An Experimental Protocol for Analyzing the Accuracy of Software Error Impact Analysis

Vincenzo Musco*, Martin Monperrus†, Philippe Preux‡
University of Lille, CRIStAL and INRIA
*Email: vincenzo.musco@inria.fr
†Email: martin.monperrus@univ-lille1.fr
‡Email: philippe.preux@univ-lille3.fr

***Abstract*—In software engineering, error impact analysis consists in predicting the software elements (*e.g.* modules, classes, methods) potentially impacted by a change. Impact analysis is required to optimize the testing effort. In this paper we present a new protocol to analyze the accuracy of impact analysis. This protocol uses mutation testing to simulate changes that introduce errors. To this end, we introduce a variant of call graphs we name the "use graph" of a software which may be computed efficiently. We apply this protocol to two open-source projects and correctly predict the impact of 30% to 49% of changes.**

I. Introduction

Software continuously evolves by changes affecting some component, such as a module, a classes, a function. A single change to a source code can impact the entire software package, and may break many parts beyond the changed element. *E.g.* a change in a file reading function can impact any class that uses it to read files. The literature discusses techniques to reason on the impact of a given change (see *e.g.* [1]).

As presented by Law and Rothermel [1], error impact analysis can be done with either static call graphs or dynamic traces. These authors claim that static call graphs are inaccurate because they may return excessively large sets of impacted elements. However, we have no clear evidence that this claim has been challenged empirically. This is the problem we address in this paper: we propose an experimental protocol to numerically assess static error impact analysis.

We present an experimental protocol to measure the accuracy of impact analysis. It is inspired from mutation testing. We consider a software equipped with a set of tests, and we introduce changes in the software. Running the software, some tests fail; we consider the set of such failing tests as being the ground truth that we have to be able to predict. To make this prediction, we consider a specific kind of dependency graph that we call the "use graph" of the software under study. Use graphs are simple static call graphs which also include field usages. Intuitively, if a node uses another faulty node, the error propagates and itself becomes faulty. The output of the protocol is the number of correctly predicted impacted nodes, as well as the number of false positives and false negatives. These figures let us estimate the accuracy of our procedure.

We use this protocol on two mainstream open-source software packages: Apache Commons Lang and Apache Commons Collections. We compare the predictions made by our protocol with the predictions made by classical test suites. We correctly predict the impact of software mutation for up to 49% of the injected errors.

To sum up, our contributions are:

- an algorithm to numerically analyze the accuracy of error impact analysis;
- a visualization of error impact;
- a study of error impact on two large open-source software packages totaling 100k+ lines of code;
- three explanations on the weaknesses of basic use graphs to correctly estimate error propagation: 1) no inclusion of calls made using Java reflexion; 2) bad boundary between tests and application functions; 3) incomplete use graph building process.

The remainder of this paper is structured as follows. Section II defines our protocol, Section III presents our experiments and results, Section IV discusses the related work and Section V concludes this paper.

II. Methodology

A. Protocol

In this paper, we want to perform static *error impact analysis*: considering a software change, we want to determine its impact on the entire software application, *i.e.* determine the parts, other than the one containing the error, to which the error propagates. To this end, we introduce a new graph that represents interactions between software elements; we name this graph the "use graph" of a software. Let us denote this graph by G. G is directed; a node represents either a method, or a field. There is an edge between a method node m and a method m' if m calls m' (the call may be recursive); there is an edge between a method node and a field node if this field is used in the method. In a use graph, field nodes are thus sinks.

We investigate how accurately we may predict the propagation of changes in a software using its use graph. Given the use graph of a software, we compute the set of nodes J which may be impacted by this change. To do so, we compute the set of nodes that may be reached from the changed node by following the use graph edges in the reverse direction. We use the technique known as "mutation injection" to simulate changes. A mutation injection is a source code modification that is made on purpose to track its consequences.

Algorithm 1: Compute the accuracy of impact analysis based on mutation.

Input: S the software package. m_{op} the mutation operator.

Output: I, I^+, I^- and I^o (as described in Section II-A) for each e and m.

```
1  begin
2      G ← useGraph(S);
3      T ← testCases(S);
4      for each e in filterElements(S, m_op) do
5          for each m in mutants(S, e, m_op) do
6              if m compiles then
7                  J ← impactedTests(m, G);
8                  F ← failingTests(m, T);
9                  if J = F then
10                     I ← I ∪ {m};
11                 else if F ⊂ J then
12                     I+ ← I+ ∪ {m};
13                 else if J ⊂ F then
14                     I− ← I− ∪ {m};
15                 else
16                     Io ← Io ∪ {m};
```

Because the use graph provides a static view of the software, the use graph only approximates the change propagation graph. Since it is static, it includes all possible connections, some of which may be ignored using a dynamic analysis. For instance, if a change propagates only if $x = 0$, this is not always known statically. The use graph is similar to the dependency graph extracted by the widely used tool Dependency Finder [1] at the "feature" level (methods and fields).

To evaluate the accuracy of our approach, we define and compute the following sets:

(i) the set I composed of changes that return the same set of tests no matter what technique is used;
(ii) the set I^+ composed of changes that return some extra tests using the use graph;
(iii) the set I^- composed of changes that return some extra tests using the test suite execution;
(iv) the set I^o composed of changes that return extra tests on both.

Algorithm 1 shows our approach. This algorithm inputs a software package to consider for impact analysis and the mutation operation that is responsible for mutation injection. The output is the subsets I, I^+, I^- and I^o described above. In line 2, we compute the use graph of the software under study. In line 3, we get the set of test cases from the input software S (`testCases`). In lines 4–6, we select (`filterElements`), mutate (`mutants`) and test the appropriate elements in the software. Appropriate elements are syntactic entities to which the specific change can be applied. In line 7, we determine the nodes impacted by the mutation (`impactedTests`) according to G. In Line 8, the function `failingTests` returns the set of tests that fail when running the mutated version of the software. In lines 9–16, we determine, for each mutant, if more tests are determined with the use graph (line 12), by test suite execution (line 14), by both (line 16) or if they are the same (line 10).

[1] http://depfind.sourceforge.net/

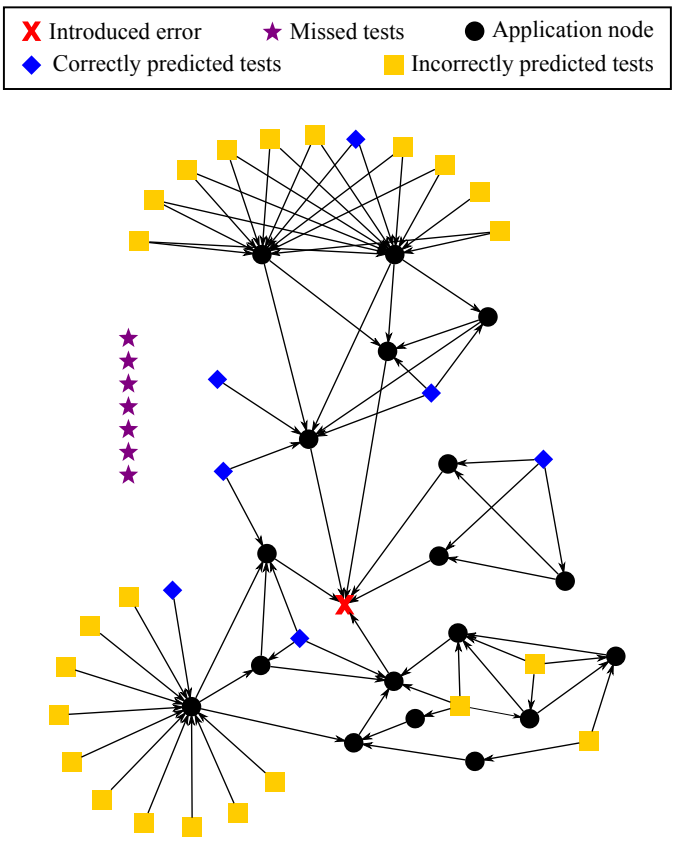

Fig. 1. This figure illustrates the effect of a particular mutation in the Apache.Commons.lang project. Only the interesting part of the use graph is represented here; the use graph is much larger, made of 5406 nodes. Nodes in black are the nodes that propagate the mutation injected in the node denoted with a red cross. Blue, yellow, and purple nodes are tests related to the injected mutation. Blue nodes are tests that are correctly predicted as impacted by the injected mutation; these are true positives. Yellow nodes are tests that are predicted as impacted, but are not; these are false positives. Purple nodes are tests that should have been predicted as being impacted but have not been; these are false negatives.

B. Analysis Metrics

In this section we present the metrics we use in this paper. Considering all changes for a specific project and a mutation operator, we can compute $|I|$, $|I^+|$, $|I^-|$, $|I^o|$ as the size of the respective sets I, I^+, I^-, I^o. It is useful to normalise these numbers, to get a feeling of the accuracy of the approach based on the use graph. So we provide the ratio between these sizes and the number of changes. $|I|+|I^+|$ is the sum of the size of the two sets I and I^+. Δ_1 and Δ_2 are respectively the median number of extra tests returned by use graph and missing tests returned by test suite execution for all changes.

TABLE I
STATISTICS ABOUT PROJECTS CONSIDERED IN THIS PAPER AND THEIR USE GRAPHS.

Project	Version	LOC	# of Nodes	# of Edges	Time	# of Tests	Time per test
Apache Commons Lang	3.1	52841	5406	10418	8.0s	2015	7ms
Apache Commons Collections	4.1	55081	6680	13478	9.7s	367	49ms

C. Visualization

We propose a visualization of error propagation: it provides developers with an idea of the potential dynamics and the complexity of the impact of a software mutation. Figure 1 illustrates an error-introducing change in Apache Commons Lang. Each node represents either a method or a field, and each edge represents a call to a method or an access to a field. The red cross is the node where the mutation occurs. Purple stars are missed tests (*i.e.* detected only by the test suite execution), yellow boxes are incorrectly predicted tests (*i.e.* detected only by the use graph), blue diamonds are correctly predicted tests (*i.e.* found by both techniques) and black circles are application nodes. As an example, the graph illustrated on Figure 1 is composed of 55 nodes with 6 correctly predicted, 7 missed tests, 23 incorrectly predicted and 18 applications nodes. As there are missed and incorrectly predicted tests, this error-introducing change belongs to the I^o set.

Those nodes categories are explained in details in section II-A. As we can see on the example, we can notice the multiple propagation paths which exist from the error-introducing change.

D. Research Questions

In this section we present the research questions to which we want to answer in this paper.

Research Question 1 *Is a basic use graph a good candidate for error impact analysis?* Since the use graph of a software is rather cheap and fast to compute, we want to know whether it can be reliably used as a preliminary impact assessment, before using more computationally intensive techniques. A use graph allows one to prioritize test cases to execute when a change occurs; a use graph can also be used to locate all methods that have to be reviewed after a change happened in the software.

Research Question 2 *Is error impact analysis project-dependent or change-dependent?* Answering to this question allows us to determine the level of genericity of use graph.

Research Question 3 *How rich are test suite scenarios?* In our context, the richness of tests relate to the number of program elements (say methods) involved in testing. The answer would help us to determine the directions we should take for our future investigations.

Research Question 4 *What are the reasons of a bad accuracy of impact analysis based on use graphs as defined in this paper?* To answer this question, we manually investigate some cases where there is no perfect match to determine the reasons leading to bad accuracy.

III. EXPERIMENTAL RESULTS

A. Setup

In this paper, we consider a dataset composed of two *Java* software packages: *Apache Commons Lang* 3.1 and *Apache Commons Collections* 4.1. Table I reports the key descriptive statistics about these projects:

- the first and second columns give the name and the version of the software being analysed;
- the third column indicates the number of lines of code of the project (computed using *cloc* [2]);
- the fourth and fifth columns give the number of nodes and the number of edges of the use graph;
- the sixth column gives the time required to build the use graph;
- the seventh column gives the number of test cases comprised in the project;
- the eighth column indicates the average time for a single test execution[3].

Our technique requires mutation operators, in this experiment, we choose the following two: (i) *if-logic* mutation that consists in changing logical operators in an `if` test (==, !=, <, >, <= and >=); (ii) *return-value* mutation consists in changing a `return` statement where the return value is changed according to its type (*e.g.* a non null integer returns a zero value).

We implement a program to extract the use graph using Spoon[4], an open-source library for analyzing and transforming Java source code.

B. Research Questions

Table II presents the accuracy obtained with the use graph for error impact analysis. The first and second columns give the project name and the mutation operator considered, the third and fourth column are the median and the maximum

[2]http://cloc.sourceforge.net/

[3]time to execute anything obviously depends on many things. Timings are reported here to give the reader a feeling about them. All experiments were made on a HP EliteBook 8570w Mobile Workstation, i7-3740QM quad core, 2.7Ghz, under Xubuntu.

[4]http://spoon.gforge.inria.fr/

value of the number of impacted nodes. The remaining columns are described in section II-B.

Research Question 1 Is a basic use graph a good candidate for error impact analysis?

Based on the limited experimental evidences presented here, the answer to this question seems affirmative. As we can see on table II, in 30 to 50% of cases, use graph can obtain a perfect match (I), which means the test set retrieved by use graph is exactly the same as the one returned by test suite execution. We consider this as a good result given that the technique is very light: as reported in Table I, the use graph construction takes no more than 10 seconds to be generated (last column). If we consider completeness (all impacted tests are predicted), Apache Commons Lang gives more encouraging values: 91% of cases returned by use graph includes all failing tests. Last, we can notice the median number of extra or missing tests are around 3.5 to 14 for Apache Commons Lang and 2 to 4 for Apache Commons Collections which represents around 1% to 7% of total cases. According to the average execution time of a test (see Table I) , the cost in time to execute those erroneously predicted tests is much less than 1 second (around 21-196ms). Again, to us, the lack of precision is small for such a computationally light technique. Still, we must nuance our answer as the sets of false positives (I^+) and false negatives (I^-) can not be neglected.

Research Question 2 Is error impact analysis project-dependent or change-dependent?

As reported in Table II: (i) the values differ strongly from a project to another for a same mutation operator (*e.g.* considering the if-logic mutation operator, 45.3% in I^+ for Apache Commons Lang versus 1.3% for Apache Commons Collections); (ii) the values differ less from a mutation operator to another for a same project (*e.g.* considering the Apache Commons Lang project, 45.3% in I^+ for if-logic mutation operator versus 42.0% for the return-value).

This indicates that the use graph is more project dependent than error-family dependent. Clearly, a stronger answer to this question requires more investigation as the dataset and the number of operators are small.

Research Question 3 How rich are test suite scenarios?

Figure 2 plots the distribution of the number of impacted nodes. Most graphs have a size smaller or equal to 10. However, there exist complex test scenarios, with much more than the median value of 6 impacted nodes; one impacts 344 nodes. If we look again at the example on Figure 1, the size of the graph is 55 nodes, and the impact propagation is not straightforward.

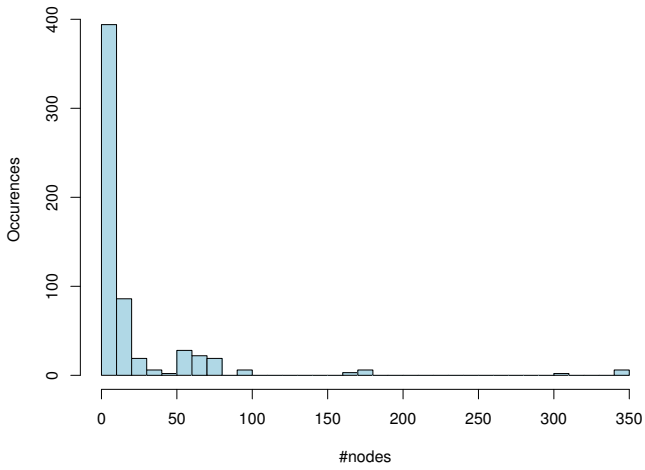

Fig. 2. Distribution of the number of impacted nodes.

Research Question 4 What are the reasons of a bad accuracy of impact analysis based on use graphs as defined in this paper?

To investigate this point, we pick at random three error-introducing changes in sets I^+, in I^- and I^o.

For the error-introducing change in set I^+ we take the case in Apache Commons Lang with a if-logic mutation. It shows an important use of Java Reflection mechanism. We are not able to handle these cases with the use graph as reflection is resolved at runtime. Nevertheless, even if the problem would not be handled, we can easily detect its uses as it refers to some specific classes/packages in Java. A warning can be raised to the user about the use of reflection mechanism so that special care could be taken when dealing with those methods.

For the error-introducing change in set I^- we analyze cases in Apache Commons Collections with a return-value mutation. We notice that the mutation occurs in an abstract test in which the call to the error-introducing change node occurs. Mutations are not expected to be done in test code, those cases highlight the difficulty to determine the boundary of test cases in a software, which turns out to be harder than expected.

For the error-introducing change in set I^o we take a case in Apache Commons Lang with a return-value mutation. This case suffers from another problem: the use of the Java notation for arbitrary number of parameters (*i.e.* `method(Object...)`). This is not correctly resolved when called with something such: `method(obj_array)` or even `method(obj1, obj2)` where `obj_array` is an array of `Objects`, `obj1` and `obj2` are `Objects`. A specific processing in the construction of the graph is required to handle this special case.

TABLE II
THE PERFORMANCE OF IMPACT ANALYSIS BASED ON LIGHTWEIGHT CALL GRAPHS ON TWO JAVA SOFTWARE PACKAGES. I VALUES ARE PROPORTIONS OF THE 600 MUTANTS GENERATED FOR EACH.

		Graph size								
Project	Mutation op.	median	maximum	$\|I\|(\%)$	$\|I^+\|(\%)$	Δ_1	$\|I\|+\|I^+\|(\%)$	$\|I^-\|(\%)$	Δ_2	$\|I^o\|(\%)$
Lang	if-logic	6	344	35.8%	45.3%	3.5	81.2%	12.0%	14	6.8%
	return-value	3	344	49.3%	42.0%	14	91.3%	7.5%	3	1.2%
Collections	if-logic	1	57	30.2%	1.3%	4	31.5%	64.2%	2	4.3%
	return-value	1	47	34.5%	2.7%	2	37.2%	62.3%	2	0.5%

C. Discussion

At a conceptual level, the main threat to the validity of our experimental results is that we consider the test suite execution as ground truth. However, it may be the case that the test cases miss the assertions that would detect the propagated error and thus fail. This threat is mitigated by manual analysis.

At the experimental level, we have yet used only a very limited set of software and of software changes. The experimental investigation should be made on a much larger set of cases in order to provide statistically significant conclusions about the relevance of the use graph approach to study the propagation of software changes.

IV. RELATED WORKS

Challet and Lombardoni [2] proposed a theoretical reflexion about impact analysis using graphs. However, they do not evaluate the validity of their "bug basins" as we do in this paper.

Law and Rothermel [1] proposed an approach for impact analysis; their technique is based on a code instrumentation to analyze execution stack traces. They compare their technique against simple call-graphs on one small software subject. On the contrary we apply our technique to two large software applications.

Robillard and Murphy [3] introduced "concern graphs" for reasoning on the implementation of features. This kind of graphs may be assessed with the protocol we have presented here.

Michael and Jones [4] alter variables during the program's execution in order to study the perturbations of the software. They focus on data-state perturbation, the if-logic mutation is another kind of perturbation: "control perturbation".

Murphy *e*t al. [5] studied nine tools for static call-graph extraction. They stated that each of them has particularities in how each part is treated by the tool. Use graphs are a specific type of call graphs. A key difference is that Murphy *e*t al. do not investigate error propagation.

V. CONCLUSION

In this paper, we aimed at assessing experimentally the accuracy of static error impact analysis. We presented a new protocol for this based on mutation testing. We discussed a set of preliminary experimental results on impact analysis in two open-source software applications. Our experiments show that use graphs enable a perfect impact prediction for 30%—49% of simulated changes. However, more experiments are required to confirm this finding. Future work will improve the use graphs to perform better impact analysis.

REFERENCES

[1] J. Law and G. Rothermel, "Whole Program Path-Based Dynamic Impact Analysis," in *Proceedings of the 25th International Conference on Software Engineering*, May 2003, pp. 308–318.

[2] D. Challet and A. Lombardoni, "Bug Propagation and Debugging in Asymmetric Software Structures," *Physical Review E*, vol. 70, p. 046109, Oct 2004.

[3] M. P. Robillard and G. C. Murphy, "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies," in *Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002, pp. 406–416.

[4] C. C. Michael and R. C. Jones, "On the Uniformity of Error Propagation in Software," in *Proceedings of the 12th Annual Conference on Computer Assurance*, June 1997, pp. 68–76.

[5] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An Empirical Study of Static Call Graph Extractors," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 2, pp. 158–191, Apr 1998.