Empir Software Eng (2006) 11: 335–367
DOI 10.1007/s10664-006-9006-4

Replaying development history to assess the effectiveness of change propagation tools

Ahmed E. Hassan · Richard C. Holt

Published online: 9 June 2006

Editors: Mark Harman, Bogdan Korel, Panos Linos, Audris Mockus, and Martin Shepperd

Abstract As developers modify software entities such as functions or variables to introduce new features, enhance old ones, or fix bugs, they must ensure that other entities in the software system are updated to be consistent with these new changes. Many hard to find bugs are introduced by developers who did not notice dependencies between entities, and failed to propagate changes correctly. Most modern development environments offer tools to assist developers in propagating changes. For example, dependency browsers show static code dependencies between source code entities. Other sources of information such as historical co-change or code layout information could be used by tools to support developers in propagating changes. We present the *Development Replay* (DR) approach which empirically assess and compares the effectiveness of several not-yet-existing change propagation tools by reenacting the changes stored in source control repositories using these tools. We present a case study of five large open source systems with a total of over 40 years of development history. Our empirical results show that historical co-change information recovered from source control repositories along with code layout information can guide developers in propagating changes better than simple static dependency information.

Keywords Change propagation · Mining software repositories · Historical co-change · Static dependency · Source control systems

1 Introduction

Change propagation is a central aspect of software development. As developers modify software entities such as functions or variables to introduce new features or fix bugs, they must ensure that other entities in the software system are updated to be

A. E. Hassan (✉)
Department of Electrical and Computer Engineering, University of Victoria, Victoria, Canada
e-mail: ahmed@ece.uvic.ca

R. C. Holt
School of Computer Science, University of Waterloo, Waterloo, Canada
e-mail: holt@plg.uwaterloo.ca

consistent with these new changes. For example, if the interface of a function changes, its callers have to be modified to reflect the new interface otherwise the source code won't compile nor link. This example of propagation is easy to determine, but this is not always the case. Many hard to find bugs are introduced by developers who did not notice dependencies between entities, and failed to propagate changes correctly.

The dangers associated with not fully propagating changes have been noted and elaborated by many researchers. Parnas (1994) tackled the issue of software aging and warned of the ill-effects of *Ignorant Surgery*, modifications done to the source code by developers who are not sufficiently knowledgeable of the code. Brooks (1974) cautioned of the risks associated with developers losing grasp of the system as it ages and evolves. Misunderstanding, lack of experience and unexpected dependencies are some of the many reasons for failing to propagate changes during the development and maintenance of a software system. Mis-propagated changes have a high tendency to introduce difficult to find bugs in software systems, as inconsistencies between entities (such as functions) increase.

Work by Atkins et al. (1999) and surveys by Sim et al. (1998) indicate that software developers would like to have tools to assist them in performing changes to the source code. These tools could guide developers by either informing them about code entities (such as functions) to change, or assisting developers in performing the actual change (such as tools that automate the code refactoring process). For our work, we focus on a tool's ability to inform developers which entities to change. A desired change propagation tool is one that would ensure that a developer changing a particular source code entity (such as a function) is informed of all relevant code entities to which the change should be *propagated*.

Such tools need to be investigated using careful, rigorous software engineering experimentation before they are adopted by practitioners (Fenton et al., 1994; Glass, 2003; Kitchenham et al., 2002; Perry et al., 2000). Unfortunately, laboratory experiments are usually not able to simulate real life industrial settings and tend to run for a short period of time, while industrial studies usually require a long and costly commitment by the practitioners. An ideal approach should strike a balance between the low cost, short duration, fast results of laboratory experiments which permit the analysis of a variety of tools; while limiting the costs and time needed for industrial studies. An ideal approach would expedite studying the effectiveness of a tool over an extended period of time. In this paper, we propose an empirical approach that attempts to strike such a balance to assess the effectiveness of a change propagation tool in assisting developers who are maintaining and enhancing large long lived software systems.

We present the *Development Replay (DR)* which permits us to replay the history of a software project since its inception till any moment in its history using data recovered from its source control repository. We can determine at any moment the *state of the software project*, such as the current developers that worked on or are currently working on the project, the cooperation history between these developers, and the structure of the dependencies between its source code entities (such as functions and variables). We can also recover *change sets* from the source control system. These change sets track the source code entities that were modified together to implement or enhance features, or to fix bugs. Using this historical data (the state of the software project and the change sets), we compare the effectiveness of several change propagation tools.

Through a case study which uses historical data from several large open source projects, we show that change propagation tools which use historical co-change information derived from source control provide better support to software developers in propagating changes than simple static dependency browsers which are usually integrated in most modern software development environments.

1.1 Organization of Paper

This paper is organized as follows. In Section 2, we discuss the change propagation process and explain how we could measure the effectiveness of a change propagation tool. Then in Section 3, we present several metrics to assess some of the claimed benefits of new change propagation tools. In Section 4, we give an overview of the DR approach and the software infrastructure we developed for assessing change propagation tools. In Section 5, we present an empirical study which applies the DR approach using large open source software systems to measure the effectiveness of several change propagation tools. Then in Section 6, we present a critical analysis of the limitations of the results derived through the DR Approach. Section 7 surveys related work and compares our approach to the surveyed work. Section 8 concludes the paper with comments about the benefits and limitations of the DR approach.

2 The Change Propagation Process

We define *change propagation* as the changes required to other entities of the software system to ensure the consistency of assumptions in a software system after a particular entity is changed. For example, a change to a function that writes data to file may require a change to propagate to the function that reads data from file. This change would ensure that both functions have a consistent set of assumptions. In some cases no change propagation may be required; for example when a comment is updated, the indentation of the text of a function is changed, the internal logic of a function is reworked, a locally scoped variable is renamed to clarify its use, or local optimizations are performed. Though developers have to tackle the problem of change propagation and locate entities to change in a software system to ensure its consistency on a daily basis, this problem and its surrounding challenges are not clearly understood. Nevertheless several approaches have been proposed to reduce hidden dependencies and assist developers in propagating changes. These proposals include, most notably, the idea of information hiding in designing systems by Parnas (1972), the Object Oriented (OO) paradigm which focuses on grouping related entities in the same structure to ease modification and understanding (Rumbaugh et al., 1991), and lately Aspect Oriented Programming (AOP) which encapsulates concepts which crosscut the structures defined by the OO paradigm (Kiczales et al., 1997).

In Fig. 1, we present a model of the change propagation process. Guided by a request for a new feature, a feature enhancement, or the need to fix a bug, a developer determines the initial entity in the software system that must change. Once the initial entity is changed, the developer then analyzes the source code to determine if there are other entities to which the change must be propagated. Then

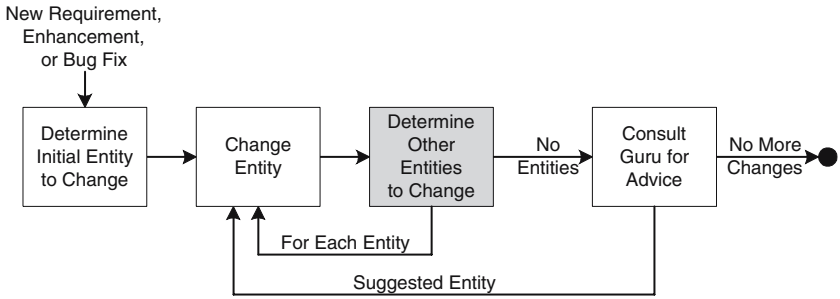

Fig. 1 Model of the change propagation process

the developer proceeds to change these other entities. For each entity to which the change is propagated the propagation process is repeated. When the developer cannot locate other entities to change, the developer consults a *Guru*. If the Guru points out that an entity was missed, then it is changed and the change propagation process is repeated for that entity. This continues until all appropriate entities have been changed. At the end of this process, the developer has determined the *change set* for the new requirement or bug fix at hand. Ideally all appropriate entities should have been updated to ensure consistent assumptions throughout the software system.

The Guru could be a senior developer, a test suite, or even a compiler. Usually consulting the senior developer is not a practical option, as the senior developer has limited time to assist each developer. Nor is it a feasible option for long lived projects where such a knowledgeable developer rarely exists. Moreover, complete and extensive test suites rarely exist. Therefore, developers find themselves forced to use other forms of advice/information such as the advice reported by a development tool to perform the activities in the grey highlighted box in Fig. 1. Ideally developers would like to minimize their dependence on a Guru. They need software development tools that enable them to confidently determine the need to propagate changes without having to seek the assistance of gurus which are not as accessible and may not even exist.

2.1 Information Sources Used to Propagate Changes

Program dependency relations, such as *call* and *use* have been proposed and used as indicators for change propagation (Finnigan et al., 1997; Gallagher and Lyle, 1991; Lee, 2000; Penny, 1992; Sniff+ Home Page). For example, if function *A* *call*s function *B*, and *B* was changed then function *A* is likely to change as well. If function *A* were to change then all other functions that call *A* may need to change as well. This ripple effect of change progresses until no more changes are required to the source code (Yau et al., 1988).

A number of information sources could be used by developers to assist them in locating other entities that should change. Builders of change propagation tools make use of these information sources to suggest entities to which a change may propagate. We now describe some of the possible sources of information. These sources of information are derived based on prior research findings and our experience in developing large software systems.

2.1.1 Entity Information

Change propagation may depend on the particular changed entity. For example, a change may propagate to other entities linked to the changed entity according to relations such as:

- A *Historical Co-change* (*HIS*) records that one entity changed at the same time as another entity. If entity *A* and *B* changed together in the past, then they are related via a historical co-change relation and are likely to change together in the future.
- A *Code Structure* (*CUD*) relation records static dependencies between entities. *Call*, *Use*, and *Define* relations are some possible sub-relations:
  - The *Call* relation records that a function calls another function or macro.
  - The *Use* relation records that a function uses a variable.
  - The *Define* relation records that a function defines a variable or has a parameter of a particular type. For example *F Define T*, means *F* defines a variable of type *T*.
- A *Code Layout* (*FIL*) relation records the location of entities relative to classes or files or subsystems in the source code. Containers such as files and classes are good indicators of a relation between entities, and related entities tend to change together.

2.1.2 Developer Information (DEV)

Change propagation may depend on the fact that the same developer changed other entities recently or frequently. This is based on the observation that over time developers gain expertise in specific areas of the source code and are likely to modify a limited set of entities in their acquired areas of expertise (Bowman and Holt, 1999).

2.1.3 Process Information

Change propagation may depend on the process employed in the development. For example, developers may always update a "change log" file or particular files with every change even though these files are independent of the specific entities that just changed.

2.1.4 Textual Information

Change propagation may depend on the fact that change propagates to entities with similar names, as the similarity in naming indicates similarities in the role of the entities and their usage, as suggested by Anquetil and Lethbridge (1998) who used such information to improve automatic clustering of files. It may also depend on the fact that entities have similar comment tokens (Shirabad, 2003).

Other sources of information may exist. Also the aforementioned information sources could be combined and extended in various ways. For example, another

possible source is the co-call entity information, where A and B both *call* C. A and B may implement similar functionality and a change to A may propagate to B.

Developers use these information sources to assist them in propagating changes. Developers spend a considerable amount of time to correctly propagate a change to other entities. Developers need tools to assist them in this labor intensive and error prone propagation process. Development tools may use the aforementioned information sources as the basis of heuristics for suggesting entities to assist developers in the change propagation process. In the following section, we investigate a number of metrics that could be used to measure the effectiveness of change propagation tools such as dependency browsers that are available in modern software development environments.

3 Measuring the Effectiveness of a Tool in Propagating Changes

In general, developers seek tools that can assist them in performing changes quickly and accurately. By quickly, we mean tools that would reduce the time needed to perform the change. By accurately, we mean tools that would ensure that a change to a source code entity is propagated to all relevant code entities. In the ideal case, a development tool would correctly suggest all the entities that should be changed without the developer resorting to asking a Guru for advice. The worst case occurs when a Guru is consulted to determine each entity that should be changed. Referring back to the change propagation model shown in Fig. 1, we would like to minimize the number of times a Guru suggests an entity to change.

The metrics discussed in this section will focus on the accuracy of the tool instead of the time required to perform the change itself. The time required to perform a change is likely to highly depend on the developer performing the change and the type of tool support (e.g., code editor) provided to the developer. Moreover, the time required may be difficult to track since most practitioners rarely record the time spent on each change.

3.1 A Simple Example

Consider the following example, Dave is asked to introduce a new feature into a large legacy system. He starts off by changing initial entity A. After entity A is changed, a tool suggests that entities B and X should change as well as. Dave changes B, but then examines X and realizes that it does not need to be changed. So Dave does not need to perform any change propagation for X. He then asks the tool to suggest another entity that should change if B were changed. The tool suggests Y and W, neither of which need to change—therefore Dave will not perform any change propagation for Y or W. Dave now consults Jenny, the head architect of the project (the Guru). Jenny suggests that Dave should change C as well. Dave changes C and asks the tool for a suggestion for an entity to change given that C was changed. The tool proposes D. Dave changes D and asks the tool for new suggestions. The tool does not return any entities. Dave asks Jenny who suggests no entities as well. Dave is done propagating the change throughout the software system.

3.2 Measuring Effectiveness for a Single Change Set

There exists a variety of metrics that could be used to measure the effectiveness of a tool in assisting developers perform changes. As highlighted earlier, we will focus on the tool's ability in locating the relevant entities that should be changed instead of focusing on the time required to perform the changes themselves.

3.2.1 Defining Recall and Precision

To measure the effectiveness of a tool in propagating changes, we use traditional information retrieval concepts: recall and precision. For our simple example, Fig. 2 shows the entities and their interrelationships. Edges are drawn from A to B and from A to X because the tool suggested that, given that the change set contains A then it should contain B and X as well. For similar reasons, edges are drawn from B to Y and W, and from C to D. We will make the simplifying assumption that a tool provides *symmetric suggestions*, meaning that if it suggests entity F when given entity E, it will suggest entity E when given entity F. We have illustrated this symmetry in Fig. 2 by drawing the edges as undirected edges.

The total set of suggested entities will be called the *Predicted* set; *Predicted* = {B, X, Y, W, D}. The set of entities that needed to be predicted will be called the *Occurred* set; *Occurred* = {B, C, D}. Note that this does not include the initially selected entity (A), which was selected by the developer (Dave) and thus does not need to be predicted. In other words, *Occurred* = *ChangeSet* − {*InitialEntity*}.

We define the number of elements in *Predicted* as P ($P = 5$), and the number of elements in *Occurred* as O ($O = 3$). We define the number of elements in the intersection of *Predicted* and *Occurred* (this intersection is {B, D}) as PO ($PO = 2$). Based on these definitions, we define:

$$Recall = \frac{PO}{O}$$

$$Precision = \frac{PO}{P}$$

In our example, $Recall = \frac{2}{3} = 66\%$ and $Precision = \frac{2}{5} = 40\%$. The rest of this paper will use these definitions of *Recall* and *Precision*.

We will make another simplifying assumption, which is that each prediction by a tool is based on a single entity known to be in the change set. For example, a heuristic may base a prediction on a single element C known to be in the change set, and not on a set of entities such as $\{A, C\}$ known to be in the change set. A further

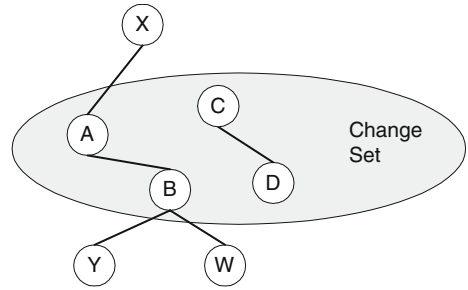

Fig. 2 Change propagation graph for the simple example—an edge between two entities indicates that a tool suggested one when informed about changes to the other one

assumption is that the developer (Dave) will query the tool for suggestions based on every so far suggested entity (which is determined to be in the change set) before querying the Guru (Jenny). An implication of these two simplifying assumptions is that a tool may not do as well in making predictions as it would without these assumptions. Nevertheless, this limitation is not a concern as we are mainly interested in comparing the relative difference between several tools using the same precision and recall model.

Our simplifying assumptions imply that *the ordering of selections and queries to a heuristic are immaterial*. For example, Dave might initially select entity B or C or D instead of A. Further, if Dave had a choice of queries to the tool (say, to get suggestions based on either entity M or N), either query could be given first. Regardless of the selections and ordering, the values determined for *Precision* and *Recall* would not be affected. The *Development Replay* approach depends on change sets that are recovered from the source control system to measure the effectiveness of a tool. These recovered change sets do not record the ordering of selections. So, not only do our assumptions simplify our analysis, they avoid the need for information that is not available in source control systems.

There is an interesting implication of our assumptions, as we will now explain. In Fig. 2, within the change set, there are two connected components, namely $\{A, B\}$ and $\{C, D\}$. With an *ideal* tool, which could be used to predict all entities in a change set without recourse to a Guru, there would necessarily be exactly one connected component. If there is more than one connected component, each of these components, beyond the initial one, implies a query to a Guru. In other words, if CC is the number of connected components and G is the number of queries to the Guru, then G = CC − 1. With an ideal tool, CC = 1 and G = 0, while with the worst tool, CC = N and G = N − 1, where N is the number of entities in the change set. Based on our previous definition of *Recall*, it can be proven that

$$Recall = 1 - \frac{(CC - 1)}{(N - 1)}$$

This is the *Recall* formula actually used in our analysis.

In addition to using *Recall* and *Precision*, we can calculate the *F*-measure which is a weighted harmonic mean of the recall and precision metrics (van Rijsbergen, 1979):

$$F_\beta = \frac{(\beta^2 + 1) * Precision * Recall}{\beta^2 * Precision + Recall}$$

The β ranges between 0 and infinity to give varying weights to recall and precision. For example to indicate that recall is half as important as precision, β would have a value of 0.5. A value of 1.0 indicates equal weight for recall and precision. A value of 2.0 indicates that recall is twice as important as precision. F_0 is the same as precision, F_∞ is recall. Higher values for F correspond to better effectiveness. Due to the dangers of failing to propagate a change to a source code entity, it may be desirable to assign β a value of 2.0 to indicate the higher importance of recall. Alternatively a senior developer may prefer not to waste a great deal of time investigating irrelevant entities, therefore she/he would prefer a tool with high precision. She/he would use a β value of 0.5. The maximum possible

value for the F-measure is 1.0 when both recall and precision are 1.0. The lowest possible value for F-measure is 0 when either recall or precision is zero.

For the results presented in this paper, we show the precision and recall values as they are more intuitive to reason about their meaning. A high recall would prevent the occurrence of bugs due to missed propagations. A high precision would save the developer's time since she/he will not need to examine incorrect suggestions. It is not clear what is the most appropriate balance that would encourage developers to adopt a particular change propagation tool. Would developers want a tool that is likely to give a large number of incorrect suggestions (*low precision*) but that is likely not to miss any of the entities that should change (*high recall*)? Or would developers prefer a more conservative tool that would suggest few correct suggestions (*high precision*) but miss other relevant entities (*low recall*)? The answers to such questions could be derived through case studies and interviews of software developers of varying experience. The answers are likely to depend on the experience of a developer and their knowledge of the software system being changed. In this paper, we use the F_1-measure which considers precision and recall of equal importance. Using the F_1-measure, we can objectively compare the effectiveness of change propagation tools. We would like the change propagation tools that we develop to perform at least as good as traditional information retrieval applications such as web search engines. An information retrieval practical range for the F_1-measure is between 0.44–0.48 where precision usually lies in the 35–40% range and recall is around 60% (Belkin, 1977).

3.2.2 Other Effectiveness Metrics

Another possible effectiveness metric is a utility function which assigns a value or a cost to each suggested entity. Such a measure is commonly used in the TREC filtering task which focuses on sorting through large volumes of dynamically generated information and presenting to the users the information details which are likely to satisfy their current information requirements (Hull, 1998). The larger the utility score, the better the filtering or, in our case, the more effective the development tool is in assisting developers perform the change. For example,

$$U = 6 * PO - 5 * (P - PO)$$

The utility of a tool (U) for each change set is added up and a final total utility is used to measure the effectiveness of the tool for all the change sets. To prevent a large change set from negatively affecting the overall outcome, a minimum utility is defined (U_{min}) that would be used if the utility for a particular change set is less than U_{min}. The results presented in this paper do not use the utility metric to study the effectiveness of a tool.

3.3 Effectiveness Measures for Multiple Change Sets Over Time

In the previous subsection, we presented metrics to study the effectiveness of a tool in assisting a developer performing a single change (*single change effectiveness metrics*). We are more interested in measuring the effectiveness of a tool when used to perform a large number of changes over an extended period of time. We now present metrics to assess the long term effectiveness of a tool.

3.3.1 Average Effectiveness of a Tool

To measure the effectiveness over time for multiple change sets, the simplest measure is the average effectiveness of a single change effectiveness metric such as the average of the recall or precision for all change sets. To calculate the average we sum up each change set and divide by the number of studied change sets (M):

$$Average\ Recall = \frac{1}{M} * \sum_{i=1}^{M} \left(Recall_i \right)$$

$$Average\ Precision = \frac{1}{M} * \sum_{i=1}^{M} \left(Precision_i \right)$$

3.3.2 Stability/Volatility of the Effectiveness of a Tool

The use of an average to measure the effectiveness of a tool has its limitations in particular, it does not take into account the fact that the effectiveness of a tool may vary widely from one usage to the next. For example, a tool may perform remarkably well for a change set but its performance may be very disappointing for the following change set.

Developers are interested in not only the average performance of a tool but in the stability of its performance as well. Developers would like tools that consistently deliver reasonable performance instead of tools whose performance varies considerably. A tool with high performance may be too volatile and may hinder its adoption by developers who seek a tool which they can depend on and trust for its consistency. This is particularly a concern when the performance of a tool is studied over a long period of time. For example, a tool may be beneficial at the beginning of a project but as the project ages and its design decays the tool may not be as helpful in propagating changes.

We use the standard deviation (σ) of the tool's effectiveness over a large number of change sets to assess the stability of a tool's effectiveness.

3.4 Relative Effectiveness of a Tool Over Time

Practitioners are interested in the relative effectiveness of a new tool in comparison to the tools they are currently using. In other words, practitioners investigate whether they should adopt tool A or tool B, or whether they should stick with their current tools. In the previous subsection, we proposed the use of the average and the standard deviation to measure the effectiveness of a single tool. We now focus on metrics to objectively compare the effectiveness of several tools over time.

Simply comparing the average effectiveness for two tools is not sufficient. For example, a particular change propagation tool may slightly outperform another tool; but once the costs associated with adopting the better performing tool are considered, it may not be worthwhile to adopt it. We therefore should ensure that difference in performance is practical to warrant adopting the new tool and we need to perform a statistical test of significance to ensure that the difference is not simply due to chance. In short to adopt a new tool, we should determine that the difference

in average is statistically significant and that the difference is large enough to matter in practice (i.e., practically significant).

We use statistical paired tests to study the difference between the performance of tools. Our statistical analysis assumes a 1% level of significance (i.e., $\alpha = 0.01$). We formulate the following test hypotheses:

$$H_0 : \mu(Perf_A - Perf_B) = 0$$
$$H_A : \mu(Perf_A - Perf_B) \neq 0$$

$\mu(Perf_A - Perf_B)$ is the population mean of the difference between the performance of each tool for the same change sets. If the null hypothesis H_0 holds (i.e., the derived p value $> \alpha = 0.01$), then the difference in average is not significant. If H_0 is rejected with a high probability (i.e., the derived p value $\leq \alpha = 0.01$), then we can be confident about the performance improvements a tool is likely to offer developers performing changes to the source code and we could recommend the adoption of the tool by developers if the difference in mean is practical.

For our empirical case study presented later in the paper, we conducted parametric and non-parametric paired tests when comparing the mean of the performance of tools. For a parametric test, we used a paired t-test. For a non-parametric test, we used a paired Wilcoxon signed rank test which is resilient to strong departures from the t-test assumptions (Rice, 1995). We studied the results of both tests to determine if there are any differences between the results reported by both types of tests. In particular for non-significant differences reported by the parametric t-test, we checked if the differences are significant according to the non-parametric Wilcoxon test. The Wilcoxon test helps ensure that non-significant results are not simply due to the departure of the data from the t-test assumptions. For the results presented later in this paper, the results of both types of tests are consistent.

3.5 Relative Stability/Volatility of the Effectiveness of a Tool

Whereas standard deviation is used to measure the stability of the effectiveness of a tool, it has its limitations. For example if the development process itself is not stable with the team varying their focus or if the quality of the design of the software system is varying over time, then the standard deviation measure is likely to show high volatility. This volatility may be attributed to the development process itself instead of being solely due to the tool's performance. Therefore, we must compare the stability of the performance of a tool against other tools as well.

In this section, we discussed a number of performance metrics. We first focused on metrics that measure the effectiveness of a tool in assisting a developer propagating a change in a single change set. We then discussed the issues surrounding applying such effectiveness metrics to measure the performance of a tool for a large number of changes over time. We highlighted the need to study the effectiveness of a tool and the stability of its performance as well. We also discussed the risks associated with the volatile performance of tools and asserted the need for tools with stable performance so developers would trust their suggestions.

Traditionally, researchers would need to conduct long term studies to examine the performance of each studied tool. Using our definitions of recall and precision, they

could gauge the performance of tools by monitoring the change process and making developers use the tool to suggest entities to change. This is a time consuming process and would require developers to adopt the tools in their development process. Also it would prevent the researchers from experimenting with several tools as they likely would only test one tool at a time. To overcome these limitations, we developed the Development Replay (DR) approach to measure the effectiveness of several tools by replaying back the development history of a software system. The following section details the DR approach.

4 The Development Replay Approach

The Development Replay (DR) approach permits researchers to assess the effectiveness of not-yet-existing or not-yet-adopted change propagation tools. Using the DR approach, researchers can investigate the effectiveness of the various heuristics used by change propagation tools. The reported results are then used to objectively compare several change propagation tools.

Each propagation tool uses heuristics to suggest entities that should be changed. The DR approach offers a software infrastructure which permits researchers to express in a programming language (Perl) the heuristics used by a tool. The heuristic implementation has access to several information sources (such as the sources discussed in Section 2) that are traditionally used to build change propagation tools.

The DR approach uses *actual* change sets that have previously occurred during the development of a software system to measure the effectiveness of change propagation heuristics. The effectiveness is measured by replaying the historical change sets and measuring the performance of the studied change propagation tools for every change set in the lifetime of a project.

An overview of the DR infrastructure is shown in Fig. 3. The DR framework provides a Perl API which a heuristic implementation can use to access timed information concerning the entities, developers, change history, and naming information of the project, such as:

1. The dependencies among the source code entities in the software system such as function call, data usage, and type definition.
2. The co-change history between source code entities such as the fact that a change to function *writeToFile* is always accompanied with a change to function *readFromFile*.
3. The name of the developers who modified each code entities.
4. The other developers who previously changed the same entities as a particular developer.

To ensure the accuracy of the replay process, the state of the software project is tracked throughout the lifetime of a project and the timed information contains details up to the current replayed change. When a particular change occurs to the software system, we can determine the state of the project at that exact moment in its history. Therefore, given the time of a change that is being investigated, the API is able to report details about the project at that exact moment in time instead of reporting details for example about the most recent dependency structure of the software system.

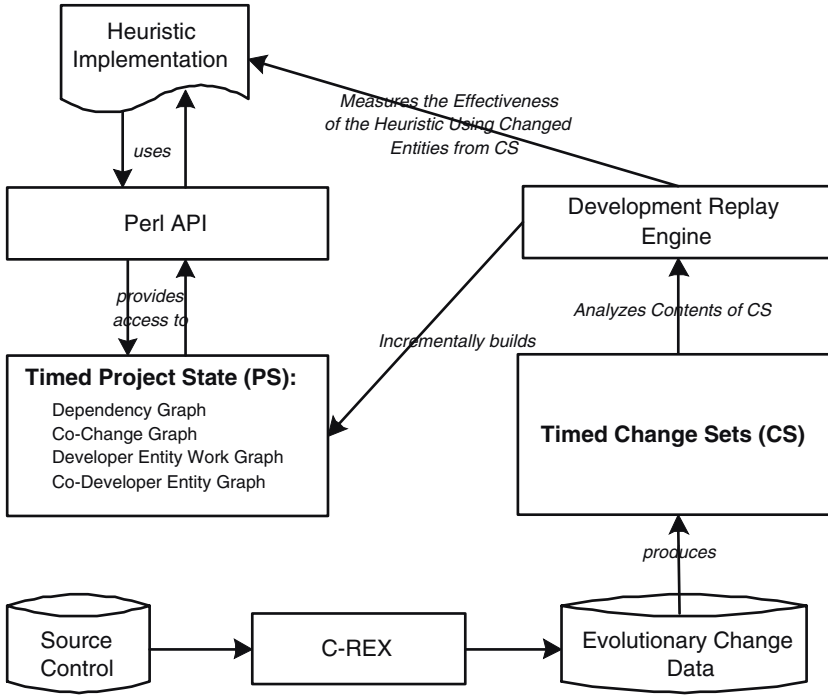

Fig. 3 The development replay infrastructure

The timed project state and the timed change sets data are derived from source control systems using a specialized evolutionary extractor, called C-REX (Hassan and Holt, 2004a; Hassan et al., 2005). The source control system usually tracks the creation, and initial content of each file. In addition, it maintains a record of each change done to a file. For each change, a *modification record* stores the date of the change, the name of the developer who performed it, the specific lines that were changed (added or deleted), a detailed explanation message entered by the developer giving the reason for the change, and other files that were changed with it. The level at which the modification record stores change information (at the file level) is too coarse for our purposes. To build a project state, C-REX preprocesses and transforms the content of the source control system into an optimized and more appropriate representation. Instead of changes being recorded at the file level we record them at the source code entity level (function, variable, or data type definition). Then we can track details such as:

- Addition, removal, or modification of a source code entity. For example, adding or removing a function.
- Changes to dependencies between the modified entities and other source code entities. For example, we can determine that a function no longer uses a specific variable or that a function now calls another function.

The C-REX extractor attaches additional information to each change set concerning the developer performing the change and the purpose of the change as well.

Using the information produced by C-REX, the Development Replay Engine (shown in Fig. 3) incrementally builds a timed project state and simulates the usage of a heuristic to propagate historical changes. Figure 4 gives an overview of how the DR engine reads the time encoded change sets and integrates the results back into the project state by updating, for example, the dependencies between the different entities in the software system. The project state contains all relevant information up to but not including the currently examined change set.

5 Empirical Case Study

We used the DR approach to empirically assess the effectiveness of different change propagation heuristics. We examined the change sets from the source control repositories of several large open source projects.

We did not study all the change sets accessible through the DR infrastructure. Instead using a lexical technique, similar to Mockus and Votta (2000), the DR infrastructure examines the content of the detailed message attached to each modification record and marks all General Maintenance (GM) modifications which are mainly bookkeeping changes. These modifications do not reflect the implementation of a particular feature. For example, modifications to update the copyright notice at the top of each source file are ignored. Modifications that are re-indentation of the source code after being processed by a code beautifier pretty-printer are ignored as well. We do not analyze GM modifications as they are rather general and we do not expect any tool heuristics to effectively predict the propagation of changes in these modifications.

We classified the remaining modification records into two types:

- Records where entities are added, such as the addition of a function, and
- Records where no new entities are added.

We chose to study the change propagation process using only modification records where no entities were added. This choice enables us to compare different change propagation heuristics fairly, as it is not feasible for any heuristic to predict propagation to or from newly created entities. We note that we still use the records where entities are added to build the historical co-change information but we do not measure the performance of any heuristic using these records. Furthermore, to avoid penalizing heuristics which make use of historical information, we do not measure the performance of the heuristic for the first 250 modification records for a software system. By skipping the first few modification records, we are giving the heuristic enough time to build a historical record of changes to give useful suggestions.

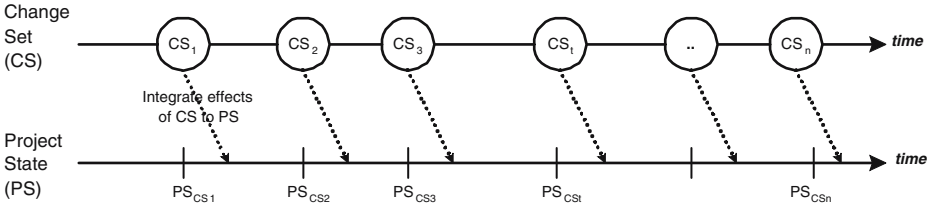

Fig. 4 Maintaining the timed project state incrementally

Table 1 Classification of the modification records for the studied systems

Project name	All records	GM records	New entities records	Studied records
NetBSD	25,839 (100%)	6,204 (24%)	4,086 (16%)	15,567 (60%)
FreeBSD	36,635 (100%)	7,703 (21%)	8,070 (22%)	20,862 (57%)
OpenBSD	13,653 (100%)	2,741 (20%)	2,743 (20%)	8,169 (60%)
Postgres	6,199 (100%)	1,461 (23%)	1,514 (24%)	3,224 (52%)
GCC	7,697 (100%)	901 (12%)	1,114 (14%)	5,682 (74%)

Table 1 gives a breakdown of the different types of modification records in the software systems we studied. The studied modification records represent on average 60% of all the available records in the history of a project, after removing GM modifications and modifications where entities are added. We believe that the studied modification records are representative of changes done to large software projects throughout their lifetime.

In Hassan and Holt (2004b), we presented a study of the change propagation process in large software systems. The study used five open software systems. The studied systems have been developed for the last 10 years and in total have over 40 years of historical modification records stored in their source control system. Table 2 lists the type of the software system, the date of initial modification processed in the source control data, and their implementation programming language.

The study examined which of the information sources (presented in Section 2) are good indicators of change propagation. We studied sequentially through time all modification records that are not General Maintenance (GM) records and where no entities were added. For each modification record, we replayed the change propagation process as outlined in Section 3. We then measured the effectiveness of heuristics based on several information sources. Each row in Tables 3, 4, 5, 6 shows the average precision and recall across change sets in a particular project for every studied heuristic. The row titled "ALL" shows the average precision and recall across all change sets in all studied projects (i.e., *change set weighted average*). The "Average" row shows the non-weighted average precision and recall across all projects. The "*F*-measure" row displays the *F*-measure derived from calculating the average precision and recall from the prior row.

The main results of our prior study are:

- Developer information in the studied software systems is not a good indicator of change propagation. The concept of code ownership is not strictly followed in these systems (Bowman and Holt, 1999) [$average\ recall = .74, average\ precision = .01$].

Table 2 Characteristics of the studied systems

Project name	Project type	Start date	Files	Prog. lang.
NetBSD	OS	March 1993	15,986	C
FreeBSD	OS	June 1993	5,914	C
OpenBSD	OS	Oct 1995	7,509	C
Postgres	DBMS	July 1996	1,657	C
GCC	C/C++ Compiler	May 1991	1,550	C

Table 3 Effectiveness of change propagation heuristics (presented in Section 2) for the five studied software systems

Project	DEV		HIS		CUD		FIL	
	Recall	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*
NetBSD	0.74	0.01	0.87	0.06	0.37	0.02	0.79	0.16
FreeBSD	0.68	0.02	0.87	0.06	0.40	0.02	0.82	0.11
OpenBSD	0.71	0.02	0.82	0.08	0.38	0.01	0.80	0.14
Postgres	0.78	0.01	0.86	0.05	0.47	0.02	0.77	0.12
GCC	0.79	0.01	0.94	0.03	0.46	0.02	0.96	0.06
All	0.73	0.01	0.87	0.06	0.39	0.02	0.81	0.13
Average	0.74	0.01	0.87	0.06	0.42	0.02	0.83	0.12
F-measure	0.02		0.11		0.04		0.21	

- Code structure dependency relations, such as Call, Use, and Define (CUD), are not a good indicators of change propagation in comparison to historical co-change or code layout (same file) information. On average only 42% of entities to which a change should propagate are due to CUD relations [$average\ recall = .42, average\ precision = .06$].
- Code layout (same file) information is a better indicator of change propagation in comparison to the historical co-change information. Unfortunately, building a tool which uses only the code layout information is not sufficient as it will only guide developers to examine entities in the current file. Thus entities in other files to which changes have to be propagated will never be suggested using such a tool [$average\ recall = .83, average\ precision = .12$].
- Historical co-change information is the best practical indicator of change propagation. Although its performance is not as good as the code layout information since it has a lower precision, it is still capable of assisting developers in propagating changes across layout boundaries (*i.e.* changes to entities that are not in the same file) [$average\ recall = .87, average\ precision = .06$].

Based on our results that show that historical information is the best practical indicator of change propagation, we sought to develop heuristics that would improve the precision of historical co-change information. We investigated several techniques to reduce the set of suggested entities to ensure that only the relevant

Table 4 Performance of FREQ(A) heuristics for the five studied software systems

Project	FREQ(60)		FREQ(70)		FREQ(80)	
	Recall	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*
NetBSD	0.43	0.49	0.35	0.60	0.30	0.66
FreeBSD	0.40	0.49	0.32	0.60	0.27	0.66
OpenBSD	0.38	0.54	0.31	0.63	0.27	0.68
Postgres	0.43	0.45	0.34	0.56	0.29	0.63
GCC	0.41	0.51	0.32	0.61	0.27	0.68
All	0.41	0.50	0.33	0.60	0.29	0.66
Average	0.41	0.50	0.33	0.60	0.29	0.66
F-measure	0.45		0.43		0.40	

Table 5 Performance of RECN(M) heuristics for the five studied software systems

Project	RECN(2)		RECN(4)		RECN(6)	
	Recall	*Precision*	*Recall*	*Precision*	*Recall*	*Precision*
NetBSD	0.42	0.41	0.52	0.30	0.58	0.24
FreeBSD	0.43	0.39	0.54	0.27	0.61	0.21
OpenBSD	0.39	0.45	0.48	0.35	0.55	0.28
Postgres	0.42	0.33	0.54	0.22	0.62	0.17
GCC	0.31	0.49	0.42	0.37	0.49	0.30
All	0.41	0.41	0.51	0.30	0.58	0.24
Average	0.39	0.41	0.50	0.30	0.57	0.24
F-measure	0.40		0.37		0.34	

entities to a change are suggested (*improving precision*), while in the same time ensuring that we do not miss suggesting entities that should change (*maintaining high recall*). We explored a few filtering techniques that would reduce the number of entities returned by heuristics which use historical co-change information:

- *Frequency* techniques return the most frequently related (co-changed) entities up to some threshold. This technique is based on the fact that the distribution of change frequency seems to follow a *Zipf* (Zipf, 1949) distribution which indicates that a limited number of entities tend to change frequently and a large number of entities change very infrequently.
- *Recency* techniques return entities that were related (co-changed) in the recent past. These techniques support the intuition that development tends to focus on related functionality and code during particular time periods.
- *Hybrid* techniques combine *Frequency* and *Recency* techniques using counting or some type of exponential decay function as done by (Graves et al., 2000; Hassan and Holt, 2005) to predict faults in software systems and assist managers in allocating testing resources.
- *Random* techniques randomly pick a set of entities to return up to some threshold such as a count. This technique might be used when there is no frequency or recency data to prune the results.

Table 6 Performance of FREQFIL(80, 10) and FREQCUD(70, 10) heuristics for the five studied software systems

Project	FREQFIL(80, 10)		FREQCUD(70, 10)	
	Recall	*Precision*	*Recall*	*Precision*
NetBSD	0.52	0.51	0.43	0.47
FreeBSD	0.50	0.49	0.40	0.45
OpenBSD	0.50	0.49	0.40	0.46
Postgres	0.48	0.48	0.43	0.44
GCC	0.54	0.48	0.41	0.48
All	0.51	0.49	0.42	0.46
Average	0.51	0.49	0.42	0.46
F-measure	0.50		0.44	

We investigated two families of heuristics—FREQ(A) and RECN(M). Both heuristic families filter entities from the historical co-change information (HIS) using recency and frequency techniques:

- *FREQ(A)*: given a changed entity E, the FREQ(A) heuristics would suggest all entities that have changed with E at least twice in the past and changed more that A% of the time with E.
- *RECN(M)*: given a changed entity E, the RECN(M) heuristics would suggest all entities that have changed with E in the past M months.

We used the DR approach to measure the performance of these two heuristics. We experimented with values $A = \{40, 60, 80\}$ for the FREQ heuristics and $M = \{2, 4, 6\}$ for the RECN heuristics. The performance results for the five studied systems are summarized in Table 4 for FREQ heuristics and Table 5 for RECN heuristics.

Tables 4 and 5 show that although both FREQ and RECN heuristics have lower recall than the HIS heuristic (shown in Table 3), both FREQ and RECN have much higher precision. Also the FREQ heuristics outperform the RECN heuristics. Comparing the top performing FREQ heuristic—FREQ(60) and the top performing RECN heuristic—RECN(2), we note the following:

- For all change sets (ALL) and for each specific project's change sets except for GCC, the difference in mean between FREQ's recall and RECN's recall is not statistically significant.[1] For GCC, the improvement of 0.10 (32%) in FREQ's recall in comparison to RECN's recall is statistically significant.
- For all change sets (ALL) and for each specific project's change sets except for GCC, the difference in mean between FREQ's precision and RECN's precision is statistically significant (p-value < 0.0001). The improvement in precision between FREQ and RECN for each project is as follows—NetBSD: 0.08 (+19%), FreeBSD: 0.1 (+26%), OpenBSD: 0.09 (+20%), Postgres: 0.12 (+37%), ALL: 0.09 (+22%).

The improvement in effectiveness of the FREQ heuristic in comparison to the RECN heuristic is statistically significant. The change sets for the GCC project exhibit different characteristics than the other project. This may be due to the fact that the GCC project during the studied time period was mainly in a maintenance phase with few new features being added to it.

5.1 Enhancing the Performance of FREQ(A) Heuristics

We sought to improve the performance of the FREQ(A) heuristics by increasing the number of suggested entities that are relevant to a change while maintaining a high precision. We adopted an approach which *relaxed* the FREQ(A) heuristic by

[1] Using a parametric paired t-test and a non-parametric paired Wilcoxon signed rank test. The t-test is performed on the square root of the precision/recall for each change set to ensure that the data has a normal distribution, a requirement for the t-test. Due to the large number of change sets used in our analysis, the normality of the data is not a major concern as the t-test is a robust test. Nevertheless we ensure the normality to guarantee the validity of our results.

incorporating other information source to suggest additional entities. In particular we defined two extensions to the FREQ(A) heuristics:

- *FREQFIL(A, B)*: For a changed entity E, FREQFIL(A, B) returns the same entities as FREQ(A). In addition, it returns all entities defined in the same file as E that have changed with E in the past at least twice and more that (A–B)% of the time.
- *FREQCUD(A, B)*: For a changed entity E, FREQCUD(A, B) returns the same entities as FREQ(A). In addition, it returns all entities related to E through a CUD relation that have changed with E in the past at least twice and more that (A–B)% of the time.

We re-ran our experiments using the DR infrastructure on the five studied systems using values for $A = \{60, 70, 80\}$, and $B = \{10, 20, 30\}$. We produced nine results for each extension—18 results in total. The results for the best performing FREQFIL and FREQCUD are shown in Table 6.

The best performing (highest F-measure) FREQFIL(A,B) extension was for A = 80 and B = 10, with precision = 0.49, recall = 0.51 and F-measure = 0.50. The results indicate that the FREQFIL(80,10) heuristic can on average suggest to a developer half of all entities to which a change must be propagated and that half of its suggestions are correct. Whereas the best performing FREQCUD(A,B) extension was for A = 70 and B = 10, with precision = 0.42, recall = 0.46, and F-measure = 0.44.

Examining the FREQFIL and FREQCUD results shown in Table 6, we observe that for all change sets (ALL) and for each specific project's change sets, the FREQFIL's recall and precision are better than FREQCUD's except for GCC where both precision's are equal. The difference in mean between FREQFIL's recall/precision and FREQCUD's recall/precision is statistically significant (p-value < 0.0001).

Comparing the FREQFIL results (shown in Table 6) to the FREQ results (shown in Table 4), we observe that:

- For all change sets (ALL) and for each specific project's change sets, FREQFIL's recall is better than FREQ's recall. Also the difference in mean between FREQFIL's recall and FREQ's recall is statistically significant (p-value < 0.0001). The improvement in recall between FREQFIL and FREQ for each project is as follows—NetBSD: 0.09 (+21%), FreeBSD: 0.1 (+25%), OpenBSD: 0.12 (+32%), Postgres: 0.05 (+12%), GCC: 0.13 (+32%), ALL: 0.1 (+24%).
- For all change sets (ALL) and for each specific project's change sets except for OpenBSD, the difference in mean between FREQFIL's precision and FREQ's precision is not statistically significant. For OpenBSD the difference is statistically significant (p- value < 0.0001)—OpenBSD FREQ's precision is 0.05 (+10%) better than the precision of its FREQFIL heuristic.

Based on our analysis we are over 99% confident that the improvement in performance of the best performing FREQFIL heuristic over the best performing FREQCUD and FREQ heuristic is statistically significant for all the studied systems. Table 7 summarizes the results of our analysis. In our analysis we compared

Table 7 Summary of the analysis of four change propagation heuristics

Compared heuristics	Improvement in *Recall*	Improvement in *Precision*
FREQ(60) vs RECN(2)	*GCC*: RECN offers a 10% statistically significant improvement *Other systems*: difference is not statistically significant	*All systems*: FREQ offers a 19–37% statistically significant improvement
FREQFIL(80,10) vs. FREQCUD(70,10)	*All systems*: FREQFIL offers a 12–32% statistically significant improvement	*All systems*: FREQFIL offers a 6–9% statistically significant improvement
FREQFIL(80,10) vs. FREQ(60)	*All systems*: FREQFIL offers a 12–32% statistically significant improvement	*OpenBSD*: FREQ offers a 10% statistically significant improvement *Other systems*: Difference is not statistically significant

the performance of four heuristics: FREQ(60), RECN(2), FREQFIL(80,10), and FREQCUD(70,10). Our analysis required us to perform the following three statistical tests to compare the performance of the various heuristics: FREQ(60) vs. RECN(2), FREQFIL(80,10) vs. FREQCUD(70,10), and FREQFIL(80,10) vs. FREQ(60). Given the number of statistical tests and to avoid the discovery of significant results by chance, we employ a Bonferroni correction procedure (Miller, 1981; Rice, 1995) to derive a stricter significance test ($\alpha = 0.01/3 = 0.003$). Our statistical findings still hold under the stricter significance level derived using the Bonferroni procedure.

To ensure that the performance results are stable over a long period of time, we compared the standard deviation of the precision and recall of the best performing FREQFIL and FREQCUD heuristics We found that the standard deviations of both heuristics are statistically equivalent. The good performance of the FREQFIL heuristic encourages us to explore building a change propagation tool that uses such a heuristic to suggest entities.

In summary, the DR approach permitted us to investigate the effectiveness of a number of heuristics, each corresponding to a not yet developed software tool, with no cost associated with conducting long term case studies or even building such tools. The results show that a tool that uses historical co-change information combined with code layout (same file) information is likely to outperform tools that are based solely on either historical co-change information, code layout, or code structure information. Using these findings, we are more confident in building such a tool and conducting more costly long term case studies to validate the effectiveness of such a tool.

In our current analysis, we associated an equal importance to precision and recall. Examining Table 4 for FREQ heuristics and Table 5 for RECN heuristics, we note that FREQ heuristics always have better precision than RECN heuristics and that

RECN heuristics usually have better recall than FREQ heuristics. If we were to consider recall twice as important as precision ($\beta = 2.0$), then the F_2-measures for FREQ tools would be (0.43, 0.36, and 0.32), whereas the F_2-measures for RECN heuristics would be (0.39, 0.44, and 0.45). In this case, the performance of the RECN heuristics is better. We could then explore improving the RECN heuristics instead of the FREQ heuristics. We could also explore creating a heuristics that combines the RECN and FREQ heuristics. For example, we could develop a heuristic that relaxes the FREQ heuristic by returning all entities that co-changed recently (RECN) with the changed entity. The DR approach permits us to easily investigate these additional heuristics.

6 Threats to Validity and Limitations of Results Derived through the DR Approach

We now discuss the limitation of the DR approach and the applicability of the results derived through the approach. The DR approach permits the empirical evaluation of the effectiveness of change propagation tools. All empirical research studies should be evaluated to determine whether they were able to measure what they were designed to assess. In particular, we need to determine if our findings that a particular heuristics/tool is more effective than another one are valid and applicable or are they due to flaws in our experimental design. Four types of tests are used (Yin, 1994): construct validity, internal validity, external validity, and reliability.

6.1 Construct Validity

Construct validity is concerned to the meaningfulness of the measurements—do the measurements quantify what we want them to? The main conjecture of our work is that developers desire a tool that permits them to perform changes quickly and accurately. We focus on the accuracy of the tool instead of the time required to perform the change itself. The time required to perform a change is likely to be highly dependent on the developer performing the change. Moreover, the time needed for each change is difficult to track since most practitioners rarely keep detailed records of their time.

The DR approach assumes that the accuracy of a tool in propagating changes is a sufficient reason to encourage the adoption of a particular tool in the software development process. The approach does not tackle the issues of the tool's user interface and the developer's interaction with the tool as well. For example a tool with a complex user interface may be abandoned by its users. Also some developers may be more proficient users of a tool than others. The approach also assumes that the training costs for adopting a tool is negligible and should not be a major concern. That last point may not be a major concern given we are interested in the long term benefits of adopting a particular tool. Nevertheless, it is a limitation of the approach.

The CUD code dependency heuristic used in our study corresponds to simple static dependency browsers that are integrated nowadays in most modern software development environments. All heuristics derived from the HIS heuristic require a learning period of around 200 change sets before they would yield useful results. Using the DR approach we could explore enhancing the CUD and HIS heuristics or

we could experiment with heuristics derived from other software development information.

6.2 Internal Validity

Internal validity deals with the concern that there may be other plausible rival hypotheses to explain our findings—can we show that there is a cause and effect relation between the usage of a specific tool and the developers ability to propagate changes accurately throughout a software system, or are there other possible explanations?. The DR approach measures and compares the performance of software development tools by simulating their usage using *actual* change sets which are recovered from source control repositories. By keeping all other project variables constant and simulating the usage of a tool through replay we have a clear cause and effect relation between the simulated tool usage and the effectiveness of the tool. That said the introduction of a tool as part of the development process may affect the type of changes that developers are likely to perform. For example, a tool that reduces the time needed to perform rather complex changes may encourage developers to perform more complex changes instead of opting for workarounds for these complex changes (Baniassad et al., 2002). Moreover the performance of the tool may depend on other factors that are not modeled by the DR approach. Software development is a complex process with a number of factors and facets interacting together and affecting its outcome (Lehman and Belady, 1985). The DR approach uses source control systems to recover and recreate the historical state of a software project, unfortunately a large number of issues and factors surrounding a software system are not present in the source control data (such as personnel communications and knowledge that resides in the heads of the software developers). The DR approach can integrate additional knowledge present in other software development repositories such as mailing list repositories to assist in improving the accuracy of the development replay process. Unfortunately, automated integration of mailing list information is not an easy task as the data is not as structured as source code and change data which are stored in source control systems.

We make use of change sets recovered from source control systems of large open source projects to measure the performance of specific tools. We make the assumption that each change set contains only changes that are related, i.e., that involve a change propagation. In principle, it is possible that a developer may check in several unrelated entities as part of the same modification record. For our purposes, we assume that this occurs rarely. We believe that this is a reasonable assumption based on the development process employed by the studied open source projects and discussions with open source developers (Bauer and Pizka, 2003; Mitchell, 2000; Weinberg, 2003). In most open source projects, access to the source code repository is limited. Only a few selected developers have permission to submit code changes to the repository. Changes are analyzed and discussed over newsgroups, email, and mail lists before they are submitted (Cubranic and Murphy, 2003; Mockus et al., 2000; Ye and Kishida, 2003). We believe that this review process reduces the possibility of unrelated changes being submitted together in a modification record. Moreover, the review process helps ensure that changes have

been propagated accurately in most cases. Thus most change sets would contain a complete propagation of the changes to all appropriate entities in the software system. Recent work by Chen et al. (2004) cautioned of the reliability of open source change logs. Change logs are summarizations of the purpose of changes that occurred to a software system as it evolves and are usually stored in a single file called the ChangeLog file. They are used to provide a high level overview of changes. The quality of these change logs is not a concern for the DR approach. The DR approach builds the change sets using the source control database instead of relying on change logs which tend to omit a large number of details about the project's evolution.

6.3 External Validity

External validity tackles the issue of the generalization of the results of our study—can we generalize our results to other software systems and projects? We believe that the external validity of our results is reasonably high.

The DR approach uses detailed historical records stored in source control systems. This ensures that the studied code development process is a realistic process which involves experienced developers working on large software systems over long periods of time. Alternatively, we could have performed controlled experiments which would run for a limited time. In that case we would not be able to confidently simulate realistic change patterns since we would not have access to experienced and knowledgeable developers to perform simulated modifications to the source code.

Concerning the results in our case study, we chose to study systems with a variety of development processes, features, project goals, personnel, and domain to help ensure the generality of our results and their applicability to different software systems. Nevertheless, all studied systems are open source infrastructure software systems with no graphical user interface. Other systems such as those with graphical interface and which may implement business logic such as banking and online purchasing systems may produce different results. Also commercial software systems may exhibit different characteristics than open source systems. Fortunately, the DR approach permits us to easily assess the effectiveness of a particular heuristic or tool for a specific project. Instead of depending on prior results, the DR approach can be used to quickly measure the effectiveness of a variety of tools to the project at hand as long as the project uses a source control system to store changes to its code as it evolves.

6.4 Reliability

Reliability refers to the degree to which someone analyzing the data would reach the same conclusions or results. We believe that the reliability of our study is very high. The data used in our study is derived from source control systems. Such systems are used by most large software systems which makes it possible for others to easily run the same experiments on other data sets to reproduce our findings and tailor the heuristics to their specific project as well.

6.5 Summary of Limitations

Although the DR approach has its limitations, we believe it can greatly assist in assessing the effectiveness of software development tools. The DR approach gives us a "*back-of-the-envelope*" measure of the effectiveness of a tool in assisting developers in propagating changes. We do not advocate fully depending on the results reported by the DR approach, instead the results should be used to evaluate the value of performing more costly analysis such as pilot studies over extended periods of time to gain more concrete and validated results. The DR approach gives us the flexibility of experimenting with a large number of alternative tool designs and ideas with no associated costs or risks as we are using historical data that has already been collected for other purposes. Moreover, the DR approach derives results that are project specific since it can perform its automated analysis using data recovered from the history of a project of interest, instead of relying on results reported for other software projects.

7 Survey of Related Work

The work presented in this paper focuses on two main research themes: change propagation and the use of historical data to assist in software development tasks. In this section, we survey prior results in both areas of research and discuss our work in light of the surveyed research.

7.1 Change Propagation

We have presented a technique that estimates the effectiveness of heuristics used by change propagation tools in assisting developers who are maintaining and enhancing software systems.

Arnold and Bohner give an overview of several models of change propagation (Arnold and Bohner, 1993; Bohner and Arnold, 1996). They propose several tools and techniques that are based on code dependencies and algorithms such as slicing and transitive closure to assist in code propagation. Rajlich (1997) proposes to model the change process as a sequence of snapshots, where each snapshot represents one particular moment in the process, with some software dependencies being consistent and others being inconsistent. The model uses graph rewriting techniques to express its formalism. Our change propagation model (presented in Section 2) builds on the intuition and ideas proposed in these models. It simplifies the change propagation process and uses it as a benchmark to measure the effectiveness of software development tools.

Assessing the effectiveness of a heuristic or a tool in assisting a developer propagate changes can be done through a number of metrics. In this paper we used precision and recall at the change set level. We also introduced the idea of using a utility function which has been traditionally used in the information retrieval community to measure the effectiveness of search techniques.

At first glance our definition of precision and recall may seem the most intuitive one, nevertheless there are a number of other several possible definitions of precision and recall used by others. In particular, given a software system with M

change sets and each change set having N_j changed entities, precision and recall could be defined relative to the changes performed to the software system or for each entity in the software system. For example, a heuristic may be able to suggest 50% of all entities that should change given any changed entity, and 50% of its suggestions may be correct. Or the performance of a heuristic may be measured for each entity, for example if entity A is changed, then the heuristic can predict correctly 50% of all entities that should change with A and 50% of its suggestions may be correct. But for another entity B it can predict only 10% of the entities that should change and 100% of its suggestions are correct. Both metrics measure the performance of a heuristic/tool, the first metric gives a measure that is based on the change (*change based metric*), whereas the second metric gives a metric that is based on each entity (class/function) in a software system (*entity based metric*). We chose to use change based metrics that estimate the overall effectiveness of a heuristic for all change sets. In contrast, an entity based metric may report excellent results for a few entities. Such measurements may give an incorrect impression of the quality of a heuristic given that these few entities with good performance may rarely change instead most of the changes could be done to entities for which the heuristic performs badly.

For change based metrics, we can define precision and recall at different level of details:

- *Query level*: For each changed entity (e_{ji}) in a change set (O_j) the developer queries the tool for suggestions (P_{ji}), the ($Precision_{ji}$) and ($Recall_{ji}$) for the suggestions for each entity (e_{ji}) is measured as follows:

$$Recall_{ji} = \frac{P_{ji} \cap O_j}{O_j}$$

$$Precision_{ji} = \frac{P_{ji} \cap O_j}{P_{ji}}$$

To measure the precision and recall for a number of change sets (M), the average of the precision and recall is taken for all queries in all change sets, i.e.,:

$$Average\ Recall = \frac{1}{\sum_{k=1}^{M} N_k} * \sum_{j=1,\ i=1}^{M,N_j} \left(Recall_{ji} \right)$$

$$Average\ Precision = \frac{1}{\sum_{k=1}^{M} N_k} * \sum_{j=1,\ i=1}^{M,N_j} \left(Precision_{ji} \right)$$

One of the limitations of this metric is that for many changed entities a tool may not be able to give a suggestion ($P_{ji} = \{\}$) implying a precision of 1 and recall of 0, therefore we may inflate the precision of a tool.

- *Change set level*: This is the metric adopted by us and it focuses on measuring precision and recall at the change set level. We chose this metric as it gives us a good measure of the perceived benefit of using a heuristic by a developer to implement a new feature, enhance a specific feature, or fix a particular bug. In

addition, we can use the results of such a metric to monitor the evolution of a software system, for example we can detect the decay of the design of a software system when the recall of a tool that is based on the CUD information drops. This is an indication that changes to a software system are being scattered throughout the code.

- *Period:* this metric combines all the query result sets and all the changed sets (M) for a period of time and measures their overall precision and recall as follows:

$$Overall\ Recall = \frac{\sum_{j=1}^{M}(P_{ji} \cap O_j)}{\sum_{j=1}^{M} O_j}$$

$$Overall\ Precision = \frac{\sum_{j=1}^{M}(P_{ji} \cap O_j)}{\sum_{j=1}^{M} P_{ji}}$$

This metric does not have to worry about queries where a tool returns no suggestions (*i.e.*, $P_{ji} = \{\}$), as it combines the results for all the queries over a period of time. This metric is similar to the query level metric but it aggregates the results over a time period instead of using an averaging technique.

For the DR approach, any of the aforementioned metrics could be used since the main purpose of the DR approach is to compare a number of heuristics using a common effectiveness metric.

The work in the area of change propagation closest to our work is by Briand et al. (1999). Briand et al. study the likelihood of two classes being part of the same change due to an aggregate value that is derived from object oriented coupling metrics. The analysis is done at the class level whereas we perform our analysis at the entity level (such as functions), in other words the suggestions by their approach are coarser than ours. They use an entity based metric for each class in the software system. Briand et al. performed their study on a single small academic object oriented system with at most three developers working on the system at any time and with six developers working on it throughout its lifetime. Our analysis was done on a number of large open source software system with hundreds of developers working on them. Furthermore, the dependency structure of the software system studied by Briand remained constant. This fact permitted them to calculate their object oriented metrics at the beginning of the study period and to use their metric data in their analysis instead of re-calculating the metrics throughout the lifetime of the studied project. The DR approach permits the analysis to be performed on an up-to-date and accurate view of the software system instead of depending on stale data and assuming it does not vary considerably. Such an assumption does not hold for most large projects with an active developer and user base.

7.2 The Use of Historical Data

Several researchers have proposed the use of historical data related to a software system to assist developers gain a better understanding of their software system and its evolution. Cubranic and Murphy (2003) presented a tool that uses bug reports, news articles, and mailing list postings to suggest pertinent software development

artifacts. These information sources (*i.e.*, bug reports and mailing list postings) could be used as information sources by developers and tools to assist in propagating changes. Once these sources of information are integrated into the DR framework, tool adopters can empirically study the effectiveness of this data in assisting developers. Other possible sources of information are design rationale graphs such as presented in Baniassad et al. (2003); Robillard and Murphy (2002). Yet these later approaches require a substantial amount of human intervention to build the data required to assist developers in conducting changes.

Chen et al. have shown that comments associated with source code modifications provide rich indexing for source code when developers need to locate source code lines associated with a particular feature (Chen et al., 2001). We extend their approach and map changes at the source line level to changes in the source code entities, such as functions and data structures. Furthermore, we map changes to the dependencies between the source code entities. We then provide time tagged access to this data through the DR framework. This timed data can then be used to study the benefits of building tools that make use of such data.

Eick et al. presented visualization techniques to explore change data (Eick et al., 1992; Stephen et al., 2002). Gall et al. propose the use of visualization techniques to show the historical logical coupling between entities in the source code (Gall et al., 1998). Xing and Stroulia (2004) investigate using historical change information to uncover code smells and propose code refactorings.

Three works are most similar to our motivation to evaluate the effectiveness of software development tools. These are work by Zimmermann et al. (2004), work by Shirabad (2003) and work by Ying (2003). These works along with our work focus on studying the effectiveness of tools and heuristics in supporting developers during the change propagation process. Work by Zimmermann and Ying uses prior co-changes to suggest other items to change; the suggestions are evaluated against actual co-changes. They do not study the relative performance against other tools (heuristics). In particular, they do not examine other sources of information such as process or entity information. Their results show that historical information is valuable in assisting developer propagate changes throughout the source code for many large open source software systems. Work by Shirabad uses code structure and layout information as well as textual tokens derived from problem reports to suggest entities to change; the suggestions are evaluated against actual co-changes. Shirabad's analysis, done at the file level, shows that textual tokens derived from problem reports are more effective in suggesting entities to propagate changes to, than simply using code structure and layout information. His results, done at the file level, agree with our findings, done at the entity level, which shows that historical co-change information is more effective than simple static dependency information in propagating changes.

Shirabad, Zimmermann and Ying perform their analysis using a batch process where the historical information stored in the source control system is divided into a training and testing periods. The training data set is used to train the system, then the effectiveness of the tool is measured using the testing data. Whereas work by us uses an adaptive approach where each suggestion done by a tool uses information from all prior changes up to the current change. We believe a batch process has its limitations as the tool's performance will not react in a timely fashion to changes and shifts in interest during the development of a software system. Zimmermann et

Table 8 Summary of related work by Briand, Shirabad, Ying, and Zimmermann et al

Work by	Level of analysis	Information sources	Performance metrics	Metric type	Analysis technique	System types
Briand et al.	Class	Entity (CUD)	Likelihood	Entity	Batch	C++ (academic)
Shirabad	File	Entity (layout naming)	Relevance	Entity	Batch	Pascal-like (commercial)
Ying	File	Entity (co-change)	Precision, recall, interestingness	Change based query	Batch	C++, Java (open source)
Zimmermann et al.	Entity, file	Entity (co-change)	Precision, recall, likelihood	Change based period	Batch, adaptive	C, C++, Java Python, Latex (open source)
Hassan and Holt	Entity, file	Entity (all sources), developer, process	Precision, recall, std. deviation, *F*-measure	Change based change set	Adaptive	C (open source)

al. recent work uses an adaptive approach and is similar to our approach in that context.

Work by Zimmermann uses a change based period metric, whereas work by Ying uses a change based query metric. Work by Shirabad defines a relevance metric which considers a suggestion to be relevant if both entities changed together in any change set in the testing period. Furthermore, work by Zimmermann proposes a metric similar to the likelihood metric proposed by Briand. Work by Ying defines an interestingness metric that measures the value of a tool's suggestion against the current dependency structure of a software system but the suggested evaluation approach is a manual approach. For example a change is not considered interesting if a tool proposes that if a *.h* file changes then its corresponding *.c* file should change as well. The DR approach permits the automation of such analysis and is able to statistically show the benefit of a tool against other types of tools.

Ying and Shirabad focus their analysis at the file level whereas Zimmermann et al. and our work focus our analysis at the entity level (and can easily lift our analysis to the file level).

All three approaches use a variety of data mining techniques to perform their analysis such as association rule mining for Zimmermann et al., machine learning algorithms for Shirabad, and market basket analysis for Ying. All data needed for the three approaches are easily available through the DR software infrastructure. Table 8 summarizes these three approaches along with our work and Briand's work.

8 Conclusion and Future Possibilities

In this paper we highlighted the importance of having a good understanding of the change propagation process in software systems. We presented several sources of information that could be used as a basis for change propagation heuristics. We discussed the need to understand the effectiveness of a tool based on these information sources in propagating changes by measuring the tool's performance when used *(1)* by practitioners, *(2)* over an extended period of time, *(3)* to perform real changes, and *(4)* on large software systems. We noted that conducting such experiments is costly and usually not feasible in an industrial setting.

We outlined the Development Replay approach which offers an infrastructure to easily investigate the performance of several change propagation tools without conducting costly long term studies. The approach *automatically* estimates the effectiveness of not yet developed change propagation tools by reenacting their use on change sets recovered from the source control repository of a project. The DR approach permits researchers to experiment ahead of time with a variety of change propagation heuristics to determine the most promising tools to build or adopt. Through the DR analysis, researchers can recognize limitations or possible improvements of a studied tool/heuristic before they conduct long term costly experiments.

Using the DR approach, we analyzed data derived from several open source projects that have been developed for a total of 40 years. Our empirical results show that historical co-change information can be extremely useful in supporting developers who are propagating changes in large software systems. Our results cast doubt on the effectiveness of simple static dependencies which are usually offered in dependency browsers that are integrated in most modern development environments, as good indicators for change propagation. We as well showed that by combining

several heuristics we could improve the effectiveness of tools in supporting developers propagate changes. Our best performing change propagation heuristic can on average suggest to a developer half of all entities to which a particular change must be propagated and that half of its suggestions are correct. The DR approach offers researchers the flexibility to experiment with a variety of other heuristics while ensuring that the effectiveness of these heuristics is measured using actual changes derived from the source control systems of long lived software projects.

Whereas in this paper we explored using the DR approach and software repositories to study the change propagation phenomena, we could use the DR approach to assess the benefits of other maintenance tools or strategies, for example:

- *The benefits of code restructuring strategies on localizing changes to the source code:* refactoring strategies, object oriented technologies, and aspect oriented techniques aim to restructure the source code to assist developers in understanding the code better and to ease future changes by localizing changes. Using historical information and a proposed code restructuring, we could replay the changes using the DR approach and measure the effectiveness of the newly proposed restructuring in localizing prior changes to the code.
- *The benefits of static source code analysis tools on pointing out error prone code:* static source code analysis tools such as lint perform static analysis on the source code and mark areas that are likely to have faults in them. One could experiment with different heuristics to warn developers about potential faults. The heuristics could be studied through the DR approach by replaying the development history and correlating the warnings to actual faults in the source code.

Acknowledgments The authors gratefully acknowledge the significant contributions from the members of the open source community who have given freely of their time to produce large software systems with rich and detailed source code repositories; and who assisted us in understanding and acquiring these valuable repositories. They also thank Lionel Briand for his very helpful comments and suggestions to improve the statistical analysis and presentation of our results.

References

Anquetil N, Lethbridge T (1998, April) Extracting concepts from file names: a new file clustering criterion. In: Proceedings of the 20th International Conference on Software Engineering. Kyoto, Japan, pp 84–93

Arnold R, Bohner S (1993) Impact analysis—toward a framework for comparison. In: Proceedings of the 13th International Conference on Software Maintenance. Montréal, Quebec, Canada, pp 292–301

Atkins D, Ball T, Graves T, Mockus A (1999, May) Using version control data to evaluate the effectiveness of software tools. In: Proceedings of the 21st International Conference on Software Engineering. Los Angeles, California, pp 324–333

Baniassad EL, Murphy GC, Schwanninger C, Kircher M (2002, April) Managing crosscutting concerns during software evolution tasks: an inquisitive study. In: Proceedings of the 1st IEEE International Conference on Aspect-oriented Software Development. Enschede, The Netherlands, pp 120–126

Baniassad EL, Murphy GC, Schwanninger C (2003, May) Design pattern rationale graphs: linking design to source. In: Proceedings of the 25th International Conference on Software Engineering. Portland, Oregon

Bauer A, Pizka M (2003, September) The contribution of free software to software evolution. In: Proceedings of the 6th IEEE International Workshop on Principles of Software Evolution. Helsinki, Finland,

Belkin NJ (1977) The problem of matching in information retrieval. In: Theory and Application of Information Research, the Second International Research Forum in Information Science. Copenhagen, Netherlands, pp 187–197

Bohner S, Arnold R (1996) Software change impact analysis. IEEE Computer Soc

Bowman IT, Holt RC (1999, May) Reconstructing ownership architectures to help understand software systems. In: Proceedings of the 7th International Workshop on Program Comprehension. Pittsburgh, Pennsylvania

Briand LC, Wüst J, Lounis H (1999, August) Using coupling measurement for impact analysis in object-oriented systems. In: Proceedings of the 15th International Conference on Software Maintenance. Oxford, England, UK, pp 475–482

Brooks FP (1974) The mythical man-month: essays on software engineering. Addison Wesley Professional

Chen A, Chou E, Wong J, Yao AY, Zhang Q, Zhang S, Michail A (2001) CVSSearch: searching through source code using CVS comments. In: Proceedings of the 17th International Conference on Software Maintenance. Florence, Italy, pp 364–374

Chen K, Schach SR, Yu L, Offutt J, Heller GZ (2004) Open-source change logs. Empirical Software Engineering 9(197):210

Cubranic D, Murphy GC (2003, May) Hipikat: recommending pertinent software development artifacts. In: Proceedings of the 25th International Conference on Software Engineering. Portland, Oregon, pp 408–419

Eick SG, Steffen JL, Eric J, Sumner E (1992) Seesoft—a tool for visualizing line oriented software statistics. IEEE Trans Softw Eng 18(11):957–968

Fenton N, Pfleeger SL, Glass RL (1994) Science and substance: A challenge to software engineers. IEEE Softw 11(4):86–95

Finnigan PJ, Holt RC, Kalas I, Kerr S, Kontogiannis K, Müller HA, Mylopoulos J, Perelgut SG, Stanley M, Wong K (1997) The software bookshelf. IBM Syst J 36(4):564–593

Gall H, Hajek K, Jazayeri M (1998, November) Detection of logical coupling based on product release history. In: Proceedings of the 14th International Conference on Software Maintenance. Bethesda, Washington, District of Columbia

Gallagher KB, Lyle JR (1991) Using program slicing in software maintenance. IEEE Trans Softw Eng 17(8):751–761

Glass RL (2003) Questioning the software engineering unquestionables. IEEE Softw 20(3):119–120

Graves TL, Karr AF, Marron JS, Siy HP (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661

Hassan AE, Holt RC (2004a, May) C-REX: an evolutionary code extractor for C.

Hassan AE, Holt RC (2004b, September) Predicting change propagation in software systems. In: Proceedings of the 20th International Conference on Software Maintenance. Chicago, USA

Hassan AE, Holt RC (2005, Sept) The top ten list: dynamic fault prediction. In: Proceedings of the 21th International Conference on Software Maintenance. Budapest, Hungary

Hassan AE, Jiang ZM, Holt RC (2005, November) Source versus object code extraction for recovering software architecture. In: Proceedings of the 12th Working Conference on Reverse Engineering. Pittsburgh, USA

Hull DA (1998) The TREC-7 filtering track: description and analysis. In: Voorhees EM, Harman DK (eds) Proceedings of TREC-7, 7th Text Retrieval Conference. National Institute of Standards and Technology, Gaithersburg, USA, pp 33–56

Kiczales G, Lamping J, Menhdhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J, (1997) Aspect-oriented programming. In: Akit M, Matsuoka S (eds) Proceedings of the 11th European Conference on Object-oriented Programming, vol. 1241. Springer, Berlin Heidelberg New York, pp 220–242

Kitchenham BA, Pickard SLPLM, Jones PW, Hoaglin DC, Emam KE, Rosenberg J (2002) Preliminary guidelines for empirical research in software engineering. IEEE Trans Softw Eng 28(8):721–734

Lee EHS (2000) Software comprehension across levels of abstraction. Master's thesis, University of Waterloo

Lehman MM, Belady LA (1985) Program evolution—process of sofware change. Academic, London

Miller RG (1981) Simultaneous statistical inference. Springer, Berlin Heidelberg New York

Mitchell M (2000, October) GCC 3.0 State of the Source. In: 4th Annual Linux Showcase and Conference. Atlanta, Georgia
Mockus A, Votta LG (2000, October) Identifying reasons for software change using historic databases. In: Proceedings of the 16th International Conference on Software Maintenance. San Jose, California, pp 120–130
Mockus A, Fielding RT, Herbsleb JD (2000, June) A case study of open source software development: the apache server. In: Proceedings of the 22nd International Conference on Software Engineering. ACM, Limerick, Ireland, pp 263–272
Parnas D (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12):1053–1058
Parnas D (1994, May) Software aging. In: Proceedings of the 16th International Conference on Software Engineering. Sorrento, Italy, pp 279–287
Penny DA (1992) The software landscape: a visual formalism for programming-in-the-large. PhD thesis, University of Toronto
Perry DE, Porter AA, Votta LG (2000, June) Empirical studies of software engineering: a roadmap. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE)—Future of SE Track. Limerick, Ireland, pp 345–355
Rajlich V (1997) A model for change propagation based on graph rewriting. In: Proceedings of the 13th International Conference on Software Maintenance. Bari, Italy, pp 84–91
Rice J (1995) Mathematical statistics and data analysis. Duxbury
Robillard MP, Murphy GC (2002, May) Concern graphs: finding and describing concerns using structural program dependencies. In: Proceedings of the 24th International Conference on Software Engineering. Orlando, Florida
Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W (1991) Object-oriented modeling and design. Prentice-Hall, Englewood Cliffs, New Jersey
Shirabad JS (2003) Supporting software maintenance by mining software update records. PhD thesis, University of Ottawa
Sim SE, Clarke CLA, Holt RC (1998, June) Archetypal source code searching: a survey of software developers and maintainers. In: Proceedings of the 6th International Workshop on Program Comprehension. Ischia, Italy, pp 180–187
Sniff+ Home Page. Available online at http://www.takefive.com/
Stephen PS, Eick G, Mockus A, Graves TL, Karr AF (2002) Visualizing software changes. IEEE Trans Softw Eng 28(4):396–412
van Rijsbergen CJ (1979) Information retrieval. Butterworths, London. Available online at http://www.dcs.gla.ac.uk/Keith/Preface.html
Weinberg Z (2003, May) A maintenance programmer's view of GCC. In: First Annual GCC Developers' Summit. Ottawa, Canada
Xing Z, Stroulia E (2004, June) Understanding class evolution in object-oriented systems. In: Proceedings of the 12th International Workshop on Program Comprehension. Bari, Italy
Yau S, Nicholl R, Tsai J, Liu S (1988) An integrated life-cycle model for software maintenance. IEEE Trans Softw Eng 15(7):58–95
Ye Y, Kishida K (2003, May) Toward an understanding of the motivation of open source software developers. In: Proceedings of the 22nd International Conference on Software Engineering. ACM, Portland, Oregon, pp 419–429
Yin RK (1994) Case study research: design and methods. Sage, Thousand Oaks, California
Ying AT (2003) Predicting source code changes by mining revision history. Master's thesis, University of British Colombia
Zimmermann T, Weißgerber P, Diehl S, Zeller A (2004, May) Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering. Edinburgh, UK
Zipf GK (1949) Human behavior and the principle of least effort. Addison-Wesley

Ahmed E. Hassan is an Assistant Professor at the University of Victoria, where his research interests include Mining Software Repositories (MSR). He spent the early part of his career helping architect the Blackberry wireless platform at Research In Motion (RIM). He contributed to the development of protocols, simulation tools, and software to ensure the scalability and reliability of RIM's global infrastructure. He worked for IBM Research at the Almaden Research Lab in San Jose and the Computer Research Lab at Nortel Networks (BNR) in Ottawa. He spearheaded the organization and creation of the MSR workshop series at ICSE and its associated research community. He recently co-edited a special issue of the IEEE Transaction on Software Engineering (TSE) on MSR.

Ric Holt is a Professor at the University of Waterloo, where his research interests include visualizing software architecture. His work includes reverse engineering of legacy systems and repairing software architecture. His architectural visualizations have included Linux, Mozilla (Netscape), IBM's TOBEY code generator, and Apache. His previous research includes foundational work on deadlock, development of a number of compilers and compilation techniques, development of the first Unix clone, and authoring a dozen books on programming and operating systems. He is one of the designers of the Turing programming language.