The Hybrid Technique for Object-Oriented Software Change Impact Analysis

Conference Paper *in* Proceedings of the Euromicro Conference on Software Maintenance and Reengineering, CSMR · March 2010

DOI: 10.1109/CSMR.2010.48 · Source: DBLP

4 authors:

Mirna Maia
Universidade Federal de Campina Grande (UFCG)
3 PUBLICATIONS **29** CITATIONS
SEE PROFILE

Roberto Bittencourt
Universidade Estadual de Feira de Santana
86 PUBLICATIONS **297** CITATIONS
SEE PROFILE

Jorge Figueiredo
Universidade Federal de Campina Grande (UFCG)
93 PUBLICATIONS **538** CITATIONS
SEE PROFILE

Dalton Serey
Universidade Federal de Campina Grande (UFCG)
50 PUBLICATIONS **363** CITATIONS
SEE PROFILE

The Hybrid Technique for Object-Oriented Software Change Impact Analysis

Mirna Carelli Oliveira Maia*, Roberto Almeida Bittencourt*†,
Jorge Cesar Abrantes de Figueiredo* and Dalton Dario Serey Guerrero*
**Department of Systems and Computing*
UFCG - Federal University of Campina Grande
Campina Grande, Brazil
Email: {mirna,dalton,abrantes}@dsc.ufcg.edu.br
†*UEFS - State University of Feira de Santana*
Feira de Santana, Brazil
Email: roberto@uefs.br

***Abstract*—Change impact analysis techniques that underestimate impact may cause important financial losses from the point of view of an IT services company. Thus, reducing false-negatives in these techniques is a goal with strong practical relevance. This work presents a technique that uses both static and dynamic analysis of object-oriented source code to improve resulting impact estimates in terms of recall. The technique consists of three steps: static analysis to identify structural dependencies between code entities, dynamic analysis to identify dependencies based on a succession relation derived from execution traces, and a ranking of results from both analyses that takes into account the relevance of dynamic dependencies. Evaluation was performed through prototype development and a multiple-case quantitative case study that compared our solution against a static technique and a dynamic one. Results showed that our hybrid technique improved recall between 90 and 115% compared to the static technique, and between 21.2 and 39% compared to the dynamic one.**

I. Introduction

Impact analysis techniques try to estimate the change impact from the analysis of software artifacts. Among other approaches, static and dynamic analysis are the most popular techniques in the literature [1], [2].

Current techniques may be inaccurate in that they either underestimate or overestimate change impact [1]. Figure 1 illustrates these issues. Starting from a change C, the aim of an impact analysis technique is finding the part of the software truly affected by the change (set A) . Inaccuracy of the technique produces the change impact set I, which may not fully match A. Overestimating impact generates false-positives, which may confuse the analyst because it contains unnecessary information. On the other hand, underestimating impact produces false-negatives that, when kept unchanged, may negatively affect the changed software, usually in the form of bugs that need to be fixed. Consequently, false-negatives may cause important financial losses to the software company.

In this paper, we present a hybrid impact analysis technique based on both static and dynamic analysis that aims to reduce the number of false-negatives and to present results according to their relevance to the change. In our solution, impact analysis is performed in three steps: (i) static analysis identifies structural dependencies between code entities; (ii) dynamic analysis identifies dependencies based on execute-after sequences from software traces; (iii) results from both analyses are joined and ranked according to their relevance.

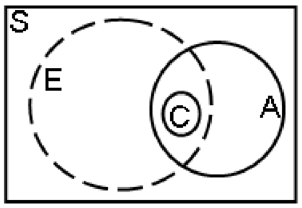

Figure 1. The impact of a change C in a software S

During evaluation, we compared our technique against a purely static technique and a purely dynamic technique in a multiple-case case study. Metrics for comparison were: number of false-negatives, number of false-positives, precision and recall. Our technique improved recall between 90 and 115% compared to the static technique. When compared to the dynamic technique, our technique improved recall between 21.2 and 39%. Thus, reduction of false-negatives was achieved when comparing our approach to the other solutions.

II. Background

Change impact analysis is a process that aids software engineers to identify the consequences of software changes.

From an impact analysis viewpoint, software entities are pieces of design or code that may undergo changes. In object-oriented software, methods/functions and fields are usually the entities at the granularity that is important to analysis. Hence, from an initial change set of entities, the challenge is finding out additional entities affected by the change in this initial set. The extended set with initial and affected entities is called the impact set. To find out these additional entities, relations between entities are used.

Our work builds on static and dynamic impact analysis techniques. In the **static approach**, structural dependencies are identified between software entities from an static representation of the software, usually extracted from source code. In the **dynamic approach**, dynamic analysis is done by instrumenting source code to record program event traces. Generally, events are method/function calls and returns and variable reads or writes. Temporal relations between events in traces are used to find out impacted entities based on system behavior.

Both static and dynamic approaches produce errors when identifying the impact set. Let us define *estimated entities* as the set of entities obtained by an impact analysis technique, and *affected entities* as the set of entities truly affected by a given change. If the analysis results contain a specific entity that was not truly affected, an error was made that is called a **false-positive**. Oppositely, when the analysis results do not contain a truly affected entity, an error was made that is called a **false-negative**. Precision and recall are two relative metrics respectively associated to false-positives and false-negatives. These metrics are popular in the information retrieval domain and we shall restate them here to the impact analysis domain [3]. **Precision** is the ratio between correctly estimated entities and the total of estimated entities: $Precision = \frac{|A \cap I|}{|I|}$. **Recall**, on the other hand, is the ratio between correctly estimated entities and the total of affected entities: $Recall = \frac{|A \cap I|}{|A|}$.

III. The Hybrid Impact Analysis Technique

We propose a pre-change hybrid impact analysis technique that combines both static and dynamic analysis and is aimed at reducing the number of false-negatives. Our solution focus on object-oriented systems, with changes described at source code level.

Figure 2 shows an overview of the hybrid technique. In short, our technique consists of three steps: (i) a depth-first search based on the static analysis of a structural program graph to determine an initial impact set; (ii) dynamic analysis of software traces to identify additional entities for the impact set and dynamic dependencies between entities based on the frequency of succession relations, and (iii) ranking the impact set entities based on an impact factor determined from dynamic analysis. The process is outlined in Figure 2. Change types may be additions, removals and modifications to code entities planned by software engineers. Entities are code entities present in object-oriented code such as classes, methods and fields.

A. Static Analysis

The first step of the hybrid technique is the static analysis of source code to identify structural dependencies. A program graph is extracted from source code, with vertices representing code entities and edges meaning structural dependencies between them. From the entities in the change set, we perform a depth-first search to find other entities that depend on them. We customize the search with a depth parameter (stop condition), to reduce the false-positives [3].

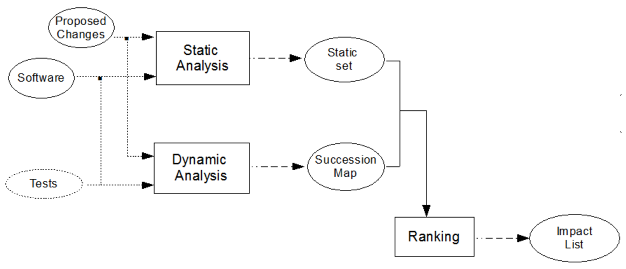

Figure 2. Overview of Hybrid Technique

B. Dynamic Analysis

Dynamic analysis is the second step, and consists of program execution, data collection and data analysis. During program runtime, event traces related to accessed code entities are collected in the order events occur. We take into account events specifically related to methods and fields: method entry, method return, field access and field write.

Event traces are then analyzed to discover dynamic dependencies. There is a dynamic dependency between two entities if events from both entities happen in succession. More specifically, the succession relation is determined by the event order associated to trace entities. Consider that E^1 and E^2 are events of entity 1 and entity 2, respectively. E^2 is successor of E^1 if there is a trace where E^2 appears after E^1. In other words, if E^1 and E^2 are events of a trace T, E^2 is successor of E^1 if and only if $\exists T | T = t_1 E^1 t_2 E^2 t_3, \forall t_1 t_2 t_3$, where t_1, t_2 and t_3 are subtraces from trace T.

From the definition above, we identify the sucessors for each event and count their occurrences. This value informs the analyst how likely a successor is dependent on the change set and it is stored in a structure called succession map. Data in this map are used to rank results of both static and dynamic analysis in the following step.

Not all successors are equally important for the dynamic analysis. It usually happens that farther successors may be repeated occurrences of closer successors. Thus, we define k-successor as the k^{th} successor of an entity, following the regular event order. For instance, suppose the trace: A_E B_E B_R A_R C_E F_W C_R B_E C_E C_R B_R B_E C_E C_R F_A B_R. In the first occurrence of a B_E method entry, B_R is a 0-successor of B_E, A_R is a 1-successor of B_E, C_E is a 2-successor of B_E, and so on. From the statistics of the successors of B_R in Figure 3, we notice that the closer successors are indeed the most likely to be affected by a change. Hence, we define a succession distance parameter as a stop condition to the process of populating the succession map, used to reduce the number of false-positives.

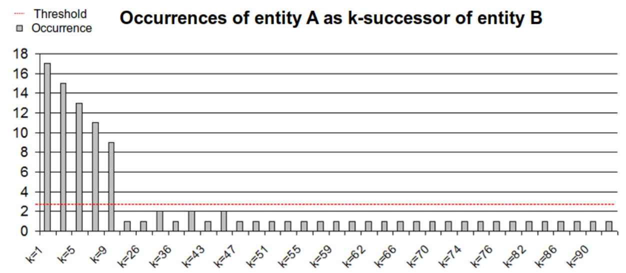

Figure 3. Succession distance parameter for dynamic analysis

C. Ranking

The hybrid technique finishes with the ranking step, which combines the impact set obtained in the previous steps and ranks their entities according to dynamic analysis. To rank this set, we define and use an impact factor, which represents the relevance of an entity to a change. It tries to capture the potential impact a change might cause in a software entity, based on the dynamic dependencies.

The value of the impact factor for an entity A is based on the probability of an event of A being a sucessor of an event of a change set entity. Computing an estimate of this probability is easily done from the succession map derived at the end of the dynamic analysis. Dynamic analysis offers the succession of events as a clue to the importance of an entity to another entity. It is likely that an entity may depend on a predecessor entity, since an entity event can change system state and, thus, affect its successors.

IV. Evaluation

We evaluated our technique regarding two aspects: feasibility and effectiveness. For feasibility, we successfully developed a prototype tool called SD-Impala that implements our hybrid technique. To evaluate effectiveness, we conducted an empirical multiple-case quantitative case study with four evolving software systems and a toy example.

A. Experimental Design

For five different software systems, we recorded the changes that should be implemented in a follow-up release. For each proposed change, we performed impact analysis using our technique and two additional ones: CollectEA, a dynamic technique [1], and Impala, a static technique [3]. To compare them, we extracted from the repository two consecutive versions of each software system: the first, before implementing changes, and the second, after implementing changes. Using a diff tool, we derived the true impact set for each change, used as an oracle in the comparison.

We selected five software systems under maintenance: Design Wizard [1], Design Model [2], Design Abstractor [3], Impala [4] and a toy example [5]. Table I shows some features of the selected projects.

	Design Wizard	Design Model	Design Abstractor	Impala	Toy Example
LOC	5825	2520	5969	5176	678
# classes	67	18	77	57	20
# events	49723968	103793	5913996	1537684	1006
Coverage	57.2%	52.49%	64.80%	63.63%	84.22%

Table I
Characteristics of the selected projects

Parameters from the competing techniques are similar to the parameters in our technique, which facilitates comparison. Impala and SD-Impala have a parameter called depth that defines the scope for the static depth-first search. SD-Impala was parameterized by the distance, a parameter that represents the number of successors taken into account during dynamic analysis.

After extracting the execution traces and the program graph from the software systems, we tuned the parameters to obtain better accuracy and devised three different scenarios for comparison: (i) depth=5 and distance=100; (ii) depth=10 and distance=500, and (iii) depth=20 and distance=1000.

B. Results

We compared results by measuring the number of false-positives, the number of false-negatives, precision and recall (see section II) for each technique and for each change in a particular software system. We plotted graphs showing the average results for the five software systems, for each impact analysys technique in each of the experimental scenarios. Table II shows results in terms of average recall and average precision.

Metric	Scenario	Impala	CollectEA	SD-Impala
Precision	i	0.585	0.226	0.285
	ii	0.506	0.226	0.269
	iii	0.511	0.226	0.269
Recall	i	0.269	0.424	0.524
	ii	0.272	0.424	0.586
	iii	0.275	0.424	0.589

Table II
Average precision and recall per technique and scenario

C. Analysis

Our technique produced less false-negatives and better average recall than both competing techniques (see Table II). Compared to the static technique, SD-Impala increased recall between 90 and 115%. The same happened against CollectEA, with increase between 21.2 and 39%. Larger depth and larger distance slightly improved results for our technique. From the first to the third scenario, we obtained

[1]API to extract design information from Java programs
[2]In-memory graph model to represent software design
[3]API that abstracts design information from software designs
[4]Static impact analyzer tool
[5]Toy prototype used to validate our technique

an improvement of 14.6% in recall. Distance plays a more important role in that than depth, since changing depth for the purely static technique only improved 2.3% in recall.

On the other hand, the static technique showed superiority for false-positives and precision (Table II), while the dynamic technique showed the worst figures. Our technique showed results inbetween the competitors. Compared to the static technique, SD-Impala decreased precision between 58.4 and 61%. Against CollectEA, SD-Impala improved precision between 61.5 and 82%. Tuning the parameters with larger depth and larger distance worsened precision results for SD-Impala. From the first to the third scenario, results decreased 11.3% in precision. For the static technique, increasing depth reduced precision by 13.4%.

V. Related Work

Arnold and Bohner have pioneered the field of change impact analysis [2]. Since their seminal work, a host of impact analysis techniques have been developed, using relations from a wide range, from structural dependencies to dynamic relations in software traces, besides additional information from source code vocabulary and traceability between artifacts. Static program analysis techniques have been thoroughly applied to impact analysis. Structural dependencies between software entities are the basis for a number of techniques. An specific work on object-oriented systems was done by Lee et al., where they represent system entities in a program graph and estimate impact from static dependencies [4]. Hattori et al. improved this work with a tool called Impala, adding a depth parameter to reduce false-positives, and performed a better analysis in terms of information retrieval measures [3]. Our work adds to Impala not only the dynamic analysis, but also ranks results in terms of the likely importance of an entity in the impact set.

Dynamic analysis techniques have more recently been applied in impact analysis. One of these is CollectEA [1]. It is based on a binary relation called ExecuteAfter to estimate affected entities. Apiwattanapong et al. have compared CollectEA to other dynamic techniques in terms of precision and performance. Results showed that CollectEA was almost as efficient as the most efficient existing technique and as precise as the most precise existing technique [5].

VI. Conclusion

Impact analysis techniques are innacurate in that they usually overestimate impact, producing false-positives, or underestimate it, generating false-negatives. From the point of view of an IT services company, the former case is awkward, since it brings irrelevant data to analysis. At times, the company may be threatened to lose market share, since impact overestimation involves higher change costs. On the other hand, the latter case can be even worse. Underestimating change impact results in financial losses, since the IT company shall receive less money for a service than it actually requires. Thus, reducing false-negatives is a crucial aspect of an impact analysis process.

This work presents a technique to reduce the number of false-negatives in impact analysis of Java source code. Our solution combines program static and dynamic analysis into a hybrid technique and ranks results based on dynamic information.

We evaluated the effectiveness of our technique through a multiple-case quantitative case study. Results for our technique were compared against two other impact analysis techniques: a purely static technique (Impala) and a purely dynamic one (CollectEA).

Results showed that our technique produced better recall than the other techniques under scrutiny. Nonetheless, the static technique showed better figures of precision than our approach, which, by the way, were better than CollectEA. In our view, recall was the most important metric, since our aim was to reduce the number of false-negatives. Compared to the static technique, our approach increased recall between 90 and 115%. The improvement in recall was also visible against CollectEA, with increase between 21.2 and 39%.

Further work should focus on end-to-end impact analysis, reaching additional software artifacts such as software features and database descriptions.

References

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM, 2005, pp. 432–441.

[2] S. B. Robert Arnold, *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996.

[3] L. P. Hattori, D. Guerrero, J. Figueiredo, J. A. Brunet, and J. Damasio, "On the precision and accuracy of impact analysis techniques," in *ICIS '08: Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 513–518.

[4] M. Lee, A. J. Offutt, and R. T. Alexander, "Algorithmic analysis of the impacts of changes to object-oriented software," in *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 61.

[5] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 491–500.