On The Precision And Accuracy Of Impact Analysis Techniques

***Abstract*—Several techniques and algorithms for impact analysis of software systems have been recently published in literature. Most of them, however, are not practical enough to be applied in the software industry because, among other reasons, they produce too many false results (either positive or negative). In this paper, we propose and evaluate the use of two measures from information retrieval, namely *precision* and *recall*, to help express and compare precision and accuracy of impact analysis techniques and algorithms.**

I. INTRODUCTION

Software change impact analysis is the process of identifying the potential consequences (side-effects) of a proposed change. It can be used to estimate what has to be modified in an implementation to accomplish a change [1]. Current approaches are based on static or dynamic analysis of the program.

Static impact analysis [1]–[4] is usually based on slicing techniques and transitive closure on call graphs. Static techniques consider all possible executions of the program and produce too many false-positives, i.e., elements identified that are not impacted by the change. Dynamic analysis [5]–[7] calculates the impact of a change at run-time. They generate less false-positives than static analysis because they calculate the result based on possible executions of the program. In contrast, as they depend on the input used to execute the program, they usually miss on identifying some impacted elements - i.e., they produce false-negatives. Both static and dynamic impact analysis techniques produce false-positives and false-negatives. One example of false-negative in static analysis is the use of reflection in Java (very late binding) where information about what class is instantiated is known only at run-time.

Some of the current work on impact analysis [6], [8], [9] investigate how to assess the accuracy of existing techniques. The measures used, however, are either not formally defined or unsound. They assume that their algorithms are safe in terms of not excluding from the identified impact set any entity that is actually impacted [9]. However, they do not prove this assumption. They measure precision in terms of the number of methods in the impact sets against the total number of methods. They consider algorithm A to be more precise than algorithm B if the results of A are, overall, a subset of the results from B. However, they do not investigate if A has produced more false-negatives than B and, consequently, is more imprecise than B.

In this paper, we propose and evaluate the use of two measures from information retrieval, namely *precision* and *recall*, to express and compare the precision and accuracy of impact analysis techniques and algorithms. *Precision* measures the amount of identified elements that are not impacted (false-positives), and *recall* evaluates the elements that are not identified but are impacted (false-negatives). The combination of the two measures gives the accuracy result.

In order to evaluate our measures, we implemented a coarse-grained static impact analysis technique on a tool, called Impala. It calculates the impacted elements by identifying all the direct and indirect dependencies of a change (e.g. class A uses class B, so A depends directly on B). In order to reduce the amount of false-positives generated by static analysis, Impala's algorithms allow the adjustment of a stop point on the identification of indirect dependencies. In the empirical study, we assess the trade-offs of reducing false-positives and enlarging false-negatives of three software systems based on *precision* and *recall* results.

The remainder of this paper is organized as follows. Section II defines the measures *precision*, *recall* and *accuracy* to assess impact analysis techniques and algorithms. Section III describes the Impala tool and its algorithms. Section IV presents an empirical study conducted by applying Impala to two small and one medium size projects, including Impala's project. In section V we present and evaluate the results and discuss some research questions. We compare our measures with the ones proposed by other impact analysis techniques in section VI. Finally, we conclude our paper and discuss future work in section VII.

II. PRECISION AND RECALL

Before introducing *precision* and *recall* concepts, it is important to define what are false-positive and false-negative in the context of impact analysis.

False-positive. Given a set I of impacted elements identified by an impact analysis technique and a set M of the real modified elements, a false-positive is an element that is in I, but not in M.

$$\text{False-positive} = I - M$$

One concrete example of where it happens is when static impact analysis is used in object oriented software and a case of inheritance is involved. The class diagram from Figure 1 shows the relations among class A, interface B and its implementations B1, B2 and B3. Suppose a change in class A impacted in B, because B has a dependency

relationship with A. B is an interface and, probably, only one of its implementations will be impacted by a change in A. However, this is known only at run-time. Static impact techniques do not have the information of which, out of B1, B2 and B3 will be affected, because they analyze the source code or the bytecode. So they include all of them in the impact set. In this case, two out of B1, B2 and B3, are false-positives.

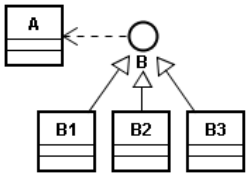

Fig. 1. Example of inheritance in an object oriented program.

False-negative. Given a set I of impacted elements identified by an impact analysis technique and a set M of the real modified elements, a false-positive is an element that is in M, but not in I.

$$\text{False-negative} = M - I$$

To exemplify, suppose all classes from the dependency graph from Figure 2 are impacted by a change. A dynamic technique is used to analyze the impact based on trace executions. This technique only analyzes two traces: {A,B,C} and {A,D,F}. One trace, {A,B,E}, is ignored, and the final change set is {A,B,C,D,F}. In this case, E is a false-negative, because it is an impacted element, but dynamic analysis missed on identifying it.

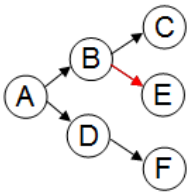

Fig. 2. Example of traces from dynamic analysis.

Now that should be clear what we call false-positives and false-negatives, we define our measures. *Precision* and *recall* are two measures from information retrieval [10] that we adapted for the impact analysis context.

Precision. The *precision P* indicates which fraction of the estimated impacted classes, I, was actually modified, M. This relation is defined by the following equation:

$$P = \frac{|I \cap M|}{|I|}$$

The relationship between false-positives and *precision* is: the fewer false-positives an impact analysis technique produces, the higher is *precision*.

Recall. The *recall R* describes the proportion of the modified classes, M, that were estimated, I. This relation is defined by:

$$R = \frac{|I \cap M|}{|M|}$$

The relationship between false-negative and *recall* is: the fewer false-negatives an impact analysis technique produces, the higher is *recall*.

Accuracy. The *accuracy A* measures the total error, considering both *precision* and *recall*. The calculation of accuracy is given by:

$$A = \frac{P+R}{2}$$

The goal of an impact analysis technique is to achieve high *precision* and high *recall*, which means that the technique estimated the impact of all classes actually modified and only them.

III. IMPALA TOOL

To evaluate the proposed measures, we conducted an empirical study using Impala, our change impact analysis tool that performs static analysis. In this section, we present the overall aspects of Impala and describe its static impact analysis algorithms.

Impala is a Java tool composed by a group of algorithms based on call graph dependencies that, given a set of changes, calculates the set of impacted elements. Impala analyzes Java systems at method level. In other words, Impala analyzes only the control flow of the system, ignoring information that can be obtained by data flow analysis.

Each change on the set of changes is composed by the *entity* to be modified and the *type of change*. We classify the types of changes as:

- add and remove class, field and method;
- change visibility of class, field and method;
- add and remove super/sub-types;
- change signature, semantics and return type of a method;
- change type or name of a field.

Impala uses the *DesignWizard* API [11] to extract a representation of the system to be analyzed as sets of entities and relations. An entity is defined as a class, a method or a field. Each relation connects two entities. They can be of the following types: *instanceof*, *contains*, *extends*, *implements*, *getstatic*, *putstatic*, *getfield*, *putfield*, *invokevirtual*, *invokespecial*, *invokestatic*, *invokeinterface*, *isinvokedby*, *isaccessedby*, *issuperclass*, *catch*, *throws*, *isdeclaredon*, *load*.

A. Impala's Algorithms

Impala is composed by a collection of algorithms that searches for impacts based on the type of each change in the set of changes. The analysis process consists of two steps: extracting a representation of the system in terms of entity sets and relations; and calculating the impacts given a set of changes. The impact analysis itself starts with `analyzeChanges` routine, shown in Pseudocode 1. It receives

as argument the change set and the depth. For each change, `analyzeChanges` invokes a specific algorithm that calculates the set of impacted entities according to the type of change. The impacted set of each change is included in `impactedSet`, which is the final result.

Pseudocode 1 Initial routine that calls impact analysis algorithms according to the change type.

```
 1 Enity[] analyzeChanges(changeSet[], depth)
 2   Entity entity
 3   Entity impactedSet[]
 4  Algorithm algorithm
 5   FOR each change in changeSet[] DO
 6     entity <- change.getEntity()
 7     algorithm <- getAlgorithm(entity.changeType)
 8     IF entity is not in impactedSet[] THEN
 9       include entity in impactedSet[]
10     ENDIF
11     Entity impactedChildren[]
12     impactedChildren[] <- algorithm.execute(entity,depth)
13     IF impactedChildren[] are not in impactedSet[]  THEN
14       include impactedChildren[] in impactedSet[]
15     ENDIF
16   ENDFOR
13   RETURN impactedSet[]
```

In line 7, there are ten algorithms that can be returned by `getAlgorithm(entity.changeType)`. These algorithms search for the dependencies that can be impacted by the change proposed according to their types. In order to reduce false-positives these algorithms are parameterized to stop the search on the entity dependency graph by using the depth as a criterion. The depth on a dependency graph represents the distance of one entity to another. For instance, if A calls B, B calls C, C calls D; distance from A to B is 1, from A to C is 2, and from A to D is 3. When depth is one, only direct dependencies will be returned as impacted entities. If depth is set to 3, two levels of indirect dependencies are considered together with direct ones. These algorithms are based on the assumption that the farther an entity is from a change, the less likely is its impact.

Due to space limitation, we are unable to show all the algorithms. In Pseudocode 2, we present the algorithm for the *remove* change type. The `remove()` algorithm was chosen because of its general aspects - most of the other algorithms derive from it - and the easiness to foresee what impacts an entity removal may cause. This algorithm recursively searches for indirect dependencies until there is no more dependency or depth is equal to zero.

When a class, a method or a field is removed, the `remove()` algorithm searches for their callers: classes that inherit from the removed class, and methods that use a field or call the removed method. First, it gets the direct callers, in line 8. Then, it recursively searches for indirect dependencies - lines 11 to 14.

The call `entity.getDirectCallers()` has different implementations for each entity type. This is due to the type of relations that can be associated to an entity. For a field, it can be accessed by a method. For a method, the logic is quite similar, and the callers of a class are a combination of the first two. Because of their similarities, we only show `getDirectCallers()` from the entity type field.

Pseudocode 2 Impact analysis algorithm from the *remove* change type.

```
 1 Entity[] execute(entity, depth)
 2   Entity entity
 3   Entity impactedSet[]
 4   IF (depth=0) THEN
 5     RETURN void
 6   ENDIF
 7   //Direct callers of the entity
 8   impactedSet[] <- entity.getDirectCallers()
 9   //Indirect callers
10   Entity indirectCallers[] <- 0
11   FOR each entity in impactedSet[] DO
12     //Gets the indirect callers
13     indirectCallers[] <- execute(entity, depth-1))
14   ENDFOR
15   IF indirectCallers[] are not in impactedSet[] THEN
16     include indirectCallers[] in impactedSet[]
17   ENDIF
18   return impactedSet[]
```

Pseudocode 3 Routine to get the callers of a field.

```
 1 Entity[] getDirectCallers()
 2   Entity callerSet[] <- 0
 3   //This relation: <field><IS_ACCESSEDBY><method>
 4   Relation relations[] <-
 5         getRelations(TypesOfRelation.IS_ACCESSEDBY)
 6   FOR each relation in relations[] DO
 7     //Gets the method side of the relation
 8     entity <- relation.getCaller()
 9     include entity in callerSet[]
10   ENDFOR
11   return callerSet[]
```

In Pseudocode 3, we show the method `getDirectCallers()` from a field representation class, called `FieldNode`. It gets all the relations from type *isaccessedby* in which the field represented by this `FieldNode` is called by a method - lines 4 and 5. Then, for each relation, it puts the caller method in the caller set that is returned to the algorithm. In the `remove()` algorithm we showed before, this caller set is added in the impact set of an entity.

IV. EMPIRICAL STUDY

In order to investigate the hypothesis of increasing the accuracy of impact analysis by adjusting the depth criterion, we conducted an empirical study with three different Java software projects, shown in Table I. Impala was used to validate itself. The second project chosen was DesignWizard [11], which is also used as part of our solution. OurGrid is a free to join peer-to-peer grid developed by the Distributed Systems Labs at UFCG [12].

TABLE I
PROJECTS SELECTED FOR THE CASE STUDY. LOCS = LINES OF CODE.

Name	Description	LOCs	Number of classes
Impala	Static impact analysis tool	1,584	45
DesignWizard	API for automated inspection of Java programs	3,644	44
OurGrid	Free to join peer-to-peer grid	48,752	627

A. Research Questions

We want to investigate the behavior of *precision* and *recall* measures using Impala's algorithms in an empirical study. It is known that static impact analysis produces too many false-positives. Impala uses the adjustment of depth criterion in order to reduce these false-positives. Our feeling is that the reduction of false-positives causes an increase in the number of false-negatives. So, the research questions we sought to answer are:

RQ1: Does the reductions of depth value increase *precision* on Impala's results?

RQ2: Do the increase of *precision* lead to the decrease of *recall*?

RQ3: Is *precision* sufficient to measure accuracy? Is *recall* as important as *precision*?

B. Setup

We analyzed Impala's algorithms with five different depth values. For the evaluation we call them:

- A_1 - searches only for direct dependencies, depth equals to one;
- A_2 - searches for direct and indirect dependencies, depth equals to two;
- A_3 - searches for direct and indirect dependencies, depth equals to three;
- A_6 - searches for direct and indirect dependencies, depth equals to six;
- A_∞ - searches for entire dependence hierarchy.

In order to evaluate all five variations, we randomly selected 5 to 15% of the total number of classes from each project and applied Impala's algorithms using a predefined methodology, show in Figure 3 and described below.

1) Randomly choose from the project's verisioning system - CVS [13] - the classes to be analyzed.
2) For each class, identify in the versioning system the revisions that incorporated at least one structural change.
3) For each of these revisions, obtain the change set C, which contains all structural changes of a determined revision. This set was obtained through structural comparison of the code from two subsequent revisions. So, each element of C represents an entity (e.g., a method) that was modified from one revision to another.
4) For each change set C, apply Impala's algorithms and obtain the sets of possible impacted entities: $A_1(C), A_2(C), A_3(C), A_6(C), A_\infty(C)$.
5) For each set of impacted entities $A_n(C)$, obtain the set of their class names: $I_1, I_2, I_3, I_6, I_\infty$.
6) For each change set C, extract from CVS what was really modified, M due to each change in C. Suppose a change c in C. We extract every change that occurred after c and because of c; and include the corresponding class name on M.
7) Compare each I_n to its correspondent M in terms of *precision*, *recall* and *accuracy*.

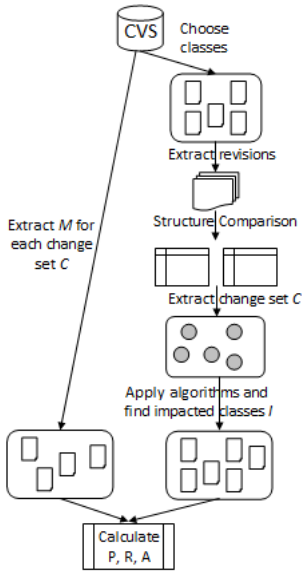

Fig. 3. Setup steps.

To get an overall measure for each software system analyzed, we summarized *precision* and *recall* results using macro-evaluation from information retrieval:

Macro-evaluation: takes the mean values of *precision* and *recall*:

$$P_M = \frac{1}{N}\sum_{i=1}^{N} P_i \qquad R_M = \frac{1}{N}\sum_{i=1}^{N} R_i$$

Macro-evaluation determines the accuracy of applying impact analysis algorithms in each change, individually. It allows us to determine with which depth the algorithms returned less false-positives and less false-negatives.

V. EVALUATION

Table II summarizes the *precision* and *recall* results obtained for OurGrid project. We chose to show the results of OurGrid because of its size, complexity and maturity. OurGrid is under development since 2004 and is considered a stable software system at the moment.

Due to space limitation, we chose to show only one result obtained for each class, which represents the analysis of one

revision. The total number of revisions analyzed is shown in column "Number of revisions". These are the revisions where one class underwent at least one structural change. Revisions due to change in license term, formatting or any other that did not represent a structural change were discarded.

From the results in Table II, we notice that there is no clear pattern that relates the increase of depth criterion and the variation of *precision* and *recall*. Both measures can remain unaltered, like what happens with RemoteAccessImpl and EBReplicaManager. One of the measures can remain unaltered while the other varies, as we can see in ReplicaExecutorThread. Finally, both measures can vary increasing, decreasing or behaving as second degree equation.

To extract an overall behavior and answer RQ1, we calculated the macro-evaluation of OurGrid's *precision* and *recall* results. Figure 4 shows the relationship between *precision* and *recall* for the different depth values. As expected, when depth value increases, *precision* decreases. However, the exact opposite happens with *recall*: when depth value increases, *recall* increases. In other words, the deeper the search goes, more false-positive and less false-negatives it produces.

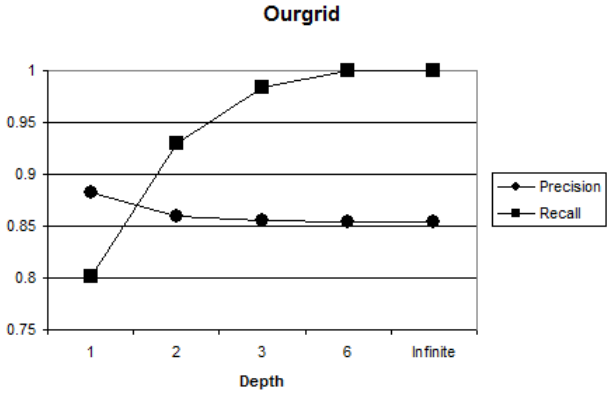

Fig. 4. *Precision* and *recall* for OurGrid.

In order to answer RQ1 and RQ2, we constructed the same graph for DesignWizard and Impala, shown in Figure 5. Although *precision* results were less significative, especially for Impala, the same behavior can be noticed: the grater the depth is, the lower is *precision* and the higher is *recall*.

From these results, we can positively answer RQ1. The reduction of depth value increases *precision* of Impala's results and vice-versa. If *precision* increases when depth is reduced it means that entities connected with indirect dependencies are less likely to be impacted for a change than entities that have direct dependencies. In other words, *precision* of A_1 $\geq$ *precision* of A_2 $\geq$ *precision* of A_3 $\geq$ *precision* of A_6 $\geq$ *precision* of A_∞ on average.

Table III shows the average *precision* and *recall* results from the three projects. From the results of Table III and the *precision* versus *recall* graphs, we a can infer that the increase of *precision* is associated with the decrease of *recall* and vice-versa, answering RQ2. The results from Impala's algorithms

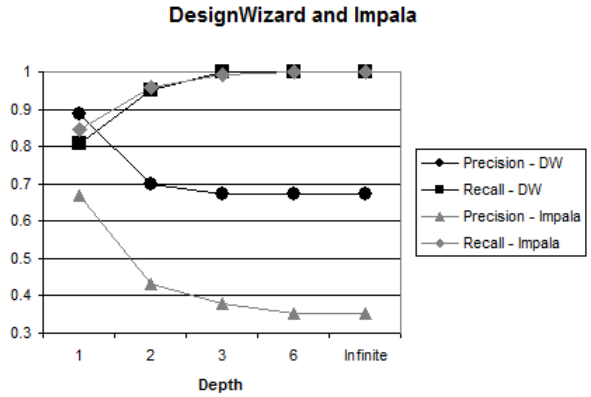

Fig. 5. *Precision* and *recall* for DesignWizard and Impala.

TABLE III
AVERAGE GAIN ON *precision* AND LOSS ON *recall* COMPARED WITH A_∞.

Depth	P_M	Gain on P	R_M	Loss on R
A_∞	73.46%	-	100%	-
A_6	72.86%	-1%	100%	0%
A_3	73.97%	1%	98.15%	2%
A_2	75.56%	3%	93.85%	6%
A_1	84.18%	15%	81.15%	19%

show that, opposite to *precision*, *recall* of A_1 $\leq$ *recall* of A_2 $\leq$ *recall* of A_3 $\leq$ *recall* of A_6 $\leq$ *recall* of A_∞ on average.

Now, we are able to discuss RQ3. Although no importance is given by recent impact analysis approaches [8], [9], [14] to the false-negatives produced, the results show that *recall* is as important as *precision* to measure the accuracy of an impact analysis technique. It is known that every approach, static or dynamic, produces both false-positives and false-negatives. From the empirical study conducted, we showed that ignoring *recall* can lead to a false result of accuracy.

Finally, Figure 6 shows the relative results of *precision*, *recall* and accuracy. In this empirical study, if only *precision* was taken into account, the best choice would be to search only for direct dependencies. However, *precision* and *recall* relative results show that the higher accuracy, 87% for infinite search, was found where *precision* is lower. There seems to be a trade-off between *precision* and *recall* for impact analysis. It is important to investigate if this trade-off exists only with Impala or it is a common behavior for other impact analysis techniques. If this is a common behavior, it is important to balance *precision* and *recall* measures in order to increase accuracy.

VI. RELATED WORK

Law et al [6] evaluate the precision of two dynamic impact analysis algorithms, CoverageImpact and PathImpact, by measuring the relative sizes of the impact sets computed, change set and set of program executions. Precision is measured in terms of the number of methods in the impact sets versus the total number of methods. They consider algorithm A to be more precise than algorithm B if the results of A are, overall,

TABLE II
Precision AND *recall* RESULTS FOR OURGRID.

Class	N. of revisions	Depth									
		1		2		3		6		∞	
		P	R	P	R	P	R	P	R	P	R
CorePeerImpl	11	1.00	0.25	0.60	0.75	0.67	1.00	0.67	1	0.67	1.00
RemoteAccessImpl	6	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
GumSpecService	1	0.80	0.57	0.88	1.00	0.88	1.00	0.88	1.00	0.88	1.00
UserAgentClient	4	0.50	0.50	0.56	0.75	0.58	0.92	0.60	1.00	0.60	1.00
EBReplicaExecutorFacade	4	0.67	0.50	0.80	1.00	0.50	1.00	0.50	1.00	0.50	1.00
EBReplicaManager	5	0.33	1.00	0.33	1.00	0.33	1.00	0.33	1.00	0.33	1.00
PermissionManager	2	1.00	0.17	1.00	0.17	1.00	0.50	1.00	1.00	1.00	1.00
ReplicaExecutor	1	1.00	0.50	0.67	0.50	0.80	1.00	0.80	1.00	0.80	1.00
ReplicaExecutorThread	10	1.00	0.14	1.00	0.43	1.00	0.71	1.00	1.00	1.00	1.00
ReplicaExecutorThreadManager	5	0.67	0.50	0.80	1.00	0.80	1.00	0.80	1.00	0.80	1.00

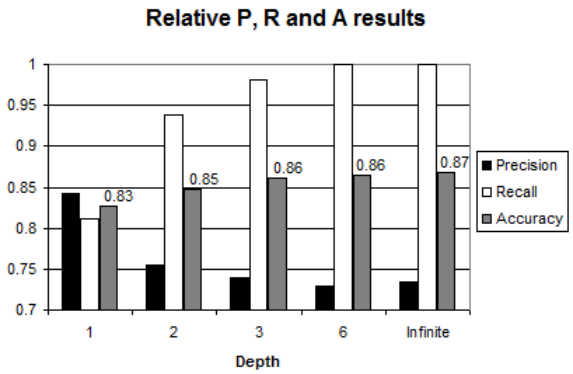

Fig. 6. Relative results of *precision*, *recall* and accuracy - in percentage.

a subset of the results from B. An important information is that they consider their algorithms to be safe in the sense that no method that is not in the impact set for a change can be affected by that change. That is, the algorithms do not produce false-negatives for the executions analyzed. However, one can never assure that all possible executions were taken into consideration.

The same approach is used by Apiwattanapong et al [8] to compare another dynamic algorithm, CollectEA, with CoverageImpact and PathImpact. It is also used by Breech et al [9] to compare one static and one dynamic algorithms with their correspondent improvements. One can never assume that a static technique is safe, because some code information, like the ones from very late binding, can only be collected at runtime.

VII. CONCLUSIONS AND FUTURE WORK

We have proposed, applied and evaluated two measures to assess accuracy of impact analysis techniques and algorithms: *precision* and *recall*. We have defined the concepts of false-positives and false-negatives in the context of impact analysis to relate *precision* to false-positives and *recall* to false-negatives. Previous works [6], [8], [9] use the term precision as a general measure, ignoring the conceptual difference between impacts that are not identified by an algorithm and identified impacts that do not really occur.

By applying the *precision* and *recall* measures to assess the accuracy of Impala's static impact analysis algorithms, we have been able to demonstrate that both measures influence on the accuracy results in a manner that reflects the usual understanding of the concept. As they are stated in formal setting, however, they can be used to compare, evaluate and improve new algorithms. In our empirical study, specifically, a higher accuracy was found by analyzing the adjustment of the depth value to balance *precision* and *recall*.

We are currently investigating if the need to balance *precision* and *recall* to achieve a higher accuracy is a common behavior for other impact analysis techniques. In the empirical study conducted for this paper, we analyzed the results on class level. We plan to analyze the same results on entity (method and field) level and to use the same methodology to evaluate some dynamic impact analysis techniques.

REFERENCES

[1] R. S. Arnold and S. Bohner, *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996.

[2] 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.

[3] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: ACM Press, 2001, pp. 46–53.

[4] R. J. Turver and M. Malcolm, "An early impact analysis technique for software maintenance," *Journal of Software Maintenance: Research and Practice*, vol. 6, no. 1, pp. 35–52, 1994.

[5] B. Korel and J. Laski, "Dynamic slicing of computer programs," *J. Syst. Softw.*, vol. 13, no. 3, pp. 187–195, 1990.

[6] 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.

[7] B. Breech, M. Tegtmeyer, and L. Pollock, "A comparison of online and dynamic impact analysis algorithms," in *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 143–152.

[8] 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.

[9] B. Breech, M. Tegtmeyer, and L. Pollock, "Integrating influence mechanisms into impact analysis for increased precision," in *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 55–65.

[10] C. J. Van Rijsbergen, *Information Retrieval, 2nd edition*. Dept. of Computer Science, University of Glasgow, 1979. [Online]. Available: http://www.dcs.gla.ac.uk/Keith/Preface.html

[11] "Design wizard," 2007, http://www.designwizard.org.

[12] W. Cirne, F. V. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray, "Labs of the world, unite!!!" *J. Grid Comput.*, vol. 4, no. 3, pp. 225–246, 2006.

[13] Free Software Foundation, "CVS - Concurrent Versions System," 2006, http://savannah.nongnu.org/projects/cvs/.

[14] L. Huang and Y.-T. Song, "Precise dynamic impact analysis with dependency analysis for object-oriented programs," in *Software Engineering Research, Management & Applications, 2007. SERA 2007. 5th ACIS International Conference on*, 2007, pp. 374–384.