

Variable Granularity for Improving Precision of Impact Analysis

Maksym Petrenko, Václav Rajlich
Department of Computer Science
Wayne State University
Detroit, MI USA 48202
{max,rajlich}@wayne.edu

Abstract

Impact analysis is a specialized process of program comprehension that investigates the nature and extent of a planned software change. Traditionally, impact analysis involves inspecting dependencies among the software components of a fixed granularity; these components constitute a dependency graph. In this paper, we argue that a single granularity is insufficient and leads to imprecise analysis. We explain how the precision can be improved by variable granularity, where the programmers choose among the granularity of classes, the granularity of class members, and the granularity of code fragments. We assess the resulting precision in a case study on open-source software.

1. Introduction

Software change plays a vital role in software evolution, agile and iterative software development, and maintenance. Impact analysis (IA) is a special topic of program comprehension, where the programmers attempt to understand the nature and the size of a particular software change that is planned as a response to a change request or a bug report. IA is a part of the software change design and belongs to a category of as-needed program comprehension techniques [16], where only a specific part of the software is understood by the programmers, just enough to be able to work on a specific task. IA estimates the set of all program components to be changed.

Iterative IA is a process that relies on a close collaboration of programmers and supporting software tools, where the programmers inspect software dependencies that are extracted by the tools. The process results in a set of components inspected by the programmers (*inspected set*). There is also a set of the components actually impacted by the change (*impact set*). The inspected set may contain components that are not impacted and they are considered to be false positives of the iterative IA; likewise, the inspected set

might miss some of the changed components and they are considered to be false negatives of the IA.

Earlier approaches emphasized a single granularity of the related program analysis [7, 28]. In this work, we study how a variable granularity improves the precision of IA, saving the programmers the extra work that is needed to inspect the false positives. It is motivated by our intuition that components at fine granularities are smaller, have fewer dependencies, and therefore the resulting inspected set is smaller.

We discuss different granularities, including the granularity of classes, the granularity of all program components that include class members, and the granularity of code fragments. We compare the precision of IA at these granularities in a case study on open-source software.

The rest of the paper is organized in the following way: section 2 summarizes the related work, section 3 explains the iterative IA approach that uses variable granularity, section 4 presents the case study design, section 5 discusses case study results, and section 6 contains conclusions and future work.

2. Related work

Software change consists of several activities [29] and one of them is IA. Usually, concept location is the preceding activity and identifies parts of the source code that are most relevant to the change request [20], [24]. The typical IA process starts with these initial components and uses program dependencies to define the inspected set. The components of this set are then inspected by the programmers who determine those components that are actually impacted by the change [5].

Iterative approaches support step-by-step navigation of program dependencies, where the control over the process alternates between the tool and the programmer. In each step, the tool proposes some components that can be affected by the change. The

programmer then takes over and corrects the mistakes made by the tool, marking the inspected components as relevant or irrelevant to the change.

One of the first models for iterative IA was presented in [26]. According to this model, programmers are able to visit the inspected set and mark the graph nodes with either the “Action” or the “No Action” label. A visual tool for this model was presented in [3], supporting the granularity of classes.

The theoretical model was further formalized and expanded with the set of extra propagation rules and marks by Rajlich in [27] and later in [28]. The tool support for this model was introduced in [10], expanded into the tool JRipples [7], and later integrated with the Eclipse development environment (<http://www.eclipse.org>). This paper presents the further elaboration on this research thread by providing support for variable granularity.

Similar, but less formal, a tool and model of iterative IA was discussed in [12]. In this model, artifacts and dependencies among them are typed and the approach is used for object-oriented programs.

Other approaches for *IA in object-oriented programs* [9, 17] categorize the types of changes and analyze software at several granularities, usually at the granularity of top classes, their members, and statements within these members.

Dynamic IA techniques [22] determine the impact set by finding the components that are executed simultaneously with the initially changed components.

Software repository mining techniques [1, 8, 32, 34] find impacted components by looking into the history of software changes, and find components that were jointly modified in the past.

IA heuristics can guide the developers by proposing the most relevant dependencies. Briand et al. [6] suggested program dependency coupling measures that indicate the likely change propagation. Poshyvanyk et al. [25] explained how the coupling between the source code of the components can be measured using information retrieval techniques. Malik and Hassan [18] examined an IA heuristic that combines repository mining technique with a coupling measure based on program dependencies.

Several recent papers empirically evaluated new techniques for impact analysis [1, 8, 13], concept location [23, 24], and software development [14] by case studies that *reenacted* past changes extracted from software repositories. The word “reenactment” originates from [14]; other papers use a different terminology for the same idea.

In the reenactment, researchers use a new technique, repeat actions undertaken by original developers, and

compare the results. Reenactment can be done manually by a human [23] or automatically by an algorithm that simulates human action. An example of automated reenactment is in [8] where a new IA tool suggests an impact set for a specific change request, and this suggested impact set is compared with the actual impact set retrieved from the repository.

3. Iterative Impact Analysis

Iterative IA is guided by the program dependencies. In the papers [27, 28] that serve as a foundation of this work, the program dependencies were not specified; it was only assumed that these dependencies can be extracted from the program and they propagate the change. In our current work, we selected the dependencies that are extracted from Eclipse AST [15].

3.1. Class and Member Dependency Graph

We view a software system as a set of components (i.e. classes, methods, and fields) and dependencies that form a graph called *Class and Member Dependency Graph* (CMDG). In CMDG, the components are represented as the nodes and the dependencies are represented as the edges. We define CMDG in the following way:

Definition 1. Let P be an OO program and let $G(V,E)$ be a directed graph, where the set of nodes $V=C \cup M \cup F$ represents the components of the program P , C represents the set of all classes in the program P , M represents the set of all methods in the program P , and F represents the set of all fields in the program P . The set of directed edges is $E = EN \cup ER$. An edge $(x,y) \in EN$ if and only if the definition of the component y is nested within the definition of the component x . An edge $(x,w) \in ER$ if and only if component w is referred within the definition of component x . Then, $G(V,E)$ is the *Class and Member Dependency Graph* (CMDG) of the program P .

Definition 2. The set of all components *nested* in d is defined as $N(d) = \{x \in V \mid (d, x) \in EN \text{ or there is } y \in N(d) \text{ such that } (y, x) \in EN\}$. If a component x nests within component y , we say y is the *parent* of x and x is the *child* of y .

Definition 3. We say two components x and y are *neighbors* if and only if there is an edge $(x,y) \in ER$ or an edge $(y,x) \in ER$.

Definition 4. A component x has the *nesting level* $nest(x) = 0$ if its definition is not present within the definition of any other component in P . A component y in program P has the nesting level $nest(y) = k$ if there is a component z in P such that $nest(z) = k-1$ and the definition of y is nested in the definition of z .

3.2. Single Granularity Impact Analysis

In *single granularity* IA, for example the granularity of classes, the programmers iteratively inspect classes that are suggested by the computer [28]. The control alternates between the computer and the programmer, in a manner depicted by the UML activity diagram in Figure 1; there are two swimlanes in the diagram, one for the computer and one for the programmer. Although the diagram of Figure 1 is intended for the granularity of all components, it can be used to describe single granularity IA also; the only difference is that in this case, the action “Select granularity” is bypassed.

The IA process starts with the computer constructing the CMDG, where all classes initially have

the blank status. The class identified during concept location receives the Changed status indicating it is scheduled to change, and the neighbors of this class receive the Next status suggesting that the programmer should inspect them for the possible secondary changes. (As mentioned above, the action “Select granularity” is ignored in the single granularity IA.)

Then the control is passed to the programmer who selects the *focus class* among the classes suggested by the computer, inspects it, and decides on its new status. The computer records this decision in CMDG, using the *marks* in Table 1. The marks are ordered by their *priority*, from the highest priority mark Changed to the lowest priority mark Blank.

If the programmer decided that the focus class should be marked Changed or Propagates, the computer also schedules its neighbors for future inspection by marking them Next. If the focus class is marked Inspected, no neighbors are marked. In the subsequent action, the computer checks if any classes marked Next still remain in CMDG. If so, the control returns to the programmer for the next iteration; otherwise, the process ends.

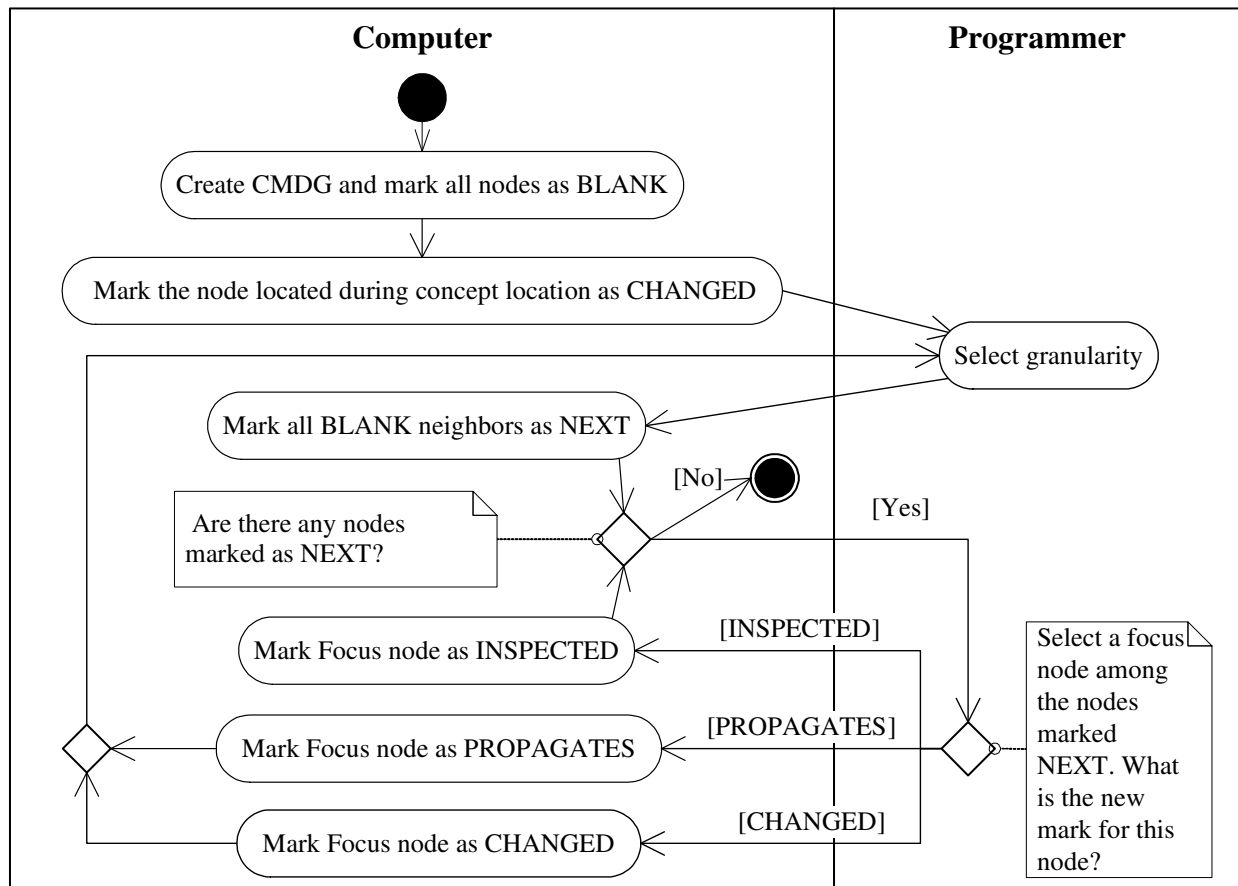


Figure 1. Activity diagram of interactive impact analysis at the granularity of all components

Table 1. Impact Analysis Marks

	Mark	Meaning
1	Changed	The component belongs to the impact set and is scheduled to change.
2	Propagates	The programmer inspected the component and found that it is not scheduled to change, but the neighbors of this component may change; the component propagates the change although it does not change itself.
3	Next	Automatically given by the computer and denotes that the component should be inspected by the programmer for possible change.
4	Inspected	The component was reviewed and found not to be impacted by the change. It also denotes that the component does not propagate the change to any of its neighbors.
5	Blank	An unknown status of the component; the component was never inspected and is not currently scheduled for the inspection.

3.3. Variable Granularity Impact Analysis

IA at the granularity of classes can be imprecise as classes often have many neighbors that are irrelevant to the change. IA at the granularity of fine-grained components should be more precise because these components have fewer neighbors; however, this IA might miss relevant neighbors as certain dependencies exist only at coarse granularities, for example inheritance. Thus, the IA process that wants to include finer granularities must allow *variable granularity*, as represented by the full activity diagram in Figure 1.

In this process, all components of CMDG, including classes and their members of an arbitrary nesting level, are at the disposal of the programmers. After inspecting the focus component, the programmers can choose whether the impact will propagate at the same, coarser, or finer granularity, see the action “Select granularity” in Figure 1. If they choose the same granularity, the computer marks as Next the neighbors of the focus component as described in the previous section. If a finer granularity is needed, the computer marks as Next the neighbors of all *children* (i.e. components that nest in the focus component) on the selected nesting level. Finally, when a coarser granularity is selected, the computer marks as Next the neighbors of the *parent* (i.e. component in which the focus component nests).

Marks propagate to the children and the parents of the focus node. The new mark of the parent is given to all children that have a mark of a lower priority than this new mark. When switching to a coarser granularity, the highest-priority mark of all children becomes the mark of the parent.

In cases when even the granularity of the finest components is insufficient, the programmers can choose the *granularity of code fragments* and specify the focus component by selecting a fragment of code they wish to change. First, the programmers select (highlight) a code fragment that may consist of one or more consecutive statements and choose the new mark for it. After that, the computer determines the focus component as the most fine-grained component containing the selected fragment, and updates its mark. Then it finds the neighbors of the fragment: the components that are referenced in the selected code, and the components that reference the focus component; these neighbors are marked as Next. In this way, even though code fragments are not a part of CMDG, the granularity of fragments relies on CMDG for the change propagation.

```
public class DepositTransaction
{
    public void deposit(int amount){
        Bank.startTransaction();
        Account.addMoney(amount);
        Account.chargeFee();
        Bank.endTransaction();
    }
}
```

Figure 2. Selected fragment in Java code

In the example in Figure 2, the programmers selected and highlighted the code fragment that contains two lines of code. The computer identifies the method ‘deposit’ that contains the fragment as the focus node, and marks this method as Changed. It then marks as Next the components that call the ‘deposit’ method. It also analyzes the selected fragment and marks as Next all methods that are called within the fragment, i.e. ‘addMoney’ and ‘chargeFee’. On the other hand, the ‘startTransaction’ and ‘endTransaction’ methods are called outside of the fragment and thus do not receive the Next mark.

3.4. The tool JRipples

An interactive Eclipse plug-in JRipples (<http://jrripples.sourceforge.net>) implements actions in the Computer swimlane in Figure 1, together with the interface that allows the programmer to take actions in

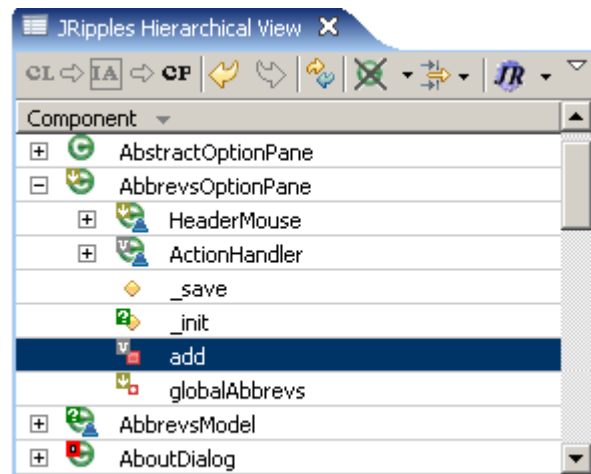
the Programmer swimlane. JRipples analyzes the program, creates the CMDG, keeps track of the marks, and gives programmers the list of suggested IA tasks. In this way, JRipples frees the programmer to concentrate on important tasks and decisions rather than bookkeeping.

Figure 3 shows a specialized JRipples view that presents relevant components and their IA marks. For the granularity of classes, it provides the list of top-level classes; for the granularity of all components, it provides lists of all members of these classes. JRipples monitors and automatically updates the marks of the components that are changed within other Eclipse tools like the refactoring tools, editors, debuggers, and so forth. For example, if the programmer changes a class in the Java editor, JRipples automatically selects this class as the focus component, marks it as Changed, and marks all its neighbors as Next.

JRipples also displays the IA marks in the standard Eclipse views and provides the possibility to change the marks in these views. Figure 4 shows the marks of JRipples displayed within the Package Explorer view.

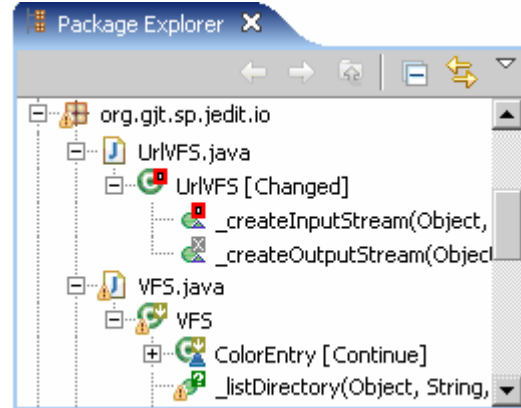
For the granularity of code fragments, JRipples provides a specialized tool that integrates with the source code editor of Eclipse and allows highlighting a specified part of the code, as shown in Figure 2.

The architecture of JRipples is based on the Eclipse extension points [19]. The open nature of the Eclipse extension points allows for further extension of JRipples functionality. The core of JRipples is CMDG that models the current state of the project. Other components of JRipples are plugged into JRipples core



Marks: Changed Next Propagates Inspected
Figure 3. The JRipples view in Eclipse¹

¹ For an explanation of the standard Eclipse icons, please visit <http://help.eclipse.org/help32/topic/org.eclipse.jdt.doc.user/reference/ref-156.htm>



Marks: Changed Next Propagates Inspected
Figure 4. Marks of JRipples displayed within the Package Explorer view of Eclipse

and interact either through the modifications of the CMDG, or directly through the defined interfaces. This architecture allows the user to switch among different modules “on the fly” and to adjust JRipples when a particular need becomes apparent. For example, the user can switch between different CMDG presentation modes, choose different analysis modules, change the granularity of the analysis, select the appropriate dependency builders, and so forth.

The dependency analysis module is responsible for building the CMDG and is based on the Eclipse Abstract Syntax Tree (AST). The process management module supports the steps of IA at the various levels of granularity.

4. Case study design

In order to evaluate the proposed approach, we performed a case study that was designed and conducted according to [31] and [11], where [11] answers common misconceptions and objections against the use of case studies.

The *hypothesis* of the case study states that IA done on variable granularity is more precise than IA done on the granularity of classes.

We have conducted the case study by reenacting the iterative IA for changes that we extracted from the change management repositories of the open-source software projects. For each change, we reenacted IA on the fixed granularity of classes, the variable granularity of all components, and the variable granularity of code fragments. We measured the precision of IA at these three granularities by comparing the respective sizes of the inspected sets built for these granularities.

As objects of our study, we selected Adempiere and Azureus software, two of the most active Java projects

on the Sourceforge.net repository of open-source software.

Adempiere (<http://www.adempiere.com>) is an enterprise resource planning application and consists of 261,130 components, 3,727 classes and 550,000 LOC.

Azureus (<http://azureus.sourceforge.net>) is a peer-to-peer file exchange system and consists of 133,008 components, 2,552 classes and 310,000 LOC.

4.1. Extracting changes

We selected the changes of the case study from the CVS and SVN repositories of the projects. In these repositories, each commit typically reflects a feature implementation or a bug fix, and provides information on the date of the change, the files modified, and the *delta*, i.e. lines of code modified in these files. For each project, we randomly selected 20 commits, each modifying at least 5 components spread over at least 5 Java files. For each commit, we extracted a snapshot of the software prior to the change, and built the CMDG.

4.2. Reenacting IA at the granularity of classes

To reenact IA at the granularity of classes, we first determined the Changed classes by analyzing the delta and identifying the classes that contain at least one modified line of code.

In order to identify the Propagates classes, we noticed that in iterative IA, the Changed classes and the Propagates classes must form an interconnected subgraph. Moreover, an effective IA process uses a minimal number of Propagates classes that interconnect these Changed classes.

In order to determine the minimal number of Propagates classes, we used the heuristic algorithm [21] that takes as an input a weighted graph with a subset of nodes identified as *terminals*, and produces a minimum-weight spanning tree that interconnects all terminals with the (approximately) smallest number of non-terminal nodes. If all edges in the input graph have an equal weight, the algorithm simply minimizes the number of the interconnecting nodes. This algorithm naturally maps to our problem: Changed classes are the terminal nodes, and the interconnecting nodes represent the Propagates classes through which the Changed classes were reached during iterative IA.

As an example, consider the dependency graph in the left part of Figure 5, where the Changed classes have the black shading. Then, the spanning tree in the right part of the figure is built using the algorithm in [21], and contains all Changed classes and a minimal number of Propagates classes.

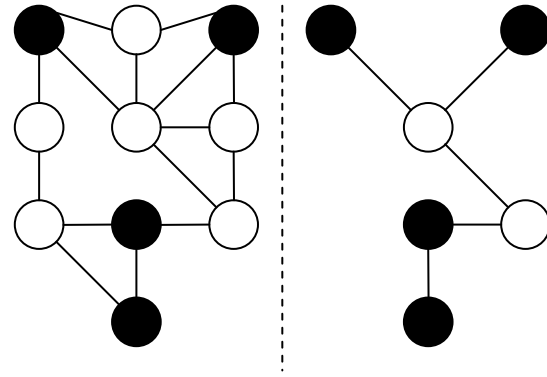


Figure 5. Identifying Propagates classes.
Left: dependency graph with Changed classes (black). Right: spanning tree with the minimal number of Propagates classes

For Inspected classes, we identified all neighbors of Changed and Propagates classes in CMDG.

4.3. Reenacting IA at the granularity of all components

For the granularity of all components, we again determined the Changed components from the delta by identifying the most fine-grained fields, methods and member classes that contain at least one modified line of code.

As in the case of the granularity of classes, we used the algorithm in [21] to find the minimal number of Propagates components that interconnect the Changed components. In order to ensure that the algorithm not only minimizes the number of Propagates components, but also simulates IA on the finest granularities which offer the highest precision, we heuristically weighted the edges in the graph:

All EN edges have a weight 1. Then the weight of a path from a parent component x to any of its children is less or equal to $|N(x)|$. Based on this fact and the observation that any parent component nests more components than any of its children, we set the weight of each ER edge as the following:

$$w(x, y) = |N(x)| + |N(y)| + \max(|N(x)|, |N(y)|)$$

As the algorithm seeks to minimize the total weight of the edges in the spanning tree, this weighting scheme ensured it was more expensive for the simulation to propagate an impact at the granularity of the parent than at the granularity of the children.

After the spanning tree was computed, we determined the Inspected components by finding all components that are neighbors of the Changed and Propagates components in CMDG.

4.4. Reenacting IA at the granularity of code fragments

To determine the Changed code fragments, we analyzed the lines in each Changed component, and extracted the Changed fragment as the first modified line of the component, the last modified line, and all lines in between.

The rest of the reenactment followed the same process as the reenactment at the granularity of all components with one exception: the Changed components had a reduced set of dependencies, because the dependencies to the methods that are called outside of their Changed fragments were not considered in the reenactment.

4.5. Calculating precision

The recall [2] compares the number of retrieved relevant documents against the number of all relevant documents. In IA, recall compares the number of impacted components that a developer was able to identify against the number of all impacted components. Recall below 100% indicates that the developer identified an incomplete impact set. In our case study, the input to the algorithm was the complete impact set extracted from the repository, for which we computed the relevant inspected sets. Because of this design, the algorithm of our reenactment always produced 100% recall.

Since the recall in our case study is always 100%, impact set is always a subset of the inspected set. Thus, we calculated the precision [2] by taking the number of relevant documents, in our case all classes that are actually changed, and comparing this number with the number of all retrieved documents, in our case the number of all inspected classes:

$$P(\text{granularity}) = \frac{I}{M(\text{granularity})}$$

In this formula, I is the number of Changed classes (i.e. the size of the impact set), and $M(\text{granularity})$ is the sum of the Changed, Propagates, and Inspected classes (i.e. the size of the inspected set). Please note that the sets of Changed and Propagates classes are the same for all three granularities.

4.6. Tool support

The algorithms for finding inspected sets at the different granularities were implemented as a plug-in for JRipples. This plug-in substitutes the programmer and simulates all actions from the Programmer swimlane in Figure 1.

5. Results of the study

The data from the case study are in the Appendix and the summary is in the Table 2. The data in the last two lines of the table show that the granularity of all components improves the average precision over the granularity of classes by a factor of 1.88. The granularity of fragments provides an additional increase in precision by a factor of 1.32 over the granularity of all components. These findings confirm our hypothesis and show the effectiveness of the proposed approach.

It should be noted that while most changes in our studies modified classes with only a moderate number of neighbors, some changes modified classes that are used by a large number of other classes; these particular changes produced the large inspected sets.

Table 2. Summary of case study data.

	Impacted classes	Propagates Classes	Inspected classes at granularity of			Precision of IA at granularity of			
			Classes	All components	Fragments	Classes	All components	Fragments	
Adempiere									
Median	7	0	101	53	44	5.61%	10.73%	14.72%	
Average	8.05	0.75	164.10	89.95	77.15	7.81%	14.88%	18.45%	
Azureus									
Median	7	1	139	66	42	5.39%	9.62%	15.21%	
Average	7.45	1	152.60	80.95	64.60	7.19%	13.40%	18.93%	
Summary over the two projects									
Median	7	1	124	59	42	5.41%	9.90%	14.72%	
Average	7.75	1	158.35	85.45	70.88	7.50%	14.14%	18.69%	
Improvement factor in average precision over granularity of classes							1.88		2.49
Improvement factor in average precision over granularity of all components								1.32	

5.1. Threats to validity

We reenacted changes by modeling an ideal human behavior through the algorithm that always produces the complete impact set, minimizes the number of Propagates components, and selects the Next components at the finest possible granularity. Real programmers might make less perfect choices and, as a result, may miss some impacted classes, may need to inspect more classes than absolutely necessary, and work on a less than ideal granularity; these imperfections will lower both the precision and recall of IA. Also, the programmers might select a different set of Propagates components, resulting in a different number of Inspected classes and a different precision of the analysis. The impacted classes may be missed by programmers particularly as a consequence of hidden dependencies [33], resulting in a lower recall. However, these potential inaccuracies influence the precision only slightly and since they influence different granularities equally, we believe that the precision improvement factors of Table 2 are influenced only minimally.

The problem of finding the minimal spanning tree connecting a specified set of nodes is known to be NP-complete, and the algorithm [21] used in our study produces the set of Propagates nodes that is only approximately minimal. The use of a different algorithm can give a different result. In addition, the algorithm is independent of the order in which the Changed components were inspected during actual IA and independent of the selection of the first Changed component; algorithm that would reenact the exact order in which the original IA was done would be more accurate.

Using CVS and SVN commits in our study assumes that each commit provides a set of changed classes that are related to a single changed feature. If a commit modifies several features, our reenactment might overestimate the effort of IA. The reenactment can also underestimate the effort of IA if the commit contains only a portion of the change related to the modified feature.

Certain changes can propagate through the dependencies that are not captured by JRipples (e.g. database dependencies, text files dependencies, etc.). As our model captures only certain dependencies, the reenactment in the case study might overestimate the actual effort of IA by selecting more Propagates components than was necessary for the change.

Different object software might produce different results as different architectures use Propagates components in different contexts, and the program

components may have a different average number of neighbors. The investigated systems are not representative of all types of Object Oriented systems. Generalization of some of our results to other systems has to be done cautiously.

6. Conclusions and future work

We described the iterative IA that uses variable granularity and provides a more precise support to the programmers than previous approaches. We studied the granularities of classes, of all components that include class members, and of code fragments. We also described the tool JRipples that supports the iterative IA process with the variable granularity. The case study showed the substantial improvement in precision when the iterative IA is supported by variable granularity.

We assessed the effectiveness of the IA at variable granularity by reenacting past changes in Azureus and Adempiere. For this, we extracted the impact sets from the repositories of the projects, and used the algorithm in [21] to simulate the IA actions of a programmer that produced these sets. In the future, we would like to evaluate the proposed IA by also observing programmers during the actual process of IA.

To further improve the precision of iterative IA, we plan to investigate how the IA process can be guided by a use of heuristics [6, 25]. We also plan to support the typology of changes as in [9, 17]; insights from this work may further improve the precision of IA.

Programmer support will be further improved by extending the functionality of JRipples to other activities of the software change, such as concept location that provides the starting point of IA, or regression testing that follows change implementation. We will investigate whether the inspected sets, built during the IA process, can be used as regression testing firewalls; firewalls are the sets of program components that have to be retested to ensure that the existing functionality is not broken when new features are introduced [30].

We also plan to investigate different types of software dependencies [4] and their suitability for IA support. Another plan extends the analysis of dependencies into the systems implemented with mixed technologies, in particular the non-code files (for example, text and xml files) that are extensively used in large Java-based interactive programs.

Acknowledgments

The authors would like to thank Radu Vanciu for his contribution to the definition of CMDG.

This work was partially supported by the grants CCF-0438970 and CCF-0820133 from the National Science Foundation (NSF). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

- [1] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A., "Identifying the Starting Impact Set of a Maintenance Request: a Case Study", in *Proceedings European Conference on Software Maintenance and Reengineering*, 2000, pp. 227-230.
- [2] Baeza-Yates, R. A. and Ribeiro-Neto, B., *Modern Information Retrieval*, Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc., 1999.
- [3] Barros, S., Bodhuin, T., Escudie, A., Queille, J. P., and Voidrot, J. F., "Supporting Impact Analysis: a Semi-Automated Technique and Associated Tool", in *Proceedings International Conference on Software Maintenance*, 1995, pp. 42-51.
- [4] Binkley, D., "Source Code Analysis: A Road Map", in *Proceedings International Conference on Software Engineering (ICSE'07)*, 2007, pp. 104-119.
- [5] Bohner, S. and Arnold, R., *Software Change Impact Analysis*, Los Alamitos, CA, IEEE Computer Society, 1996.
- [6] Briand, L. C., Wuest, J., and Lounis, H., "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems", in *Proceedings International Conference on Software Maintenance*, 1999, pp. 475-483.
- [7] Buckner, J., Buchta, J., Petrenko, M., and Rajlich, V., "JRipples: A Tool for Program Comprehension during Incremental Change", in *Proceedings IEEE International Workshop on Program Comprehension*, 2005, pp. 149-152.
- [8] Canfora, G. and Cerulo, L., "Impact Analysis by Mining Software and Change Request Repositories", in *Proceedings 11th IEEE International Symposium on Software Metrics*, 2005, pp. 9-29.
- [9] Chaumun, M. A., Kabaili, H., Keller, R. K., and Lustman, F., "A change impact model for changeability assessment in object-oriented software systems", *Software maintenance and reengineering*, 45, 2-3, 2002, pp. 155 - 174.
- [10] Chen, K. and Rajlich, V., "RIPPLES: Tool for Change in Legacy Software", in *Proceedings International Conference on Software Maintenance*, 2001, pp. 230 - 239.
- [11] Flyvbjerg, B., "Five Misunderstandings About Case-Study Research", *Qualitative Inquiry*, 12, 2, 2006, pp. 219-245.
- [12] Han, J., "Supporting Impact Analysis and Change Propagation in Software Engineering Environments", in *Proceedings Workshop on Software Technology and Engineering Practice*, 1997, pp. 172-183.
- [13] Hassan, A. E. and Holt, R. C., "Replaying development history to assess the effectiveness of change propagation tools", *Empirical Software Engineering*, 11, 3, 2006, pp. 335 - 367.
- [14] Jensen, C. and Scacchi, W., "Discovering, Modeling, and Reenacting Open Source Software Development Processes", *New Trends in Software Process Modeling, Series in Software Engineering and Knowledge Engineering*, 18, 2006, pp. 1-20.
- [15] Khun, T. and Thomann, O., "Abstract Syntax Tree", Date Accessed: August 2007, Online at http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html, 2007.
- [16] Lakhoria, A., "Understanding Someone Else's Code: An Analysis of Experience", *The Journal of Systems and Software*, 23, 3, 1993, pp. 269-275.
- [17] Lee, M., Offutt, A. J., and Alexander, R. T., "Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software", in *Proceedings Technology of Object-Oriented Languages and Systems 2000*, pp. 61-71.
- [18] Malik, H. and Hassan, A. E., "Supporting Software Evolution Using Adaptive Change Propagation Heuristics", in *Proceedings International Conference on Software Maintenance*, 2008, pp. 177-186.
- [19] Marchal, B., "Working XML: Define and load extension points", Date Accessed: August 2005, Online at <http://www-128.ibm.com/developerworks/library/x-wxxm29.html>, 2005.
- [20] Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., and Sergeyev, A., "Static Techniques for Concept Location in Object-Oriented Code", in *Proceedings IEEE International Workshop on Program Comprehension*, 2005, pp. 33-42.
- [21] Mehlhorn, K., "A faster approximation algorithm for the Steiner problem in graphs", *Information Processing Letters*, 27, 1988, pp. 125-128.
- [22] Orso, A. A., Law, T., Rothermel, J., and G. Harrold, M. J., "An empirical comparison of dynamic impact analysis algorithms", in *Proceedings International Conference on Software Engineering*, 2004, pp. 776-786.
- [23] Petrenko, M., Rajlich, V., and Vanciu, R., "Partial Domain Comprehension in Software Evolution and Maintenance", in *IEEE International Conference on Software Comprehension*, 2008, pp. 13-22.
- [24] Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval", *IEEE Transactions on Software Engineering*, 33, 6, 2007, pp. 420-432.
- [25] Poshyvanyk, D. and Marcus, A., "The Conceptual Coupling Metrics for Object-Oriented Systems", in *IEEE International Conference on Software Maintenance*, 2006, pp. 469-478.
- [26] Queille, J.-P., Voidrot, J.-F., Wilde, N., and Munro, M., "The Impact Analysis Task in Software Maintenance: A Model and a Case Study", in *Proceedings International Conference on Software Maintenance*, 1994, pp. 234 - 242.
- [27] Rajlich, V., "A Model for Change Propagation Based on Graph Rewriting", in *Proceedings International Conference on Software Maintenance*, 1997, pp. 84-91.
- [28] Rajlich, V., "Modeling Software Evolution by Evolving Interoperation Graphs", *Annals of Software Engineering*, 9, 1-4, 2000, pp. 235-248.
- [29] Rajlich, V. and Gosavi, P., "Incremental Change in Object-Oriented Programming", *IEEE Software*, 21, 4, 2004, pp. 62 - 69.

[30] White, L., Jaber, K., Robinson, B., and Rajlich, V., "Extended firewall for regression testing: an experience report", *Extended firewall for regression testing: an experience report*, 20, 6, 2008, pp. 419-433.

[31] Yin, R. K., *Applications of Case Study Research*, 2 ed., Thousand Oaks, CA, USA, Sage Publications, Inc, 2003.

[32] Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C., "Predicting Source Code Changes by Mining Change History", *IEEE Transactions on Software Engineering*, 30, 9, 2004, pp. 574 - 586.

[33] Yu, Z. and Rajlich, V., "Hidden Dependencies in Program Comprehension and Change Propagation", in *Proceedings International Workshop on Program Comprehension*, 2001, pp. 293 - 299.

[34] Zimmermann, T., Weisgerber, P., Diehl, S., and Zeller, A., "Mining Version Histories to Guide Software Changes", *IEEE Transactions on Software Engineering*, 31, 6, 2005 pp. 429-445.

Appendix: Case Study data

Date of commit	Classes in impact set	Propagates Classes	Classes in inspected set at granularity of			Precision of IA at granularity of		
			Classes	All components	Fragments	Classes	All components	Fragments
Azureus								
17-Apr-04	7	4	224	183	181	3%	4%	4%
20-May-04	8	2	98	72	69	8%	11%	12%
24-May-04	5	0	91	56	31	5%	9%	16%
30-Jun-04	5	2	147	57	40	3%	9%	13%
18-Jul-04	7	2	151	82	70	5%	9%	10%
9-Nov-04	6	1	193	83	75	3%	7%	8%
2-Feb-05	7	1	88	55	42	8%	13%	17%
27-Jun-05	5	1	92	51	18	5%	10%	28%
26-Sep-05	7	2	238	129	120	3%	5%	6%
27-Nov-05	12	0	170	77	44	7%	16%	27%
23-Mar-06	14	1	262	86	71	5%	16%	20%
27-Apr-06	7	0	21	16	12	33%	44%	58%
27-Jun-06	6	0	42	20	16	14%	30%	38%
7-Aug-06	6	0	130	77	42	5%	8%	14%
11-Sep-06	5	1	167	53	29	3%	9%	17%
11-Jan-07	8	1	83	76	72	10%	11%	11%
19-Mar-07	14	1	506	317	267	3%	4%	5%
10-Apr-07	8	0	128	43	37	6%	19%	22%
9-Nov-07	7	0	70	26	17	10%	27%	41%
4-Feb-08	5	1	151	60	39	3%	8%	13%
Adempiere								
24-Jan-07	11	2	251	96	36	4.38%	11.46%	30.56%
3-Feb-08	17	0	64	50	46	26.56%	34.00%	36.96%
19-Jun-07	7	2	69	50	38	10.14%	14.00%	18.42%
21-May-07	5	0	108	16	11	4.63%	31.25%	45.45%
2-Dec-08	5	0	56	24	21	8.93%	20.83%	23.81%
18-Dec-07	5	0	64	56	53	7.81%	8.93%	9.43%
18-Feb-08	5	1	212	152	142	2.36%	3.29%	3.52%
4-Jan-09	8	0	72	23	23	11.11%	34.78%	34.78%
16-Feb-07	9	0	60	47	32	15.00%	19.15%	28.13%
8-Nov-07	18	4	567	433	381	3.17%	4.16%	4.72%
21-Nov-06	9	0	692	110	95	1.30%	8.18%	9.47%
21-Feb-08	6	0	124	115	104	4.84%	5.22%	5.77%
30-Nov-08	5	1	131	76	54	3.82%	6.58%	9.26%
26-Feb-07	16	1	274	265	252	5.84%	6.04%	6.35%
7-Oct-07	6	1	70	63	61	8.57%	9.52%	9.84%
29-Jan-08	5	1	124	50	35	4.03%	10.00%	14.29%
12-Feb-08	7	0	43	20	18	16.28%	35.00%	38.89%
15-Feb-08	5	2	72	67	66	6.94%	7.46%	7.58%
13-Feb-08	5	0	93	37	33	5.38%	13.51%	15.15%
19-Dec-07	7	0	136	49	42	5.15%	14.29%	16.67%