Predicting Change Propagation in Software Systems

Ahmed E. Hassan and Richard C. Holt
Software Architecture Group (SWAG)
School of Computer Science
University of Waterloo
Waterloo, Canada
{aeehassa,holt}@plg.uwaterloo.ca

ABSTRACT

Software systems contain entities, such as functions and variables, which are related to each other. As a software system evolves to accommodate new features and repair bugs, changes occur to these entities. Developers must ensure that related entities are updated to be consistent with these changes.

This paper addresses the question: *How does a change in one source code entity propagate to other entities?* We propose several heuristics to predict change propagation. We present a framework to measure the performance of our proposed heuristics. We validate our results empirically using data obtained by analyzing the development history for five large open source software 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 consistent with these new changes. For example, if the interface for 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 that 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 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 [25]. Brooks cautioned of the risk associated with developers losing grasp of the system as it ages and evolves [23]. Misunderstanding, lack of experience and unexpected dependencies are some of the reasons for failing to propagate changes throughout the development and maintenance of source code.

Mis-propagated changes have a high tendency to introduce hard to find bugs in software systems, as inconsistencies between entities (such as functions) increase. Researchers have proposed various approaches to control the amount of change propagation and to avoid hidden dependencies. These proposals include, most notably, the ideas of information hiding in designing systems by Parnas [24], the Object Oriented (OO) paradigm which focuses on grouping related entities in the same structure to ease modification and understanding [29], and lately Aspect Oriented Programming (AOP) which encapsulates concepts which crosscut the structures defined by the OO paradigm [15].

Moreover, researchers have proposed tools and algorithms to assist developers in understanding their software and determining the extent of propagation for each change to the source code. For example, dependency analysis algorithms such as slicing [12] in association with entity dependency browsers [17], software understanding tools and architecture visualization tools [10] are used by developers and researchers to maintain and evolve large complex long lived industrial systems.

Program dependency relations, such as *call* and *use* have been proposed and used as indicators for change propagation. Thus if a 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 [33]. Given the central role of change propagation in software development, a good understanding of the reasons and drivers for change propagation is needed. Researchers have focused mainly on their intuition and a limited number of observational studies to understand the nature of change propagation in software systems. Using their intuition, they built several tool prototypes. Using small observational studies, they examined the performance of these tools [18, 31].

In this paper, we ask the question:

> *How does a change in one source code entity propagate to other entities?*

The answer to this question helps us understand the value of software development tools in assisting developers by suggesting entities where a particular change will likely propagate during software development and maintenance. We validate our approach empirically using a large data set which is based on the development a history of five large open source software systems, developed for a total of over 40 years by hundreds of developers spread around the globe. We study changes to these large code bases using data derived from their source control repositories. Using this large data set, we follow a two step approach. First we empirically study several general heuristics that predict change propagation. For example, we study the probability that change propagates due to source code entities being in the same file, or having changed in the past together. By studying these general heuristics, we can measure the overall value of the data used by these heuristics. Then we use our newly acquired understanding to build enhanced heuristics and measure their effectiveness in predicting change propagation.

Organization of Paper

The paper is organized as follows. In Section 2, we present source control repositories along with a classification of the data stored in these repositories. In Section 3, we give an overview of the change propagation process and present the steps taken by developers to propagate changes. Then in section 4, we present a framework to study the performance of heuristics that predict change propagation. Several general heuristics for predicting change propagation are proposed in Section 5. Then in Section 6 we empirically study the performance of some of these heuristics. In Section 7, we survey related work in software evolution, program extraction, impact analysis and change propagation. Finally in Section 8 we summarize our work and propose future work directions and challenges based on our results.

2 SOURCE CONTROL REPOSITORIES AND CHANGE

To carry out our study of change propagation, we used data stored in the source control repository. In this section we give an overview of the type of data stored in these repositories. We also present the processing we performed to conduct the study presented herein.

Source control systems such as CVS and Perforce [26] are used by large software projects. These control systems track changes to the source code over time. This permits multiple developers to work simultaneously on a large project without worrying about their changes being lost. Most source control systems reconcile changes made simultaneously by developers working on the same file. Moreover, they keep a full record of all changes to each file in the system. This permits developers at any time to retrieve older versions of the code. Older versions of the code can be studied to get a clearer understanding of the current state of the system. Also in some cases developers may choose to revert back to one of these older versions as they discover that their current version contains bugs or is too complex to maintain or understand.

The data stored in the source repository presents a great opportunity to study the change propagation process for large software systems over extended periods of time. The data collection costs are minimal as the collection is done automatically when changes are done to the source code. For each file in the software system, the repository tracks its creation, and its initial content. 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. For our purposes, the level at which the modification record stores change information (at the file level) is too high. Thus we preprocess and transform 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.

Source control systems, such as CVS, record changes to each file in the software system and do not associate changes occurring to various files as being part of the same modification record. We developed techniques to rebuild this missing association. Using our derived low level information about changes to source code, we can generate dependency relations that reflect the state of the source code when a change occurred. In short, for each change to the source code we can associate other changes that occurred in other files. We also know the dependencies between the source code entities at that moment in time when a change occurred. Furthermore, we have a record of entities which previously changed with each entity.

3 THE CHANGE PROPAGATION PROCESS

Studies indicate that at least fifty percent of the life cycle and budget of large software systems are spent on maintaining it [38]. The maintenance phase lasts for many years after the initial release. As new features are added, others are enhanced, and bugs are fixed, developers are faced with the challenge of determining appropriate propagation of their changes in these large evolving code bases. In this section, we examine the process of change propagation and give a breakdown of the various steps it involves.

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 a file may require a change to propagate to the function that reads data from file. This 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 function text 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.

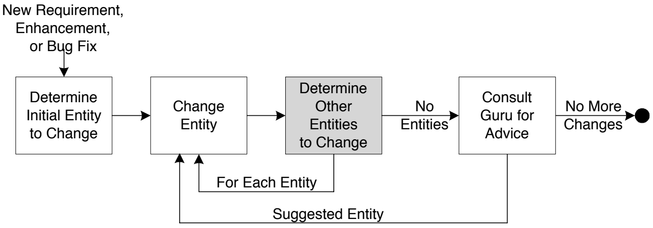
New Requirement, Enhancement, or Bug Fix
Determine Initial Entity to Change
Change Entity
Determine Other Entities to Change
No Entities
Consult Guru for Advice
No More Changes
For Each Entity
Suggested Entity

Figure 1: Model of the Change Propagation Process

In Figure 1, we propose 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 she/he 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, she/he 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 just changed 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 at hand – Ideally all appropriate entities should have been updated to ensure consistent assumptions throughout the system.

The Guru can be a senior developer, a software development tool, or even a suite of tests. 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 no such all knowing developer exists. Therefore, developers find themselves forced to use other forms of advice/information such as the results of test suites or a development tool. 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.

This propagation process is necessary because there are *interdependencies* among the changed entities of a software system. The goal of change propagation is to ensure the consistency of assumptions among these interdependent entities. In many cases these interdependencies are obvious to developers changing the code, based on their experience, domain knowledge, and the output of development tools and code browsers. In other cases, tools such as compilers or linkers point out inconsistencies among interdependent entities by means of error messages. Failure to update any of these interdependent entities would cause the software system to have inconsistencies in its assumptions. As a result new faults may appear in the code.

Developers spend a considerable amount of time trying to correctly propagate a change to other entities. This is a labor intensive task that is error prone and not well understood. We investigate if there are good indicators such as call graph relations that could assist a developer in determining other entities to change. By studying this process, we can measure the value of tools, such as dependency browsers that are provided by modern software development environments.

4 MEASURING THE PERFORMANCE OF CHANGE PROPAGATION HEURISTICS

In the following section, we propose several heuristics to generate the set of entities that should be changed in response to a changed entity – These correspond to the "*Determine Other Entities to Change*" step in Figure 1. In this section we present our approach to measuring the performance of these change propagation heuristics.

In the ideal case, a heuristic would correctly suggest all the entities that represent a change set without asking the Guru for any advice. The worst case occurs when the Guru is consulted to determine each entity in the change set. Referring back to the change propagation model shown in Figure 1, we would like to minimize the number of times the Guru is consulted for an entity to change.

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, one of our heuristics 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 heuristic to suggest another entity that should change if B were changed. The heuristic 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 heuristic for a suggestion for an entity to change given that C was changed. The heuristic proposes D. Dave changes D and asks the heuristics for new suggestions. The heuristic does not return any entities. Dave asks Jenny who suggests no entities as well. Dave is done propagating the change throughout the software system.

Defining Recall and Precision

To measure the performance of a heuristic we use traditional information retrieval concepts: recall and precision. For our simple example, Figure 2 shows the entities and their interrelationships. Edges are drawn from A to B and from A to X because the heuristic suggested that, given that the change set contains A, 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 heuristic provides *symmetric predictions*, meaning that if it predicts entity F when given entity E, it will also predict E when given F. We have illustrated this symmetry in Figure 2 by drawing the edges as undirected edges.

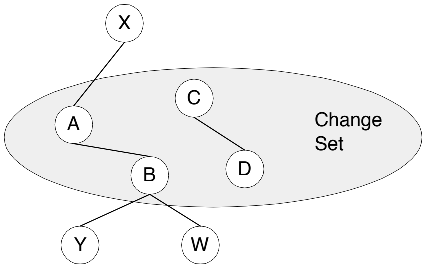

Figure 2: Change Propagation Graph for the Simple Example - An edge between two entities indicates that a heuristic suggested one when informed about changes to the other one.

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 $\{\text{B}, \text{D}\}$) as PO (PO = 2). Based on these definitions, we define:

$$\begin{aligned} Recall &= \frac{PO}{O} \\ Precision &= \frac{PO}{P} \end{aligned}$$

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*. If no elements are predicted ($i.e.$ P and PO are empty), then we define $Precision$ as 1, and if no other elements are changed ($i.e.$ O is empty, we define recall as 1.

We will make another simplifying assumption, which is that each prediction by a heuristic is based on a single entity known to be in the change set. For example, a heuristic may base a prediction on 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 assumption is that the developer (Dave) will query the heuristic 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 our simplifying assumptions is that the heuristics may not do as well in making predictions as they would without these assumptions.

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 heuristic (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 change records from the source control system 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 from the change records.

There is an interesting implication of our assumptions, as we will now explain. In Figure 2, within the change set, there are two connected components, namely {A, B} and {C, D}. With an *ideal* heuristic, 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, 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 heuristic, CC = 1 and G = 0, while with the worst heuristic, 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.

Average Performance

We have presented a framework to measure the recall and precision of a heuristic for a particular change set. To measure the performance over time we sum up the recall and precision for each change set and divide by the number of change sets (M) in the history of a studied project:

$$\begin{aligned} Average\ Recall &= \frac{1}{M} * \sum_{i=1}^{M} \Big(Recall_i \Big) \\ Average\ Precision &= \frac{1}{M} * \sum_{i=1}^{M} \Big(Precision_i \Big) \end{aligned}$$

5 HEURISTICS FOR PREDICTING CHANGE PROPAGATION

In this section, we introduce several heuristics which could be used to predict change propagation by suggesting entities that should change based on an entity that has changed. These heuristics are based on our analysis of the modification records, our intuition and previous research in software engineering.

Each heuristic is characterized by two main aspects:

Heuristic Data Source – This determines the source of data, such as call graph or historical co-change, used by the heuristic algorithm to suggest other entities to change.

Pruning Technique – The pruning technique defines the algorithms used by heuristics to prune their suggested change set. For example, a heuristic may choose to return the ten entities most recently changed in the past with a specific entity instead of returning all entities that ever changed with it. The pruning technique would assist in improving the precision of a heuristic. When no pruning technique is used, a heuristic maximizes recall while sacrificing precision.

Heuristic Data Sources

There exist several sources of data that a heuristic can use to predict the set of entities that should change. These heuristics aim to reduce the number of predicted entities that are not needed to change while ensuring all the entities that should change are predicted. Some of the possible sources of data are the following:

Entity Data
An Entity based heuristic proposes that the change propagation process is dependent on the particular changed entity. For example, a change may propagate to other entities interdependent on the changed entity according to relations such as:

- A *Historical Co-change* records that one entity changed at the same time as another entity. If entity A and B changed together in previous change sets, then they are related via a Historical co-change relation.
- A *Code Structure* 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 that is of a particular type. For example F *Define* T, means F defines a variable of type T.
- A *Code Layout* relation records the location of entities relative to classes, 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.

Developer Data
A Developer based heuristic assumes that change propagates to other entities changed recently or frequently by the same developer. This heuristic is based on the observation that over time developers gain expertise in specific areas of the source code and are likely to modify the same entities in their acquired areas of expertise.

Process Data
A Process based heuristic assumes that change propagation depends on the process employed in the development. For example a change to a particular entity tends to propagate changes to other frequently or more recently changed entities independent of the specific entity that changed, as these recently changed entities may be responsible for a specific feature being modified throughout the software system.

Name Similarity Data
A Name Similarity heuristic assumes that change propagates to entities with similar names, as the similarity in naming indicates similarities in the role of the entities and their usage. We don't use this heuristic in our study, but work in [1] has shown its value in improving automatic clustering of files.

Random Data
A heuristic which uses a random generator to produce the set of suggested entities to change. Such a heuristic assumes that change propagation is a random process that is chaotic and unpredictable. Clearly such a heuristic is counter intuitive and we don't use it in our study instead we focus on the other mentioned heuristics.

Other sources of data may exist. Also the aforementioned data sources can be combined and extended in various ways. For example, another possible heuristic is the co-call heuristic, where A and B both *call* C. A and B may implement similar functionality and a change to A may propagate to B.

Pruning Techniques

There are several techniques to reduce the set of suggested entities by a heuristic. Some of the possible pruning techniques are:

- *Frequency* techniques return the most frequently related entities up to some threshold. For example, the distribution of change frequency seems to follow a *zipf* distribution which indicates that a limited number of entities tend to change frequently and a large number of entities change very infrequently [37].
- *Recency* techniques return entities that were related in the recent past. These techniques support the intuition that development tends to focus on related functionality during particular time periods.
- *Hybrid* techniques combine *Frequency* and *Recency* techniques using counting or some type of exponential

decay function as done by [13] to predict faults in software systems.

- *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.
- *No pruning* returns the results without any deletions.

6 EMPIRICAL STUDIES

Using our definitions of recall and precision we could measure the performance of heuristics by monitoring the change process and making developers use our heuristic tool to suggest entities to change. Unfortunately, this is a time consuming process and would require developers to adopt our tool in their development process. Also it would prevent us from experimenting with several heuristics as we could only test one heuristic at a time. Instead we use the historical change data stored in the source control repository to measure the performance of our heuristics. We study each change set in the history of the project and determine the performance of the proposed heuristics.

We start this section by giving an overview of the software systems and the modification records studied. We follow this by selecting several change propagation heuristics and measuring their performance using the historical data stored in the source control system.

Studied Systems

We used five open software systems to validate our work. 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 1 lists the type of the software system, the date of initial modification processed in the source control data, and the programming language used. We chose to study systems with a variety of development processes, features, project goals, personnel, and domain of the studied software systems to help ensure the generality of our results and their applicability to different software systems.

Application Name	Application 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 1: Characteristics of the Studied Systems

Studied Modification Records

In our empirical work, we studied change sets in several software systems. Empirically, a change set is derived from the modification records stored in a source control repository for the projects. Using a lexical technique, similar to [21], we studied the content of the detailed message attached to each modification record throughout the history of a project. We extracted and removed using this technique all General Maintenance (GM) modifications which are mainly bookkeeping changes. They do not reflect the implementation of a particular feature. These modifications are not considered in our analysis of the change propagation process. 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 chose not consider these GM modifications as they are rather general and we do not expect any heuristics to 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 for our historical based heuristics we still use the records where entities are added or removed to improve future suggestions but we do not measure the performance of any heuristic using these records.

Table 2 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 a representative sample of changes done to large software projects throughout their lifetime.

In our analysis we make the assumption that each modification records contains only related changes, *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 [4, 19, 32]. 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 mailing lists before they are submitted [9, 20, 34]. 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

Application Name	All Records	GM Records	New Entities Records	Studied Records
NetBSD	25,839	6,204	4,086	15,567
	(100%)	(24%)	(16%)	(60%)
FreeBSD	36,635	7,703	8070	20,862
	(100%)	(21%)	(22%)	(57%)
OpenBSD	13,653	2,741	2,743	8,169
	(100%)	(20%)	(20%)	(60%)
Postgres	6,199	1,461	1,514	3,224
	(100%)	(23%)	(24%)	(52%)
GCC	7,697	901	1114	5682
	(100%)	(12%)	(14%)	(74%)

Table 2: Classification of the Modification Records for the Studied Systems

cases. Thus most change sets are likely to contain a complete propagation of a change to all appropriate entities in the software system.

Measuring the Performance of Change Propagation Heuristics

Due to size limitations, we will not address all possible heuristics presented in section 5. We chose to study the following heuristics:

1. Developer Based (DEV): This heuristic returns all entities that were previously changed by the same developer who is performing the current change.
2. Entity Based Historical Co-change (HIS): This heuristic returns all entities that previously changed with the just changed entity.
3. Entity Based Code Structure using Call, Use, and Define (CUD): This heuristic returns all entities that are related to the just changed entity via a *Call*, *Use* or *Define* relation.
4. Entity Based Code Structure using Code Layout (FIL): This heuristic returns all entities that are defined in the same file as the just changed entity.

As these heuristics do not employ any pruning techniques, we expect their precision to be low in comparison with their recall. We chose to present these unpruned heuristics to determine the *maximum* possible recall for heuristics that are based on specific data sources. For example, we would know the best possible recall for a heuristic which uses code structure information or developer information. We can later focus on improving the precision by experimenting with a variety of pruning techniques for each general heuristic. Moreover, the maximum recall indicates as well the probability that a change propagates due to a particular heuristic.

We study all the software systems using these basic heuristics then we combine the best performing ones to build a hybrid heuristics with pruning techniques. The hybrid heuristic will seek a middle ground by suggesting entities that are expected to change (high recall) and in the same time punning incorrect suggestions (high precision). The performance results for the five studied systems are summarized in Table 3.

The results shown are derived by examining sequentially through time all modification records that are not GM record and where no entities were added. For each modification record, the heuristics had to predict the change propagation process outlined in section 4. Then the performance of the heuristic is measured. To avoid penalizing heuristics based on historical data as they work on building a historical record of changes to give useful predictions, we did not measure the performance of the heuristics for the first 250 modification records for a software system.

Discussion of Results

Examining Table 3, we notice that DEV heuristic has a high recall but low precision. This indicates that through the lifetime of a project, developers of the studied systems tend to work on many entities that are not necessarily related. They do not focus on a specific set of subsystems and entities. Also the concept of code ownership is not strictly adhered to [7]. The very low precision values discourage us from pursuing this heuristic for the design of the hybrid heuristics later in this section.

Again looking at Table 3, we conclude that the FIL heuristic has the best balance of precision versus recall compared to the other three heuristics. It has a high recall without considerably sacrificing its precision. We consider this as empirical validation for our previous work that hypothesized that a source code file represents a coherent conceptual grouping of related items [14]. It would be interesting to compare these results to an Object Oriented heuristic which would suggest that change propagates to entities in the same object. Using the FIL heuristic is not sufficient as it will only guide developer to examine entities in the current file. Thus entities in other files for which changes have to be propagated will never be suggested using this heuristic.

Further examination of the results in Table 3 reveals that the code structure dependency relation (CUD) is not a good indicator of change propagation in comparison to historical records or code layout information (on average only 42% of entities for which a change should propagate are due to CUD relations). This is an interesting finding; it suggests that code and relation browsers are not particulary effective at indicating which entities to propagate changes to. Software development environments should offer more advanced code and relation browsers that use other sources of data to show the interdependencies between entities in a software system. These new browsers could assist developers as they maintain their evolving systems. For example, Table 3 shows that the HIS heuristic outperforms the CUD heuristic. It has the best recall and second best precision after the FIL heuristic. These results suggest that the development history can

Application	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
Average	0.74	0.01	0.87	0.06	0.42	0.02	0.83	0.12

Table 3: Performance of the Change Propagation Heuristics for the Five Studied Software Systems

be of considerable value to maintainers of software systems.

Although, the systems used in our study represent several types of software systems, they are all infrastructure software systems with no graphical user interface. Other systems such as those with graphical interface or which may implement business logic such as banking and online purchasing systems may produce different results. By investigating other types of software systems, we can determine the generality of our findings. Nevertheless, the findings suggest that historical information can assist developers in propagating changes and that code structure (*i.e.* dependency browsers) are not as helpful as historical information or code layout (FIL) information.

Improving Precision With Hybrid Heuristics

The results shown in Table 3 indicate that the FIL and HIS heuristic are the two best performing heuristics. Unfortunately, the FIL heuristic can only suggest entities in the same file, which is a limiting factor. We investigated pruning the suggestions returned by the HIS heuristic to increase its precision. We then combined the results with suggestions from the FIL heuristic. This lead to the development of a family of hybrid heuristics (HYB). These heuristics are based on the following intuition:

1. Entities that changed together in the past have a high tendency to change together in the future as we observed in the performance of the HIS heuristic.
2. Developers tend to group related entities in the same file, as shown through the performance of the FIL heuristic.

Based on this intuition, we define a family of heuristics HYB(A,B) with parameters A and B. Given a changed entity E, the HYB heuristics return:

1. Entities that have changed with E in the past at least twice together and more that A% of the time. This prunes entities from the HIS heuristic.
2. Entities defined in the same file as E that have changed with E in the past at least twice together and more that (A-B)% of the time. This *relaxes* the previous rule by using the fact that the entities reside in the same file to compensate for a lower percentage of co-change.

We developed this heuristic family through trial and error using the Postgres historical development database. For example, we attempted to use the recency of a co-change to prune entities but the performance of such heuristics were disappointing.

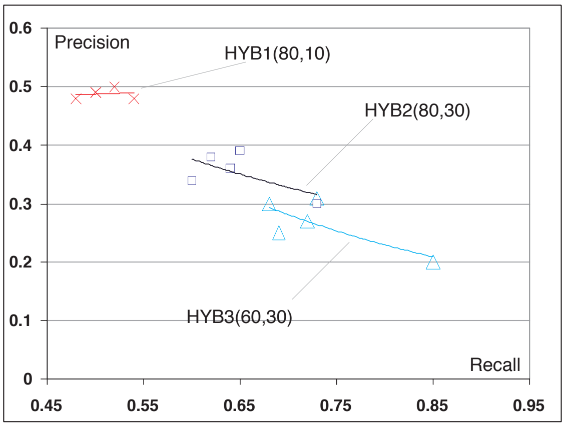

Figure 3: Performance of the HYB Heuristic

To measure the performance of the HYB(A,B) heuristic family, we re-ran our experiment on the five studied systems using values for $A = \{60, 70, 80\}$, and $B = \{10, 20, 30\}$. We produced results for nine versions of the hybrid heuristic. In Figure 3, we show only three of these results to make the figure readable, namely, we show the best (HYB1), average (HYB2), and worst (HYB3) performing heuristics of the nine heuristics. The figure has recall as the x axis and precision as the y axis. We would like to have high recall and high precision heuristics, therefore we prefer heuristics that have data points in the upper right corner of the figure. For each of the five studied system, we plot its precision and recall for each of the three heuristics as a dot in the figure. Then for each of the three heuristics (HYB1, HYB2, and HYB3), we draw the best fitted curve for the five pairs of (recall, precision) for that heuristic. At the end we get one curve per heuristic as shown in Figure 3. The results for the best performing heuristic are summarized in Table 4. The results indicate that using HYB1, we are able on average to suggest to a developer half of all entities to which a change must be propagated and that half of our suggestions are correct. We believe that the precision may be improved as these heuristics currently predict entities based on a change to a single entity and not on a set of changed entities. Nevertheless, the results are in par with typical information retrieval practical boundaries where precision usually lies in the 35%-40% range and recall is around 60% [5]

	NetBSD	FreeBSD	OpenBSD	Postgres	GCC	Average
Recall	0.52	0.50	0.50	0.48	0.54	0.51
Precision	0.50	0.49	0.49	0.48	0.48	0.49

Table 4: Performance of the HYB1 - the Best Performing Hybrid Heuristic

The HYB1 heuristic is one of many possible heuristics that can be created by combining a number of sources of data and pruning techniques proposed in Section 5. Ideally, researchers and tool developers can describe various other heuristics and use the technique presented herein and the development history of large software projects to measure the value of such tools even before they proceed to implement prototypes for such tools.

7 RELATED WORK

Arnold and Bohner give an overview of several formal models of change propagation [2, 6]. The models 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 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 [27]. The model uses graph rewriting techniques to express its formalism. Our change propagation model builds on top of the intuition and ideas proposed in these models. It simplifies the change propagation process and empirically measures the effectiveness of various sources of data that could be used to assist developers as they maintain their source code. In addition, we use an empirical approach supported by a large data set derived from open source projects to design and validate change propagation heuristics.

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 *et al.* presented a tool that uses bug reports, news articles, and mailing list posting to suggest pertinent software development artifacts [9]. Our approach focuses on the source code and its evolutionary history as a good source of data for change propagation heuristics. Other types of data sources such as bug reports and mailing list posting can be used as data sources for heuristics as well. Once these sources of data are integrated into our performance measurement framework, we can determine their effectiveness in assisting developers. Other possible source of data are design rationale graphs such as presented in [3, 28]. Yet these later approaches require a substantial amount of human intervention to build the data needed 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 [8]. 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 measure the benefit of these data sources. Our technique used to build the historical dependency graph is an extension of the work presented in [22] and is similar to work presented in [36]. Concurrently with our work, Zimmermann *et al.* have mined source control repositories using association rules to produce heuristics for change propagation. They do not use static dependencies, such as the CUD relations covered in our work, in their heuristics. 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 measures [16].

Work by Shirabad [30] which uses machine learning techniques to build co-change relations can be integrated in our heuristics and its performance measured in contrast to several proposed heuristic in this paper. Work by Ying *et al.* advocates the use of market basket analysis techniques on the historical data in the source code to assist developers as they maintain the source code [35]. Gall proposes the use of visualization techniques to show the historical logical coupling between entities in the source code [11].

8 CONCLUSIONS AND FUTURE WORK

In this paper we examined the change propagation process in software development. We highlighted the importance of having a good understanding of change propagation. We presented several change propagation heuristics. We proposed an approach to study the performance of various change propagation models and we studied the results empirically using data derived from several open source projects that have been developed for a total of 40 years.

Our results cast doubt on the effectiveness of code structures such as call graphs as good indicators for change propagation. In addition, we have shown that the historical co-change data can be used to develop heuristics to assist developers during the change propagation process with great success.

We believe that the approach and results presented herein should encourage researchers and tool developers to search for different and more sophisticated change propagation heuristics with better performance. These new heuristic and ideas can be validated easily using data derived from the development history of large software projects.

ACKNOWLEDGEMENTS

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.

REFERENCES

[1] N. Anquetil and T. Lethbridge. Extracting concepts from file names: A new file clustering criterion. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 1998)*, pages 84–93, Kyoto, Japan, Apr 1998.

[2] R. Arnold and S. Bohner. Impact analysis - toward a framework for comparison. In *IEEE International Conference Software Maintenance (ICSM 1997)*, pages 292–301, Montral, Quebec, Canada, 1993.

[3] E. L. Baniassad, G. C. Murphy, and C. Schwanninger. Design Pattern Rationale Graphs: Linking Design to Source. In *IEEE 25th International Conference on Software Engineering*, Portland, Oregon, USA, May 2003.

[4] A. Bauer and M. Pizka. The contribution of free software to software evolution. In *IEEE International Workshop on Principles of Software Evolution (IWPSE03)*, Helsinki, Finland, Sept. 2003.

[5] N. J. Belkin. The problem of matching in information retrieval. In *Theory and Application of Information Research, The Second International Research Forum in Information Science*, pages 187–197, Copenhagen, Netherlands, 1977.

[6] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Computer Soc. Press, 1996.

[7] I. T. Bowman and R. C. Holt. Reconstructing Ownership Architectures To Help Understand Software Systems. In *IEEE 7th International Workshop on Program Comprehension*, 1999.

[8] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through source code using CVS comments. In *IEEE International Conference Software Maintenance (ICSM 2001)*, pages 364–374, Florence, Italy, 2001.

[9] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2000)*, pages 408–419, Portland, Oregon, May 2003. ACM Press.

[10] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[11] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *IEEE International Conference on Software Maintenance (ICSM98)*, Bethesda, Washington D.C., Nov. 1998.

[12] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[13] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *Software Engineering*, 26(7):653–661, 2000.

[14] A. E. Hassan and R. C. Holt. Studying the chaos of code development. In *Proceedings of WCRE 2003: Working Conference on Reverse Engineering*, Victoria, British Columbia, Canada, Nov. 2003.

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[16] L. C. Briand and J. Wüst and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 475–482, Oxford, England, UK, Aug. 1999.

[17] E. H. S. Lee. Software Comprehension Across Levels of Abstraction. Master's thesis, University of Waterloo, 2000.

[18] T. C. Lethbridge and N. Anquetil. Architecture of a Source Code Exploration Tool: A Software Engineering Case Study. Tr-97-07, School of Information Technology and Engineering, University of Ottawa, 1997.

[19] M. Mitchell. GCC 3.0 State of the Source. In *4th Annual Linux Showcase and Conference*, Atlanta, Georgia, Oct. 2000.

[20] A. Mockus, R. T. Fielding, and J. D. Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 263–272, Limerick, Ireland, June 2000. ACM Press.

[21] A. Mockus and L. G. Votta. Identifying reasons for software change using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 120–130, San Jose, California, Oct. 2000.

[22] G. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.

[23] J. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley Professional, 1995.

[24] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053 – 1058, 1972.

[25] D. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE 1994)*, pages 279 – 287, Sorrento Italy, May 1994.

[26] Perforce - The Fastest Software Configuration Management System. Available online at <`http://www.perforce.com`>

[27] V. Rajlich. A model for change propagation based on graph rewriting. In *IEEE International Conference Software Maintenance (ICSM 1997)*, pages 84–91, Bari, Italy, 1997.

[28] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *IEEE 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002.

[29] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., Englewood Cliffs, NJ., USA, 1991.

[30] J. S. Shirabad. *Supporting Software Maintenance by Mining Software Update Records*. PhD thesis, University of Ottawa, 2003.

[31] A. von Mayrhauser and S. Lang. On the Role of Static Analysis during Software Maintenance. In *Proceedings of International Workshop on Program Comprehension*, pages 170–177, Pittsburgh, Pennsylvania, May 1999.

[32] Z. Weinberg. A Maintenance Programmer's View of GCC. In *First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.

[33] S. Yau, R. Nicholl, J. Tsai, and S. Liu. An integrated life-cycle model for software maintenance. *IEEE Transactions on Software Engineering*, 15(7):58–95, 1988.

[34] Y. Ye and K. Kishida. Toward an understanding of the motivation of open source software developers. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 419–429, Portland, Oregon, May 2003. ACM Press.

[35] A. T. Ying, G. C. Murphy, R. T. Ng, and M. C. Chu-Carroll. Using version information for concern inference and code-assist. In *Tool Support for Aspect-Oriented Software Development Workshop*, Seattle, Washington, Nov 2002.

[36] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *IEEE International Workshop on Principles of Software Evolution (IWPSE03)*, Helsinki, Finland, Sept. 2003.

[37] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.

[38] N. Zvegintzov. Nanotrends. *Datamation*, pages 106–116, Aug. 1983.