Araújo *et al. Journal of Software Engineering Research and Development* (2016) 4:5
DOI 10.1186/s40411-016-0031-8

Journal of Software Engineering Research and Development

RESEARCH Open Access

Correlating automatic static analysis and mutation testing: towards incremental strategies

Cláudio A. Araújo[1†], Marcio E. Delamaro[2], José C. Maldonado[2†] and Auri M. R. Vincenzi[3*†]

*Correspondence: auri@dc.ufscar.br
†Equal contributors
[3]Departamento de Computação, UFSCar, Rod. Washington Luís, Km 235, 13565-905 São Carlos, SP, Brazil
Full list of author information is available at the end of the article

Abstract

Background: Traditionally, mutation testing is used as test set generation and/or test evaluation criteria once it is considered a good fault model. This paper uses mutation testing for evaluating an automated static analyzer. Since static analyzers, in general, report a substantial number of false positive warnings, the intention of this study is to define a prioritization approach of static warnings based on their correspondence with mutations. On the other hand, knowing that Mutation Test has a high application cost, another possibility is to try to identify mutations of some specific mutation operators, which an automatic static analyzer is not adequate to detect. Therefore, this information can be used to prioritize the order of incrementally applying mutation operators considering, firstly, those with no correspondence with static warnings. In both cases, contributing to the establishment of incremental strategies on using automatic static analysis or mutation testing or even a combination of them.

Methods: We used mutation operators as a fault model to evaluate the direct correspondence between mutations and static warnings. The main advantage of using mutation operators is that they generate a large number of programs containing faults of different types, which can be used to decide the ones most probable to be detected by static analyzers.

Results: We provide evidences on the correspondence between mutations and some types of static warnings. The results obtained for a set of 19 open-source programs indicate that: 1) static warnings may be prioritized based on their correspondence level with mutations; 2) specific set of mutation operators and their mutations may be prioritized based on their correspondence level with warnings.

Conclusion: It is possible to provide an incremental testing strategy aiming at reducing the cost of both static analysis and mutation testing using the correspondence information between these activities/artifacts.

Keywords: Software testing, Warnings, Mutants, Static analysis, Mutation testing, Static analyzer evaluation

1 Introduction

In software development environments, static analysis tools are used to support the verification of violations of code standards. Examples of violations detected by these tools are the access to invalid objects (uninitialized), the usage of deprecated methods, the encoding in disagreement with a determined established standard, among others.

In these environments, it is also common the existence of maintenance and development activities to find (and correct) software faults. Software faults are introduced in the source code due to mistakes made by developers. A wrong command or instruction in the source code are examples of software faults (IEEE 1990).

Mutation Testing is a very effective testing criterion once the mutation operators or a subset of them, responsible to perform the syntactic changes into the original program for mutant generation, represent a plausible fault model. In general, such a fault model is used as test set generation and/or test evaluation criteria which makes Mutation Testing a good tool for experimentation (Andrews et al. 2005).

Besides effective, Mutation Testing has some drawbacks, mainly related to the high number of generated mutants. A way to reduce its cost is decreasing the number of mutation operators we need to use. This is called selective mutation and there are several alternatives to identify this subset of mutation operators (Acree et al. 1979; Mathur 1991; Mresa and Bottaci 1999; Offutt et al. 1993).

Automatic Static Analysis does not require software execution like Mutation Testing. On the other hand, it uses a set of well defined bug patterns aiming at, by static analysing the source code, issuing warnings related to some possible source code line problem. The main disadvantage of automatic static analyzers is the high number of warnings which do not correspond to a fault (false positive) but demands time to be analysed.

Besides the concern in usage of static analysis tools (Ayewah et al. 2007b; Hovemeyer and Pugh 2004; Louridas 2006), there is no agreement of their real benefits since it is not clear the relation between static warning and faults (Ayewah et al. 2007a).

Inspired by the work of Araújo Filho et al. (2010) and Couto et al. (2013), and trying to overcome the problem they faced of evaluating the direct correspondence between warnings and faults due to the small number of real faults, we revisited their work evaluating the correspondence between static warnings and mutations. We decided to use mutation testing due to the following reasons: 1) it is considered a good fault model for experimentation and has been successfully used for test set evaluations (Andrews et al. 2005); 2) mutants are generated by mutation operators which can be seen as fault categories so we can try to correlate warnings and specific types of faults; 3) it allows to increase fault concentration per Kilo Lines of Code (KLOC) by summing up the number of mutants derived from each source code line; and 4) it eases experimentation with a large number of software products. These reasons help overcoming limitations of previous works on these subjects. Moreover, from this study, we intend to define a strategy for static warnings prioritization based on their correspondence with mutations. The results can be used to evolve either static analyzers or mutation testing, or both. The former in the sense it will be possible to compare different static analyzers against the same fault model to decide which kind of mutations they are really adequate to detect. This information can be used to prioritize the warnings resolution starting from the ones more correlated with some mutation operator and also to guide the evolution of static analyzers by creating additional static verification rules to detect uncovered mutations. The later by avoiding the generation of mutants for that mutation operators which static analyzers are adequate to detect their faults statically.

In this paper, we report an approach that makes use of information obtained through the application of FindBugs (Hovemeyer and Pugh 2004) on detecting mutations generated by μJava (Ma et al. 2005). The objective is to identify correspondence between bug kinds with mutation operators. Based on this information we can establish incremental strategy for using bug kinds and mutation operators on an incremental way, aiming at mitigating the problem of both automated static analysis and mutation testing.

Given this scenario, the research questions of interest are:

- **Question Q_1 (direct correspondence)**: is there any correspondence between static location of warnings and program elements with mutations' concentration?
- **Question Q_2 (direct correspondence at source code line level)**: the analysis is performed at a lower level considering each source code line individually, i.e., is there any correspondence between static location of warnings and program elements with mutations' concentration at the source code line level?

Observe that a positive answer to these questions suggests that FindBugs is adequate to detect specific types of mutations. By analyzing which static warnings are better to detect specific types of mutations one may prioritize warnings to avoid earlier analysis of false positives. Moreover, we can also know which mutations are or are not detectable by static analysis tools.

We can summarize the contributions of this work as:

- the identification of existence of direct correspondence between warnings and some mutation operators;
- the identification of specific mutation operators which FindBugs is more prone to detect;
- the identification of specific mutations operators which FindBugs is not adequate to detect;
- the establishment of prioritization strategies for incremental use of bug kinds based on their correspondence level with mutations at the line level; and
- the establishment of prioritization strategies for incremental use of mutation operators based on their inverse correspondence with static warnings at the line level.

The remainder of this paper is organized as follows: Section 2 presents basic information with respect to static analysis and mutation testing. Section 3 defines Direct Correspondence per Line (DCL) and describes the experimental study and the data collection process. Section 4.1 presents how DCL is used on the establishment of an incremental strategy for applying FindBugs bug kinds. Section 4.2 performs the same analysis but considering how DCL is used on the establishment of an incremental strategy for applying μJava mutation operators. Section 5 illustrates how the incremental testing strategies can be employed to reduce the cost of static analysis and mutation testing. Section 6 describes the lessons learned and threats to validity of this study. Section 7 describes related works and the main points and contributions of this work. In Section 8, we draw the conclusions and present future work. Finally, Appendix A provides complementary information about μJava mutation operators and FindBugs bug kinds.

2 Background

Mutation Testing (MT) enables the generation of a high number of versions of a given determined system. This generation is performed by small syntactic changes which are done in the original system by mutation operators that simulate the mistakes more commonly committed by developers (DeMillo et al. 1978). To each change done by a mutation operator a new version of the system, called mutant, is generated. From a theoretical perspective, each mutant represents a possible fault that could be present in the original system (Copeland 2004).

There are several mutation tools available for Java programs (Coles 2015; Ferrari et al. 2011; Just et al. 2011; Ma et al. 2005). We are using the set of mutation operators implemented by μJava system which supports MT for Java programs (Ma et al. 2005). It creates object-oriented mutants for Java according to 47 mutation operators specialized to object-oriented faults: 19 Traditional Operators responsible to model faults at method level (Ma and Offutt 2005), and 28 Class Operators, responsible to model faults at class level (Offutt et al. 2006). Besides the advantage of having a well defined set of program faults, generated by mutation operators, is that the faults introduced by the operators are not detectable by Eclipse IDE as the injected faults used in the work of (Daimi et al. 2013).

Each μJava mutation operator has an acronym for identification. The first letter of this acronym determine the language features the operator is related to. For instance, method level operator AORB stands for "Arithmetic Operator Replacement (Binary)" and is one of the operators of Arithmetic (A) group. The reduced number of method operators implemented by μJava is due to the selective approach adopted to create such a set of mutants (Offutt et al. 1996). Appendix A has additional information about μJava mutation operators set.

An automated static analyzer is a tool that reads the source code of a given system and issues a set of warnings based on rules which look for deviations with respect to a given code standard. In general, warnings are issued with respect to a given source code line in which the tool detects any possible fault with respect to its supported rules.

Automated static analysis vocabulary includes the following terms: false positives, true positives and false negatives. A false positive occurs when a tool alerts to the presence of a non-existent fault. A false negative occurs when a fault exists, but it is not detected due to the fact that static analysis tools are not perfectly accurate and may not detect all faults. Finally, a true positive occurs when a tool produces a warning to indicate the presence of a real fault in the system under analysis.

There are several static analysis tools available for different programming languages (Burn 2014; Copeland 2005; Daimi et al. 2013; Evans and Larochelle 2002; Hovemeyer and Pugh 2004; Microsoft 2014; Pohl 2001). We use FindBugs (Hovemeyer and Pugh 2004) in this study for two reasons. First, because the same tool was used by Araújo Filho et al. (2010) and Couto et al. (2013) which inspired this work. They used FindBugs it to evaluate the correspondence between warnings and real faults. In our work, the actual faults were replaced by mutants. To make feasible future comparisons, we use the same static analysis tool. Second, several experiments involving static analysis uses FindBugs, and Tomas et al. (2013) pointed out that its false positive rate is less than 50 %.

FindBugs is one of the most popular static analysis tools and is widely used in Java community. It implements a set of bug detectors for a variety of common bug patterns. According to the structure of FindBugs, each bug category includes many bug kinds and each bug kind consists of several bug patterns. Figure 1 shows a sample structure of FindBugs categories, subcategories, patterns and bug patterns. Bug patterns in FindBugs are divided into categories: Bad Practice, Correctness, Malicious code vulnerability, Multithreaded correctness, Internationalization, Performance, Security, and Dodgy.

In Fig. 1, bug kinds, like BC[1], NP[2], DLS[3], and DMI[4], belong to correctness category. And the bug kind BC (the abbreviation for bad casts of object references) contains four bug patterns: Impossible cast, Impossible downcast, Impossible downcast of toArray() result, and instanceof will always return false.

3 Experimental study

3.1 Definitions

Let $S = \{s_1, s_2, \ldots, s_t\}$ be a set of t systems and $s_x \in S$ a system under analysis. s_x is composed by a set of source code files $F = \{f_1, f_2, \ldots, f_n\}$ where n is the number of source files of s_x. Consider $f_j \in F$ and Mf_j a set of mutants generated from f_j by applying a set of mutation operators MO. Therefore, $Ms_x = \{Mf_1 \cup Mf_2 \cup \ldots \cup Mf_n\}$ is the set of all mutants generated from s_x.

Consider $m_i o_k f_j \in Mf_j$ the i-th mutant of operator $o_k \in MO$ on the file f_j. f_j and $m_i o_k f_j$ differ from each other at least on some line of source code. Let Wf_j be the resultant set of warnings of applying FindBugs on f_j, and let $Wm_i o_k f_j$ be the resultant set of warnings of applying FindBugs on $m_i o_k f_j$.

We illustrated situations that occur when a warning w_i is reported by FindBugs on the original file f_j (Wf_j) and/or on the mutant $m_i o_k f_j$ ($Wm_i o_k f_j$) in Figs. 2, 3 and 4. Consider w_1, w_2, and w_3 as warnings, $Wm_i o_k f_j$ the set of warnings generated in a specific mutated file f_j, and Wf_j the set of warnings generated in the original file f_j, according to Figs. 2, 3 and 4:

1. Case 1: $w_1 \in \{Wm_i o_k f_j \setminus Wf_j\}$ is a warning that was reported in the mutant $m_i o_k f_j$ and that was not reported in the original file f_j, considering the same line where mutation occurs. In other words, w_1 represents a true positive, since it indicates FindBugs only generate the warning due to the mutation;

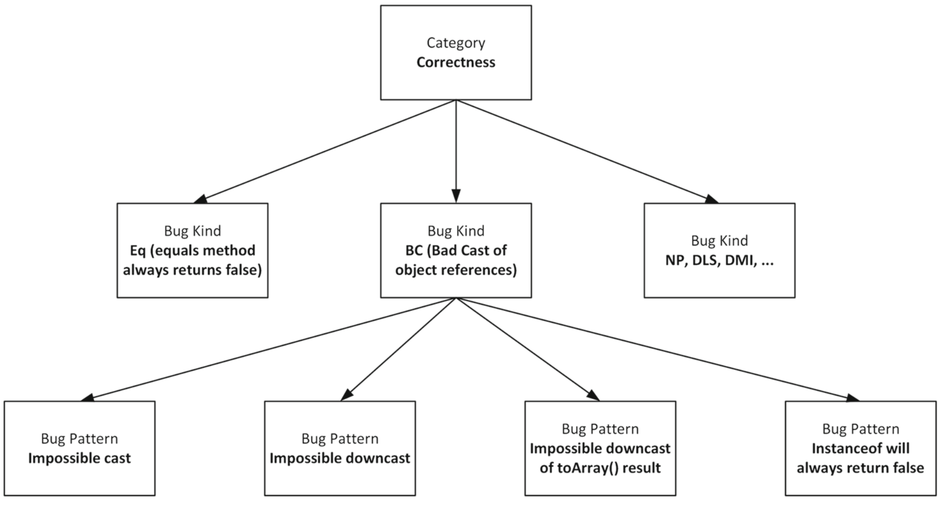
Category
Correctness
Bug Kind
Eq (equals method always returns false)
Bug Kind
BC (Bad Cast of object references)
Bug Kind
NP, DLS, DMI, ...
Bug Pattern
Impossible cast
Bug Pattern
Impossible downcast
Bug Pattern
Impossible downcast of toArray() result
Bug Pattern
Instanceof will always return false

Fig. 1 Partial structure of FindBugs: correctness category (Shen et al. 2011)

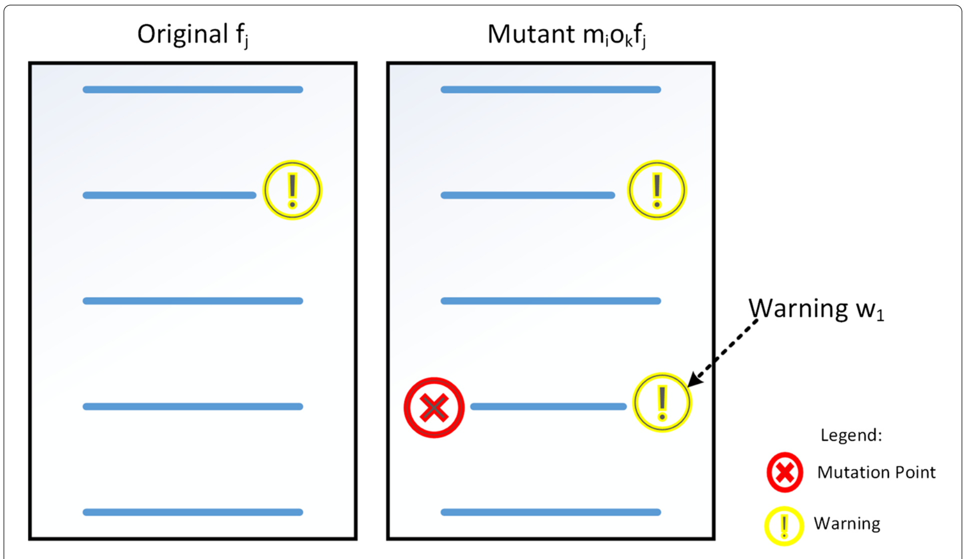

Fig. 2 Warning w_1 related in the mutant $m_i o_k f_j$ at the mutation point

2. Case 2: $w_2 \in \{Wm_i o_k f_j \cap Wf_j\}$ is a warning that was reported in mutant $m_i o_k f_j$, in the same line in which mutation happened, but w_2 was also reported in original file f_j. In this way, w_2 does not depend on the mutation;
3. Case 3: $w_3 \in \{Wf_j \setminus Wm_i o_k f_j\}$ is a warning which was reported in original file f_j, and that was not reported in mutant $m_i o_k f_j$, considering the line in which mutation occurs. In other words, the mutation actually corrects the warning reported in f_j.

Figures 2, 3 and 4 illustrate Cases 1, 2 and 3, respectively. As an example of Case 1, in the mutant shown in Code 2 (Additional file 1: Figure S2), w_1 was reported in the mutated line 4, and w_1 was not reported in correspondent line 4 of the original file (Code 1 - Additional file 1: Figure S1). As an example of Case 2, in the mutant shown in Code 4 (Additional file 1: Figure S4), w_2 is associated with both original (Code 3 - Additional file 1: Figure S3) and mutated files. Finally, as an example of Case 3, w_3 is reported in the original file (Code 5 - Additional file 1: Figure S5) but not on the mutant (Code 6 - Additional file 1: Figure S6).

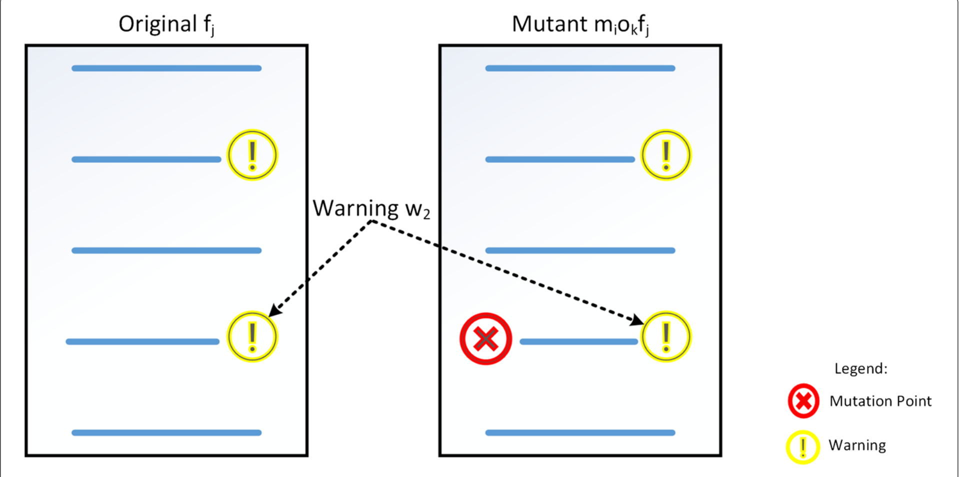

Fig. 3 Warning w_2 related in the mutant and original file, in the same line of the mutation

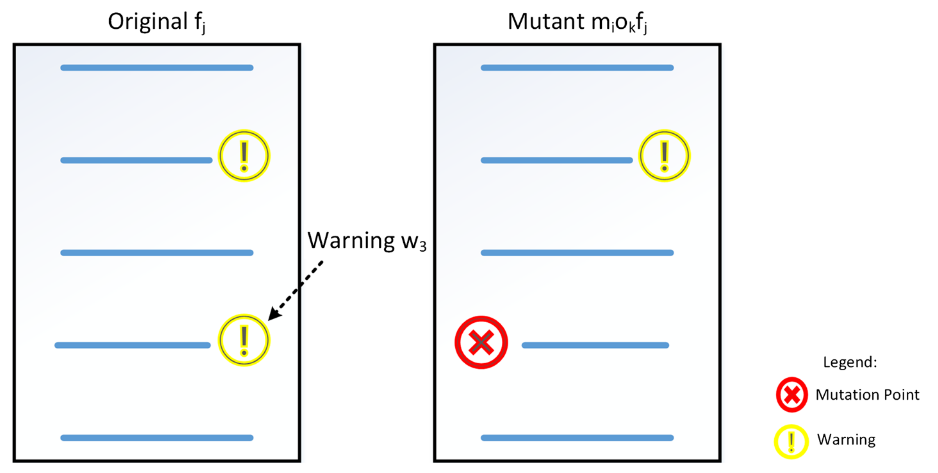

Fig. 4 Warning w_3 related in the original file but not in the mutant

In this study, we are specially interested in the situation represented by Case 1), where $w_1 \in Wm_io_kf_j$ and $w_1 \notin Wf_j$. Thus, if we can find a set of warnings adequate to detect specific kinds of mutations, we may prioritize the warning analysis starting from them since, in general, they are sensible to specific types of mutations, i.e., they probably are not **false positives**. Cases 2 and 3 are also interesting to be investigated but are out of the scope of this work.

All the analysis is performed based on the concept of **direct correspondence**. As defined by (Couto et al. 2013), the direct correspondence occurs when the warning and the fault are relatively close. Relatively close means at the method level, i.e., if warnings and faults exist in the same method they considered the existence of a correspondence between warning and fault.

In our work, we evaluated the direct correspondence at the source code line level. In this sense, relatively close means at the same source code line instead of at the same method. Therefore, we adopt a fine grain to identify the correspondence between mutations and warnings more precisely, considering the so called Direct Correlation per Line (DCL) defined below.

Once each mutant is generated by a specific mutation operator we can identify classes of mutations which static analyzers are or are not adequate to detect.

3.1.1 Direct Correspondence per Line (DCL)

To calculate the DCL of each mutation operator and of each warning category, the following functions are defined:

- $TW(w)$ = total number of warnings of the type w reported in all mutants.
- $DCL_A(w)$ = absolute number of warnings of the type w which are reported exactly in a mutation point, but the warning w does not exist on the same line in the original file.

$$DCL_R(w) = \begin{cases} DCL_A(w)/TW(w) & \text{if } TW(w) > 0 \\ 0 & \text{if } TW(w) = 0 \end{cases} \tag{1}$$

The aim of the functions $DCL_A(w)$ and $DCL_R(w)$ is to classify each bug kind according to its warnings w capability in detecting faults represented by mutants. The bug kinds with higher $DCL_R(w)$ rates should be prioritized in relation to the other bug kinds with lower

$DCL_R(w)$ rates, since warnings of bug kinds with higher $DCL_R(w)$ are more probable to be true-positives and should be analyzed first.

- $TM(o_k)$ = total number of generated mutants by operator o_k.
- $DCL_A(o_k)$ = absolute number of mutants of operator o_k with at least one warning of any type reported in the mutation point but the same warning does not exist on the same line in the original file.

$$DCL_R(o_k) = \begin{cases} DCL_A(o_k)/TM(o_k) & \text{if } TM(o_k) > 0 \\ 0 & \text{if } TM(o_k) = 0 \end{cases} \quad (2)$$

The aim of the functions $DCL_A(o_k)$ and $DCL_R(o_k)$ is to classify the types of faults represented by mutants of mutation operators according to the capability that these faults are detected by FindBugs. Operators with higher $DCL_R(o_k)$ correspondence rates represent types of faults that are easier detected by FindBugs. On the other hand, operators with lower $DCL_R(o_k)$ represent types of faults rarely detected by FindBugs, and may suggest new bug patterns that can be added to the static analyzer tool to improve its capability.

3.2 Experimental process

The process used to collect the data in the study is described in the steps below:

1. Let S be a set of systems, i.e, $S = \{s_1, s_2, \ldots, s_t\}$
2. For each system $s_x \in S, 1 \leq x \leq t$
3. For each source file f_j in s_x, $1 \leq j \leq n$, where n is the number of source files of s_x

    3.1 Execution of FindBugs in f_j and generation of a XML with the set of warnings Wf_j
    3.2 Execution of ParserXMLFindBugs to read the XML and to include the Wf_j on database (DB)
    3.3 Execution of μJava tool in f_j and generation of a set of mutants Mf_j
    3.4 For each mutant $m_i o_k f_j \in Mf_j$
    (a) Execution of FindBugs in $m_i o_k f_j$ and generation of a XML with the set of warnings $Wm_i o_k f_j$
    (b) Execution of ParserXMLFindBugs to read the XML and to include $Wm_i o_k f_j$ on DB
    (c) Execution of diff between f_j and $m_i o_k f_j$ to include lines number and textual difference on DB

4. Execution of SQL scripts to extract DCL data of S from DB

Figure 5 depicts the way data is collected and processed to support the experimental process. In case of the original system, observe that both FindBugs and μJava are executed considering the entire system. On the other hand, in case of mutants, once μJava generates mutants per file, we executed FindBugs only on each mutated file. In the later it is not necessary to run FindBugs on the entire system since it differs from the original only in the mutated file. In our cost analysis this point is clear.

FindBugs is executed on each file f_j of a system s_x. The result of FindBugs execution on f_j produces the set of warnings Wf_j, which is stores in a XML file. ParserXMLFindBugs is an application we developed to read the XML file and stores the collected information into a database. In the next step, the mutants of file f_j are generated through the application of

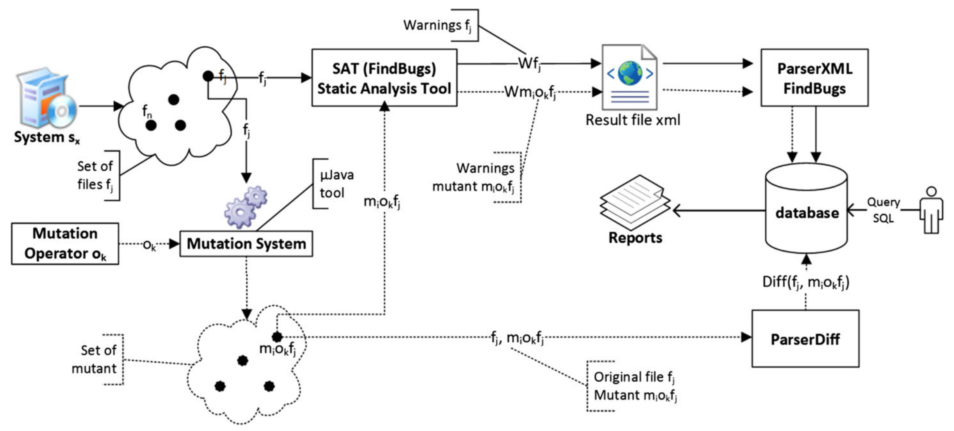

Fig. 5 Data collection and processing for supporting experimental process

all μJava mutation operators. The mutation operator o_k, applied to each file f_j, generates a set of mutants Mo_kf_j, which can be identified as $m_io_kf_j$ (the i-th mutant of o_k operator from file f_j). FindBugs is then executed on each mutant $m_io_kf_j$ generating a set of warnings $Wm_io_kf_j$ for such a mutant in a XML file. The parser ParserXMLFindBugs is used again to read the XML file and to store the warning information with respect to mutant $m_io_kf_j$ into the database. ParserDiff is another application we developed to calculate the syntactical difference between the original file f_j and the mutant $m_io_kf_j$, indicates on each source code line the mutation occurred, and stores such information into the database to identify the mutation produced. At the end of the process, SQL scripts are executed to extract matching information between mutations and static warning.

In this study, 19 systems were used ($t = 19$), as shown in Table 1. These systems, which are available at Apache Foundation, are a subset of systems used by (Couto et al. 2013). For instance, considering the entire system s_{11}, Open JPA, version 1.0.0, it has 102,682

Table 1 Set of systems used in the experimetal study

s_x	System	Version	LOC	Warnings (*W*)	Mutants (*M*)	*W*/KLOC	*M*/KLOC
s_1	Beehive	1.0	55,129	340	27,282	6.167	494.876
s_2	Cayenne	2.0.2	68,523	517	76,140	7.545	1111.160
s_3	Cfx	2.1	5344	51	3065	9.543	573.540
s_4	Ddlutils	2.0.0	13,892	133	15,294	9.574	1100.921
s_5	Hadoop Hbase	0.2.0	24,718	61	3827	2.468	154.826
s_6	Ibatis	2.3.0	10,781	135	13,102	12.522	1215.286
s_7	Ivy	2.1	26,893	135	21,570	5.020	802.067
s_8	James Server	2.2.0	18,599	204	17,431	10.968	937.201
s_9	Jdo	2.1	2748	27	1166	9.825	424.309
s_{10}	Lucene	2.9.4	38,152	215	28,316	5.635	742.189
s_{11}	Open JPA	1.0.0	102,682	359	65,529	3.496	638.174
...	...	...	...	...	...	...	...
s_{18}	Xalan J	2,7,2	93,335	76	12,280	0.814	131.569
s_{19}	Xmlbeans	2.0.0	46,927	103	12,572	2.195	267.905
Total			749,966	3709	493,522	–	–
Average			39,472	195	25,975	6.353	702.180
Median			38,152	135	17,431	5.635	638.174

LOC[5], to which FindBugs reported 359 warnings; μJava generated 65,529 mutants; which give an average of 3.5 warnings/KLOC[6]; and an average of 638.2 mutants/KLOC.

Considering all the 19 systems, they sum up 749,966 LOC with 3709 static warnings and 493,522 mutants. Averages of these proportions are 6.4 and 702.2 for warnings/KLOC and mutants/KLOC, respectively, and medians 5.6 and 638.2, respectively.

For each one of the selected systems, mutants were generated using all the 47 mutation operators supported by μJava tool (Ma et al. 2005). To collect the amount of warnings reported by FindBugs, it was executed in each one of the selected systems using its default configuration, i.e., employing all the warning categories available. To collect the warnings present in each one of the mutants, FindBugs was executed in each one of them with the same configuration. All data reported by FindBugs and the data from mutants were stored in a database to verify the direct correspondence by line.

Data of FindBugs execution on each mutant are presented in Table 2. Information related to the warnings (grouped by bug kinds) reported by FindBugs and the mutation operator used to generate each mutant is registered.

In Table 2, the following information is presented: the first column is a line identifier; column W (Warning) represents the types of warnings reported by FindBugs in the mutants; columns ISD, JTI, JTD, JSD, . . . , represent each one of the 47 μJava operators; column $TW(w)$ is the result of function $TW(w)$ defined in Section 3.1.1.

Table 2 Bug kinds versus mutation operator

#	*W*	Mutation operators					*TW*(*w*)
		ISD	JTI	JTD	JSD	⋯	
1	AT∗	0	0	0	1	⋯	5
2	BC	0	45	35	117	⋯	22,549
3	BIT	0	0	0	0	⋯	482
4	BSHIFT	0	0	0	0	⋯	6
5	Bx	2	392	125	368	⋯	107,060
6	CI∗	0	2	0	1	⋯	188
7	CN∗	0	186	166	13	⋯	10,350
8	DB	0	2	2	1	⋯	1275
9	DC∗	0	17	15	1	⋯	1634
10	DE	0	119	65	44	⋯	15,692
11	DLS	2	76	2656	178	⋯	46,286
12	DMI	0	45	15	9	⋯	1450
13	DP∗	0	55	42	10	⋯	3037
14	Dm	12	380	229	526	⋯	95,171
15	EC	0	0	0	0	⋯	729
⋯	⋯	⋯	⋯	⋯	⋯	⋱	⋯
80	UW∗	0	2	0	0	⋯	450
81	UrF	1	496	324	981	⋯	25,871
82	UuF∗	0	125	81	33	⋯	4775
83	UwF	1	2364	1,913	65	⋯	36,170
84	VO∗	0	10	4	0	⋯	1011
85	WMI∗	0	0	0	5	⋯	3072
86	Wa∗	0	5	3	9	⋯	1474
87	$TW(o_k)$	88	14,303	11,789	7882	⋯	980,533

It was reported warnings with respect to 86 different bug kinds, however Table 2 presents only a subset of these bug kinds. As an example of warning reported in mutants, there is warning of Bx bug kind (line 5) which was reported twice in mutants of ISD operator, 392 times in mutants of JTI operators, 125 times in mutants of JTD operator, 368 times in mutants of JSD operator, and so on. Considering the mutants of the 47 mutation operators, the total warning of Bx bug kind is 107,060 (column $TW(w)$ of line 5). Column ISD shows that in mutants of ISD operator, 88 warnings of different types were reported. Column JTI shows that in mutants of JTI operator 14,303 warnings were reported. The total of warning obtained in all the mutants is 980,533 (last column of last line).

From the data collected and presented in Table 2, we applied the functions $DCL_A(w), DCL_R(w), DCL_A(o_k)$, and $DCL_R(o_k)$ defined in Section 3.1.1, which produced Tables 3 and 5, employed in the definition of two incremental strategies, presented on Sections 4.1 and 4.2, respectively.

Table 3 DCL by Warning: $DCL_R(w)$

#	W	Mutation operators					$DCL_A(w)$	$TW(w)$	$DCL_R(w)$
		ISD	JTI	JTD	JSD	...			
1	QF	0	0	0	0	...	81	81	100.00 %
2	RE	0	0	0	0	...	13	13	100.00 %
3	UM	0	0	0	0	...	4	4	100.00 %
4	QBA	0	0	0	0	...	2	2	100.00 %
5	INT	0	0	0	0	...	1897	1963	96.64 %
6	BIT	0	0	0	0	...	444	482	92.12 %
7	UR	0	1855	0	0	...	2292	2593	88.39 %
8	UCF	0	0	0	0	...	7619	10,046	75.84 %
9	SA	2	3326	2793	0	...	6739	11,086	60.79 %
10	IL	35	0	0	183	...	776	1515	51.22 %
11	DLS	2	0	2629	0	...	17,785	46,286	38.42 %
12	SS	0	0	0	1814	...	1814	5367	33.80 %
13	BSHIFT	0	0	0	0	...	1	6	16.67 %
14	DB	0	2	1	0	...	148	1275	11.61 %
15	RpC	0	0	0	0	...	27	237	11.39 %
...	...	...	...	...	...	⋱	...	...	...
31	EC	0	0	0	0	...	1	729	0.14 %
32	IP	0	0	3	0	...	3	2290	0.13 %
33	DE	0	0	0	0	...	14	15,692	0.09 %
34	IS	0	0	0	0	...	18	21,114	0.09 %
35	SBSC	0	0	0	0	...	2	2882	0.07 %
36	BC	0	0	0	0	...	15	22,549	0.07 %
37	OS	0	0	0	0	...	5	10,174	0.05 %
38	REC	0	0	0	0	...	15	54,741	0.03 %
39	Dm	0	2	0	0	...	11	95,171	0.01 %
40	ES	0	1	0	0	...	1	10,152	0.01 %
41	Bx	0	2	1	0	...	7	107,060	0.01 %
42	EI2	0	0	0	0	...	1	27,574	0.00 %
43	IIO	0	0	0	0	...	1	29,517	0.00 %
44	Total	39	6362	5431	2249	...	50,768	980,533	5.18 %

4 Results and Discussion

Based on DCL we defined two incremental strategies. One, described in Section 4.1, intends to prioritize bug kinds. The other, described in Section 4.2, prioritizes mutation operators.

4.1 Using DCL for bug kinds prioritization

In this strategy we used the correspondence of warnings and mutants to identify bug kinds more probable to produce true positive warnings. Table 3 presents the direct correspondence by line between bug kinds and mutation. It contains information about $DCL_A(w)$ and $DCL_R(w)$ for all warnings (group by bug kinds) reported in mutants.

Table 3 presents the following information: the first column is a line identifier; column W (warnings) represents all bug kinds that have at least one warning related to some mutation point (at line level); columns ISD, JTI, JTD, JSD, . . . , represent each one of the 47 operators of μJava; columns $DCL_A(w)$ and $DCL_R(w)$ contain the values of each one of these functions defined in Section 3.1.1. Column $TW(w)$ is the same information presented in Table 2, replicated here to ease the analysis. In the first lines of Table 3 are the bug kinds that were more adequate to detect faults modeled by some μJava mutation operators, and, in the last lines, are the bug kinds with lower correspondence with mutations.

Note that in Table 3 it is presented only the 43 bug kinds which reported at least one warning in the same line of mutation (column $DCL_A(w) \geq 1$). In Table 2 are presented all the 86 bug kinds reported in the experiment. Therefore, there are 43 bug kinds that had $DCL_A(w) = 0$. For this bug kinds, no warning w is reported in any mutant, or, when reported, w is also reported in the original file in the same line the mutation occurs. The bug kinds with no warning in the mutation point are identified with symbol '$*$' in Table 2.

In general, 50,768 warnings were reported exclusively on the mutated lines, which gives a DCL of 5.18 % (50,768/980,533). As can be seen in Table 3, there are 4 bug kinds (lines 1 to 4) with $DCL_R(w) = 100.0$ % and there are 5 bug kinds (last lines) with $DCL_R(w) \leq$ 0.01 %.

As it is shown in Table 3, there is a variation in $DCL_R(w)$. Bug kinds with $DCL_R(w) >$ 10.0 % are presented in the first 15 rows of Table 3, of which 12 have $DCL_R(w) > 33$ %.

Bug kinds with $DCL_R(w) > 88$ % are: QF, RE, UM and QBA with 100.0 %, INT with 96.64 %, BIT with 92.12 %, and UR with 88.39 %, providing evidences that there is a direct correspondence between certain types of warnings and certain types of faults (mutation), motivating further work in this direction, for instance, analyzing the subcategories and priorities of the warnings.

Each bug kind of FindBugs has one or more bug patterns and each bug pattern has a priority (the greater the priority the greater the criticality of such a warning for FindBugs) and belongs to a specific category indicating which kind of fault such a warning subtype is related to (Hovemeyer and Pugh 2004). More information about FindBugs warnings is available on Appendix A. Table 4 presents additional information related to bug kinds with $DCL_R(w) > 88$ % (QF, RE, UM, QBA, INT, BIT, UR).

The top 6 bug kinds have 14 bug patterns classified on the following categories (see Table 4): CORRECTNESS(8), STYLE(4), PERFORMANCE(1) and BAD_PRACTICE(1).

Table 4 Details of the bug kinds

#	*W*	Subtype	Priority	Category	Description
1	QF	QF_QUESTIONABLE_FOR_LOOP	2	STYLE	Complicated, subtle or wrong increment in for-loop
2	RE	RE_BAD_SYNTAX_FOR_REGULAR_EXPRESSION	1	CORRECTNESS	Invalid syntax for regular expression
3	UM	UM_UNNECESSARY_MATH	3	PERFORMANCE	Method calls static Math class method on a constant value
4	QBA	QBA_QUESTIONABLE_BOOLEAN_ASSIGNMENT	1	CORRECTNESS	Method assigns boolean literal in boolean expression
5	INT	INT_VACUOUS_COMPARISON	2	STYLE	Vacuous comparison of integer value
6	INT	INT_VACUOUS_BIT_OPERATION	2	STYLE	Vacuous bit mask operation on integer value
7	INT	INT_BAD_REM_BY_1	1	STYLE	Integer remainder modulo 1
8	INT	INT_BAD_COMPARISON_WITH_SIGNED_BYTE	3	CORRECTNESS	Bad comparison of signed byte
9	INT	INT_BAD_COMPARISON_WITH_NONNEGATIVE_VALUE	1	CORRECTNESS	Bad comparison of nonnegative value with negative constant
10	BIT	BIT_ADD_OF_SIGNED_BYTE	2	CORRECTNESS	Bitwise add of signed byte value
11	BIT	BIT_SIGNED_CHECK	3	BAD_PRACTICE	Check for sign of bitwise operation
12	BIT	BIT_AND	1	CORRECTNESS	Incompatible bit masks
13	BIT	BIT_IOR	1	CORRECTNESS	Incompatible bit masks
14	BIT	BIT_AND_ZZ	1	CORRECTNESS	Check to see if ((...) & 0) == 0
15	Dm	DM_DEFAULT_ENCODING	1	I18N	Reliance on default encoding
16	Dm	DM_CONVERT_CASE	3	I18N	Consider using Locale parameterized version of invoked method
17	Dm	DM_STRING_VOID_CTOR	2	PERFORMANCE	Method invokes inefficient new String() constructor
18	Dm	DMI_BLOCKING_METHODS_ON_URL	1	PERFORMANCE	The equals and hashCode methods of URL are blocking
19	Dm	DMI_COLLECTION_OF_URLS	1	PERFORMANCE	Maps and sets of URLs can be performance hogs
20	Dm	DM_GC	1	PERFORMANCE	Explicit garbage collection; extremely dubious except in benchmarking code
21	Dm	DM_STRING_CTOR	2	PERFORMANCE	Method invokes inefficient new String(String) constructor
22	Dm	DM_BOOLEAN_CTOR	2	PERFORMANCE	Method invokes inefficient Boolean constructor; use Boolean.valueOf(...) instead
23	Dm	DM_STRING_TOSTRING	3	PERFORMANCE	Method invokes toString() method on a String
24	Dm	DM_NEXTINT_VIA_NEXTDOUBLE	2	PERFORMANCE	Use the nextInt method of Random rather than nextDouble to generate a random integer
25	Dm	DM_EXIT	3	BAD_PRACTICE	Method invokes System.exit(...)
26	ES	ES_COMPARING_PARAMETER_STRING_WITH_EQ	1	BAD_PRACTICE	Comparison of String parameter using == or !=
27	ES	ES_COMPARING_STRINGS_WITH_EQ	2	BAD_PRACTICE	Comparison of String objects using == or !=

Table 4 Details of the bug kinds *Continued*

28	Bx	DM_BOXED_PRIMITIVE_TOSTRING	2	PERFORMANCE	Method allocates a boxed primitive just to call toString
29	Bx	DM_FP_NUMBER_CTOR	3	PERFORMANCE	Method invokes inefficient floating-point Number constructor; use static valueOf instead
30	Bx	DM_NUMBER_CTOR	2	PERFORMANCE	Method invokes inefficient Number constructor; use static valueOf instead
31	Bx	DM_BOXED_PRIMITIVE_FOR_PARSING	1	PERFORMANCE	Boxing/unboxing to parse a primitive
32	Bx	BX_BOXING_IMMEDIATELY_UNBOXED	2	PERFORMANCE	Primitive value is boxed and then immediately unboxed
33	Bx	BX_BOXING_IMMEDIATELY_UNBOXED_TO_PERFORM_COERCION	2	PERFORMANCE	Primitive value is boxed then unboxed to perform primitive coercion
34	Bx	BX_UNBOXING_IMMEDIATELY_REBOXED	2	PERFORMANCE	Boxed value is unboxed and then immediately reboxed
35	EI2	EI_EXPOSE_REP2	2	MALICIOUS_CODE	May expose internal representation by incorporating reference to mutable object
36	IIO	IIO_INEFFICIENT_INDEX_OF	3	PERFORMANCE	Inefficient use of String.indexOf(String)
37	IIO	IIO_INEFFICIENT_LAST_INDEX_OF	3	PERFORMANCE	Inefficient use of String.lastIndexOf(String)

Moreover, we can analyze the priority of these 14 bug patterns according to FindBugs classification. In this case we observe that 50 % are of priority 1 (most important for FindBugs), 28.6 % are of priority 2, and 21.4 % are of priority 3. This suggests that even bug patterns which FindBugs classifies as priority 2 or 3 may have a good detection capability of specific types of faults.

On the other hand, several other bug kinds presented $DCL_R(w) \approx 0.00$ %. These bug kinds ($DCL_R(w) \approx 0.00$ %) are not capable of detecting the difference between the original and the mutated source code. Examples of these bug kinds are: Dm, ES and Bx with $DCL_R(w) = 0.01$ %, and EI2 and IIO with $DCL_R(w) = 0.00$ % (lines 39 to 43 in Table 3).

By analyzing the 5 bug kinds with $DCL_R(w) \approx 0.00$ % we observe that they group 23 bug patterns belonging to: PERFORMANCE(17), BAD_PRACTICE(3), II8N(2) and MALICIOUS_CODE(1). In terms of priority, the distribution of these 23 bug patterns is: 26,1 % of priority 1, 47,8 % of priority 2, and 26,1 % of priority 3. Observe that, although in this case, bug patterns with medium/low priority correspond to more than 73 %, there are bug patterns with priority 1 which have no correspondence with faults modeled by the μJava mutation operator.

Observe that this does not mean these bug kinds are good/bad predictor of faults or generate only true/false positive warnings. It only indicates that they are good/bad predictors on detecting faults modeled by these set of mutation operators. Nevertheless, once mutation testing has confirmed as an effective criterion for test set evaluation (Andrews et al. 2005) we consider the set of faults modeled by its mutation operators a good starting point for FindBugs bug kind prioritization.

This variation in the correspondence between bug kinds and mutation operators illustrates that warnings of some bug kinds are more sensitive in identifying certain types of faults, represented by specific mutation operators.

4.2 Using DCL for mutation operators prioritization

In the same way, correspondence information can be used to prioritize mutation operators. In this case, the idea is to use first mutation operators for that there is a lower correspondence with warnings, i.e., mutation operators which represent faults difficult to be detected by FindBugs.

Table 5 presents information about $DCL_A(o_k)$ and $DCL_R(o_k)$ for each mutation operator of μJava. Operators are classified by decreasing order of $DCL_R(o_k)$.

In columns of Table 5, the first column is a line identifier; the second and third columns present the operator name; the column Type is a category of each operator ((C) class and (T) traditional); column $TM(o_K)$ is the result of function of same name defined in Section 3.1.1; column "$TM(o_k)$ with Warning" contains, for each operator, the number of mutants that had warnings (column total$_1$) and the relative amount of these mutants with the total (column total$_1/TM(o_k)$); column "$TM(o_k)$ without Warning" contains, for each operator, the amount of mutants that did not have warnings (column total$_2$) and the relative amount of these mutants with the total (column total$_2/TM(o_k)$); at last, columns $DCL_A(o_k)$ and $DCL_R(o_k)$ are the absolute and relative correspondence of warning and mutant, as defined in Section 3.1.1.

The Total line of Table 5 shows that 493,522 mutants were generated (considering all μJava mutation operators). On 38 % of these mutants (187,892 mutants) no warning was

Table 5 DCL by Mutation Operator: $DCL_R(o_k)$

#	Operator o_k	Type	$TM(o_k)$	$TM(o_k)$ with warning		$TM(o_k)$ without warning		$DCL_A(o_k)$	$DCL_R(o_k)$
				Total_1	$\text{Total}_1/TM(o_k)$	Total_2	$\text{Total}_2/TM(o_k)$		
1	JTD	C	3243	3094	95.41 %	149	4.59 %	2850	87.88 %
2	JTI	C	4969	4372	87.99 %	597	12.01 %	3605	72.55 %
3	JSD	C	3229	2806	86.90 %	423	13.10 %	2118	65.59 %
4	ISD	C	75	51	68.00 %	24	32.00 %	37	49.33 %
5	OAN	C	2632	2073	78.76 %	559	21.24 %	561	21.31 %
6	AOIS	T	55,182	40,189	72.83 %	14,993	27.17 %	10,716	19.42 %
7	LOR	T	484	341	70.45 %	143	29.55 %	93	19.21 %
8	COR	T	8659	5629	65.01 %	3030	34.99 %	1241	14.33 %
9	SDL	T	102,500	66,644	65.02 %	35,856	34.98 %	12,201	11.90 %
10	COI	T	27,399	18,963	69.21 %	8436	30.79 %	2529	9.23 %
11	EOC	C	67	45	67.16 %	22	32.84 %	6	8.96 %
12	AORB	T	10,287	6844	66.53 %	3443	33.47 %	851	8.27 %
13	PRV	C	8010	5344	66.72 %	2666	33.28 %	661	8.25 %
14	ROR	T	47,917	32,617	68.07 %	15,300	31.93 %	3519	7.34 %
15	AODU	T	451	308	68.29 %	143	31.71 %	22	4.88 %
...	...	...	...	...	...	...	...	...	...
31	OMR	C	6361	5549	87.23 %	812	12.77 %	4	0.06 %
32	IHD	C	19	4	21.05 %	15	78.95 %	0	0.00 %
33	LOD	T	21	6	28.57 %	15	71.43 %	0	0.00 %
34	OMD	C	73	27	36.99 %	46	63.01 %	0	0.00 %
35	JDC	C	144	72	50.00 %	72	50.00 %	0	0.00 %
36	EOA	C	166	91	54.82 %	75	45.18 %	0	0.00 %
37	ISI	C	170	88	51.76 %	82	48.24 %	0	0.00 %
38	PPD	C	209	174	83.25 %	35	16.75 %	0	0.00 %
39	PCC	C	211	46	21.80 %	165	78.20 %	0	0.00 %
40	PMD	C	221	87	39.37 %	134	60.63 %	0	0.00 %
41	IOR	C	426	190	44.60 %	236	55.40 %	0	0.00 %
42	IOP	C	489	177	36.20 %	312	63.80 %	0	0.00 %
43	IPC	C	866	159	18.36 %	707	81.64 %	0	0.00 %
44	IHI	C	2304	2045	88.76 %	259	11.24 %	0	0.00 %
45	JID	C	2828	1882	66.55 %	946	33.45 %	0	0.00 %
46	IOD	C	3631	1430	39.38 %	2201	60.62 %	0	0.00 %
47	PCI	C	48,444	11,769	24.29 %	36,675	75.71 %	0	0.00 %
Total			493,522	305,630	61.93 %	187,892	38.07 %	42,138	8.54 %
Average			10,500	6,503	60.22 %	3,998	39.78 %	897	9.05 %
Median			2,446	1,613	59.51 %	540	40.49 %	11	0.76 %

reported. On 305,630 (61.93 %) mutants, FindBugs reported at least one warning different of the ones reported in the original file.

$DCL_R(o_k)$ of each operator o_k is presented in the last column of Table 5. In the first lines of this table are the mutation operators which generate faults easier to be detected by FindBugs. In the last lines of this same table are the mutation operators that represent fault categories FindBugs has difficult to identify. In general, considering all the mutation operators, $DCL_R(o_k)$ was of 8.54 % (42,138/493,522) (last column of Total line).

There is a variation in $DCL_R(o_k)$ rate among the several kinds of mutation operators. There are 4 class mutation operators (JTD, JTI, JSD and ISD), which have $DCL_R(o_k)$ above 49 %, with JTD reaching 87.88 %. Observe that one may suggest to prioritize the analyses of warnings associated with these types of faults since, according to these data, they have more chance to be true positives warnings. At least in 49 to 87.88 % of the cases they are able to point specific source code lines containing mutants of these mutation operators.

The operator JTD, from Java Specific feature and responsible for simulating faults by removing `this` keyword, has a direct correspondence by line of 87.88 %. Operators JTI and JSD, also from Java Specific feature category, are responsible to model faults related to `this` keyword insertion and `static` modifier deletion, and have correspondence rates of 72.55 and 65.59 %, respectively. Finally, the forth mutation operator is ISD, from Inheritance category. It has a correspondence rate of 49.33 % and models faults relate to `super` keyword deletion.

This does not mean that by correcting warnings of a warning category with higher correspondence to mutations, faults will be removed. But, if we do not have enough resources to deal with all the warning categories, this strategy, at least, provides information about which bug kinds generate warnings with some correspondence to specific fault categories, represented by the mutants.

On the other hand, there are mutation operators whose faults are not sensible by FindBugs. Warning insensitive mutations are generated by only one method mutation operator: LOD - responsible for removing logical operators (see second line of Table 6).

There are also 15 class mutation operators which generate mutants which are warning insensitive. They model different fault categories: 1 related to common mistake (EOA) on using reference assignment instead of cloning the object content; 7 related to Inheritance feature on deleting and removing a attribute on a subclass with the same name of a attribute in the parent class (IHD and IHI), or deleting or renaming methods on a subclass with the same name methods in the parent class (IOD and IOR), or removing the call to `super()` on the subclass constructor (IPC), or inserting `super` keyword (ISI), or moving the calling position of overriding methods (IOP); 2 of Java specific features related to removing default constructor if it exists (JDC) or removing the initialization of instance variables in declaration (JID); 1 related to overloading feature removing the overload method of a subclass (OMD); and 4 related to polymorphism feature changing

Table 6 Warning insensitive mutation operator

#	MO	Category	MO description
32	IHD	Inheritance	Hiding variable deletion
33	LOD	Logical	Logical Operator Deletion
34	OMD	Overloading	Overloading method deletion
35	JDC	Java Specific	Java-supported default constructor creation
36	EOA	Common Mistake	Reference assignment and content assignment replacement
37	ISI	Inheritance	`super` keyword insertion
38	PPD	Polymorphism	Parameter variable declaration with child class type
39	PCC	Polymorphism	Cast type change
40	PMD	Polymorphism	Instance variable declaration with parent class type
41	IOR	Inheritance	Overriding method rename
42	IOP	Inheritance	Overriding method calling position change
43	IPC	Inheritance	Explicit call of a parent's constructor deletion
44	IHI	Inheritance	Hiding variable insertion
45	JID	Java Specific	Member variable initialization deletion
46	IOD	Inheritance	Overriding method deletion
47	PCI	Polymorphism	Type cast operator insertion

the type of a variable declaration for a parent type (PMD), changing the cast type of a parent class to on of its subclass (PCC), changing the type of a parameter variable by the type of a child class type (PPD), and inserting type cast before reference variables (PCI). In static analyzers terminology these faults are called false negative, which means faults that exist in the source code but the static analyzers is not adequate to detect.

As stated previously, such information are useful to improve FindBugs to detect additional kinds of faults for instance, by written a new rule to check the existence of the default constructor (JDC) or checking for the need to overload parent class methods (OMD). Moreover, assuming an incremental testing strategy when combining static and dynamic analysis, this set of mutation operators which generate warning insensible mutants should be used during testing once the chance the faults they modeled be detected during automatic static analysis is reduced.

We consider the results obtained so far very promising and it is expected that, with the increase of the number of evaluated systems, it is possible to identify other fault categories, represented by mutation operators, which can contribute to an optimization in the establishment of incremental strategies which combine static and dynamic analysis in a more efficient and effective way.

Considering these results, we may suggest from Tables 3 and 5 incremental strategies for applying warning categories of FindBugs and mutation operators of μJava.

In the case of warning categories, the order is from the ones with higher direct correspondence rates to the lower correspondence rates. This strategy is illustrated in Section 5.1. In the case of mutation operators, we use the inverse order of correspondence since the lower the correspondence more difficult the specific fault types to be detected by automatic static analyzer and the mutation operator should be considered during testing. This strategy is illustrated in Section 5.2.

Observe that this knowledge database can be always improved. As soon as more information about warnings and mutations are collected, the incremental strategy can be updated aiming at improving its capability, contributing to reduce the cost of static analysis/mutation testing and guiding the reviewer to firstly analyze warnings more probably to lead to fault detection and quality improvement before to conduct mutation testing.

5 Incremental strategies: example of application

In this section we illustrate how the incremental strategies defined on Sections 4.1 and 4.2 can be used incrementally to reduce the cost of application of either static analysis or mutation testing.

5.1 Prioritization of bug kinds based on $DCL_R(w)$

To illustrate how the direct correspondence can be employed to prioritize the analysis of warnings reported by FindBugs, we applied the bug kinds as described above on three additional systems: Cassandra, Hibernate and Apache POI. Table 7 presents the complexity of these systems based on its size and the number of warnings generated by FindBugs on its default configuration.

The suggested approach is to prioritize the analysis of bug kinds with higher $DCL_R(w)$ rates in detriment to the ones with lower $DCL_R(w)$ rates, respecting the bug kinds order defined on Table 3.

Table 7 Metrics of example systems

ID	System	Version	LOC	Mutants	Warnings
C	Cassandra	2.1.0	43,792	3289	899
P	Apache POI	3.9	61,460	19,717	790
H	Hibernate	4.3.6	129,337	110,677	2204
Total			234,589	133,683	3893

Table 8 presents the data of applying the bug kinds incrementally. For instance, DLS warning rules (line 11) reported 5 warnings in Cassandra system (W_C column), 26 in Apache POI (W_P column), 392 in Hibernate (W_H column), with the total of 423 (W_{total} column) warnings reported in the three systems.

Columns CC_C, CC_P and CC_H of Table 8 store the cumulative cost of warnings by each category of each system. The column CC_{total} stores the total warnings of the three systems. The $CC(i)$ (each system and total) on the line i is obtained by the Eq. 3. Columns CR_C, CR_P, CR_H and CR_{total} show the cost reduction to consider the other bug kinds. $CR(i)$ (each system and total), on the line i is obtained by Eq. 4.

Cumulative Cost of Warning ($CC(i)$):

$$CC(i) = W_1 + W_2 + \cdots + W_{i-1} + W_i \tag{3}$$

Cost Reduction of Warning $CR(i)$**):**

$$CR(i) = 1 - (CC(i)/Total) \tag{4}$$

In Fig. 6, we illustrate the data for CC_{total} and CR_{total}, when the warnings are analyzed according to the order suggested in Table 8. Note, in Fig. 6, that initially $CR_{total} = 100$ % because no warning was analyzed ($CC_{total} = 0$). We suggest the warnings with higher $DCL_R(w)$ should be analyzed first, since these warnings are more likely to correspond to a fault according to our study.

From Table 8, by summing up the number of warnings reported from bug kinds of lines 1 to 11, the cumulative cost were 8, 42, and 413 for each system individually (columns CC_C, CC_P and CC_H, respectively), which means that, if only these warning rules were analyzed the cost reduction with respect to all 79 bug kinds were 99.15, 94.59, and 80.62 % (columns CR_C, CR_P and CR_H, respectively), respectively. Therefore, overall, 463 warnings were generated (CC_{total} column), with means a cost reduction of 87.98 % (CR_{total} column).

If the top 18 warning rules were used, the 694 warnings were generated at all, representing a cost reduction around 81 % with respect to the total number of warnings. Moreover, observe that Cassandra and Apache POI present both cost reduction above 90 % for these set of bug kinds which may indicate that Apache community handled the problems reported by these bug kinds during its development processes.

In the last line of Table 8 we can see the total number of warnings the warning rules composing the incremental strategy generate on each system: 946, 776, and 2,131, respectively.

5.2 Prioritization mutation operators based on $DCL_R(o_k)$

Considering the prioritization of mutation operators, based on $DCL_R(o_k)$, presented in Table 5 (p. 16), we applied the mutation operators incrementally. The idea is to use the historical data previously collected about the correspondence rate between warnings and

Table 8 Incremental strategy for applying bug kinds

#	W	Cassadra			Apache POI			Hibernate			Total		
		W_C	CC_C	CR_C	W_P	CC_P	CR_P	W_H	CC_H	CR_H	W_{total}	CC_{total}	CR_{total}
1	QF	0	0	100.00 %	0	0	100.00 %	0	0	100.00 %	0	0	100.00 %
2	RE	0	0	100.00 %	0	0	100.00 %	3	3	99.86 %	3	3	99.92 %
3	UM	0	0	100.00 %	0	0	100.00 %	0	3	99.86 %	0	3	99.92 %
4	QBA	0	0	100.00 %	0	0	100.00 %	0	3	99.86 %	0	3	99.92 %
5	INT	2	2	99.79 %	3	3	99.61 %	0	3	99.86 %	5	8	99.79 %
6	BIT	0	2	99.79 %	4	7	99.10 %	0	3	99.86 %	4	12	99.69 %
7	UR	0	2	99.79 %	3	10	98.71 %	4	7	99.67 %	7	19	99.51 %
8	UCF	0	2	99.79 %	5	15	98.07 %	13	20	99.06 %	18	37	99.04 %
9	SA	1	3	99.68 %	0	15	98.07 %	0	20	99.06 %	1	38	99.01 %
10	IL	0	3	99.68 %	1	16	97.94 %	1	21	99.01 %	2	40	98.96 %
11	DLS	5	8	99.15 %	26	42	94.59 %	392	413	80.62 %	423	463	87.98 %
12	SS	0	8	99.15 %	0	42	94.59 %	1	414	80.57 %	1	464	87.96 %
13	BSHIFT	0	8	99.15 %	0	42	94.59 %	0	414	80.57 %	0	464	87.96 %
14	DB	4	12	98.73 %	2	44	94.33 %	3	417	80.43 %	9	473	87.72 %
15	RpC	0	12	98.73 %	1	45	94.20 %	1	418	80.38 %	2	475	87.67 %
16	NP	64	76	91.97 %	7	52	93.30 %	90	508	76.16 %	161	636	83.49 %
17	DMI	1	77	91.86 %	0	52	93.30 %	1	509	76.11 %	2	638	83.44 %
18	UwF	12	89	90.59 %	3	55	92.91 %	41	550	74.19 %	56	694	81.99 %
19	BC	49	138	85.41 %	92	147	81.06 %	86	636	70.15 %	227	921	76.10 %
20	MS	29	167	82.35 %	71	218	71.91 %	30	666	68.75 %	130	1,051	72.72 %
...	...	...	...	...	...	...	...	...	...	...	...	...	...
61	Nm	6	551	41.75 %	28	701	9.66 %	18	1260	40.87 %	52	2512	34.80 %
62	ODR	0	551	41.75 %	0	701	9.66 %	2	1262	40.78 %	2	2514	34.75 %
63	PZLA	5	556	41.23 %	8	709	8.63 %	54	1316	38.24 %	67	2581	33.01 %
64	RC	0	556	41.23 %	0	709	8.63 %	1	1317	38.20 %	1	2582	32.99 %
65	RI	3	559	40.91 %	0	709	8.63 %	28	1345	36.88 %	31	2613	32.18 %
66	RR	0	559	40.91 %	8	717	7.60 %	0	1345	36.88 %	8	2621	31.98 %
67	SC	4	563	40.49 %	0	717	7.60 %	0	1345	36.88 %	4	2625	31.87 %
68	SF	143	706	25.37 %	26	743	4.25 %	4	1349	36.70 %	173	2798	27.38 %
69	SIC	164	870	8.03 %	10	753	2.96 %	99	1448	32.05 %	273	3071	20.30 %
70	STCAL	2	872	7.82 %	0	753	2.96 %	0	1448	32.05 %	2	3073	20.24 %
71	Se	48	920	2.75 %	3	756	2.58 %	106	1554	27.08 %	157	3230	16.17 %
72	SnVI	9	929	1.80 %	9	765	1.42 %	551	2105	1.22 %	569	3799	1.40 %
73	UL	0	929	1.80 %	0	765	1.42 %	1	2106	1.17 %	1	3800	1.38 %
74	UPM	6	935	1.16 %	1	766	1.29 %	13	2119	0.56 %	20	3820	0.86 %
75	USELESS_ STRING	0	935	1.16 %	9	775	0.13 %	2	2,121	0.47 %	11	3831	0.57 %
76	UuF	1	936	1.06 %	0	775	0.13 %	2	2123	0.38 %	3	3834	0.49 %
77	VO	4	940	0.63 %	0	775	0.13 %	2	2125	0.28 %	6	3840	0.34 %
78	WMI	4	944	0.21 %	1	776	0.00 %	6	2131	0.00 %	11	3851	0.05 %
79	Wa	2	946	0.00 %	0	776	0.00 %	0	2131	0.00 %	2	3853	0.00 %
80	Sum	946	—	—	776	—	—	2131	—	—	3853	—	—

mutants. The incremental strategy would prioritize, initially, the use of the operators that generate mutants that were not detected by FindBugs and have a lower cost in terms of the number of generated mutants.

Table 9 shows the number of mutants generated for each type of μJava mutation operator in Cassandra (column M_C), Apache POI (column M_P) and Hibernate (column M_H).

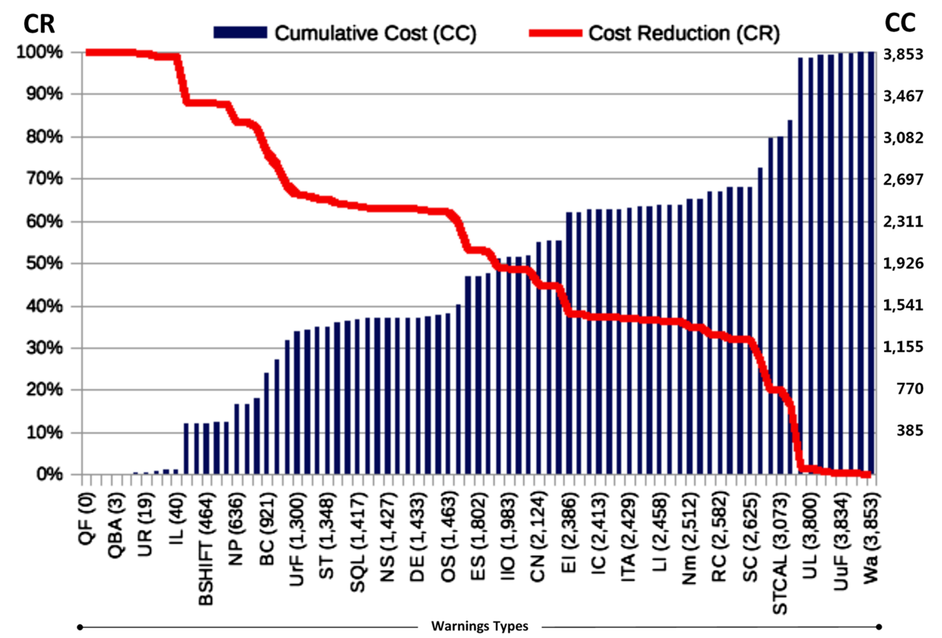

Fig. 6 Cumulative Cost versus Cost Reduction of warning category incremental strategy

The M_{total} column equals $M_C + M_P + M_H$. For example, in line 15 we find the number of mutants generated regarding the IOD operator: 47 mutants in Cassandra (column M_C), 70 mutants in the Apache POI (column M_P), and 1,389 mutants in Hibernate (column M_H). On all the three systems, IOD generated 1,506 mutants (column M_{total}).

In order to reduce the cost of Mutation Testing, Table 9 shows the mutation operators from μJava applied incrementally. The goal is to prioritize the order of applying the mutation operators based on increasing order by $DCL_R(o_k)$. When $DCL_R(o_k)$ of two mutation operators are the same, we apply first the mutation operator which generates less mutantes. The rationality is that this order privileges lower cost mutation operators which represent fault categories difficult to be detectable by FindBugs. In other words, considering the collected data, operators with smaller $DCL_R(o_k)$ indicate the changes that FindBugs was less effective or even unable to generate warnings that could detect them.

Table 9 presents mutation operators in increasing order by $DCL_R(o_k)$ and cost in terms of number of mutants. Equations below are defined, aiming at evaluating the cost of the incremental strategy.

Cumulative Cost of Operator ($CC(o_k)$):

$$CC(o_k) = M_1 + M_2 + \cdots + M_{k-1} + M_k \tag{5}$$

Cost Reduction of Operator ($CR(o_k)$):

$$CR(o_k) = 1 - (CC(o_k)/Total) \tag{6}$$

Equation 5 is the cumulative cost in terms of the number of mutants to be analyzed, following the priority order established in Table 9. It represents the number of mutants generated from the operator in the first line of Table 9, IHD, until the operator at the k-th line.

Table 9 Incremental strategy for applying mutation operator

#	*W*	Cassandra			Apache POI			Hibernate			Total		
		M_C	CC_C	CR_C	M_P	CC_P	CR_P	M_H	CC_H	CR_H	M_{total}	CC_{total}	CR_{total}
1	IHD	0	0	100.00 %	1	1	99.99 %	3	3	100.00 %	4	4	100.00 %
2	LOD	0	0	100.00 %	0	1	99.99 %	0	3	100.00 %	0	4	100.00 %
3	OMD	0	0	100.00 %	3	4	99.98 %	2	5	100.00 %	5	9	99.99 %
4	JDC	0	0	100.00 %	0	4	99.98 %	64	69	99.94 %	64	73	99.95 %
5	EOA	0	0	100.00 %	0	4	99.98 %	0	69	99.94 %	0	73	99.95 %
6	ISI	1	1	99.97 %	1	5	99.97 %	35	104	99.91 %	37	110	99.92 %
7	PPD	0	1	99.97 %	0	5	99.97 %	27	131	99.88 %	27	137	99.90 %
8	PCC	1	2	99.94 %	20	25	99.87 %	16	147	99.87 %	37	174	99.87 %
9	PMD	0	2	99.94 %	0	25	99.87 %	8	155	99.86 %	8	182	99.86 %
10	IOR	5	7	99.79 %	16	41	99.79 %	83	238	99.78 %	104	286	99.79 %
11	IOP	2	9	99.73 %	1	42	99.79 %	43	281	99.75 %	46	332	99.75 %
12	IPC	18	27	99.18 %	30	72	99.63 %	32	313	99.72 %	80	412	99.69 %
13	IHI	4	31	99.06 %	5	77	99.61 %	102	415	99.63 %	111	523	99.61 %
14	JID	6	37	98.88 %	16	93	99.53 %	254	669	99.40 %	276	799	99.40 %
15	IOD	47	84	97.45 %	70	163	99.17 %	1389	2058	98.14 %	1506	2305	98.28 %
16	PCI	90	174	94.71 %	880	1043	94.71 %	7975	10,033	90.93 %	8945	11,250	91.58 %
17	OMR	15	189	94.25 %	3	1046	94.69 %	2215	12,248	88.93 %	2233	13,483	89.91 %
18	AOIU	172	361	89.02 %	1647	2693	86.34 %	2907	15,155	86.31 %	4726	18,209	86.38 %
19	AORS	9	370	88.75 %	57	2750	86.05 %	408	15,563	85.94 %	474	18,683	86.02 %
20	EMM	0	370	88.75 %	9	2759	86.01 %	1437	17,000	84.64 %	1446	20,129	84.94 %
21	LOI	238	608	81.51 %	2274	5033	74.47 %	4830	21,830	80.28 %	7342	27,471	79.45 %
22	EAM	38	646	80.36 %	780	5813	70.52 %	11,851	33,681	69.57 %	12,669	40,140	69.97 %
23	COD	12	658	79.99 %	22	5835	70.41 %	727	34,408	68.91 %	761	40,901	69.40 %
24	VDL	66	724	77.99 %	612	6447	67.30 %	1719	36,127	67.36 %	2397	43,298	67.61 %
25	ASRS	52	776	76.41 %	674	7121	63.88 %	256	36,383	67.13 %	982	44,280	66.88 %

Table 9 Incremental strategy for applying mutation operator (*Continued*)

26	AODS	2	778	76.35 %	3	7124	63.87 %	87	36,470	67.05 %	92	44,372	66.81 %
27	PNC	6	784	76.16 %	0	7124	63.87 %	1751	38,221	65.47 %	1757	46,129	65.49 %
28	ODL	241	1025	68.84 %	2087	9211	53.28 %	8489	46,710	57.80 %	10,817	56,946	57.40 %
29	CDL	46	1071	67.44 %	523	9734	50.63 %	2092	48,802	55.91 %	2661	59,607	55.41 %
30	SOR	2	1073	67.38 %	2	9736	50.62 %	0	48,802	55.91 %	4	59,611	55.41 %
31	JSI	40	1113	66.16 %	141	9877	49.91 %	1509	50,311	54.54 %	1690	61,301	54.14 %
32	PCD	0	1113	66.16 %	1	9878	49.90 %	5	50,316	54.54 %	6	61,307	54.14 %
33	AODU	4	1117	66.04 %	7	9885	49.87 %	53	50,369	54.49 %	64	61,371	54.09 %
34	ROR	528	1645	49.98 %	1246	11,131	43.55 %	9999	60,368	45.46 %	11,773	73,144	45.29 %
35	PRV	21	1666	49.35 %	19	11,150	43.45 %	1183	61,551	44.39 %	1223	74,367	44.37 %
36	AORB	192	1858	43.51 %	1484	12,634	35.92 %	2284	63,835	42.32 %	3960	78,327	41.41 %
37	EOC	0	1858	43.51 %	0	12,634	35.92 %	26	63,861	42.30 %	26	78,353	41.39 %
38	COI	152	2010	38.89 %	290	12,924	34.45 %	6542	70,403	36.39 %	6984	85,337	36.16 %
39	SDL	467	2477	24.69 %	2502	15,426	21.76 %	24,284	94,687	14.45 %	27,253	112,590	15.78 %
40	COR	56	2533	22.99 %	46	15,472	21.53 %	2442	97,129	12.24 %	2544	115,134	13.88 %
41	LOR	12	2545	22.62 %	88	15,560	21.08 %	8	97,137	12.23 %	108	115,242	13.79 %
42	AOIS	602	3147	4.32 %	3858	19,418	1.52 %	7792	104,929	5.19 %	12,252	127,494	4.63 %
43	OAN	15	3162	3.86 %	16	19,434	1.44 %	2102	107,031	3.29 %	2133	129,627	3.03 %
44	ISD	0	3162	3.86 %	0	19,434	1.44 %	20	107,051	3.28 %	20	129,647	3.02 %
45	JSD	27	3189	3.04 %	191	19,625	0.47 %	736	107,787	2.61 %	954	130,601	2.31 %
46	JTI	89	3278	0.33 %	49	19,674	0.22 %	2133	109,920	0.68 %	2271	132,872	0.61 %
47	JTD	11	3289	0.00 %	43	19,717	0.00 %	757	110,677	0.00 %	811	133,683	0.00 %
48	Total	3289	–	–	19,717	–	–	110,677	–	–	133,683	–	–

The CC_C, CC_P and CC_H columns store, respectively, the result of Function 5 on the generated mutants in Cassandra, Apache POI and Hibernate systems. The column CC_{total} is the result of Function 5 applied to the three systems. The costs accumulated to analyze mutants related to the first 10 mutation operators (IHD, LOD, OMD, JDC, EOA, ISI, PPD, PCC, PMD and IOR) are 7 mutants in the Cassandra, 41 mutants in the Apache POI, 238 mutants in Hibernate, and 286 mutants considering the three systems together.

Considering the 16 first mutation operators with $DCL_R(o_k) = 0$ % (IHD, LOD, OMD, JDC, EOA, ISI, PPD, PCC, PMD, IOR, IOP, IPC, IHI, JID, IOD and PCI), the cumulative cost is 174 mutants in Cassandra, 1,043 mutants in Apache POI, 10,033 mutants in Hibernate, summing up 11,250 mutants considering all the three systems together. Observe that this represents only 8.42 % $(11,250/133,683)$ of all possible mutants μJava is able to generate considering the entire mutation operator set. Moreover, faults modeled by such mutation operators are not easily detected by FindBugs.

On the other hand, Eq. 6 represents the cost reduction (in %) relative to the total of mutants if only the operators until the k-th line are used.

Columns CR_C, CR_P, CR_H and CR_{total} store, respectively, the result of Eq. 6 in Cassandra, Apache POI, Hibernate, and the three systems together. The cost reductions provided for analyzing only the top 10 mutation operators (IHD, LOD, OMD, JDC, EOA, ISI, PPD, PCC, PMD and IOR) are 99.79 % for Cassandra, 99.79 % for Apache POI, 99.78 % for Hibernate, and 99.79 % considering all the three systems.

Note that the cost reduction still remains above 90.0 % considering the 16 operators $DCL_R(o_k) = 0.0$ %. The cost reductions for analyzing the mutants of these 16 mutation operators (IHD, LOD, OMD, JDC, EOA, ISI, PPD, PCC, PMD, IOR, IOP, IPC, IHI, JID, IOD and PCI) are 94.71 % for Cassandra, 94.71 % for Apache POI, 90.93 % for Hibernate, and 91.58 % considering all the three systems together.

Figure 7 shows the curves of cumulative cost ($CC(o_k)$) and cost reduction ($CR(o_k)$), respecting the order defined in the prioritization strategy suggested by $DCL_R(o_k)$.

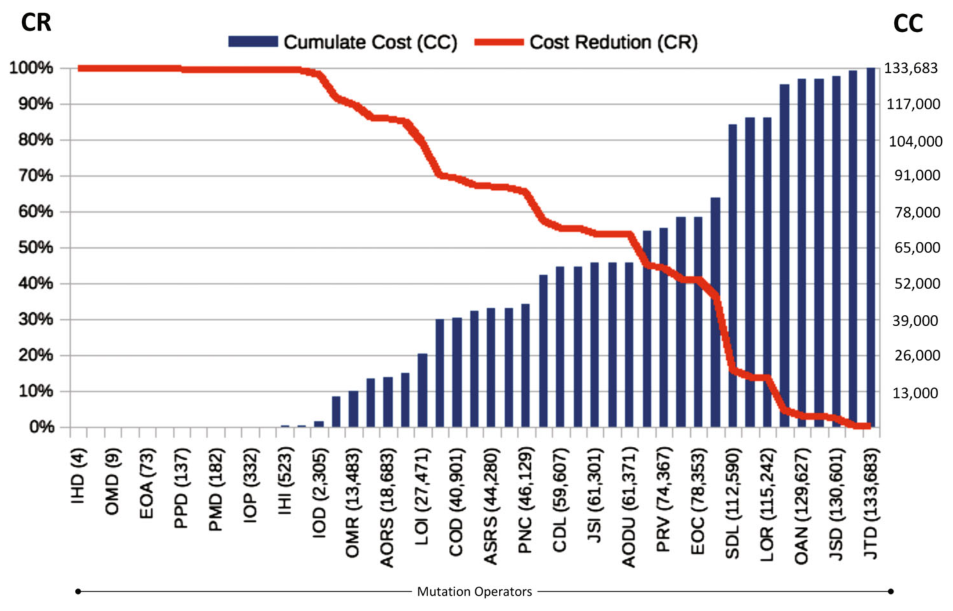

Fig. 7 Cumulative Cost versus Cost Reduction of mutation operator incremental strategy

Didactically, Figs. 6 and 7 can be understood by using three complementary scenarios:

- **Scenario 1: no warning category/mutation operator**. This case illustrates the scenario in which Cumulative Cost is zero (0 %) and Cost Reduction is 100 %;
- **Scenario 2: all warning categories/mutation operators are used**. This case illustrates the worst scenario where Cumulative Cost is 100 % and Cost Reduction is zero (0 %);
- **Scenario 3: any stage between Scenarios 1 and 2**. This case illustrates situation where only a subset of warning categories or mutation operators is used. In this case, the reviewer/tester can combine both strategies in a complementary way considering the trade off between Cost Reduction between 0 % $\leq CR \leq$ 100 % and a Cumulative Cost between 0 % $\leq (100 - CR) \leq$ 100 %. For instance, considering the warning categories in Fig. 6, if we choose a CR of 60 % we would consider the warning categories from QF to DE, implying a CC of 40 %.
  In the same way, when applying mutation operators as illustrated in Fig. 7, the tester may want to obtain a cost reduction around 80 %. In this case, he/she may consider to use operators from IHD to LOI, implying a CC around 20 %.

6 Lessons learned and threats to validity

For direct correspondence, mutation test showed (Table 5) a variation of direct correspondence among the operators, that is, among the fault categories simulated by the mutants. This result adds on the one obtained by Couto et al. (2013) since in that study it was not possible to establish direct correspondence per line. The results herein presented show strong evidence that a direct correspondence exists and it is established for specific fault kinds, enabling the establishment of complimentary test strategies integrating static and dynamic analysis.

Observe that our intention is to restrict the types of mutation operators we should use to prioritize true positive static warnings and to identify possible types of faults which FindBugs are not adequate to detect such that we can combine static and dynamic analysis in a coordinated way to take the advantage of each other.

Cost and benefit of the suggested approach:

Benefit: the value obtained by the $DCL(w)$ of each type of warning w is used to establish a prioritization order to analyze the warnings. With the use of the suggested priority order, it is expected that the true positive warnings are analyzed as soon as possible, leaving the "less important" alarms (with smaller DCL) to be analyzed later if there are still available resources. On the other hand, the incremental strategy for applying mutation operator allows to consider firstly faults which FindBugs was not adequate to detect.

Cost: There is some cost to run FindBugs on the original program and on its mutants for the database generation and DCL computation. However, as the mutants do not need to be executed, the generation time is lower than the time demanded by mutation test.

To analyze the cost of database generation and DCL computation, first we need to run FindBugs on all Java files of each system. Considering a notebook computer with an Intel Core i5 processor and 8 Gb of RAM memory, FindBugs took 6.5 seconds, on average, to run on each file. We compute the average runtime per file once when running FindBugs on mutants, we need to run it only on the mutated file and not on the entire system. In this way, the most expensive system in terms of generated mutants is s17, which is composed by 438 Java files and μJava generated 79,968 mutants. When running FindBugs on s17, it took 48 minutes and 32 seconds, and reported 219 warnings in the original system. This fact gives an average time for reporting warnings around 6,6 second for system S17. With respect to mutants' warnings computation time it was about 531,659.4 seconds ($6.6 \times 79{,}968$) which corresponds to 6.15 days to finish the data collection.

Observe that the biggest system we analyzed in terms of lines of code, s11, with 102,682 lines of code, has 519 Java files and FindBugs reported 359 warnings in 49 minutes and 50 seconds, giving an average time of 5.8 seconds for reporting warning per file. This was the lowest runtime per file FindBugs obtained in our experiment. Considering that μJava generated 65,529 mutants for s11, we finished the data collection for this system in 4,37 days. The smallest system in terms of lines of code, s9, was also the one which took the lowest time to finish data collection, around 2 hours.

The cost of mutant generation and the cost for loading XML data into the database are not representative taking only a few seconds and are not considered in this analysis.

Observe that, if we have seven cloud computers, similar to the one we have used, we can collect all the data for the prioritization strategy in one day, considering the most expensive program. Therefore, it is evident that our strategy has a manageable cost only dependent on available hardware resources which, in general, are cheaper than human resources.

Once it is determined the prioritization order, as presented in Sections 4.1 and 4.2, the cost to use it, as shown in Section 5.1, is only to apply FindBugs and to analyze the warnings reported, respecting the suggested order. In the same way, when applying mutation testing, the tester may decided how much mutation operators he/she wants to applying respecting the time and cost constraint available. The suggested order prioritizes faults more difficult to be detected by FindBugs.

As any other study, this one also presents threats to validity and limitations as presented below.

External Validity: In this study, we used 19 of 30 systems used by Couto et al. (2013). One of the reasons for the absence of the other 11 systems is due to the difficulty in obtaining its dependencies (jars files) so that μJava tool could not generate the corresponding mutants.

Other limitations observed in the experiment are related to the programming language and the static analysis tools used. In the experiment, it was considered only the Java language and, therefore, the results cannot be extended to other languages, specially to those of dynamic typing. In relation to static analysis tools, it was used only FindBugs and the results cannot be automatically extended to other static analysis tools as well.

About equivalent mutants: one of the objectives of the study is to verify if a given mutation produced is detected by FindBugs Tool, and this is done statically, without the necessity of the mutant execution. Thus, in this first stage of the study, as the mutants were not executed, there is no data about live, dead or equivalent mutants. However, we shall explore the generation of test sets in automated and/or manual ways executing the mutants, evaluating the effectiveness of such test suites, analysing live mutants to determine equivalence and to construct a historical data base for Java mutation operators, similar to the one built to mutation operators for C language, enabling the automatic treatment of equivalent mutants based on Bayesian learning technique (Vincenzi et al. 2002).

Internal Validity: We do not identified internal validity threats. Initially, mutants were generated, through μJava tool (version 4), from the systems shown in Table 1. For this, all available operators in μJava tool were selected with the aim of generating the highest amount of mutants (fault) as possible.

After that, FindBugs Tool (version 3.0.0) was used with option `-low` so that warnings of any priority could be reported. At this stage, FindBugs reports all possible warnings. To calculate the place where mutation occurred, it was necessary to store in the data base the textual difference between each mutant and the original file (diff).

Construction Risk: To control the experimentation process and reduce the risk of an analysis based on incorrect data in systems shown in Table 1, data collection process and the effective usage of the technique based on the faults were validated in simpler systems used in previous experiments (Polo et al. 2009).

7 Related works

Ayewah et al. (2007a) examined different types of warnings reported by FindBugs classifying them into false positives, trivial bugs and serious bugs. They concluded that, besides the high false positive rates, static analysis tools often uncover true but trivial bugs.

Nagappan and Ball (2005) developed an empirical methodology for the early projection of pre-release defect density based on the outcomes of two different static analysis tools. They were able to establish a strong correspondence between the number of warnings reported by the static analysis tools and the actual pre-release fault density for Windows Server 2003 obtained through testing.

Daimi et al. (2013) also carried out a performance evaluation of five Java static analyzers. They evaluated the five tools using Eclipse according to three different criteria: the total number of violations (warnings) found, run time, and memory usage. Based on these criteria each tool was evaluated against six fault categories: data faults, control faults, interface faults, measurement faults, duplicated code, and code convention violations. Based on these categories, faulty codes were temporary injected in three different programs and each tool was evaluated against such a fault version. Although such fault categories were plausible, they reported that Eclipse IDE for Java was able to detect almost all the faults injected immediately once they were syntax errors.

To investigate the direct correspondence, the work of Couto et al. (2013) used three systems that together add up to around 118 thousand lines of code (LOC) and to those were reported 277 corrective faults through Bugzilla and Jira tools. The approach used in the studies presented in Couto et al. (2013) shows the following results concerning correspondence between faults and warnings: non-existence of direct correspondence. In these works, it was used change's history from different versions store in iBugs repository (Dallmeier and Zimmermann 2007) and reports of software faults registered in Bugzilla (Bugzilla 2014) and Jira (Atlassian 2014) tools.

To the best of our knowledge we did not identify any research trying to combine mutation and static analyzers as proposed in this paper. Our work is complementary to the others described above in the sense it uses mutations to evaluate the capability of static analyzers in detecting such mutations. The results obtained so far indicate we can use the collected information to prioritize warning categories in an incremental strategy, allowing the reviewer to apply specific warning categories more adequate to detect mutations first by increasing the chance to analyze a true positive warning. On the other hand, it was also possible to identify warning categories which are not adequate to detect any kind of mutation suggesting these warning categories should be used only if there are time and resources available.

Mutation operators not detected by static analyzers are also source of information to improve the capability of static analyzers in detecting new kinds of faults. Such set of mutation operators were identified and can be further investigated.

8 Conclusion and future work

In this study we provided evidences of direct correspondence by line between warnings issued by FindBugs and mutations generated by μJava mutation operators.

Based on this correspondence we defined two incremental strategies for using FindBugs bug kinds and μJava mutation operators in a complementary way.

In the case of the incremental strategy for using FindBugs, we observed that for four specific bug kinds it is possible to have 100 % of direct correspondence with mutations, i.e., bug patterns of these bug kinds are adequate to detect the difference between the mutants and the original program. Even if we extend the analysis for the top 12 bug kinds, the correspondence with mutations is above 33 %.

On the other hand, we also identified that for a group of 16 mutation operators, FindBugs was not adequate to issue warnings on any of the mutants generated by such operators. Again, we used this information to define another incremental strategy for applying mutation operators considering faults modeled by these operators are very difficult to be statically detected by FindBugs. Moreover, these 16 mutation operators are responsible for less than 10 % of the total number of mutants and should be considered a good starting point for selective Mutation Testing.

As more resources are becoming available, additional bug kinds or mutation operators can be included to improve the fault detection capability of the desired strategy.

A collateral effect of the direct correspondence by line is the possibility to know fault categories FindBugs is not good enough to detect and to write additional bug patterns to improve its capability once, in general, static analysis is cheaper than dynamic analysis.

As future work we intend: 1) to incorporate other static analysis tools for Java (e.g. PMD, and JLint); 2) to extend this study to other programming languages; 3) to evaluate the use of mutation testing as a fault model for static analyzers comparison; 4) to investigate the Cases 2 and 3 shown in Figs. 3 and 4; and 4.2) to create a knowledge database which can be evolved automatically, based on the continuous collection of correspondence data, or manually, based on experts' recommendations.

Endnotes

[1] Bad casts of object references

[2] Null pointer dereference

[3] Dead local store

[4] Dubious method invocation

[5] The lines of code of each one of the systems presented in Table 1 were obtained through JavaNCSS Tool.

[6] KLOC is a acronym to Kilo Lines of Code (LOC/1,000).

Appendix A: Descriptions of mutation operators and warnings

A.1 μJava mutation operators

With respect to mutation operators, μJava has 19 method mutation operators and 28 class mutation operators, as illustrated in Tables 10 and 11, respectively.

The reduced number of method operators implemented by μJava is due to the selective approach adopted to create such a set of mutants (Offutt et al. 1996). As illustrated in Table 10, fault categories are related to Arithmetic, Relational, Conditional, Shift, Logical and Assignment, Variable, Constant, Operator and Statement operators.

Considering the class operators implemented by μJava, they are developed based on object-oriented language features and Java specific features, as illustrated in Table 11. The considered object-oriented features which have mutation operator implemented are: Inheritance, Polymorphism and Overloading, with 8, 7 and 3 mutation operators per feature, respectively. Additionally, it has 6 Java specific mutation operators and 4 related to common programming mistakes.

For more information about these set of mutant operators and examples of the mutations they performed the interested reader may refer to Ma and Offutt (2005) and Ma et al. (2005).

A.2 FindBugs warning categories

The version 3.0 of FindBugs there are 9 different warning categories. Each warning category has a set of bug kinds. At all, there are 121 different bug kinds, and each bug kind has a set of bug patterns, resulting in a set of 408 possible warnings. Table 12 presents all warning categories supported by the FindBugs version used in our experiment, the total of bug patterns on each category, and a sample of specific bug kind of each category.

Table 10 μJava traditional mutation operator (adapted from (Ma and Offutt 2005))

Language feature	Operator	Description
Arithmetic (6)	AORB	Arithmetic Operator Replacement (Binary)
	AORS	Arithmetic Operator Replacement (Short-cut)
	AOIU	Arithmetic Operator Insertion (Uniry)
	AOIS	Arithmetic Operator Insertion (Short-cut)
	AODU	Arithmetic Operator Deletion (Uniry)
	AODS	Arithmetic Operator Deletion (Short-cut)
Relational (1)	ROR	Relational Operator Replacement
Conditional (3)	COR	Conditional Operator Replacement
	COD	Conditional Operator Deletion
	COI	Conditional Operator Insertion
Shift (1)	SOR	Shift Operator Replacement
Logical (3)	LOR	Logical Operator Replacement
	LOI	Logical Operator Insertion
	LOD	Logical Operator Deletion
Assignment (1)	ASRS	Short-cut Assignment Operator Replacement
Statement (1)	SDL	Statement Deletion
Variable (1)	VDL	Variable Deletion
Constant (1)	CDL	Constant Deletion
Operator (1)	ODL	Operator Deletion

Table 11 μJava class mutation operator (adapted from Ma et al. (2005))

Language feature	Operator	Description
Inheritance (8)	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISI	`super` keyword insertion
	ISD	`super` keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism (7)	PNC	`new` method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PCI	Type cast operator insertion
	PCD	Type cast operator deletion
	PCC	Cast type change
	PRV	Reference assignment with other comparable type
Overloading (3)	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAN	Argument number change
Java-Specific (6)	JTI	`this` keyword insertion
	JTD	`this` keyword deletion
	JSI	`static` modifier insertion
	JSD	`static` modifier deletion
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common Programming Mistakes (4)	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method changes

Table 12 FindBugs warning categories

#	Category	Warning count by category	Example warning
1	Bad practice	84	ES: Comparison of String parameter using == or !=
2	Correctness	145	RV: Method ignores return value
3	Experimental	3	OBL: Method may fail to clean up stream or resource
4	Internationalization	2	Dm: Consider using Locale parameterized version of invoked method
5	Malicious code vulnerability	15	DP: Classloaders should only be created inside doPrivileged block
6	Multithreaded correctness	45	STCAL: Static DateFormat
7	Performance	30	UuF: Unused field
8	Security	11	Dm: Empty database password
9	Dodgy code	73	BC: Unchecked/unconfirmed cast
Sum		408	–

FindBugs also has different priority levels for each warning pattern from 1 to 3, higher to lower priority. In general, priority 1 means warnings that should be corrected and probably represents a fault reviewer would like to correct; priority 2 represents not so critical warnings but that may be interesting to be analyzed after the ones with priority 1; and priority 3 means warnings related to coding style and are considered informational.

Additional file

Additional file 1: Original versus mutated program samples. (ZIP 2 kb)

Acknowledgements
The authors would like to thank the Brazilian Funding Agency: CAPES, CNPq and FAPESP. The authors would also like to thank the anonymous referees for their valuable comments.

Authors' contributions
CAA carried out the experiment and data collection. AMRV participated in the organization of the experimental study and proposed the use of mutantion testing to evaluate static analysis tools. JCM and MED participated in the work suggestions and helped in the revision of the manuscript. All authors contributed on data analysis, conclusions and future work sections. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Instituto de Informática, UFG, Alameda Palmeiras, Quadra D, Campus II, 74690-900 Goiânia, GO, Brazil. [2]Instituto de Ciências Matemáticas e de Computação, USP, Av. Trabalhador São Carlense, 400, 13566-590 São Carlos, SP, Brazil. [3]Departamento de Computação, UFSCar, Rod. Washington Luís, Km 235, 13565-905 São Carlos, SP, Brazil.

Received: 18 December 2015 Accepted: 8 October 2016
Published online: 03 November 2016

References
Acree AT, Budd TA, DeMillo RA, Lipton RJ, Sayward FG (1979) Mutation analysis. Tech rep DTIC Document
Andrews JH, Briand LC, Labiche Y (2005) Is mutation an appropriate tool for testing experiments. In: XXVII International Conference on Software Engineering – ICSE'05. ACM Press, New York. pp 402–411. doi:10.1145/1062455.1062530
Atlassian (2014) Jira. Tool's Homepage, available at: https://www.atlassian.com/software/jira/. Accessed 2 July 2014
Ayewah N, Pugh W, Morgenthaler JD, Penix J, Zhou Y (2007a) Evaluating static analysis defect warnings on production software. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07. ACM, New York. pp 1–8. doi:10.1145/1251535.1251536
Ayewah N, Pugh W, Morgenthaler JD, Penix J, Zhou Y (2007b) Using findbugs on production software. In: Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion, OOPSLA '07. ACM, New York. pp 805–806. doi:10.1145/1297846.1297897
Bugzilla (2014) Bugzilla – bug-tracking system. Tool's Homepage, available at: http://www.bugzilla.org/. Accessed 2 July 2014
Burn O (2014) Checkstyle – coding standard verifier. Tool's Homepage, available at: https://github.com/checkstyle/checkstyle. Accessed 2 July 2014
Coles H (2015) Pitest: real world mutation testing. Web page. http://pitest.org/. Last access: Accessed 2 July 2014
Copeland L (2004) A practitioner's guide to software test design. Artech House
Copeland T (2005) PMD Applied: An Easy-to-use Guide for Developers. An easy-to-use guide for developers, Centennial Books
Couto C, Montandon JaE, Silva C, Valente MT (2013) Static correspondence and correlation between field defects and warnings reported by a bug finding tool. Softw Qual J 21(2):241–257. doi:10.1007/s11219-011-9172-5
Daimi K, Banitaan S, Liszka K (2013) Examining the performance of java static analyzers. In: XI International Conference on Software Engineering Research and Practice – SERP'13. CSREA Press, Las Vegas. pp 225–230
Dallmeier V, Zimmermann T (2007) Extraction of bug localization benchmarks from history. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE'07. ACM, New York. pp 433–436. doi:10.1145/1321631.1321702
de Araújo Filho JE, de Moura Couto CF, de Souza SJ, Valente MT (2010) A Study on the Correlation Between Field Defects and Warnings Reported by a Static Analysis Tool. In: IX Brazilian Symposium on Software Quality – SBQS'2010. SBC, Belém. pp 9–23
DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: Help for the practicing programmer. Computer 11(4):34–43
Evans D, Larochelle D (2002) Improving security using extensible lightweight static analysis. IEEE Softw 19(1):42–51. doi:10.1109/52.976940
Ferrari FC, Nakagawa EY, Maldonado JC, Rashid A (2011) Proteum/AJ: A mutation system for AspectJ programs. In: X International Conference on Aspect-oriented Software Development Companion – AOSD'11. ACM, New York. pp 73–74. doi:10.1145/1960314.1960340

Hovemeyer D, Pugh W (2004) Finding bugs is easy. SIGPLAN Not 39(12):92–106. doi:10.1145/1052883.1052895
IEEE (1990) IEEE Standard Glossary of Software Engineering Terminology. IEEE Standards Board, New York
Just R, Schweiggert F, Kapfhammer GM (2011) Major: An efficient and extensible tool for mutation analysis in a java compiler. In: XXVI IEEE/ACM International Conference on Automated Software Engineering – ASE'11. IEEE Computer Society, Washington, DC. pp 612–615. doi:10.1109/ASE.2011.6100138
Louridas P (2006) Static code analysis. IEEE Softw 23(4):58–61. doi:10.1109/MS.2006.114
Ma YS, Offutt J (2005) Description of method-level mutation operators for Java. On-line document, available at: http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf. Accessed 10 Aug 2015
Ma YS, Offutt J, Kwon YR (2005) Mujava: an automated class mutation system: Research articles. Softw Test Verification Reliab 15(2):97–133. doi:10.1002/stvr.v15:2
Mathur AP (1991) Performance, effectiveness, and reliability issues in software testing. In: Computer Software and Applications Conference, 1991. COMPSAC'91., Proceedings of the Fifteenth Annual International. IEEE, New York. pp 604–605. 10.1109/CMPSAC.1991.170248
Microsoft (2014) StyleCop. Tool Homepage, available at: https://stylecop.codeplex.com/. Accessed 2 July 2014
Mresa E, Bottaci L (1999) Efficiency of mutation operators and selective mutation strategies: an empirical study. J Softw Test Verification Reliab 9(4):205–232
Nagappan N, Ball T (2005) Static analysis tools as early indicators of pre-release defect density. In: Proceedings of the 27th International Conference on Software Engineering, ICSE '05. ACM, New York. pp 580–586. doi:10.1145/1062455.1062558
Offutt AJ, Rothermel G, Zapf C (1993) An experimental evaluation of selective mutation. In: 15th International Conference on Software Engineering. IEEE Computer Society Press, Baltimore. pp 100–107
Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C (1996) An experimental determination of sufficient mutant operators. ACM Trans Softw Eng Methodol 5(2):99–118
Offutt J, Ma YS, Kwon YR (2006) The class-level mutants of mujava. In: Proceedings of the 2006 International Workshop on Automation of Software Test, AST '06. ACM, New York. pp 78–84. doi:10.1145/1138929.1138945
Pohl J (2001) Lint - C program verifier. Tool's Man Page, available at: http://www.unix.com/man-page/FreeBSD/1/lint. Access on: 2 July 2014
Polo M, Piattini M, García-Rodríguez I (2009) Decreasing the cost of mutation testing with second-order mutants. Softw Test Verification Reliab 19(2):111–131. doi:10.1002/stvr.v19:2
Shen H, Fang J, Zhao J (2011) Efindbugs: Effective error ranking for findbugs. In: Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST '11. IEEE Computer Society, Washington, DC. pp 299–308. doi:10.1109/ICST.2011.51
Tomas P, Escalona MJ, Mejias M (2013) Open Source Tools for Measuring the Internal Quality of Java Software Products. A Survey. Computer Standards & Interfaces 36(1):244–255. 10.1016/j.csi.2013.08.006
Vincenzi AMR, Nakagawa EY, Maldonado JC, Delamaro ME, Romero RAF (2002) Bayesian-learning based guidelines to determine equivalent mutants. Int J Softw Eng Knowl Eng 12(06):675–689