on a specific class of Mandelbugs, namely the aging-related bug, distinguishing memory-related problems (e.g., leaks), storage-related (e.g., fragmentation, leaks), wrong management of other resources (e.g., handles, locks), and numerical errors, observing an impact of about 5%. Authors in Lu et al. (2008) focus on concurrency bugs; they analyze 105 faults from 4 complex systems distinguishing among atomicity violations, deadlocks, and order-violations. This is a study explicitly analyzing bug manifestation, limited to concurrency bugs, in terms of number of threads, accesses, and variables involved in the manifestation. Recent papers (Tan et al., 2013; Li et al., 2006) focus on open source software, and propose a taxonomy distinguishing the root cause (memory, concurrency, semantic), the impact (i.e., the failure), and the component affected. Authors in Sahoo et al. (2010) analyzed bug reports in server applications, and considered the reproducibility characteristic distinguishing bugs as deterministic, timing dependent, or non-deterministic, studying the number of inputs to trigger a failure. Chandra and Chen (2000) studied the reports of Apache, GNOME, and MySQL to investigate recovery strategies, and distinguished *environment-dependent (transient or not)* from *environment-independent* faults.

Several other papers perform a classification on bug reports to analyze characteristics not related to the bug manifestation, but that we also consider in our work, e.g., the bug fixing time and severity. Specifically, several studies analyzed the fixing process of defects. Zhang et al. (2012) studied the bug fixing time and factors influencing it on three open source software applications. They found the assigned severity, the bug description, and the number of methods and changes in the code as impacting factors. In Cinque et al. (2014), the fixing time is showed to impact the accuracy of the quality estimation made during testing. The work in Ihara et al. (2009) reports a study specifically focused on finding bottlenecks in the issue management process, highlighting that the main cause of inefficiency is the time lag in which the correction is verified. In our analysis, we adopt the same approach to approximate the fixing time by selecting only the phase of the bug lifecycle in which the bug report is actually modified. Authors in Mockus et al. (2002) analyzed two projects, Apache and Mozilla, to evaluate several aspects of the open source development, also considering the fixing time, by extracting the actual modification phase of bugs as in Ihara et al. (2009). Nistor et al. (2013) manually inspected performance bugs, analyzing, among other characteristics, their fixing process by looking at patches. We also analyzed the bug fixing process, along with testing, in an industrial context, highlighting a remarkable heterogeneity among components in mission-critical systems (Carrozza et al., 2014). The severity attribute is analyzed in Lamkanfi et al. (2010), Lamkanfi et al. (2011) and Menzies and Marcus (2008). They all analyze the severity as reported by the user, recognizing that it is a difficult attribute to assess. They investigated on methods to predict the severity assignment, so as to check the conformance with the user-assigned severity, by using text mining on bug reports and machine learning techniques. We opted for the same choices to analyze severity – that is, grouping severity into *high vs. low* categories and discarding the default assignment as it is shown to be unreliable. In our work, we also analyze fixing time and severity, but relating them to the bug manifestation type to figure out if these attributes are impacted by how bugs surface. A different set of studies argues about the quality of the bug report itself, and thus on the reliability of analyzes performed on it. Herzig et al. (2013) and Antoniol et al. (2008) raised the problem of issue report misclassification (i.e., reports classified as bugs, but actually referring to non-bug issues) highlighting that a relevant share (33% and 18–22% in the two cases) is misclassified. Bettenburg et al. (2008), 2007) also discuss the quality of bug reports by conducting a survey among developers to determine the information that is actually used (e.g., steps to reproduce, stack traces, and test cases) and the information supplied by users. In Wang et al. (2011), authors remark the importance of an efficient defect reporting on the testing process, demonstrating that improving reporting decreases the percentage of invalid reports from 26% to 19.53%. These studies suggest that, although analyzes on bug reports are extremely important to understand characteristics of

bugs, care must be taken in interpreting the results, which might be affected by incorrect or biased classification.

From all the surveyed papers, it is clear that several high-level characteristics (directly or indirectly) attributable to the bug manifestation process are studied separately. Furthermore, these characteristics are inconsistently assigned sometimes to bugs, sometimes to failures, and conflate disparate factors such as concurrency, memory, timing, interaction with OS or other applications, wrong resource management, wrong error handling. In few cases (Lu et al., 2008; Sahoo et al., 2010) , the research explicitly targets bug manifestation, analyzing few aspects (e.g., number of inputs required) or coarse categories (e.g., deterministic, non-deterministic), often embedded in studies with a different primary objective. Other studies examine important attributes of bugs (e.g., fixing time and severity) but unrelated to the bug manifestation. In this work, we have a special focus on bug manifestation characteristics, and examine them against several different attributes, such as opening time, fixing time, severity, and failure mode.

3. Study methodology

3.1. Characteristics of bug manifestation process

In this Section, we introduce the set of triggering conditions ("*triggers*") considered in the analysis. Triggers are viewed as the conditions necessary for the bug to be activated and propagated up to the user interface as failure. We first introduce a simple system model framing the considered triggers.

3.1.1. System model

We consider the *application* and the main external entities with a potential impact, namely the *user* and the *execution environment*, similarly to Chandra and Chen (2000). We assume an application as composed of processes and/or threads communicating with each other to accomplish the intended function, with communication channels implemented by either a global (e.g., shared memory) or a local model (e.g., message exchange). The state of the application includes the states of local processes (and/or threads), and of communication channels among them. A local state is the set of data (e.g., stored as variables in memory or files) which the processes/threads can operate on (i.e., read from/write to). We assume the execution environment as made up of concurrently running software and hardware possibly deployed across multiple machines. Software includes the system software of each machine (i.e., both operating system kernel and other system software such as compilers, linkers, debuggers, editors, user interface, utility software, libraries), middleware (e.g., virtual machines, middleware for distributed computing), and application-level programs. Hardware includes the physical machines on which the application is deployed, I/O devices, as well as the network connecting them. The user is the other external entity that interacts with the application by submitting workload requests and getting results. We assume that a workload request can be represented as a generic *request for service* (e.g., a query to a DBMS), characterized by a *request type* (e.g., query type, like INSERT) among a set of types, and by *input parameters* (e.g., values of an INSERT), in turn characterized by a type and a value. The request is processed and produces an output result (returned as value(s) or as a state change). To accomplish a well-defined task, the user can submit a sequence of (one or more) serial/concurrent requests.[1]

Triggers refer to *conditions of the bug manifestation process (not of the bug itself) necessary for the bug activation and propagation up to the user interface*. They are required to have the following properties: (i) A trigger is defined in the context of a system model like the described one, namely, it refers exclusively to the elements of the system model (e.g., workload requests, execution environment entities as abstracted by the model); (ii) a trigger is not meant to be general with respect to all systems; but it should be general with respect to the considered system model. In other words, triggers do not refer to one or few specific failure, but they should represent a condition that, with respect to the system model, is general, i.e., a trigger observed in one failure should be, in principle, applicable to failures of a system that can be described by the same model; (iii) triggers should be combinable with each other, under some composition rules, to describe multiple activation/propagation conditions, where failures are caused by more conditions together.

3.1.2. Trigger types

Considering the model and requirements above, we distinguish the macro-conditions possibly determining the bug manifestation, which may depend on: the submitted *workload*, the application's *(initial) states*, the *execution environment*, and the *user* behavior. Considering these factors, we identify 13 basic conditions as potential failure-causing triggers. We group them in *workload- and state-dependent*, and *user- and environment-dependent* triggers.

3.1.2.1. Workload- and state-dependent triggers. When the workload request(s) are the only condition necessary to expose a failure, the bug manifestation process is systematically reproducible. Intuitively, this means that repeating the same steps that led to a failure, always causes the same failure to be observed. In the easies case, the "same steps" means that it is sufficient to resubmit the last request (or few requests), and the failure reappears. In the worst case, the same requests sequence submitted by the user from the beginning (i.e., from the initial state) is needed, denoting a strong dependence on the application state. Any intermediate case makes sense, where there is a partial state dependence only from a certain point on in the sequence of requests.

More formally, *the bug manifestation is "workload- and state-dependent" if resubmitting (at most a subset of) the same workload requests that caused a failure always produces the same failure, for every valid state of the environment (i.e., for every state of the environment in which the traversed application states are allowed to occur) and for every admissible user inputs' timing/ordering in each request of the sequence*.

These are bugs whose activation and propagation are not affected by the environment or by the user inputs' timing/ordering. In practice, there may be different workload conditions activating the bug; we identify a set of triggering conditions characterizing the failure-exposing workload:

- ANY. This is the trivial case, when any workload request causes the failure. It means that the bug is activated with so many failure-causing inputs that any request type from the initial state will cause the bug activation and the failure to appear.
- REQUEST TYPE. The necessary failure-exposing condition is a request of a certain type. This means that whatever the initial and current (valid) state[2] of the application, a request of that type always causes the application to fail regardless of the type of input parameters and their values.
- SEQUENCE. More than one workload request is needed to lead the system to failure. For every (valid) initial state, a specific sequence of (serial/concurrent) requests is necessary to cause the failure. The bug-failure path is dependent on the state of the application. When the sequence is of "concurrent" request, we distinguish the trigger as CONCURRENT SEQUENCE.
  Note that one (and only one) of the first three triggers is necessarily present in any failure; these are therefore considered "basic" triggers.

[1] It may happen that a sequence of requests to accomplish the task is allowed to be submitted with different timing and/or ordering among requests, i.e., various timing/ordering alternatives are admissible for accomplishing that task.

[2] *Valid* means admissible with respect to the input request to submit.

- INPUT TYPE. The failure-causing condition is that one or more input parameters must be of a specific type.
- INPUT VALUE. The failure-causing condition is the value of one or more input parameters. We distinguish further among SPECIFIC, RANGE, or CLASS, depending on the value necessary to make the program fail. In the former case, one or more specific values are needed (e.g., boundary values); RANGE requires a set of values grater/less than a value or comprised in given range; CLASS are values with a (not-range) property (e.g., all string containing commas).
- CONFIGURATION. A specific application configuration is required, i.e., a proper subset of the initial states in which at least one configuration parameter must take a specific value, or in which a component is present or not.

It is important to remark that many bugs require more of these conditions together to surface, e.g., a specific REQUEST TYPE, a specific INPUT VALUE, and a certain CONFIGURATION. Each of the first three triggers represents a potential *necessary and sufficient* condition for the failure occurrence; if one of these (mutually exclusive) triggers is not sufficient by itself, one (or more) of the latter three trigger is also needed. If all of them are still not sufficient, then an external condition is required: the trigger belongs to the following high-level category.

3.1.2.2. User- and environment-dependent triggers. Bug reproduction might depend not only on the submitted program's inputs. More subtle bugs may be activated or not depending on the state of the environment. We capture these factors by *user- and environment-dependent* triggering conditions. Specifically:

The bug manifestation is "user- and environment-dependent" if resubmitting (at least a subset) of the same workload requests that caused a failure, there exist at least one (valid) state of the environment or user inputs' timing/ordering [3] *causing the same failure to not be reproduced (namely, either user- or environment-dependent).* Again, in such a range of possibilities, we abstract a set of triggering conditions capturing the user and environment influence. We have:

- USER TIMING/ORDERING.The same sequence of requests to accomplish the intended task may be submitted with several timing and/or ordering among requests: if these admissible alternatives affect the bug surfacing, the bug manifestation is said to depend on this trigger. Failures caused by concurrency whose activation depends on the user's request timing/ordering (e.g., atomicity violation, order violation Lu et al., 2008) are also labeled with this trigger. The trigger may appear together with one of the conditions on the execution environment introduced hereafter.

We identify execution environment conditions as characterized by three-dimensional categories, distinguishing whether: (i) the condition is *direct* or *indirect*; (ii) the condition affects the *activation* or the *propagation*; (iii) the condition involves one of these elements of the execution environment, distinguished according to the system model: *OS kernel's subsystems* (namely: *OS memory management, OS device drivers, OS filesystem, OS networking, OS process management* Love, 2010 and Bovet and Cesati, 2005), *other system software* (i.e.: *utility software, development software, user interface, libraries, other*), *middelware, application-level interacting software, hardware platform/resource* (e.g., CPU, disk, physical memory, I/O devices, buses, physical network, others). In particular, with respect to the first two dimensions, the four triggers are:

- EXEC-ENVIRONMENT – Direct Conditions (DC), affecting the Activation (A): there is an environmental condition, which caused the *bug activation*, that: (i) *directly* affects the state of the application (i.e., it causes a state change) before the bug activation, and (ii) does not hold in a successive attempt to repeat the failure-causing steps (namely, the bug activation is not deterministic). Examples are failures caused by particular thread scheduling provoking a race condition (i.e., system-dependent concurrency bugs).
- EXEC-ENVIRONMENT – Indirect Conditions (IC), affecting the Activation (A): The same as before, but the state of the application is not directly affected before the bug activation. For instance, a resource is temporarily unavailable.
- EXEC-ENVIRONMENT – Direct Conditions (DC), affecting the Propagation (P): The bug is deterministically activated in an environment-independent way, but there is an environmental condition affecting the *error propagation*, which: (i) causes the error being propagated in a different way on retry (and a different failure, or no failure, reaches the interface), (ii) *directly* affects the state of the application before the bug activation, and (iii) does not hold in a successive attempt to repeat the failure-causing steps. For instance, the application is designed to change a policy when an environmental condition occurs (e.g., free memory threshold exceeded), and this change does not affect the bug activation, but affects the error propagation (i.e., on a retry, the bug is always activated, but the policy change causes the error being propagated differently).
- EXEC-ENVIRONMENT – Indirect Conditions (IC), affecting the Propagation (P): The same as before, but the state is not affected before the bug activation. For instance, if a bug causing memory leak is activated deterministically on a requests' sequence, the way it leads to failure changes on a retry, depending on the available free memory and on the other applications' tasks during that retry. Indirect triggers, IC-A and IC-P, may be seen as *pure* environment-dependent conditions, as the state of the application is not influenced by the environment before the bug activation.

These four triggers on execution environment are mutually exclusive. We denote them as a triple, e.g.: ⟨*DC, A, OS Memory*⟩ is a memory state that directly influences the bug activation. Note that what typically and ambiguously intended as "bug type" (e.g., resource leak, concurrency, OS interaction) may fall in any of the presented categories depending on how the bug is triggered and surfaces.

TRANSIENT: it is a further trigger about possible transient behavior. In fact, the velocity at which an environmental condition changes plays a key role in the bug manifestation (Chandra and Chen, 2000). There may be cases (e.g., race condition due to OS threads scheduling) in which it is almost sure that the environment changes in a successive retry and the same bug may not surface. Other cases (e.g., a disk temporarily full) may require more time, and the bug may re-appear in a given time interval. We introduce the condition TRANSIENT, added to the previous ones whenever an environmental change occurs faster than the time to process the failure-causing requests' sequence: in such a case, the request repetition will find a diverse state.

EXEC-ENVIRONMENT CONFIG: this condition refers to bug manifestations exhibiting a deterministic behavior, but requiring a specific execution environment. We refer to the same environment values defined above (namely, OS memory, device drivers, hardware platform, etc.), but, in this case, they play a different role. For instance, a bug surfaces with a given workload sequence, but only under a specific filesystem, or because of a specific library. Unlike the previous environment conditions, repeating the steps that led to failure *in that specific environment* causes always the same failure to appear again.

3.2. Bug sources and classification procedure

The analysis is conducted on two open source software applications, Apache Web Server and MySQL DBMS. These are instances of modern large and complex software systems widely adopted in

[3] As before, *valid* means admissible, compatible environment state w.r.t. the input requests; in the case of user, it means that the same workload request(s) could be submitted in different timing/ordering producing the same result.

Table 2
Criteria to classify bugs according to the analysis.

Analysis of:	Category	Description
	NOT A BUG	*Request for new feature, enhancements, documentation, build, operator errors*
Trigger	**WL-DEPENDENT**	*Bug report assigned with only workload triggers*
	ENV-DEPENDENT	*Bug report assigned with at least one environmental trigger*
	NOT SUFF. INFO	*Reported information is not enough to classify it as WL or ENV-dependent*
Complexity	**UNMODIFIED**	*Bug report lifecycle goes directly from the "untreated" to the "resolved" phase*
	MODIFIED	*Bug report lifecycle goes from the "untreated" to the "modification" phase*
Temporal evolution	*Bugs from MySQL 5.1 and Apache 2.0 release time up to the second next major release (5.6 and 2.4)*	
	OMISSION	*The server does not respond to an input*
	TIMING	*The server responds untimely*
Failure mode	**RESPONSE**	*The server responds incorrectly*
	CRASH	*The server, after a first omission, keeps on not responding until restart*
	UNKNOWN	*Reported information is not enough to distinguish the failure mode*
Severity	**HIGH**	*Severity is "major", "critical", "blocker" for Apache; "critical" and "serious" for MySQL*
	LOW	*Severity is "trivial" and "minor" for Apache, "performance" for MySQL*
	DEFAULT	*Severity is "normal" for Apache, "non-critical" for MySQL*

business-critical contexts in their respective categories (Web Servers and DBMSs). The analysis is not intended to have a generally valid empirical characterization of triggers occurrence; we want to understand how triggers may occur in real-scale systems, not how they generally occur. Observed patterns and findings will therefore refer to the specific systems under analysis.

Apache has been using Bugzilla[4] as Bug Tracking System (BTS), whereas MySQL BTS is based on a PHP BTS[5]. Since both Apache and MySQL development started many years ago, tens of thousands of problems have been reported since then: to conduct the analysis, we focused on the following (stable) versions and on their principal components: Apache v2 *core* and MySQL v5.1 *server*. In order to work with mature and reliable bug descriptions, we filtered the repositories by extracting only reports that have been solved (marked as "closed"). Bugs explicitly marked by the reporter as *duplicate* and bugs marked as *enhancement* or *feature request* in the "severity" field have been excluded from the analysis. From this, a set of 666 problem reports came out, namely 98 and 568 for Apache and MySQL respectively. These have been manually inspected in each of the following sections provided in a report: (i) the textual description of the steps to repeat the failure; (ii) the textual discussion and comments of developers/users working on that bug; (iii) the final patch that has been committed, along with the description note in the change log; (iv) the possible attached files (e.g., test cases, environment configuration files). The criteria used to categorize the bugs for each of the conducted analyzes are reported hereafter and summarized in Table 2.

3.2.1. Criteria for trigger analysis

Each report is categorized into the following macro-classes: NOT A BUG, NOT SUFFICIENT INFO, WORKLOAD-DEPENDENT, ENVIRONMENT-DEPENDENT. Each report not falling in the categories NOT A BUG and NOT SUFFICIENT INFO is labeled with the failure-causing *triggers* responsible for the bug manifestation. Specifically, to assign bugs to classes, the following rules are followed:

- NOT A BUG: a report is classified as NOT A BUG if it reports: (i) a request for new features or for enhancement not marked erroneously as such[6]; (ii) a documentation issue (either of the source code or of test cases), such as missing, ambiguous, or outdated documentation; (iii) build-time errors (e.g., compilation or linking errors); (iv) operator errors.
- In each section of the report mentioned above, the inputs required for the bug manifestation (i.e., the failure-causing *workload*) are examined, along with the *application configuration*. One or more workload- and state-dependent triggers (in the following, simply WL triggers) are assigned to the report whenever the workload and/or application configuration match the definition provided in Section 3.1.2, representing necessary conditions for the failure. The report is labeled with these triggers.
- In each section of the report mentioned above, the possible conditions related to the *execution environment* or to *user actions* (in accordance with the definitions of Section 3.1.2) are examined (user- and environment-dependent triggers, in the following USR/ENV). One or more USR/ENV triggers are assigned to a report if there is any user/environment conditions described in the report matching the definitions of Section 3.1.2. Besides WL triggers, the report is labeled also with USR/ENV triggers if any of them is applicable.
- WORKLOAD-DEPENDENT: the report is classified as WORKLOAD-DEPENDENT if it has been assigned only WL triggers.
- ENVIRONMENT-DEPENDENT: the report is classified as ENVIRONMENT-DEPENDENT if, beside WL triggers, at least one ENV trigger is present as necessary failure condition.
- In each report (WORKLOAD-DEPENDENT or ENVIRONMENT-DEPENDENT), we examine if a specific execution *environment configuration* is reported as condition to make the bug surface. In such a case, the EXEC-ENVIRONMENT CONFIG trigger is also added as label to that report.
- NOT SUFFICIENT INFO: a report is classified as NOT SUFFICIENT INFO if the information contained in any of the inspected sections does not allow to classify a report in any of the previous categories (e.g., a bug that is closed as soon as it is reported because corrected by a new minor release). Note that even though the report contains some information useful for the subsequent analyzes (i.e., complexity, temporal, and impact analysis), but not enough information to be classified in terms of triggers, it is discarded from all the analyzes, because, in any case, we are interested in investigating the relation of triggers with the factor under analysis.

From the initial set of 666 reports, 18 out of 98 reports for Apache (18.37%) and 51 out of 568 reports for MySQL (8.97%) are identified as NOT A BUG; 4 out of 98 (4.08%) and 28 out of 568 (4.93%) reports for Apache and MySQL, respectively, are classified as NOT SUFFICIENT INFO. The remaining 565 reports (76 and 489 for Apache and MySQL, respectively) are WORKLOAD-DEPENDENT or

[4] https://www.bugzilla.org.
[5] https://bugs.php.net.
[6] The indication reported in the severity field, which should help distinguishing bugs from feature requests or enhancements, are known to be unreliable (Herzig et al., 2013; Antoniol et al., 2008) – thus the exclusion performed solely based on the severity field is not enough, and is integrated with the manual inspection

ENVIRONMENT-DEPENDENT bugs. This dataset of 565 reports (let us call it D1) is used in the other analyzes described in the following.

3.2.2. Criteria for complexity and temporal analysis

We analyze the correlation of triggers with fixing time of bug. To this aim, the history of each bug report is examined. The *lifetime* of a bug report is considered as the time difference between the bug closing time and the bug reporting time. To extract the fixing time of a bug, we adopt the same method as Ihara et al. (2009), and Mockus et al. (2002) as well: the lifetime of a report is split in three phases: *untreated phase, modification phase*, and *verification phase*. Bugs reported in the BTS but that have not been accepted or assigned yet are *untreated*. As soon as a bug is accepted and assigned to anyone, it is in the *modification* phase. In this phase, there is the actual discussion and action by developers to find a patch for the bug. When the developers finish to modify the bug, it is marked as "resolved". At that point, the *verification* phase occurs, where the correction is verified by the quality assurance team. For the purpose of our analysis, we thus consider the *modification* phase as actual working phase by the developers on the bug report. This phase is hereafter denoted as *time to fix*. Moreover, in the case of bugs got re-opened (i.e., going from "resolved" state back to the modification phase), the total time to fix is computed as sum of times in which the bug has been in the modification phase. Finally, as in Ihara et al. (2009), bug reports going directly from untreated to verification (i.e., unmodified but closed bugs) are labeled as UNMODIFIED and not considered in the time to fix computation. Thus, each report is labeled either as MODIFIED or as UNMODIFIED. Out of the initial 565 bug reports of D1, 19 are found to be UNMODIFIED and discarded from this analysis. The final dataset for complexity analysis is of 546 reports: we denote it as D2.

Regarding the temporal analysis, we consider the opening time of each report of the dataset D1. This is then used to distinguish the temporal evolution of WORKLOAD-DEPENDENT and ENVIRONMENT-DEPENDENT bugs. However, comparing bugs of the two applications from their release time up to now is inconclusive, because MySQL 5.1 was released in 2005, while Apache 2.0 in 2002. Therefore, we consider the bugs from the release time of MySQL 5.1 and Apache 2.0 up to the second successive major release, namely to MySQL 5.6 and Apache 2.4 (in fact, we may generally consider that after new major releases, the usage of older versions, and thus the bugs reported, progressively decreases). This choice led us to discard 4 bugs in MySQL and 2 bugs in Apache from D1, with temporal frames going from 2005 to 2011 for the former (i.e., up to MySQL 5.6) and from 2002 to 2012 for the latter (up to Apache 2.4). The data set for temporal analysis is of 559 bug reports denoted as D3.

3.2.3. Criteria for impact analysis

To perform the analysis relating the bug manifestation to the perceived impact by the user, we categorize each report of D1 with respect to the perceived *failure mode* and *severity*. Specifically, as for failure modes, we borrow the following well-known classification by Cristian (1991a), Cristian (1991b):

- OMISSION: a report is labeled with this failure mode when the server omits to respond to an input.
- TIMING: a report is labeled with this failure mode when the server's response is functionally correct but untimely. Timing failures thus can be either early or late timing failures (performance failures).
- RESPONSE: a report is labeled with this failure mode when the server responds incorrectly: either the value of its output is incorrect (*value* failure), or the state transition that takes place is incorrect.
- CRASH: a report is labeled with this failure mode when, after a first omission, the server omits to produce output to all subsequent inputs until its restart.
- UNKNOWN: a report is labeled in this way when it is not possible to distinguish the failure mode from the filed bug report description.

The *severity* field assigned by the reporters has been shown to be unreliable (Herzig et al., 2013; Antoniol et al., 2008) although there exist guidelines on how to assign the severity of a bug. One of the main reasons is that many BTSs use default values which are never updated by the reporter, because they might not be able to assess severity at the initial stage of bug lifecycle, and then there is no benefit to update the value when the bug is going to be fixed. To reduce the impact of incorrect assignments, we have:

- Enlarged the grain of the analysis, by splitting data into HIGH *vs.* LOW severity categories, similarly to the work by Lamkanfi et al. (2011). HIGH severity class for Apache reports includes: *major, critical, blocker*, while for MySQL it includes *critical* and *serious*. severity for Apache includes: *trivial* and *minor*, while, for MySQL, it includes *performance. Enhancement* (for Apache) and *feature request* (for MySQL) are discarded.[7]
- Discarded the default assignment field values (labeled as DEFAULT), i.e., *normal* for Apache, and *non-critical* for MySQL so as to rely only on cases where we are sure that the severity was set by the reporter (Lamkanfi et al., 2011). Specifically, from the considered dataset D1, we have discarded 40 bugs for Apace and 260 for MySQL, getting to a dataset, for this analysis, of 265 out of 565 bug reports; we call it D4. This way, we have a considerably reduced but more reliable dataset for the impact analysis.

3.2.4. Procedure

Summarizing, based on the above criteria, the manual classification was conducted in the following way:

- Two authors independently inspected all the reports classifying them according to criteria described above. From the initial set of 666 reports, 69 reports (18+51) are identified as NOT A BUG; 32 (4+28) are classified as NOT SUFFICIENT INFO. The remaining reports are classified as either workload-dependent or environment-dependent ones, and assigned the corresponding *triggers* (WL and or ENV triggers).
- During the inspection, both authors examined the bug history, and label the bug as MODIFIED or UNMODIFIED according to the criterion described above. If the bug is MODIFIED, they compute and record the *time to fix* as time in which the bug is in the modification phase (possibly accounting for re-openings as described). The bug reporting timestamp is also recorded, for the temporal analysis.
- During the inspection, both authors assign a label for the failure mode analysis, among: OMISSION, TIMING, RESPONSE, CRASH, or UNKNOWN, and a label for severity: HIGH, LOW, or DEFAULT.
- Conflicting cases for each field are resolved by discussions to reach a consensus among all the authors. An immediate consensus was reached for all but 7 reports (1.05%), for which an agreement about triggers was not achieved even after the discussion. In those cases, we needed to actually reproduce the failures to dispel any doubt about necessary triggers.
- Once labels are assigned, we performed the counting for each analysis using the corresponding datasets: D1 for trigger and failure mode analysis; D2 for complexity analysis; D3 for temporal analysis; D4 for severity analysis.

The impact of the assumptions entailed by the described empirical study set up are discussed in Section 8.

[7] As mentioned above, reports marked explicitly as feature requests and enhancements are not inspected and soon discarded from the analysis; reports found to be feature requests and enhancements during the manual inspection are classified as NOT A BUG and discarded.

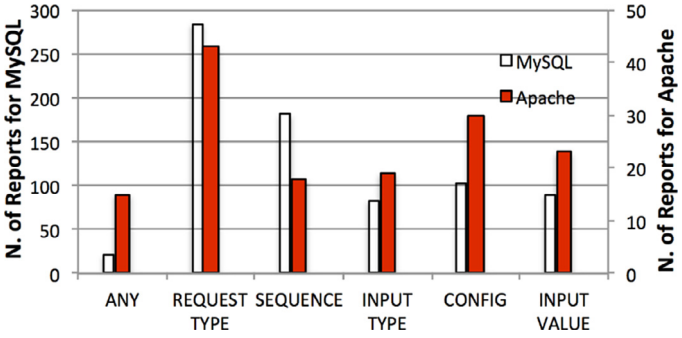


Fig. 1. WL triggers as necessary failure condition.

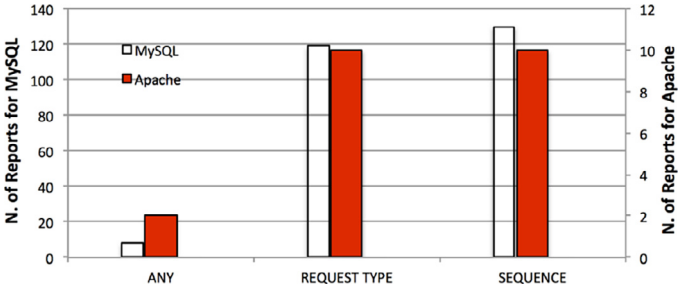


Fig. 2. WL triggers as necessary and sufficient WL condition.

4. Trigger analysis

This Section reports the results of the trigger analysis in terms of findings and their implication. We use, for brevity, "*workload*-dependent" (WL) and *environment*-dependent" (USR/ENV) triggers in lieu of *workload- and state-dependent* and *user- and environment-dependent* triggers, respectively. The whole set of findings is listed hereafter.

Finding #1: *the manifestation of most of the reported bugs (80.71%) needs only one or more workload conditions to surface, with no environmental condition required.*

In particular, this percentage is 69.73% (53 out of 76) in Apache and 82.41% (403 of 489) in MySQL, while the remaining 23 and 86, respectively, are USR/ENV. Workload-dependent bugs are easier to expose, as their activation and propagation is not affected by the current environment. The share of environment-dependent bugs (≈20%) basically confirms the similar percentages reported in those studies that in some way consider the environment as a factor for reproduction (e.g., Trivedi et al., 2011; Grottke et al., 2010). More details follow in the analysis of workload and environment-dependent bug surfacing characteristics.

4.1. Workload conditions

4.1.1. Basic conditions

Fig. 1 shows the total number of reports in which each workload trigger was necessary for the reproduction. Since, for many bugs, more than one condition is needed, the sum over all the triggers is greater than the total number of reports analyzed, while the sum of the first three mutually exclusive triggers amounts to the total. Fig. 2 shows the number of cases in which each workload trigger is the *only necessary workload condition* for making the bug surface (namely, the trigger appears alone or in conjunction with a USR/ENV trigger).

Bugs activated only by the ANY trigger are in principle too trivial to escape testing, resulting in very few cases. Most of bugs are either dependent on a request type or on a sequence of requests. Specifically:

Finding #2: *for 57.87% of total bugs, the manifestation requires a specific request type; in most cases (35.04%) it requires a further additional condition to expose the failure.*

This is consistent between MySQL and Apache, with REQUEST TYPE being 58.04% and 56.58% respectively. It also appears relatively many times as a *necessary and sufficient* WL condition (Fig. 2), but in that case it does not overcome the SEQUENCE trigger. Examples are bugs activated with a specific query type (e.g., bug #24562: MySQL raises an assertion on an '*ALTER TABLE t ORDER BY n*' instruction; or bug #23379: MySQL provides a wrong time value on the 'SHOW PROCESSLIST' request).

Finding #3: *the manifestation of 35.22% of total bugs are due to sequences of requests, and are dependent on the state of the application. In 24.78% of cases, the sequence is sufficient by itself to make the bug surface.*

The SEQUENCE trigger is very relevant especially for MySQL, where many more "state-dependent" failures (activated through series of queries) are observed compared to Apache. We observe that it often appears as necessary and sufficient workload trigger (131 cases out of 181 for MySQL and 10 out of 18 for Apache – Fig. 2), being the most frequent condition able of causing a failure by itself. Moreover, 20% of them for MySQL and 22% for Apache refer to sequences of *concurrent* requests, which, in many cases, are present along with environment triggers. An example is bug #11983 in Apache, where connection is closed prematurely on a sequence of concurrent CGI requests.

Implications: while the bugs that need a specific request type plus some other conditions may be complex, and thus naturally escape testing, there is a percentage of bugs requiring only a request type as condition that is still unexpectedly high. This might indicate that: (i) a poor basic functional testing on input commands was conducted (compared with other V&V strategies more suitable for other triggers), which should be improved, and/or that (ii) a continuous introduction of changes (e.g., due to frequent development and maintenance) are likely to introduce new bugs (e.g., regression bugs) that remain undetected. In both cases, a persistency of testing strategies for basic functional testing (with respect to other V&V activities targeting other trigger types) should be implemented along all the product lifecycle.

Finally, the high percentage of bugs requiring "sequences" of requests suggest that a majors focus on bug detection activities that consider the state of the application can expose more bugs. Examples of these techniques are: state-based (e.g., model-based) testing, stress (long-running) testing, interaction testing.

More conditions together

Besides basic triggers, bugs surfacing is often due to more workload conditions together:

Finding #4: *the manifestation of 50.62% of total bugs require a secondary WL condition, related to the input type (present in 18.05% of total bugs), to the application configuration (23.54%), or to the input value (20.00%).*

INPUT TYPE, CONFIG, and INPUT VALUE triggers have similar values for MySQL (Fig. 1), whereas we notice a slight peak of the CONFIG trigger in Apache, indicating more failures due to untested configuration options. As for sub-categories of the INPUT VALUE trigger, we observed that:

Finding #5: *INPUT VALUEs causing more often failures are due to one or more specific values. Particularly, the manifestation of 11.68% of total bugs require a specific value, 4.78% require a specific range of values, and 3.54% require a class of values.*

The conditions on the INPUT VALUE triggers are: 56, 22, and 12 for SPECIFIC, RANGE, CLASS respectively, in MYSQL; whereas they are: 10, 5, 8 for the same categories in Apache. The most frequent condition is when one or more specific values are needed (e.g., boundary values), the least frequent one is the "class" values (e.g., values with a property in common, as multiples of a given value, string

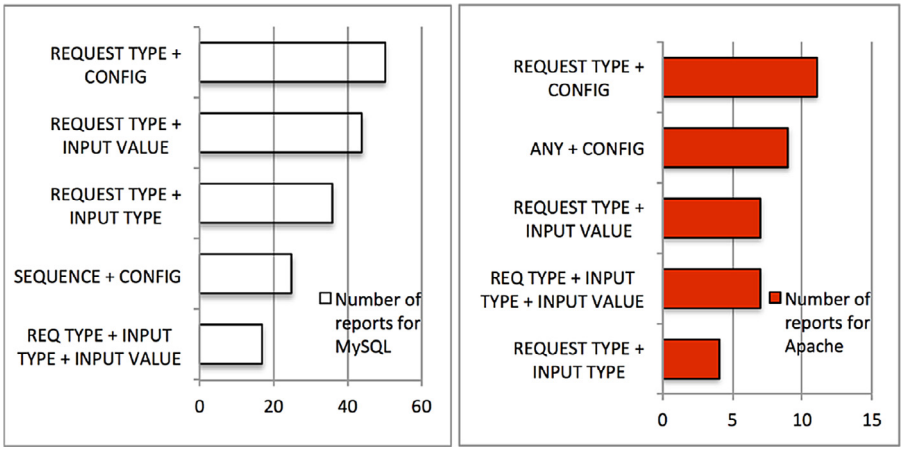


Fig. 3. Counts of top-5 WL trigger combinations.

containing commas, non-Latin chars, etc.). Examples of input-related triggers are: bug #23653, where MySQL crashes if the *last_day()* function is used with a zero date; bug #20404 of MySQL: query fails with inputs containing Turkish char; bug #31237 of Apache triggered when the input file given in the URL is greater than 2GB.

Fig. 3 reports the count of the top-5 combinations of WL triggers, highlighting the most common patterns that we encountered.

Finding #6: *most of bugs requiring at least two workload conditions is caused by a specific request type together with a specific configuration (10.79% of the total bugs). The occurrence of a request together with a specific input value is also a relevant combination (9.02% of the total bugs). The third most relevant combination is a request type together with a specific input type, occurring in 7.08% of cases.*

The most frequent patterns are the REQUEST TYPE together with one more trigger (with CONFIG being the most frequent in both systems). It is interesting to note that among the top-5 combinations there is also one requiring three triggers together, that can represent quite hard-to-reach bugs.

Implications: the second workload condition, when present, is quite evenly distributed; this means that, as half of the analyzed bugs require at least two workload conditions, bug detection activity must consider different types of testing/analysis to capture them (Cotroneo et al., 2013c). While configuration problems require more extensive field testing, the detection of bugs related to the input type or input value calls for strategies acting on input data classes, e.g.: robustness testing, boundary-value analysis, interaction/integration testing. For instance, the mentioned example bugs (#23653, #20404, #31237) could have been detected with boundary-value and robustness testing.

4.2. Environment conditions

Fig. 4 reports the number of times each environment trigger is present as necessary condition, along with some workload trigger(s). An EXEC-ENV-$*$ trigger can occur along with a USR-TIM/ORD trigger. Note that the figure, for sake of clarity, does not distinguish the environment subsystem involved, which is reported in Table 3. Patterns for MySQL and Apache are different. In MySQL, relatively many more failures depend on user timing (and less often on ordering), which is badly managed by the system. Apache presents few failures exposed by user timing/ordering, and more cases regarding the bug propagation (i.e., deterministically activated) than the activation (13 *vs.* 9) differently from MySQL (29 *vs.* 39). About the execution environment (excluding the user timing/ordering), we can distinguish some patterns common to the two studied application, while others are completely different. Specifically, we have that:

Finding #7: *the manifestation of most of total bugs due to the execution environment is caused by indirect environmental conditions (60.29% and 72.73% for MySQL and Apache respectively) – namely, what we called "pure" environment-dependent bugs.*

Table 3
Involved execution environment item.

	MySQL				**Apache**			
	DC		*IC*		*DC*		*IC*	
	A	*P*	*A*	*P*	*A*	*P*	*A*	*P*
OS memory mng.	2	0	3	28	0	1	2	7
OS device dirvers	0	0	1	0	0	0	0	0
OS filesystem	0	0	3	1	0	0	0	1
OS network	0	0	1	0	0	0	0	2
OS process mng.	23	0	4	0	3	0	0	2
Other system sw	1	0	0	0	0	0	0	0
Middleware	0	0	0	0	0	0	0	0
App-level sw	1	0	0	0	0	0	0	0
Hardware	0	0	0	0	0	0	0	0
Unknown	0	0	0	0	2	0	2	0
Total	**27**	**0**	**12**	**29**	**5**	**1**	**4**	**12**

As an example, consider the bug #13543 of MySQL: the server crashes when creating a stored procedure from the command line because of the data files being corrupted.

On the other hand, there are differences in the most impacting environment factors. Table 3 reports the details. Unlike Apache, most of the ⟨*DC, A,* $*$⟩ in MySQL are race conditions, and some of them occur along with a user timing trigger. A big share is due to ⟨*IC, P, OSMemory*⟩, where problems related to memory corruption, use of uninitialized data, buffer overflows, and memory leak are preponderant. ⟨*IC, A,* $*$⟩ failures are due to memory management, network resources, process management; one ⟨*DC, P,* $*$⟩, a memory-related problem, is observed only in Apache. An interesting pattern regards concurrency bugs:

Finding #8: *the manifestation of 7.56% of total bugs of MySQL are due to concurrency. In Apache, this percentage is lower, 5.26% over the total. The total share of concurrency bugs is 7.25% of bugs*

This percentage refers to bugs requiring (i) either a specific timing or ordering of user requests (USR TIM/ORD trigger), (ii) or a system-related concurrency trigger (e.g., a ⟨*DC, A, OS Process management*⟩ like a race condition among scheduled processes/threads), (iii) or requiring both of these triggers. However, the opposite is not true: there are few USR TIM/ORD cases not related to concurrency problems, e.g., a delay between two sequential requests is required for the bug to occur. In particular, concurrency bugs are those requiring also a "concurrent sequence" as workload condition (and not just a sequence), besides these environmental triggers. As for instance, consider MySQL bugs: the manifestation of 9 bugs (i.e., 1.84% of the total) requires both USR TIM/ORD and ⟨*DC, A, OS Process management*⟩

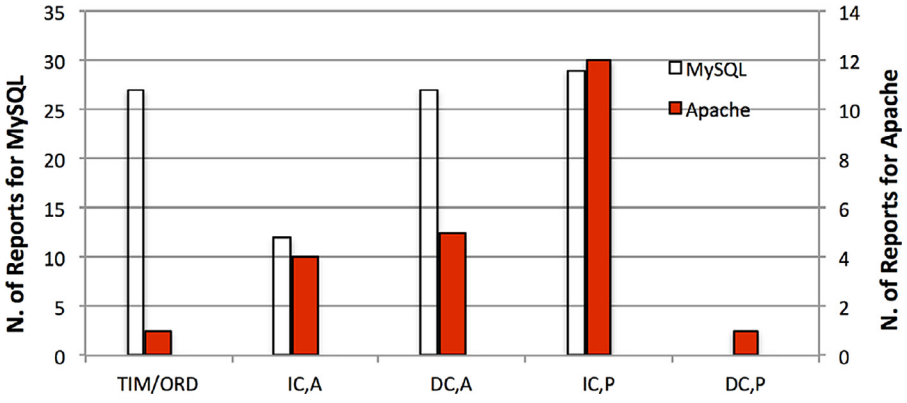


Fig. 4. USR/ENV reports per trigger.

trigger; 16 bugs (i.e., 3.27%) require only USR TIM/ORD, and 14 bugs (i.e., 2.86%) require only ⟨*DC, A, OS Process management*⟩: in total, 7.97%. The residual 0.41% (7.97 %−7.56%) are USR TIM/ORD not due to concurrency. Examples of concurrency bugs are: bug #38691 of MySQL, which provokes a server crash caused by concurrent execution of multitable update with a join and a 'flush table' or 'alter table' query; Apache bug #11983: connection closed prematurely on concurrent high load of CGI requests.

A second observation coming from the same Table is the following one:

Finding #9: *the 6.19% of total bugs (32.11% of environment-dependent ones) are related to memory management issues (⟨IC,P, OS Memory management⟩).*

This percentage is 5.72% and 9.21% in MySQL and Apache, respectively, equivalent to 32.55% and 30.43% of environment-dependent ones, revealing a high sensitivity to memory problems in both cases (e.g., memory leaks, uninitialized memory, buffer overflow). An examples is the apache bug #29962, due to a wrong management of memory acquired via the 'byterange' filter and no longer released, or the MySQL buffer overflow bug (#28361). Overall, user/system concurrency and memory issues are the most influencing factors for manifestation of environment-dependent bugs.

Implications: in general, the influence of conditions external to the application, as the *IC,P* category, suggests including the real operational environment in bug detection activities, where the influence of other applications, of configurations (e.g., disk usage, memory usage, concurrency with other programs), and of field usage is exercised. For instance, the mentioned bug #28361 of MySQL was activated by the interaction with JDBC on windows and was detected by running the JDBC compliance testsuite. Operational fault tolerance means are also a powerful tool to deal with environment-related problems and mitigate their effect (Trivedi et al., 2011), such as *restart* application/components, *retry* operations, *failover* to replicas. Activities targeting concurrency and memory problems would improve the detection of many of the observed bugs, such as static and dynamic memory and point-to analyzes, concurrency (e.g., lock-set) analysis, workload stress and robustness testing, finite state verification (e.g., model checking) techniques, and manual inspection. For instance, several memory-related bugs (e.g., #24403, #24486, #27733 of MySQL) was actually detected by the dynamic analysis tool *Valgrind*.

4.3. Workload and environment conditions

Fig. 5 reports environment-dependent bugs separated by workload trigger. The third column confirms that SEQUENCE is by far the most common workload trigger for environment-dependent bugs.

Finding #10: *the manifestation of 76.15% of environment-dependent bugs requires a sequence of requests as basic workload condition, besides the environmental condition.*

SEQUENCE is the most complex of the three basic triggers; when appearing together with USR/ENV triggers, the corresponding bug is usually a hard to reach one. This behavior is less pronounced with Apache (47.82%), in which REQUEST TYPE is also remarkably present along with USR/ENV triggers: in line with previous remarks, Apache seems to have less residual state-dependent bugs, but more bugs activated by a single request, which should have been easier to reveal through testing. As for transient bugs:

Finding #11: *the manifestation of 7.26% of total bugs occurs only in presence of environmental conditions that are "transient".*

Percentages are very similar in MySQL and Apache: 7.15% (35 of 489) and 7.89% (6 of 76), respectively. These bugs never appeared with only user timing-ordering trigger, but along with an environment trigger (in most cases), or with both an environment and user timing-ordering trigger. Many of them are, in fact, due to system-level concurrency management, appearing together with "concurrent sequence" and ⟨*DC, A, OS Process management*⟩ triggers. Note that the difficulty in reproducing TRANSIENT failures may be exacerbated by the other triggers required, like a case in MySQL requiring 5 conditions together: SEQUENCE, CONFIG, USR TIM/ORD, ⟨*DC, A, OS Process management*⟩, TRANSIENT. An example of such bugs is bug #36579 of MySQL, in which the dump of information about locks in use may lead the server to crash because of a wrong management of a mutex causing potential race conditions among threads.

Finally, we analyze the bugs depending on a specific environment configuration, which we tagged as EXEC-ENVIRONMENT CONFIG:

Finding #12: *the manifestation of 21.24% of total bugs requires a specific environment configuration. This condition is more pronounced in Apache than in MySQL.*

Specifically, we found this to be a necessary condition for Apache in 31 cases (i.e., 40.79%), many of which (14 of 31) are USR/ENV. For MySQL, we found 89 reports (i.e., the 18.2%), of which much fewer (12) were USR/ENV. Thus, bugs of Apache requiring a specific environment are much more prone to be also USR/ENV-dependent (namely, a fixed environment is required, as a specific filesystem, and an environment condition must also occur during the execution, e.g., the disk is temporary full), while in MySQL most of them are completely "deterministic" in that fixed environment (e.g., a given request type is needed, but it causes failure only with a specific filesystem, or because of a specific library).

Table 4 shows the count separated by type of environmental configuration item required for reproduction.[8]

Finding #13: *most of bugs requiring a specific environment configuration occur only under a specific OS (67 of 120, namely, 55.83%); 24.17*

[8] There is the additional category "OS other" including: the cases where there is no sufficient information to understand which part of the OS the report refers to, as well as the cases where the problem is due to the specific implementation of the software for that OS.

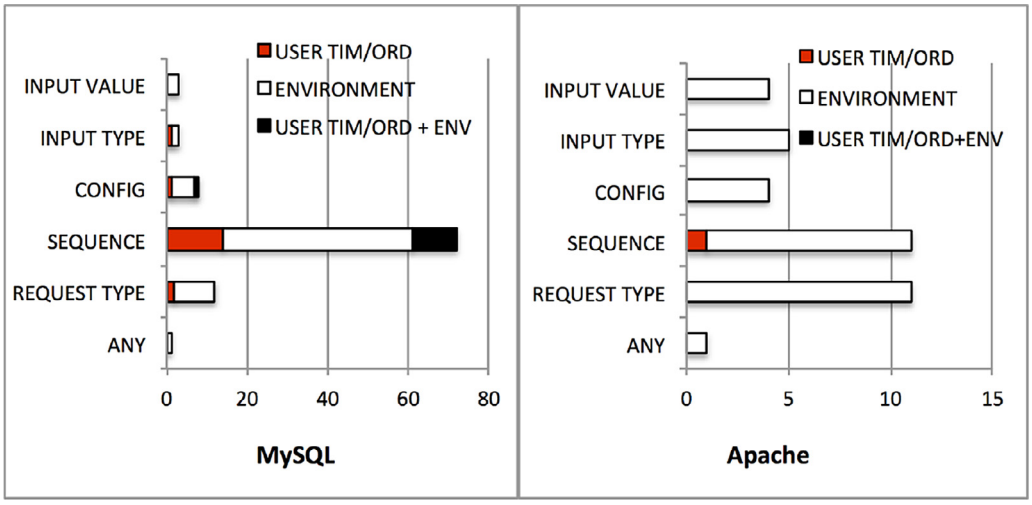


Fig. 5. WL Triggers in USR/ENV triggers.

Table 4
Environment-specific bugs count and % share.

	Apache	%	MySQL	%
OS device drivers	0	*0.00*	1	*1.12*
OS filesystem	5	*13.51*	16	*17.97*
Network	2	*5.40*	2	*2.25*
OS other	23	*74.19*	22	*24.72*
Other system sw	0	*0.00*	22	*24.72*
Middelware	0	*0.00*	0	*0.00*
App-level sw	1	*2.70*	6	*6.74*
Hardware platform/resources	0	*0.00*	20	*22.47*
Total	31	100	89	100

% require other (system or application-level) software, the remaining 20.00% require a specific hardware/network configuration to surface.

In particular: in Apache, we have a big percentage of bugs due to a specific OS, where there is often no detail to discriminate which part of the OS is involved; the second category is a particular filesystem required for the bug to appear. In MySQL, there is a high percentage of *Other system software*, namely, of bugs strictly requiring a specific library or a complier (option), and a high number of hardware-dependent bugs (i.e., requiring a specific platform); filesystem is a relevant item also in MySQL. Sometimes, bugs require a specific interacting software to surface, such as specific clients or connectors.

Implications: transient behaviors are very difficult and expensive to catch. Depending on the context, developers might decide to avoid spending effort for this category of bugs in favor of other categories (i.e., finding more but simpler bugs), or they might be forced to address as many bugs as possible (e.g., in critical systems, where these behaviors cannot be neglected). Often, such bugs may result in undefined behaviors and may appear or not depending on uncontrolled factors like the thread scheduling. These problems can be addressed either by bug detection or by runtime fault tolerance strategies, or by both of them. As for bug detection, suggested activities include all those hunting very rare situations, such as model checking, stress/concurrency and robustness testing, testing or analysis for exception handling. Formal methods as model checking, symbolic execution, and/or static/dynamic analysis tools on concurrency and memory properties are the most indicated ones for a preliminary analysis. Stress/robustness testing can be a valuable follow-up at system/acceptance testing stage. Fault tolerance includes operation retry, application restart, node reboot, failover. If a system already foresees fault tolerance means, a technique that can be particularly useful to verify the appropriateness of such means against transient behaviors is fault-based testing. As for bugs depending on the environment configuration, an intensive activity of testing on different configurations, e.g., operating systems, hardware platforms, and of testing the interaction with dependent software, as libraries, complier, and third-party software applications, is definitely a suggested task to reduce the occurrence of runtime problems.

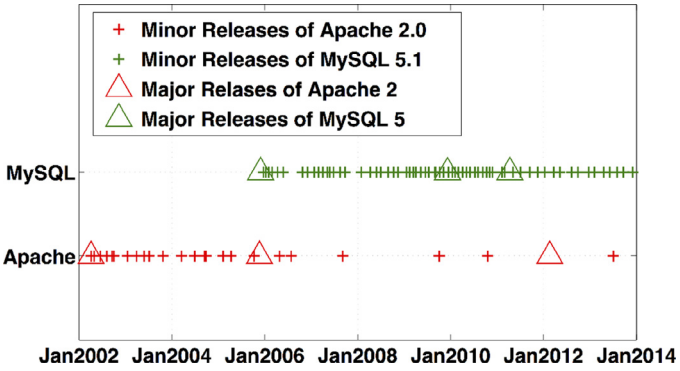
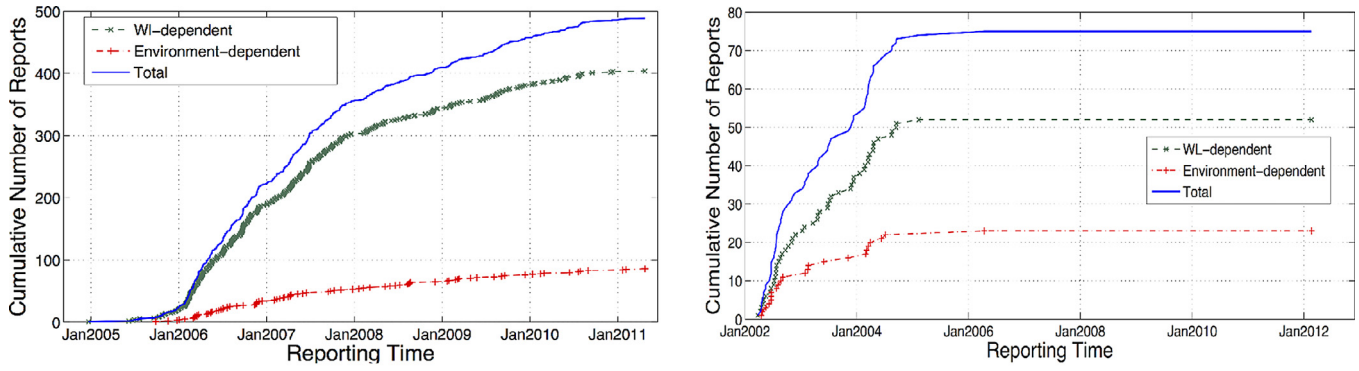


Fig. 6. Major and minor release dates for Apache and MySQL.

5. Temporal analysis

We analyze the temporal evolution of workload-dependent (WL) and environment-dependent (USR/ENV) bug triggers. For a more accurate analysis, we consider how the observed trends are related to the occurrence of major and minor releases. Fig. 6 shows minor and major releases of MySQL and Apache, starting from MySQL 5.1 and Apache 2.0., while Fig. 7(a) and (b) shows bugs reported over time. We assume that minor releases of a version (which can be due to improvements, new features, or bug fixes) indicate that the community is still active on that version. Based on this, *upon a major release* it may happen that:

1. There are many following minor releases and many bugs being detected. This might be the case of MySQL soon after the 5.5 release (December 2009): there are several minor releases and bugs were still being found roughly at the same rate of the previous year (almost 50 bugs per year).
2. There are many following minor releases and few bugs being detected. This is the case of MySQL after the 5.6 release (April 2011): there are still many minor releases up to 2014, as can



(a) MySQL (b) Apache

Fig. 7. Reporting time of bug reports.

be seen in Fig. 6, but the rate of reported bugs is considerably lower: 4 bugs are found since April 2011 up to 2014.

3. There are few following minor releases and few detected bugs. This is the case of Apache, as there is a sudden decrease of reported bugs soon after Apache 2.2 (3 bugs after November 2005 up to 2014) and also a sudden stopping of minor releases. This suggests that people migrated to Apache 2.2.
4. There are few following minor releases and many detected bugs. We did not encounter such a situation; it might be due to people still using a version, but developers focused only on subsequent versions.

Therefore, although we adopt the same criterion, there is a remarkable difference between the two case studies. For instance, for the purpose of analyzing triggers temporal evolution, data of Apache 2.0 are not much indicative after the 2.2 release time, as people likely migrated to the latter version. Oppositely, in MySQL, we have a smoother decrease and a different relation with the number of releases. There is an inflection around January 2008 (from more than 100 bugs per year to about 50), which is not related to the occurrence of any major release (it is in the middle between 5.5 and 5.6 release time); from 2008 to half 2010, there is an intensive usage and a product still not mature (many bugs still found at a constant pace); finally, form half 2010 on, there are many minor releases and few bugs reported up to 2014 (just 4), likely indicating a stabilization of MySQL 5.1 in quality (minor releases are likely more focused on small improvements and on few remaining bug fixes). Since MySQL 5.1 is actively used up to 2014, we consider these bugs as a good material to conjecture about triggers temporal evolution.

With this difference in mind, let us observe the temporal trend of WL and USR/ENV bugs manifestation. Bugs in MySQL span from 2005 to 2011. USR/ENV bugs start appearing later than workload-dependent ones. Then, their manifestation rate increases almost constantly until the last report time, whereas WL bugs have an inflection around January 2008. The trend of USR/ENV bugs supports the conjecture that they surface more rarely, and tend to require more operational time for the necessary conditions to occur. But, at the same time, we observe also a continuous increase of WL bugs, despite the inflection in January 2008: the hypothesis that these simpler bugs disappear early is not supported by data. This can be explained by considering that not all the WL bugs are so simple as one may believe: for instance, bugs requiring long and complex input sequences or specific values to be activated need time to be discovered, even though they are viewed as “deterministic”. Moreover, the introduction of new features or the fixing of older bugs has a non-negligible chance of introducing new (and regression) bugs, keeping the WL bugs trend increasing.

In the case of Apache, USR/ENV bugs start appearing only slightly later than WL ones, and the two curves have similar shapes from 2002 to 2006, although the rate of WL bug is higher. However, as mentioned, data from 2006 to 2012 of Apache are not indicative (as there is no bug and very few minor releases). Fig. 8(a) and (b) shows the (non-cumulative) proportion between USR/ENV and WL bugs. As for MySQL, since the beginning of 2008, the ratio has higher values (there are 8 four-month periods with a ratio over 0.25 – i.e., at least 1 bug out of 4 was USR/ENV, while, before 2008, this happened only in 2 four-months slots). The inflection of WL bugs around January 2008 (Fig. 7(a)), along with the nearly constant increase of USR/ENV-dependent bugs, confirm the proportion trend. The same does not stand for Apache, as both curves are saturated at Apache 2.2 release time, 2006. Overall, we can infer that:

Finding #14: *environment-dependent bug manifestations start appearing later and have a slower increase rate than workload-dependent ones.* This is more evident in MySQL than Apache, but essentially confirmed in both cases.

Finding #15: *the ratio of environment-dependent over workload-dependent bug manifestations in MySQL is lower in the first 4 years than in the period 4–8 years after the release time.* In Apache, this trend is not observed because of the sudden stopping of both bug types, which makes data less reliable for this analysis.

Implication: besides what already discussed about USR/ENV triggers, findings of this analysis (especially referred to MySQL) suggest that: (i) after an initial operational stage, whose duration depends on the application, a special focus could be put to hard-to-activate WL bugs (i.e., where more WL triggers together are needed). These, however, may require expensive bug detection activities, like combining different testing strategies, or using static analysis and inspection. Depending on the effort spent in previous phases and on bug types that are being observed, one could opt whether to devote more effort to reduce USR/ENV bugs or hard WL bugs. The second implication is that: (ii) the characteristics of development paradigms with frequent releases, which is a common practice today, calls for greater attention to bugs introduced in new features and to regression bugs. A continuous functional testing activity is required to implement this type of development; such an activity is also to be considered when deciding how to spend testing effort at later development (or even operational) stage.

6. Complexity analysis

We analyze reports from the complexity perspective in terms of bug manifestation and bug fixing. We first consider the number of contemporary conditions required to make a bug surface, and then explore the relation between the bug manifestation and the fixing time. Fig. 9 shows the number of triggers required for bug

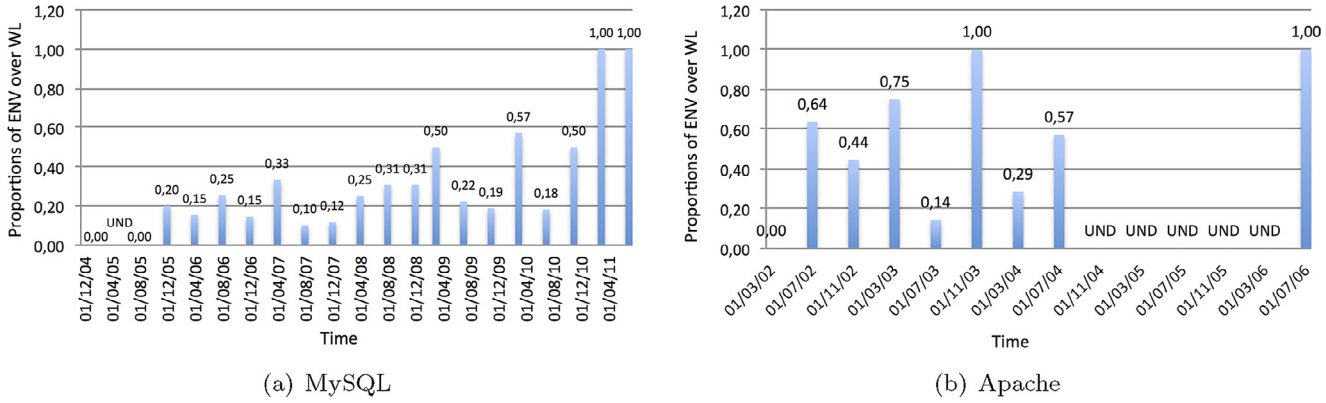


Fig. 8. Proportion of environment-dependent bugs: number of USR/ENV over number of WL occurred in a time frame of 4 months. 'UND' indicates 0 USR/ENV over 0 WL bugs.

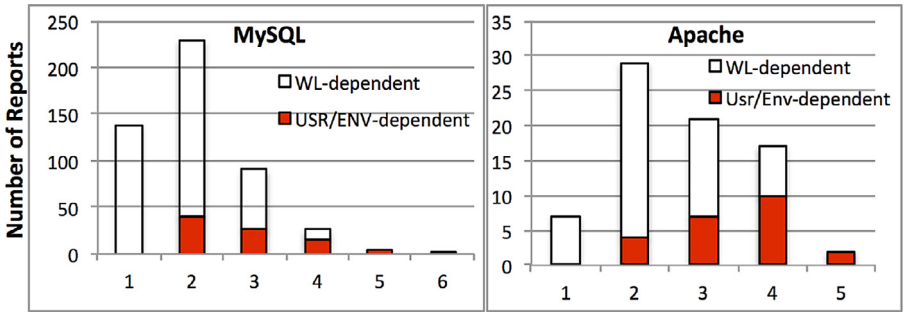


Fig. 9. Number of reports *vs.* number of required triggers.

manifestation. It is a rough indication of the difficulty for making a bug surface, as bugs requiring more conditions together to appear are likely to present a more complex manifestation process. However, complexity depends also on the type of trigger required and on the operational profile (e.g., environment triggers are typically rarer conditions than the workload ones). Hence, a bug with two triggers such as a requests "sequence" and an environment-dependent trigger may be much harder to expose than a bug with three WL triggers.

Finding #16: *the 45.66% of the total bugs require two conditions to surface, more often two workload conditions (in 37.87% of total bugs)*

Finding #17: *bugs requiring one condition to manifest themselves are less common than bugs requiring two conditions, meaning that the former are too trivial to survive early testing.*

Finding #18: *the 7.61% of total bugs require 4 conditions together to surface; the 1.06% require 5 conditions. All these represent hard-to-reach bugs, that are very difficult to uncover by bug detection activities and are likely to surface only after some operational time.*

Patterns of Fig. 9 reveal no remarkable differences between Apache and MySQL.

While the number and type of conditions indeed affect the bug manifestation, it is not clear if they have an influence on the debugging process. On one hand, the bug surfacing characteristics should have no effect on the difficulty in fixing the bug, since, at reporting time, the bug could already have a test case to be reproduced and then fixed. On the other hand, if the way in which a bug is activated and propagates to the interface is hard to reproduce, it could be difficult to obtain a test case for it. This would impact also the reported debugging time. We study the correlation of triggers with the time to fix a bug, computed as the time in which the report lifecycle is in the *modification* phase (detailed in Section 3.2). We apply hypothesis testing on different combinations. First, we test the *time to fix* difference between WL *vs.* USR/ENV bugs, to see if the type of trigger(s) has an impact. The Wilcoxon rank-sum test (Siegel and Castellan, 1988) is used to assess whether one of two independent samples tends to attain larger values; the null hypothesis that the *time to fix* of both categories is sampled from the same distribution. Results are in Table 5.[9] The null hypothesis of no significant difference cannot be rejected (with 95% of confidence). Thus, although USR/ENV bugs actually take more time to be fixed, we cannot reject the hypothesis that this difference occurred by chance.

Table 5
Impact of bug triggers on time to fix.

Project	*Median time to fix (st. dev.)*		*p-Value (adj.)*
	WL	**USR/ENV**	
MySQL	39.81 (534.87)	59.59 (278.69)	*0.3162*
Apache	19.00 (278.97)	27.34 (106.81)	*0.61*

In Table 6, we test the hypothesis that the number of triggers impacts the time to fix. Here, the Kruskal–Wallis test is adopted (Siegel and Castellan, 1988), being multiple levels involved. Results show again no significance at 95% of confidence, although the median time to fix is higher as the number of triggers increases. Finally, to take the type of trigger into account along with the number of triggers, we consider two combinations that can be assimilated to high and low complexity bug activation conditions: *Combination 1*: bugs with at most 2 triggers (≤ 2) and 0 USR/ENV triggers, as low complexity situations; *Combination 2*: bugs with at least 3 triggers (≥ 3) and at least one USR/ENV trigger. Table 7 shows that even in this case the time to fix is not impacted significantly. Results tell that:

Finding # 19: *we cannot state that there is a relation between the bug manifestation, expressed by triggers, and the time to fix a bug.* The

[9] As multiple comparisons are performed, we adopt the procedure of Benjamini and Hochberg (1995) to derive adjusted *p-Values*, to control the false rejection probability.

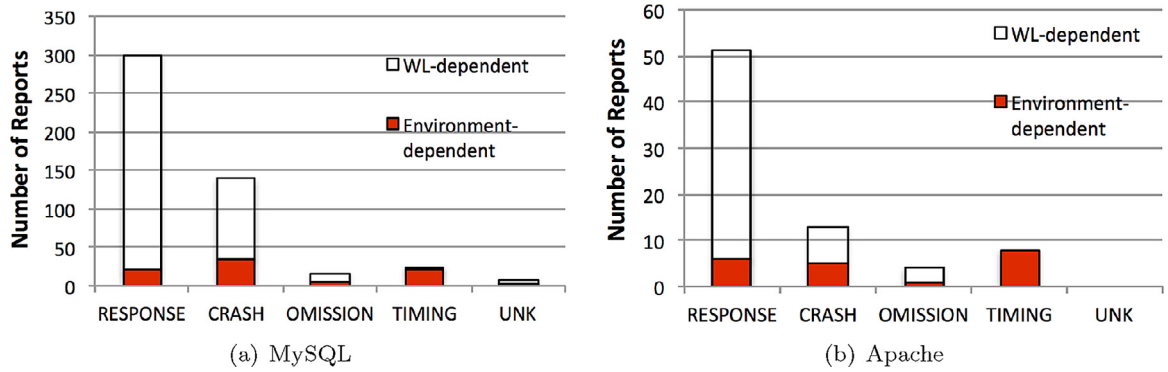


Fig. 10. Failure modes distribution.

Table 6
Impact of number of triggers on time to fix.

Project	*Mediantime to fix (st. dev.)*					*p-Value (adj.)*
#Triggers	**1**	**2**	**3**	**4**	**5**	
MySQL	32.97 (1147.82)	42.02 (253.71)	57.81 (242.32)	63.11 (287.47)	229.23 (263.37)	*0.1620*
Apache	5 (11.04)	18.41 (357.20)	57.57 (135.43)	15.49 (134.48)	66.25 (9.61)	*0.2754*

Table 7
Impact of trigger number and type combination on time to fix. Combination 1: reports with at most 2 triggers (≤ 2) and 0 environment triggers. Combination 2: reports with at least 3 triggers (≥ 3) and at least one environment trigger.

Project	*Median time to fix (st. dev.)*		*p-Value (adj.)*
	Combination 1	**Combination 2**	
MySQL	63.41 (233.30)	37.08 (803.72)	*0.056*
Apache	28.59 (113.75)	13.52 (341.79)	*0.1746*

time to fix a bug in the analyzed data was not impacted by the type of trigger, by the number of triggers, as well as by both factors together. This corroborates the hypothesis that (number and type of) triggers impact more on the way in which a bug is detected and exposed (a step typically done before the actual fixing work) than the fixing time itself. However, the high standard deviation of data suggests that more samples are needed to verify more accurately the influence of these factors in the future.

Implications: the type of occurred triggers is indeed a valuable feedback for the bug detection process, as observed in previous Sections. The number of triggers is also a suitable indication in this regard, since more triggers means more conditions together to detect a bug. Any unbalance in the patterns of Fig. 9 can reveal anomalies in the operational bug occurrence, and thus in future testing activities with respect to the operational profile. For instance, although in both cases two-trigger bugs are the majority, MySQL counts a high number of one-trigger bugs (almost 140); these are supposed to be easy cases to remove during testing, and call for a better basic testing. Oppositely, suppose to have a lot of multiple-triggers cases (i.e., with 4 or 5 triggers together), including many USR/ENV triggers; then, this would push engineers to invest either in more specialized testing tools or in runtime fault tolerance. As for debugging, different tools could be needed depending on the most frequent triggers, ranging from simple single-stepping executions to execution trace analysis through program instrumentation (e.g., for memory-related problems). A thorough characterization of bug detection and correction techniques for different trigger types would provide a feedback to understand where to improve, and, more in general, where to invest research and development efforts.

7. Impact analysis

Fig. 10 (a) and (b) shows the failure mode counts for MySQL and Apache.

Finding #20: *the manifestation of most of total bugs ended up in an incorrect response provided to the user (62.12%) or in a crash of the application (26.90%). The remaining bugs resulted in performance issues (5.84%) or omission failures (3.72%).*

The percentages for MySQL and Apache are quite close for RESPONSE (61.34% and 67.10%, respectively),OMISSION (3.47% and 5.26%), and TIMING (5.11% and 10.52%), while a larger difference is about CRASH failures, which are 28.4% in MySQL and 17.10% in Apache. Table 8 shows the contingency tables for bug type and failure mode. We aim at figuring out whether failure modes depend on the trigger type (USR/ENV or WL). The *Pearson chi-square test* (Agresti, 2007) is adopted, assessing the null hypothesis that two categorical variables are independent. The null hypothesis is clearly rejected at values greater than 99% for both projects; namely, the failure mode is influenced by the bug type. We conclude that:

Finding #21: *for the considered projects, the failure mode is affected by the type of bug manifestation classified as environment- or workload-dependent.*

USR/ENV bugs are observed to be consistently more related to crash (40.19%) and to timing (28.04%). Crash failures are tied to bugs activated with high non-determinism, which lead often to memory corruption problems; timing failures are mainly *late* timing, i.e., related to performance issues, due to erroneous management of resources (e.g., transient unavailability of resources, resource leaks, or more generally software aging (Huang et al., 1995)) whose propagation and perceived failure is “environment-dependent”. With respect to the transient behavior, failure modes counts are: of 35 transient failures in MySQL, 13 are crash failures, 10 are due to an incorrect response, 9 are performance issues, 3 are omission failures; in Apache, 3 were crash failures, 2 were incorrect responses and 1 was timing. “Crash” is the most commonly observed failure mode subject to transient behaviors. Finally, Table 9 provides the contingency tables for bug type and severity classified in *HIGH vs. LOW* by the reporter. We aim at figuring out if USR/ENV bugs are perceived to be more severe than WL ones. The null hypothesis is rejected for both projects, at values greater than 98%:

Table 8
Contingency tables for bug type and failure mode.

(a) MySQL (outcome = reject, adj. p-Value = < **0.0001**)			(b) Apache (outcome = reject, adj. p-Value = < **0.0001**)		
	WL	**ENV**		**WL**	**ENV**
Response	280 (93.33%)	20 (6.67%)	Response	45 (88.23%)	6 (11.77%)
Crash	104 (74.82%)	35 (25.18%)	Crash	5 (38.46%)	8 (61.54%)
Omission	12 (70.59%)	5 (29.41%)	Omission	3 (75.00%)	1 (25.00%)
Timing	3 (12.00%)	22 (88.00%)	Timing	0 (0.00%)	8 (100.00%)

Table 9
Contingency tables for bug type and severity assigned by the reporter.

(a) MySQL (outcome = reject, adj.p-Value = < **0.0001**)			(b) Apache (outcome = reject, adj. p-Value = **0.01752**)		
	WL	**ENV**		**WL**	**ENV**
HIGH	175 (78.13%)	49 (21.87%)	HIGH	14 (45.16%)	17 (54.83%)
LOW	114 (0.8%)	1 (0.2%)	LOW	5 (100%)	0 (0%)

Finding #22: *for the considered projects, the relative proportions of high and low severity bugs are influenced by the bug type, with the percentage of environment-dependent bugs being more relevant for the high severity class.* This can be explained by considering that USR/ENV bugs often involve factors as memory, or OS scheduling, whose failure is likely to lead to severe consequences.

Implications: bugs manifesting as workload-dependent cause more often as incorrect responses; bugs manifesting as environment-dependent cause relatively more crashes and performance issues than other failure types. Focusing more on one type of failure mode than another depends basically on the type of system, as their seriousness depends on how the system impacts the end user. For instance, in safety-critical systems, one cannot tolerate a so high percentage of crashes; consequently, countermeasures would be devoted to reduce the bugs with triggers more related to these types of failures (e.g., concurrency and memory-related problems). Instead, in a business-critical system, performance degradation issues might be more important: the typical phenomenon affecting performance of long-running systems is a gradual progressive degradation Grottke et al., 2006; Cotroneo et al., 2013, which can cause a greater user dissatisfaction. In such a case, long-running workload test methods (Bovenzi et al., 2011), possibly aided by dynamic analysis (Nethercote and Seward, 2007), can better serve the purpose.

8. Threats to validity

The validity of real world studies are naturally subject to limitations. We identify the following threats:

Selection of bug reports: we based our analysis exclusively on fixed bugs, since, for bugs that have not yet been fixed, the reports may contain inaccurate or incomplete information. On one hand, this allows relying on more stable information; on the other hand, results do not refer to non-closed bugs, and could be different if considering also reports still open. Closed bugs includes those bugs that got re-opened and at the time of the analysis are marked as "closed". Of course, at the time of the analysis, we cannot know which bugs will get re-opened in the future because of an incomplete fix. This means that some bugs might contain incorrect information that will be modified after a re-opening; hence, closed bugs not having a correct and final fix may impact the results.

*Manual inspection:*the manual inspection is conducted by examining several details of a bug report, including: reporter's description, forum discussion, attached test case and/or test data, patches applied for correcting the bug. Results were cross-checked to reduce possible misclassification. Nonetheless, as any paper where manual inspection is needed, we cannot exclude possible classification mistakes that can affect the results.

*Triggers definition:*the bug manifestation properties are defined in terms of triggers with respect to a system model defined in Section 3.1.1. Triggers are valid only with respect to that model; any system that cannot be described by the model could refer to other triggers. Although the model is very generic, the generality of triggers cannot be claimed; thus, care must be taken in studying other systems not falling in the outlined model.

*Triggers information correctness:*information provided by the reporter to reproduce the bug might be incorrect. However, for almost all MySQL reports, the developer, as first action, tries to reproduce the bug by running the steps or the test case described by the reporter. The discussion and any fixing attempt do not start until the bug is not reproduced by developers. In these cases, we are reasonably sure that the bug is reproduced. Whenever there is no information about the reproduction to determine the triggers, we filtered the bug out as NOT SUFFICIENT INFO. Instead, for those reports where we have enough information to decide about triggers but the reproduction was not explicitly described by the developer (e.g., most of Apache bug reports), we rely on the fact that the report was confirmed to be a bug, then fixed by a patch, and then verified (this gives us a certain confidence that it was reproduced): in these cases, the threats are in the process followed to reproduce the bug (not explicit as in MySQL), which might be different from the one described by the reporter (namely, using a different set of triggers). Similarly, it could happen that a bug duplicate of the analyzed one could have been activated by a different set of triggers (e.g., a simplified set of the considered one). The analysis therefore refers to one specific set of triggers for a given bug, and not to all potential sets of triggers. There could be further manifestations that will never be exposed; thus, the analysis will necessarily refer to "instances" of triggers manifestation that have been observed.

Fixing time: the time to fix a bug (used in the complexity analysis) is computed as the time in which the report is in the *modification* phase, namely from when the discussion starts until the bug is resolved but not yet tested. Although this better approximates the actual fixing time compared to the entire lifetime of the report (i.e., closed-opened difference), it does not reflect yet the actual time a developer works on bug fixing, especially if reporters misuses the tracking tool (e.g., opens the bug only when he has the fixing ready). Indeed, the actual start time of modification by developers can be earlier than the time when developers report bugs. Thus, results assume an average low impact of potential misuses of the tracking tool.

Severity: the indication of severity by the reporter may be unreliable. To mitigate this threat, we have simplified the analysis by considering *high vs. low* macro-categories, so as to reduce misclassification error, and excluded the default values that are likely to bias the results. Nonetheless, there might be still mistakes in the assignment

of severity between macro-classes, i.e., a HIGH severity bug marked as LOW, or vice-versa.

Temporal analysis: several factors may impact the temporal evolution of bugs. For instance: (i) a new major release can cause people to migrate to the new version and no longer use the previous one – thus reporting less bugs; (ii) the product usage patterns may vary with time, as usage behavior of early adopters can be different from late adopters (e.g., companies tend to adopt products later); (iii) bug reporting might decrease because of fixes of the release taking longer. While we do not have data about usage patterns and fixes of the release (which remain potential threats), we considered the occurrence of major and minor releases to figure out how occurrence trends are related to releases and have more accurate indications on time evolution. Analysis accounting for usage patterns and fixes of the release are left to future research.

External validity: we selected two well-known open source projects, both server-side applications written in C/C++ and widely used in their respective categories (e.g., DBMS and Web servers). However, findings of this study may not hold for other types of software applications with different features (e.g., different target, language, scale). Examined applications have specific characteristics, which impact the types of bugs reported on them. For instance, the occurrence of triggers may well depend on the variety of possible configurations and environments in which an application can be used; in a DBMS and a web server the variety of configurations and environments is much more limited and stable with respect to, for instance, mobile applications: this can indeed favor a predominance of WL bugs with respect to USR/ENV bugs. Thus, generalizing the findings to other system types requires further studies. Also, the examined dataset is limited to 666 reports. On one hand, the analysis regarded many aspects of the bug properties related to bug manifestation; on the other hand, the manual effort needed to study more bugs in such a way is an inherent limitation of the study. While studying more bugs could highlight different patterns of triggers, we expect that the output of our analyzes provides a valuable characterization of bugs in terms of manifestation process properties. In this sense, findings should be viewed as a framework to formulate substantiated hypotheses about bugs manifestation, to be confirmed or rejected by further studies, rather than as general findings.

9. Conclusion

This paper described the comprehensive analysis of a set of software bugs from the bug manifestation perspective. First, we defined the characteristics of the process of bug manifestation, which we captured in terms of those essential conditions (called *triggers*) that are necessary, in a bug-failure chain, to relate the cause (input, environment, and bug) to the effect (failure). The resulting set of triggers, covering both simple and complex cases, is analyzed against 666 bugs from two large and well-known software applications: MySQL and the Apache web server. Occurrence patterns of different types of bugs surfacing are derived, and several findings are reported about the most relevant trigger types, their occurrence times, their relation with the bug activation "complexity" and fixing time as well as with user-perceived failure impact.

There are several findings from our research, with many implications in terms of bug detection (e.g., testing, static and dynamic analysis) and fault tolerance strategies. The feedback provided by trigger analysis supports engineers to understand what to improve and where to invest efforts for quality assurance of next releases. For instance, consider finding #16 and #17; they reveal that too many bugs (in terms of percentage) with only one or two workload triggers (about 62%) escaped testing: these should have been removed prior to release, and call for better basic functional testing in subsequent releases. Similarly, findings #8 and #9 highlight a prevalence of concurrency and memory problems in environment-dependent bugs, demanding a better implementation of certain type of V&V or runtime fault tolerance actions. Along this line, the paper outlined more specific actions that could be implemented by observing the (relative) occurrence patterns of each trigger type. Of course, the final choices of V&V or fault tolerance solutions will depend on the context (e.g., on quality requirements, on available time for completing the V&V phase, on cost of the actions to implement *vs.* available budget, on development and organizational process, on testers skill). In general, from findings of this study, it is clear that characterizing the way in which a bug surfaces and its relation with the environment are crucial to the effectiveness of countermeasures to be used via V&V and fault tolerance.

There is a lot of future research in this field. We outline some directions we wish to pursue: (i) further empirical characterization to confirm and/or refine the features of the bug manifestation to a larger extent; (ii) analysis of the relation between bug manifestation and development process activities for process improvement purposes; (iii) definition of environment-aware strategies for V&V and fault tolerance driven by trigger patterns. More generally, our effort will be mainly devoted to explore the increasing role of the environment in bug manifestation, since we believe that software-based systems and their environment are, from the failure occurrence perspective, no longer separable from each other.

Acknowledgments

Pietrantuono's research was supported also by the PRIN Project "TENACE" (no. 20103P34XC) funded by the Italian Ministry of Education, University and Research. Trivedi's research was supported in part by NASA under Grant number NNX14AL90G.

References

Agresti, A., 2007. An Introduction to Categorical Data Analysis, second ed. Wiley-Interscience.

Antoniol, G., Ayari, K., Penta, M.D., Khomh, F., Gueheneuc, Y.-G., 2008. Is it a bug or an enhancement? A text-based approach to classify change requests. In: Proceedings of the ACM Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, pp. 23:304–23:318.

Benjamini, Y., Hochberg, Y., 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. J. R. Stat. Soc. Ser. B (Method.) 57 (1), 289–300.

Bettenburg, N., Just, S., Schröter, A., Weiß, C., Premraj, R., Zimmermann, T., 2007. Quality of bug reports in eclipse. In: Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange. ACM, New York, NY, USA, pp. 21–25. http://dx.doi.org/10.1145/1328279.1328284.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., Zimmermann, T., 2008. What makes a good bug report? In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. SIGSOFT '08/FSE-16. ACM, New York, NY, USA, pp. 308–318. http://dx.doi.org/10.1145/1453101.1453146.

Bovenzi, A., Cotroneo, D., Pietrantuono, R., Russo, S., 2011. Workload characterization for software aging analysis. In: Proceedings of the IEEE 22nd International Symposium on Software Reliability Engineering. ISSRE, 2011, pp. 240–249. http://dx.doi.org/10.1109/ISSRE.2011.18.

Bovenzi, A., Cotroneo, D., Pietrantuono, R., Russo, S., 2012. On the aging effects due to concurrency bugs: a case study on mysql. In: Proceedings of the IEEE 23rd International Symposium on Software Reliability Engineering. ISSRE, pp. 211–220.

Bovet, D., Cesati, M., 2005. Understanding the Linux Kernel. O'Reilly.

Carrozza, G., Pietrantuono, R., Russo, S., 2014. Defect analysis in mission-critical software systems: a detailed investigation. J. Softw. Evol. Proc. 27, 22–49.

Cavezza, D., Pietrantuono, R., Russo, S., Alonso, J., Trivedi, K., 2014. Reproducibility of environment-dependent software failures: an experience report. In: Proceedings of the IEEE 25th International Symposium on Software Reliability Engineering. ISSRE, pp. 267–276. http://dx.doi.org/10.1109/ISSRE.2014.19.

Chandra, S., Chen, P.M., 2000. Whither generic recovery from application faults? A fault study using open-source software. In: Proceedings of International Conference on Dependable Systems Networks, pp. 97–106.

Chillarege, R., 2011. Understanding Bohr-Mandel Bugs through ODC triggers and a case study with empirical estimations of their field proportion. In: Proceedings of the Workshop on Software Aging and Rejuvenation.

Chillarege, R., Bassin, K.A., 1995. Software triggers as a function of time – ODC on field faults. In: Proceedings of the Fifth IFIP Working Conference on Dependable Computing for Critical Applications. DCCA-5.

Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.-Y., 1992. Orthogonal defect classification – a concept for in-process measurements. IEEE Trans. Softw. Eng. 18 (11), 943–956.

Chillarege, R., Prasad, K.R., 2002. Test and development process retrospective – a case study using ODC triggers. In: Proceedings of the IEEE International Performance and Dependability Symposium. IPDS.
Cinque, M., Gaiani, C., De Stradis, D., Pecchia, A., Pietrantuono, R., Russo, S., 2014. On the impact of debugging on software reliability growth analysis: a case study. In: Murgante, B., Misra, S., Rocha, A., Torre, C., Rocha, J., Falcão, M., Taniar, D., Apduhan, B., Gervasi, O. (Eds.), Proceedings of International Conference on Computational Science and Its Applications. ICCSA 2014, Vol. 8583 of Lecture Notes in Computer Science. Springer International Publishing, pp. 461–475.
Cotroneo, D., Grottke, M., Natella, R., Pietrantuono, R., Trivedi, K., 2013a. Fault triggers in open-source software: an experience report. In: Proceedings of the IEEE 24th International Symposium on Software Reliability Engineering. ISSRE, pp. 178–187.
Cotroneo, D., Natella, R., Pietrantuono, R., 2010. Is software aging related to software metrics? In: Proceedings of the IEEE Second International Workshop on Software Aging and Rejuvenation. WoSAR 2010, pp. 1–6.
Cotroneo, D., Natella, R., Pietrantuono, R., 2013b. Predicting aging-related bugs using software complexity metrics. Perform. Eval. 70 (3), 163–178.
Cotroneo, D., Orlando, S., Pietrantuono, R., Russo, S., 2013. A measurement-based ageing analysis of the JVM. Softw. Test. Verif. Reliab. 23, 199–239.
Cotroneo, D., Pietrantuono, R., Russo, S., 2013c. A learning-based method for combining testing techniques. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13. IEEE Press, Piscataway, NJ, USA, pp. 142–151. http://dl.acm.org/citation.cfm?id=2486788.2486808
Cotroneo, D., Pietrantuono, R., Russo, S., 2013d. Testing techniques selection based on ODC fault types and software metrics. J. Syst. Softw. 86 (6), 1613–1637. http://dx.doi.org/http://dx.doi.org/10.1016/j.jss.2013.02.020. http://www.sciencedirect.com/science/article/pii/S0164121213000319
Cristian, F., 1991a. Basic concepts and issues in fault-tolerant distributed systems. In: Proceedings of the International Workshop on Operating Systems of the 90s and Beyond. Springer-Verlag, London, UK, UK, pp. 119–149. http://dl.acm.org/citation.cfm?id=647368.723610
Cristian, F., 1991b. Understanding fault-tolerant distributed systems. Commun. ACM 34 (2), 56–78. http://dx.doi.org/10.1145/102792.102801.
Grady, R., 1992. Practical Software Metrics for Project Management and Process Improvement. Prentice Hall.
Gray, J., 1985. Why do Computers Stop and What Can be Done About it?. Techinical report 85.7. Tandem Computers.
Grottke, M., Li, L., Vaidyanathan, K., Trivedi, K., 2006. Analysis of software aging in a web server. IEEE Trans. Reliability 55 (3), 411–420.
Grottke, M., Nikora, A.P., Trivedi, K.S., 2010. An empirical investigation of fault types in space mission system software. In: Proceedings of the International Conference on Dependable Systems and Networks, pp. 447–456.
Grottke, M., Trivedi, K.S., 2005. Software faults, software aging and software rejuvenation. J. Rel. Eng. Ass. Jpn. 27 (7), 425–438.
Grottke, M., Trivedi, K.S., 2007. Fighting bugs: remove, retry, replicate, and rejuvenate. IEEE Comput. 40 (2), 107–109.
Herzig, K., Just, S., Zeller, A., 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the International Conference on Software Engineering, pp. 392–401.
Huang, Y., Kintala, C., Kolettis, N., Fulton, N., 1995. Software rejuvenation: analysis, module and applications. In: Proceedings of the International Symposium on Fault-Tolerant Computing, pp. 381–390.
I. S.1044-2009, Standard Classification for Software Anomalies2010.
Ihara, A., Ohira, M., Matsumoto, K., 2009. An analysis method for improving a bug modification process in open source software development. In: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops, pp. 135–144.
Lamkanfi, A., Demeyer, S., Giger, E., Goethals, B., 2010. Predicting the severity of a reported bug. In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories. MSR, 2010, pp. 1–10. http://dx.doi.org/10.1109/MSR.2010.5463284.
Lamkanfi, A., Demeyer, S., Soetens, Q.D., Verdonck, T., 2011. Comparing mining algorithms for predicting the severity of a reported bug. In: Proceedings of the 15th European Conference on Software Maintenance and Reengineering. CSMR '11. IEEE Computer Society, Washington, DC, USA, pp. 249–258. http://dx.doi.org/10.1109/CSMR.2011.31.
Lee, I., Iyer, R., 1995. Software dependability in the Tandem GUARDIAN system. IEEE Trans. Softw. Eng. 21 (5), 455–467.
Lee, I., Iyer, R.K., 1993. Faults, symptoms and software fault tolerance in the Tandem GUARDIAN90 operating system. In: Proceedings of the International Symposium on Fault Tolerant Computing, pp. 20–29.
Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C., 2006. Have things changed now? An empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, pp. 25–33.
Love, R., 2010. Linux Kernel Development, third ed. Addison-Wesley.
Lu, S., Park, S., Seo, E., Zhou, Y., 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 329–339.
Menzies, T., Marcus, A., 2008. Automated severity assessment of software defect reports. In: Proceedings of the IEEE International Conference on Software Maintenance. ICSM 2008, pp. 346–355. http://dx.doi.org/10.1109/ICSM.2008.4658083.
Mockus, A., Fielding, R.T., Herbsleb, J.D., 2002. Two case studies of open source software development: Apache and Mozilla. ACM Trans. Softw. Eng. Methodol. 11 (3), 309–346.
Nethercote, N., Seward, J., 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan Not. 42 (6), 89–100.
Nistor, A., Jiang, T., Tan, L., 2013. Discovering, reporting, and fixing performance bugs. In: Proceedings of the 10th Working Conference on Mining Software Repositories. MSR '13. IEEE Press, Piscataway, NJ, USA, pp. 237–246. http://dl.acm.org/citation.cfm?id=2487085.2487134
Sahoo, S.K., Criswell, J., Adve, V., 2010. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In: Proceedings of the International Conference on Software Engineering, pp. 485–494.
Siegel, S., Castellan, N., 1988. Nonparametric Statistics for the Behavioral Sciences, second ed. McGraw–Hill, Inc.
Sullivan, M., Chillarege, R., 1991. Software defects and their impact on system availability-a study of field failures in operating systems. In: Proceedings of the International Symposium on Fault-Tolerant Computing, pp. 2–9. http://dx.doi.org/10.1109/FTCS.1991.146625.
Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C., 2013. Bug characteristics in open source software. Empir. Softw. Eng. 19 (6), 1665–1705.
Trivedi, K.S., Mansharamani, R., Kim, D.S., Grottke, M., Nambiar, M., 2011. Recovery from failures due to mandelbugs in it systems. In: Proceedings of the International Symposium on Pacific Rim International Symposium on Dependable Computing, pp. 224–233.
Wang, D., Wang, Q., Yang, Y., Li, Q., Wang, H., Yuan, F., 2011. Is it really a defect? An empirical study on measuring and improving the process of software defect reporting. In: Proceedings of the International Symposium on Empirical Software Engineering and Measurement, pp. 434–443.
Zhang, F., Khomh, F., Zou, Y., Hassan, A., 2012. An empirical study on factors impacting bug fixing time. In: Proceedings of the 19th Working Conference on Reverse Engineering. WCRE.

Prof. Domenico Cotroneo, Ph.D., graduated cum laude in Computer Engineering at Federico II University of Naples in 1998. From 1998 to 2001 he was Ph.D. student in Computer Engineering, and became Ph.D. in 2001. From 2001 to 2002 he was research fellow at CINI, investigating dependability issues of middleware platforms. From 2004 to 2011 he was an Assistant Professor at Federico II University of Naples. Since 2012 he is an Associate professor at the same university. His main interests include dependability aspects of middleware architectures, dependability analysis of mobile computing infrastructures, and field-based measurements techniques.

Roberto Pietrantuono, Ph.D., IEEE Member, received the B.S. and the M.S. degrees in computer engineering in 2003 and 2006, respectively, from the Federico II University of Naples, Italy. He received the Ph.D. degree in computer and automation engineering from the same university in 2009. He is currently a postdoc researcher at the same university. His main research interests are in the area of software engineering, particularly in the software verification of critical systems and software reliability modeling and evaluation. In this context, he collaborated in several national and international projects. He published in the field of software testing, dependability assessment, and reliability modeling and analysis.

Stefano Russo, Ph.D., IEEE Member, is Professor and Deputy Dean at the Department of Computer and Systems Engineering (DIS) of the Federico II University of Naples, Italy, where he leads the Distributed and Mobile Computing Group. He was Assistant Professor at DIS from 1994 to 1998, and Associate Professor from 1998 to 2002. He is Chair of the Curriculum in Computer Engineering, and Director of the National Laboratory of CINI (National Inter-universities Consortium for Informatics) in Naples. He teaches the courses of Advanced Programming and Distributed Systems. His current scientific interests are in the following areas: (i) distributed software engineering; (ii) middleware technologies; (iii) mobile computing.

Kishor S. Trivedi holds the Hudson Chair in the Department of Electrical and Computer Engineering at Duke University, Durham, NC. He has been on the Duke faculty since 1975. He is the author of a well known text entitled, Probability and Statistics with Reliability, Queuing and Computer Science Applications, originally published by Prentice-üHall; a thoroughly revised second edition (including its Indian edition and a Chinese translation) of this book has been published by John Wiley. He has also published two other books entitled, Performance and Reliability Analysis of Computer Systems, published by Kluwer Academic Publishers and Queueing Networks and Markov Chains, John Wiley. He is a Fellow of the Institute of Electrical and Electronics Engineers. He has published over 500 articles and has supervised 46 Ph.D. dissertations. He is the recipient of IEEE Computer Society Technical Achievement Award for his research on Software Aging and Rejuvenation. He works closely with industry in carrying our reliability/availability analysis, providing short courses on reliability, availability, performability modeling and in the development and dissemination of software packages such as SHARPE and SPNP.