Predicting Source Code Changes by Mining Change History

Annie T.T. Ying, Gail C. Murphy, *Member*, *IEEE Computer Society*,
Raymond Ng, and Mark C. Chu-Carroll

Abstract—Software developers are often faced with modification tasks that involve source which is spread across a code base. Some dependencies between source code, such as those between source code written in different languages, are difficult to determine using existing static and dynamic analyses. To augment existing analyses and to help developers identify relevant source code during a modification task, we have developed an approach that applies data mining techniques to determine change patterns—sets of files that were changed together frequently in the past—from the change history of the code base. Our hypothesis is that the change patterns can be used to recommend potentially relevant source code to a developer performing a modification task. We show that this approach can reveal valuable dependencies by applying the approach to the Eclipse and Mozilla open source projects and by evaluating the predictability and interestingness of the recommendations produced for actual modification tasks on these systems.

Index Terms—Enhancement, maintainability, clustering, classification, association rules, data mining.

1 Introduction

Many modification tasks to software systems require software developers to change many different parts of a system's code base [23]. To help identify the relevant parts of the code for a given task, a developer may use a tool that statically or dynamically analyzes dependencies between parts of the source (e.g., [27], [1]). Such analyses can help a developer locate code of interest, but they cannot always identify *all* of the code relevant to the change. For example, using these analyses, it is typically difficult to identify dependencies between platform-dependent modules and between modules written in different programming languages.

To augment these existing analyses, we have been investigating an approach based on the mining of change patterns—files that were changed together frequently in the past—from a system's source code change history. Mined change patterns can be used to recommend possibly relevant files as a developer performs a modification task. Specifically, as a developer starts changing a set of files, denoted by the set f_S, our approach recommends a set of additional files for consideration, denoted by the set f_R. Our initial focus has been on the use of association rule mining to determine the change patterns. In this paper, we report on our use of an association rule mining algorithm: frequent pattern mining [2], which is based on frequency counts.

To assess the utility of our approach, we evaluate the recommendations our tool can make on two large open source projects, Eclipse[1] and Mozilla,[2] based on the *predictability* and the *interestingness* of the recommendations. Predictability quantitatively evaluates the recommendations against the files actually changed during modification tasks recorded in the development history. The interestingness of recommendations is evaluated primarily by determining whether or not a recommended file has strong structural dependencies with the initial file(s) a programmer starts changing: Files that are not structurally related are deemed more interesting as these files are not typically determinable using existing static and dynamic analyses.

This paper makes two contributions. First, we show the utility of change pattern mining. This corroborates the findings of Zimmermann et al. who independently have been investigating a similar approach using different data mining algorithms [29]. Second, we introduce a set of interestingness criteria for evaluating the usefulness of change pattern recommendations.

The rest of this paper is organized as follows: Section 2 further motivates our approach through scenarios from the development history of Eclipse and Mozilla. Section 3 describes our approach. Section 4 presents a validation of our approach on Eclipse and Mozilla. The paper ends with a discussion of outstanding issues (Section 5), related work (Section 6), and a conclusion (Section 7).

2 Motivating Scenarios

To illustrate the difficulty developers sometimes face in finding relevant source code during a modification task, we outline the changes involved in two modification tasks:[3] one from the Mozilla source code change history and one

- *A.T.T. Ying and M.C. Chu-Carroll are with the IBM T.J. Watson Research Center, 19 Skyline Dr., Hawthorne, NY 10532. E-mail: {aying, markcc}@us.ibm.com.*
- *G.C. Murphy and R. Ng are with the Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver, BC, Canada V6T 1Z4. E-mail: {murphy, rng}@cs.ubc.ca.*

Manuscript received 25 Oct. 2003; revised 9 June 2004; accepted 21 June 2004.
Recommended for acceptance by D. Rombach.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0171-1003.

[1] http://www.eclipse.org/.
[2] http://www.mozilla.org/.
[3] A *modification task* is often reffered to as a *bug*.

from Eclipse's. Mozilla is an open source Web browser written primarily in C++. Eclipse is an open source integrated development environment implemented primarily in Java.

2.1 Scenario 1: Incomplete Change

The Mozilla Web browser includes a Web content layout engine, called Gecko, that is responsible for rendering text, geometric shapes, and images. Modification task #150339 for Mozilla, entitled "huge font crashes X Windows," reports on a bug that caused the consumption of all available memory when a Web page with very large fonts was displayed. As part of a solution[4] for this modification task, a developer added code to limit the font size in the version of Mozilla that uses the `gtk` UI toolkit, but missed a similar required change in the version that uses the UI toolkit `xlib`. The comments in the modification task report include the following:

- 2002-06-12 14:14: "This patch [containing the file `gtk/nsFontMetricsGTK.cpp`] limits the size of fonts to twice the display height."
- 2002-06-12 14:37: "The patch misses the Xlib gfx version." [A patch was later submitted with the correct changes in the X-windows font handling code in the file `xlib/nsFontMetricsXlib.cpp`.]

The source code in `gtk/nsFontMetricsGTK.cpp` does not reference the code in `xlib/nsFontMetricsXlib.cpp` because the code in the `gtk` version and the code in `xlib` version are used in different configurations of Mozilla. However, an analysis of the CVS[5] change history for Mozilla indicates that these two files were changed 41 times together in the development of Mozilla. When we applied our approach to Mozilla, we extracted a change pattern with three files: `gtk/nsFontMetricsGTK.cpp`, `xlib/nsFontMetricsXlib.cpp`, and `gtk/nsRenderingContextGTK.cpp`. Changing the `gtk/nsFontMetricsGTK.cpp` triggers a recommendation for the other two files, one of which had been missed in the first patch.

2.2 Scenario 2: Focusing on Pertinent Code

Eclipse is a platform for building integrated development environments. "Quick fix" is a feature of the Java development support in Eclipse that is intended to help a developer easily fix compilation errors. Modification task #23587 describes a missing quick fix that should be triggered when a method accesses a nonexistent field in an interface.

The solution for the modification task involved the class `ASTResolving`[6] in the text correction package. This class was structurally related to many other classes: It was referenced by 45 classes and referenced 93 classes; many of these classes were involved in a complicated inheritance hierarchy. Such a large number of structural dependencies can complicate the determination of which classes should be changed with `ASTResolving`.

Using our approach, we found that `ASTResolving` had been changed more than 10 times in conjunction with three classes: 11 times with `NewMethodCompletionProposal`, 10 times with `NewVariableCompletionProposal`, and 12 times with `UnresolvedElementsSubProcessor`. These three classes were part of the solution for the modification task, as indicated by comments in the revision system. However, the other four classes that were part of the solution were not recommended because they had not been modified together a sufficient number of times.[7]

3 Approach

Our approach consists of three stages. In the first stage, which we describe in Section 3.1, we extract the data from a software configuration management (SCM) system and preprocess the data to be suitable as input to a data mining algorithm. In the second stage, which we describe in Section 3.2, we apply an association rule mining algorithm to form change patterns. In the final stage, which we describe in Section 3.3, we recommend relevant source files as part of a modification task by querying against mined change patterns. Having extracted the change patterns in the first two stages, we do not need to regenerate the change patterns each time we query for a recommendation.

3.1 Stage 1: Data Preprocessing

Our approach relies on being able to extract information from a SCM system that records the history of changes to the source code base. In addition to the actual changes made to the source code, these systems typically record metadata about the change, such as the time-stamp, author, and comments on the change. Most of these systems manage software artifacts using a file as the unit. Some support finer-grained artifacts such as classes and methods in an object-oriented programming language (e.g., Coop/Orm [16]). Our initial focus has been on change histories stored in a CVS repository, which records and manages changes to source files. Before attempting to find patterns in the change history, we need to ensure that the data is divided into a collection of atomic change sets, and we need to filter atomic changes that do not correspond to meaningful tasks.

3.1.1 *Identifying Atomic Change Sets*

The first preprocessing step involves determining which software artifacts—in our case, files—were checked in together. This step is not needed when the history is stored in an SCM system that provides atomic change sets, such as ClearCase [15]. However, other systems such as CVS, which is used for the systems we targeted in our validation, do not track this information; as a result, we must process the change history to attempt to recreate these sets. We form the sets using the following heuristic: An atomic change set consists of file changes that were checked in by the same author with the same check-in comment close in time. We follow Mockus et al. in defining proximity in time of check-ins by the check-in time of adjacent files that differ by less

[4] We refer to the files that contribute to an implementation of a modification task as a *solution*.

[5] http://www.cvshome.org/.

[6] In Java, the name of a file that contains a publicly accessible class `Foo` is `Foo.java`.

[7] "Sufficient number of times" corresponds to the *support* threshold, which is further explained in Section 3.2. The choice of the support threshold is discussed in Section 4.2.

than three minutes [20]. Other studies (e.g., [11]) describe issues about identifying atomic change sets in detail.

3.1.2 Filtering

The second preprocessing step involves eliminating transactions consisting of more than 100 files because these long transactions do not usually correspond to meaningful atomic changes, such as feature requests and bug fixes. An example is when an integrated development environment, such as Eclipse, is used to remove unnecessary import declarations in all of the Java files in a project. This kind of organize-import operation tends to change many files but does not correspond to a meaningful modification task.

3.2 Stage 2: Association Rule Mining

Association rule mining algorithms extract sets of items that happen *frequently enough* among the transactions in a database. In our context, such sets, called change patterns, refer to source files that tend to change together. In our study, we investigate frequent pattern mining [2] for extracting change patterns from the source code change history of a system.

The idea of frequent pattern mining is to find recurring sets of items—or source files in our context of finding change patterns—among transactions in a database D [2]. The strength of the pattern $\{s_1, \ldots, s_n\}$, where each s_i is the name of a source file, is measured by *support*, which is the number of transactions in D containing $s_1, \ldots,$ and s_n. A frequent pattern describes a set of items that has support greater than a predetermined threshold called *min_support*.

The problem of finding all frequent patterns efficiently is not trivial because the performance can be exponential with respect to the number of items in D when the support threshold *min_support* is low. Efficient algorithms to this problem have been proposed (e.g., [3], [22], [13]). The algorithm we chose to find frequent patterns uses a compact data structure called FP-tree to encode a database [13]. The idea of the algorithm is to find frequent patterns by a depth-first approach of recursively mining *a pattern* of increasing cardinality from the data structure FP-tree that encodes D, as opposed to a breadth-first approach of finding *all patterns* of the same cardinality before finding patterns of a larger cardinality. In an FP-tree, each node represents a frequent item in D, except for the root node that represents an empty item. Each path from the root to a node in the FP-tree corresponds to a collection of transactions in D, each of which contains all of the items on the path. Items in a path are in descending order of support (and in lexicographical order of item name if two items have the same support). The mining process involves decomposing an FP-tree associated with database D into smaller FP-trees, each of which corresponds to a partition of D. This divide-and-conquer approach allows the determination of frequent patterns to focus on the decomposed database rather than the whole database.

3.3 Stage 3: Query

Applying a data mining algorithm to the preprocessed data results in a collection of change patterns. Each change pattern consists of sets of the names of source files that have been changed together frequently in the past. To provide a recommendation of files relevant to a particular modification task at hand, the developer needs to provide the name of at least one file that is likely involved in the task. The files to recommend are determined by querying the patterns to find those that include the identified starting file(s); we use the notation $recomm(f_S) = f_R$ to denote that the set of files f_S results in the recommendation of the set of files f_R. When the set of starting files has cardinality of one, we use the notation $recomm(f_s) = f_R$. The recommendation set f_R is the union of matched patterns—patterns with at least one file from f_S—not including f_S.

We illustrate the query process with an example based on Mozilla's modification task #150339 described in Section 2.1, using the frequent pattern mining algorithm with the support threshold *min_support* equal to 20 and transactions from the Mozilla development history from 27 March 1998 to 8 May 2002. Suppose a developer starts changing the file `gtk/nsFontMetricsGTK.cpp`. Our approach would retrieve the two change patterns that contain the file `gtk/nsFontMetricsGTK.cpp`:{`gtk/nsFontMetricsGTK.cpp`, `gtk/nsFontMetricsGTK.h`, `gtk/nsRenderingContextGTK.cpp`} and {`gtk/nsFontMetricsGTK.cpp`, `xlib/nsFontMetricsXlib.cpp`}. The recommendation is generated (as described in Section 3.3) by taking the union of the two change patterns not including the starting file `gtk/nsFontMetricsGTK.cpp`: *recomm* (`gtk/nsFontMetricsGTK.cpp`) = {`xilb/nsFontMetricsXlib.cpp`, `gtk/nsFontMetricsGTK.h`, `gtk/nsRenderingContextGTK.cpp`}.

4 Validation

To assess the utility of change patterns in a modification task, we need to apply the approach to systems with many source code changes and for which information about modification tasks is available. To satisfy this requirement, we applied the approach to the development histories of two large open-source systems—Eclipse and Mozilla. The two projects are written in different programming languages: Eclipse is mainly written in Java and Mozilla is mainly written in C++. The two projects also differ in development history: Mozilla has more than six years of development history, whereas Eclipse only has three years of development history. Each of these projects also involves a large number of developers, reducing the likelihood that peculiar programming practice of a particular programmer dramatically affects the results. However, both Eclipse and Mozilla are open source projects and use CVS and Bugzilla bug tracking system[8] to track modifications. In this section, we begin by describing our validation strategy (Section 4.1) and the parameter settings (Section 4.2). We then present an analysis of the results (Sections 4.3 to 4.5).

4.1 Validation Strategy

The validation process involved determining if source code recommended from a change pattern was relevant for a given modification task. This validation process required dividing the development history information for a system

[8] http://www.bugzilla.org/.

into training and test data. The training data was used to generate change patterns that were then used to recommend source for the test data.

To determine if our approach can provide good recommendations, we investigated the recommendations in the context of completed modification tasks made to each system. These modification tasks are recorded in each project's Bugzilla, which also keeps track of enhancement tasks. We refer to both bugs and enhancements as *modification tasks*, and we refer to the files that contribute to an implementation of a modification task as a *solution*.

Since Bugzilla does not record which source files are involved in a solution of a modification task, we use heuristics based on development practices to determine this information. One common practice is that developers record the identifier of the modification task upon which they are working as part of the CVS check-in comments for a solution of the modification task. Another common practice is that developers commit the files corresponding to a solution of the modification task into CVS close to the time at which they change the status of the modification task report to "fixed" [9]. For the validation, we chose tasks for which a solution can be associated, for which the files involved in the solution were between the dates designated as the test data, and for which at least one file involved in the solution was covered by a change pattern extracted from the training data.

To recommend possibly relevant files using our approach, at least one file that is likely involved in the solution must be specified by the developer. In our validation, we chose to specify exactly one file f_s to generate a set of recommended files f_R; we chose this approach because it represents the minimum amount of knowledge a developer would need to generate a recommendation. We evaluate the usefulness of the recommended files f_R in terms of two criteria, predictability and interestingness, described in the rest of Section 4.1.

4.1.1 Predictability

The predictability of the recommendations is measured in terms of *precision* and *recall*. The precision of a set of recommendations refers to how well our approach provides concise recommendations. The recall of a set of recommendations refers to how well our approach recommends the files in the solution. More precisely, the precision $precision(m, f_S)$ of a recommendation $recomm(f_S)$ is the fraction of recommendations that did contribute to the files in the solution $f_{sol}(m)$ of the modification task m. The recall $recall(m, f_S)$ of a recommendation $recomm(f_S)$ is the fraction of files in the solution $f_{sol}(m)$ of the modification task m that are recommended.

$$\begin{aligned} precision(m, f_S) &= \frac{|correct(m, f_S)|}{|recomm(f_S)|} \\ recall(m, f_S) &= \frac{|correct(m, f_S)|}{|f_{sol} - f_S|}, \text{ where} \\ correct(m, f_S) &= |recomm(f_S) \cap (f_{sol} - f_S)|. \end{aligned} \tag{1}$$

We use precision and recall to evaluate our approach quantitatively in two ways: the *average* precision and recall (denoted by $precision_{avg}$ and $recall_{avg}$, respectively), which measure the accuracy of recommendations given by each query (with starting file f_S), and the *limit* of precision and recall (denoted by $precision_{lim}$ and $recall_{lim}$, respectively), which evaluate how many recommendations are never correct and how many files in the solution are never recommended by any query.

In the computation of the average measures, $precision_{avg}$ and $recall_{avg}$, we included only the precision and recall for recommendations from modification tasks M in the test data where each task's solution contained at least one file from a change pattern. We used each file f_s in the solution of a modification task $m \in M$ to generate a set of recommended files f_R and calculated f_R's precision and recall in the equations in (1). The average precision is the mean of such precision values and analogously for the average recall.

Equations (2) show the limit precision and recall. The $precision_{lim}(M)$ value evaluates, for all modification tasks $m \in M$, the fraction of recommended files that *could possibly be correct* by any query. More specifically, this measure captures the average fraction of files that can be recommended from the union of all change patterns in which at least one file in the change pattern is in the solution. We denote the files in the union of change patterns as restricted above by $P(f_{sol}(m))$. The $recall_{lim}(M)$ value evaluates, for all modification tasks $m \in M$, the fraction of files in the solution that could *possibly be recommended* by any query. More specifically, this measure captures the average fraction of files in the solution that are in $P(f_{sol}(m))$. As in the computation of $precision_{avg}$ and $recall_{avg}$, M refers to modification tasks in the test data where each task's solution contained at least one file from a change pattern. What we want the limit measures to convey is the average percentage of recommendations by any query that are never correct, which is $1 - precision_{lim}(M)$, and the average percentage of files in the solution that are never recommended by any query, which is $1 - recall_{lim}(M)$.

$$\begin{aligned} precision_{lim}(M) &= mean_{m \in M} \frac{|P(f_{sol}(m)) \cap f_{sol}(m)|}{|P(f_{sol}(m))|} \\ recall_{lim}(M) &= mean_{m \in M} \frac{|P(f_{sol}(m)) \cap f_{sol}(m)|}{|f_{sol}(m)|}, \text{ where} \\ P(f_{sol}(m)) &= \bigcup_{\text{all change patterns } p \text{ such that } p \cap f_{sol}(m) \neq \emptyset} p. \end{aligned} \tag{2}$$

For example, suppose the solution of a modification task m contains files $\{a, b, c, d, e\}$ and the change patterns computed from past history are $\{a, b, f\}$, $\{i, j, h\}$, and $\{b, c\}$. The files that *can possibly be recommended*—$P(f_{sol}(m))$ —is $\{a, b, c, f\}$. Regardless of what starting files are used, d and e can never be recommended although they are part of the solution; as a result, 40 percent of the files in the solution cannot possibly be recommended. Of files that can possibly be recommended, the files that are also in the solution are $\{a, b, c\}$. Regardless of what starting files are used, f can never be correct although it can be recommended; as a result, 25 percent of the files cannot possibly be correct.

TABLE 1
Structural Relationships Used in Our Criteria

Structural relationship $rel(s \mapsto r)$	**Granularity**	**Language**	**Description**
$reads(m \mapsto f)$ $isRead(f \mapsto m)$	fine	C++, Java	Method m accesses the value of field f.
$writes(m \mapsto f)$ $isRead(f \mapsto m)$	fine	C++, Java	Method m writes a value to field f.
$calls(m_1 \mapsto m_2)$ $isCalled(m_2 \mapsto m_1)$	fine	C++, Java	Method m_1 contains a method invocation that can bind to m_2.
$creates(m \mapsto c)$ $isCreated(c \mapsto m)$	fine	C++, Java	Method m creates an object of class c.
$checks(m \mapsto c)$ $isChecked(c \mapsto m)$	fine	C++, Java	Method m checks or casts an object to class c.
$headerOf(h \mapsto c)$ $conformsToHeader(c \mapsto h)$	coarse	C++	File h contains the declaration of class c.
$implements(c \mapsto i)$ $conformsToInterface(i \mapsto c)$	coarse	Java	Class c is an implementation of interface i.
$extends(c_1 \mapsto c_2)$ $isExtended(c_2 \mapsto c_1)$	coarse	C++, Java	Class c_1 is a subclass of a class c_2.
$declaresMethodReturnParamTypeAs(c_1 \mapsto c_2)$ $isDeclaredAsMethodReturnParamType(c_2 \mapsto c_1)$	coarse	C++, Java	Class c_1 declares a method such that the object returned or passed as a parameter can be of class c_1.
$declaresFieldAs(c_1 \mapsto c_2)$ $isDeclaredAsField(c_2 \mapsto c_1)$	coarse	C++, Java	Class c_1 declares a field of class c_2, or ancestor of c_2 in the inheritance hierarchy.
$samePackage(c_1 \mapsto c_2)$	coarse	Java	Class c_1 and class c_2 are in the same package.
$sameDirectory(c_1 \mapsto c_2)$	coarse	C++	Class c_1 and class c_2 are stored in the same directory.

4.1.2 Interestingness

Even if a recommendation is applicable, we have to consider whether or not the recommendation is *interesting*. For example, a recommendation that a developer changing the C source file `foo.h` should consider changing the file `foo.c` would be too obvious to be useful to a developer. To evaluate recommendations in this dimension, we assign a qualitative interestingness value to each recommendation of one of three levels—*surprising*, *neutral*, or *obvious*—based on structural and nonstructural information that a developer might easily extract from the source.

Structural information refers to relationships between program elements that are stated in the source using programming language constructs. Table 1 lists the structural relationships we considered in our analysis for the Java and C++ languages. The "Structural relationship" column of Table 1 lists the structural relationships $rel(s \mapsto r)$ between two source code fragments s and r, where s refers to a source code fragment that a developer starts to look at and r refers to a source code fragment that is structurally related to s. The "Granularity" column indicates the unit of the source code fragments involved in the relationship: "coarse" indicates that the fragments are at a class or interface granularity; "fine" indicates that the granularity of the fragments is finer than class-granularity, for instance, the relationship is between methods. The column "Language" indicates to which language—C++ or Java—the relationship is applicable. The last column provides a description for each relationship.

Nonstructural information refers to relationships between two entities in the source code that are not supported by the programming language. Nonstructural information includes information in comments, naming conventions, string literals, data sharing (e.g., code exchanging data through a file), and reflection.

The interestingness value of a recommendation, f_r where $recomm(f_S) = f_R$ and $f_r \in f_R$, is based on how likely it is that a developer pursuing and analyzing f_s would consider the file f_r as part of the solution of a modification task. We assume that such a developer has access to simple search tools (e.g., `grep`) and basic static analysis tools that enable a user to search for references, declarations, and implementors of direct forward and backward references for a given point in the source, as is common in an integrated development environment.

We categorize a recommendation f_r as *obvious* when

- a method *that was changed* in f_s has a direct fine-grained reference—$reads$, $isRead$, $writes$, $isWritten$, $calls$, $isCalled$, $creates$, $isCreated$, $checks$, $isChecked$, as described in Table 1—to a method, field, or class in f_r, *or*
- a class *that was changed* in f_s has a strong coarse-grained relationship—the coarse-grained relationships described in Table 1—to a class in f_r.

We categorize a recommendation as *surprising* when

- f_s has no direct structural relationships with f_r, *or*

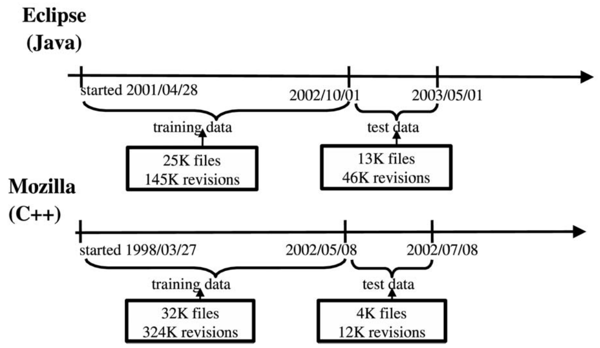

Fig. 1. Statistics on the Eclipse and Mozilla CVS repositories.

- a fragment in f_s contains nonstructural information about f_r.

A recommendation is *neutral* when

- a method in f_s, *other than the one that was changed* in f_s, has a direct fine-grained reference to a method, field, or class in f_r, *or*
- a class *that was changed* in f_s has a weak coarse-grained relationship—it indirectly inherits from, or is in the same package or directory that has more than 20 files—with a class that was changed in f_r.

If f_s and f_r have more than one relationship, the interestingness value of the recommendation is determined by the interestingness value of the most obvious relationship.

4.2 Validation Settings

Fig. 1 presents some metrics about the Eclipse and Mozilla developments and outlines the portions of the development history we considered in our analysis. In both systems, the training data comprised changes to over 20,000 files and over 100,000 versions to those source files.

Table 2 shows the number of transactions involving different cardinalities of files as well as the total number of transactions. For the period of time that corresponds to the training data, both Eclipse and Mozilla have a similar number of transactions. In both systems, transactions of two items have the highest counts and the number of transactions decreases as the cardinality of the transaction increases.

Table 3 describes the parameters we used in the data mining algorithm. The first column lists the support threshold. The second column indicates whether the data mining algorithm was applied to Eclipse and Mozilla. The third column presents the number of files that were generated from the patterns extracted using the algorithm with the specified parameter applied to either Eclipse or Mozilla.

TABLE 2
Transaction Statistics of Eclipse and Mozilla

Cardinality of transactions	Mozilla	Eclipse
2	13,734	13,608
3	5,606	6,053
4	3,605	3,822
5	2,046	2,326
> 5	7,657	9,228
total	32,648	35,037

TABLE 3
Statistics from Patterns Formed in the Training Data of Eclipse and Mozilla

min_support	Target	No. of files
20	Eclipse	134
15	Eclipse	319
10	Eclipse	877
5	Eclipse	3462
30	Mozilla	598
25	Mozilla	807
20	Mozilla	1139
15	Mozilla	1715

For the frequent pattern algorithm, the value of the support threshold *min_support* was varied so that a reasonably large number of files (over 200) were involved in patterns and the support was not too low (not below 5). Comparing the patterns generated for Eclipse and Mozilla using the frequent pattern algorithm with the same parameter setting (*min_support* equals 20 and 15), Mozilla has more than five times more files involved in the change patterns than Eclipse. We were careful to choose thresholds that were neither too restrictive nor too relaxed. An overly restrictive threshold results in too few patterns. This situation affects the recall value as the recommendations do not cover the changes needed for a modification task. An overly relaxed threshold affects the precision since too many patterns result in a number of recommendations, only a few of which are correct.

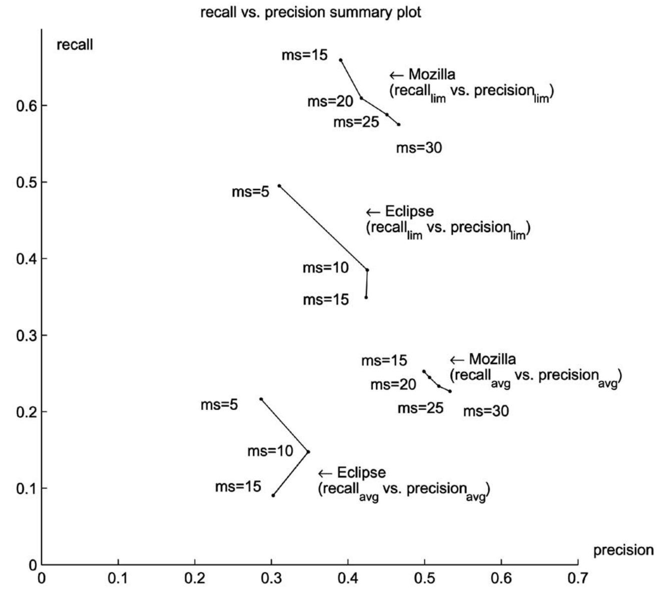

Fig. 2. Recall versus precision plot showing the frequent pattern algorithm applied to the two target systems, Eclipse and Mozilla.

TABLE 4
Mozilla Recommendation Categorization by Interestingness Value

Interestingness	Description	Modification task ids (and no. of recommendations)
surprising	cross-language	150099 (6)
surprising	duplicate code bases	92106 (2), 143094 (2), 145560 (2), 145815 (2), 150339 (2)
neutral	distant instance creation/being created and call dependence	123068 (2)
neutral	distant inheritance	74091 (2)
obvious	header-implementation	92106 (2), 99627 (2), 104603 (2), 135267 (8), 144884 (2), 150735 (2), 74091 (2), 123068 (2)
obvious	interface-implementation	135267 (2)
obvious	direct inheritance	74091 (12)
obvious	direct call dependence	99627 (2)
obvious	same package with less than 20 files	135267 (6)

4.3 Predictability Results

Fig. 2 shows the precision and recall values that result from applying the frequent pattern algorithm to each system. The lines connecting the data points on the recall versus precision plot show the trade-off between precision and recall as the parameter values are altered. The recall and precision axes are adjusted to show only values up to 0.7. The label beside each data point indicates the *min_support* threshold used in the frequent pattern mining algorithm; for example, "ms = 5" stands for *min_support* = 5.

For the two groups of connected points labeled "$recall_{avg}$ versus $precision_{avg}$," each data point represents average recall and precision. The recall and precision values for the generated change patterns for Mozilla are encouraging; precision is around 0.5 with recall between 0.2 and 0.3 (meaning that, on average, around 50 percent of the recommended files are in a solution and 20 percent to 30 percent of the files in a solution are recommended). The line plot shows a steady trade-off between recall and precision when *min_support* changes. However, the recall and precision values in the Eclipse case are less satisfactory; precision is only around 0.3, recall is around 0.1 to 0.2, and the line plot shows some strange behavior, namely, when *min_support* threshold equals 15, there is a sudden decrease in both precision and recall. The small number of patterns and small number of files covered by patterns may cause this behavior because few recommendations can be made.

For the two groups of connected points labeled "$recall_{lim}$ versus $precision_{lim}$," each data point represents the limit of recall and precision, which evaluate the recommendations that can possibly be recommended ($recall_{lim}$) and recommendations that can possibly be correct ($precision_{lim}$). For Mozilla, the precision is between 0.4 and 0.5, which means that, on average, around 50-60 percent of the recommended files are never correct; the recall is around 0.6, which means that, on average, around 40 percent of the files in a solution are never recommended. For eclipse, the precision is around 0.4 meaning that, on average, around 60 percent of the recommended files are never correct, and the recall is around 0.4 meaning that, on average, around 60 percent of the files in a solution are never recommended.

Although the recall and precision may seem low, the value of the approach must be evaluated based on its ability to provide helpful recommendations; the scenarios presented in Section 2 provide some examples. We assess the interestingness of recommendations further below. We argue that recommendations with precision presented in Fig. 2 are useful as long as the gain to the developer when the recommendation is helpful is greater than the cost to the developer of determining which recommendations are false positives. Our approach can augment existing approaches that provide recommendations to relevant source code in a modification task.

4.4 Interestingness Results

To assess the interestingness of the recommendations, we randomly chose, for each project, 20 modification tasks from the period of time covered by the test data for the project. For each modification task, we determined, for each file (f_s) that was part of the check-in for the modification and that was contained in at least one change pattern, the files (f_R) recommended using f_S. For each file recommended, we assessed its interestingness level according to the criteria described in Section 4.1.

Tables 4 and 5 summarize the recommendations resulting from change patterns generated with the frequent pattern algorithm where *min_support* threshold equals 20 in Mozilla and *min_support* equals 10 in Eclipse. We chose to investigate the results from the frequent pattern algorithm with these threshold settings because they resulted in the best trade-off in terms of precision and recall. We grouped the recommendations with the same interestingness level into similar cases and gave each a descriptive name. Recommendations with the same descriptive name and with the same interestingness level can come from different modification tasks; the identifiers of the relevant tasks are shown in the last column. The number in parentheses beside each identifier is the sum of the cardinality of recommended sets, where each recommended set is the result of querying with a different f_s associated with the modification task.

TABLE 5
Eclipse Recommendation Categorization by Interestingness Value

Interestingness	Description	Modification task ids (and no. of recommendations)
surprising	cross-platform/XML	24635 (230)
neutral	distant call dependence	21330 (2), 24567 (2)
neutral	distant inheritance	25041 (12)
obvious	containment	13907 (2), 23096 (2), 24668 (2)
obvious	framework	21330 (2)
obvious	same package with less than 20 files	21330 (2)
obvious	instance creation/being created	23587 (4)
obvious	method call dependence	21330 (2), 23587 (4), 24657 (2), 25041 (2)
obvious	direct inheritance	25041 (8)
obvious	interface-implementation	24730 (2)

4.4.1 Mozilla Interestingness Results

Table 4 presents a categorization of recommendations for Mozilla. Our approach generated recommendations for 15 of the 20 selected modification tasks. The recommendations for the 15 tasks include two kinds of *surprising* recommendations, two kinds of *neutral* recommendations, and five kinds of *obvious* recommendations. We focus the discussion on the cases categorized as *surprising*. Recommendations classified as *neutral* and even *obvious* may still be of value to a developer, as shown by the scenario described in Section 2.2, but we do not provide a detailed analysis of these cases.

The "cross-language" case in the *surprising* category demonstrates how our approach can reveal interesting dependencies on files written in different languages and on noncode artifacts that may not be found easily by a developer.

- For Mozilla, a developer specifies the layout of widgets in XUL (XML-based User interface Language), which eases the specification of the UI and provides a common interface for the UI on different platforms. XUL does not solely define the UI; a developer must still provide supporting code in a variety of formats, including XML schema files and Javascript files. This situation occurred in the solution of modification task #150099, which concerned hiding the tab bar in the Web browser by default. The solution involved adding a new menu item for displaying the user's preference of showing or hiding the tab bar in an XUL file, declaring the menu item in a XML schema file, and initializing the default settings of the menu item as call-back code in a Javascript file. Our approach generated six *surprising* recommendations involving Javascript-XUL,[9] XML schema-XML, and XML schema-Javascript.

The "duplicate code bases" case from the *surprising* category demonstrates how our approach can reveal potentially subtle dependencies between evolving copies of a code base.

- As part of the solution of modification task #92106, two scripts that built different applications in the XML content model module `TransforMiix` needed to be updated. One script was for building an application for transforming XML documents to different data formats, and the other script was for building a benchmarking application for the former application. Much of the two build scripts shared the same text and changes to one script usually required similar changes to the other script. In fact, this code duplication problem was later addressed and eliminated (modification task #157142). When either of these build scripts was considered to be modified, our approach was able to provide a recommendation that the developer should consider the other script, resulting in two *surprising* recommendations.
- For the Mac OS X operating system, Mozilla includes two code bases, `Chimera` and `Camino`. The second is a renamed copy of the first code base, but changes were still occurring in parallel to each code base. For modification tasks #143094 and #145815, our approach was able to recommend that a change to a file in one code base should result in a change to the corresponding file in the other code base.
- Modification task #145560 concerned fixing a typographical error in a variable named `USE_NSPR_THREADS` in the configuration template file `configure.in`. The shell script `configure`—the other file involved in the solution of the modification task—was generated using `autoconfig` for configuring different platforms using appropriate parameter values specified in the template file `configure.in`. Our approach was able to recommend that a change in file `configure.in` required a change in `configure` because `configure` must be regenerated from `configure.in`.
- Modification task #150339, which was described in Section 2.1, presented a situation where a developer missed changing code that was duplicated in two

[9] The terminology "Javascript-XUL" means that, given a Javascript file, an XUL file was recommended and, given an XUL file, a Javascript file was recommended.

versions of the system, one that uses the `gtk` UI toolkit and another one that uses the `xlib` UI toolkit. Our approach generated recommendations suggesting parallel changes in the code.

4.4.2 Eclipse Interestingness Results

Table 5 shows the categorization of recommendations for Eclipse. Fewer of the selected modification tasks for Eclipse resulted in recommendations than for Mozilla. We could not provide recommendations for 11 of the modification tasks because the changed files were not covered by a change pattern. Of the nine tasks that resulted in recommendations, eight of them had solutions involving files in which the classes were structurally dependent. We group these into seven cases involving *obvious* recommendations, two cases involving *neutral* recommendations, and one case involving *surprising* recommendations that we focus on.

The "cross-platform/XML" case involved recommendations for noncode artifacts, which, as we argued above, may not be readily apparent to a developer and may need to be determined by appropriate searches.

- Modification task #24635 involved a missing URL attribute in an XML file that describes which components belong to each platform version. The change involved 29 files that spanned different plug-ins for different platform versions. Our approach generated 230 *surprising* recommendations. This large number of recommendations was made because the solution for the problem contains 29 files of which 18 match at least one change pattern. Each of these 18 files, when used as the starting file, typically generated recommendations of over 10 files, summing to 230 recommendations.

4.4.3 Conclusion of the Interestingness Results

The majority of the recommendations made by our approach were classified as neutral or obvious and involved files that were structurally related. Even though existing static or dynamic analysis approaches could detect such relationships, in some cases, our approach may provide more specific recommendations; for instance, the case described in Section 2.2. Many of the cases we listed as surprising involved files implemented in different languages or used in different platforms. These dependencies would be difficult to find with existing static and dynamic analysis approaches.

4.5 Performance

The queries that we performed in this validation took less than 5 seconds on a Sun Ultra 60 system with 1280 MB RAM with 2 x 360 MHz UltraSPARC-II processors. The computation time of populating the database is more time-consuming, but is performed less often. Populating the database with file change data took about two hours for each of Mozilla and Eclipse. Transaction computation took 6 minutes on Eclipse and 11 minutes on Mozilla. The computation time of mining change patterns increases as the support threshold decreases: Table 6 shows that the performance ranges from 1 minute to 55 minutes for Eclipse and from 1 minute to 3 minutes for Mozilla.

TABLE 6
Performance on Pattern Computation on Eclipse

Target	Support threshold	Time (s)
Eclipse	20	44
Eclipse	15	86
Eclipse	10	439
Eclipse	5	3302
Mozilla	30	52
Mozilla	25	65
Mozilla	20	102
Mozilla	15	209

5 Discussion

In this section, we begin by discussing the criteria we used to validate the results of our experiment (Sections 5.1 and 5.2). We then comment on some issues with the approach (Sections 5.3 to 5.5), including the granularity of the source code used in the change pattern mining process, and an alternative algorithm we have investigated (Section 5.6).

5.1 Interestingness Validation Criteria

Evaluating the usefulness of a particular file recommendation is difficult because the relevance of a particular file to a change task is subjective. For instance, even if a file does not need to be changed as part of the modification task, some developers may assess the file as relevant if it aids in their understanding of some functioning of the code relevant to the modification task. Since we cannot determine absolutely if a file is interesting, we chose to base the interestingness criteria on a set of heuristics, several of which are based on structural relationships between parts of the code. Since existing tools, such as compilers and integrated development environments, provide support to find some structurally related code, we categorized recommendations which could be found using such tools as obvious. On the other hand, recommendations involving nonstructural relationships may be more difficult for a developer to find; these recommendations are thus classified as neutral or surprising.

The interestingness of recommendations might be determined more accurately through an empirical study involving human subjects. As we have argued, we believe our interestingness criteria are a good approximation of what would interest a software developer. An assessment of recommendations by human subjects and a correlation of the interest to our criteria remains open for future work.

5.2 Predictability Validation Criteria

Our evaluation of recall and precision is conservative in the sense that we measure these values with respect to whether a recommended file f_r was part of the set of files f_{sol} that was checked in as part of the solution. We cannot determine if a recommended file that did not have a version stored as part of the change might have been helpful to a developer in understanding the source to make the desired changes. Additionally, in the calculation of $recall_{avg}$ and $precision_{avg}$, the recommendations were made using one starting file, which represents the minimum amount of knowledge a developer would need to apply our approach.

In the calculation of predictions in the test data, we have not included the modification tasks whose solutions do not contain any files in a change pattern. The rationale behind this is that our approach is based on the assumption that the target project has a rich history of changes. Modification tasks whose solutions do not contain any files from any change patterns indicate that the part of the system the modifications tasks are concerned with does not have a rich history of changes. For example, the test data of the Mozilla project contains 827 modification tasks between the designated dates that we could associate with a CVS check-in, of which the solutions of 337 to 487 (41 to 59 percent) tasks are covered by at least one file from a change pattern: The range of task solution covered depends on the value of *min_support*.

5.3 Weaknesses of Our Approach

Several possibilities may affect the predictability of the recommendations in our evaluation. One possibility is that there were too few transactions for each system: Association rule mining usually assumes a large number of transactions. For example, in the first frequent pattern algorithm literature, the number of transactions used in the experiment is more than 20 times greater than the number of items in a validation of the frequent pattern algorithm [13]. However, in Eclipse and Mozilla, the number of transactions and the number of items—files in our context—are approximately the same because items do not occur in as many transactions as in other applications. This may be one reason that the recall and precision are not high. The use of CVS by these projects further impacts our approach since historical information is not maintained by CVS when a file or directory is renamed. Moreover, significant rewrites and refactoring of the code base can affect our approach. Such changes affect the patterns we compute because we do not track the similarity of code across such changes.

5.4 Granularity

Currently, the change associations we find are among files. Applying our approach to methods—where change patterns describe *methods* instead of *files* that change together repeatedly—may provide better results because a smaller unit of source code may suggest a similar intention behind the separated code. However, refining the granularity weakens the associations (each pattern would have lower support), which may not be well-handled by our current approach.

5.5 Implementation Trade-Offs

In this paper, we have presented an initial evaluation of an implementation of an association rule algorithm. In our implementation, we made several choices. First, we chose to use an offline approach. More time is needed up-front to compute the patterns, but the query time is faster once the patterns are computed if the same query is performed multiple times. Second, the implementation of the approach we described in this paper does not handle incremental changes that are checked into the SCM system. An incremental implementation that updates the list of transactions of atomic changes as files are checked into the SCM system and that incrementally mines patterns [8] is possible.

5.6 Alternative Algorithms for Finding Change Patterns

In this paper, we have presented one association rule mining algorithm for finding change patterns, namely, frequent pattern mining. We also considered another algorithm, called correlated set mining [7], which considers statistical correlation that may be implied by cooccurrences of items, unlike frequent pattern mining which considers only cooccurrences of items. The idea of correlated set mining is to find all sets of items with sufficient correlation as measured by the chi-squared test. In our context, the chi-squared test compares the frequencies of files changed together (called *observed frequencies*) with the frequencies we would expect if there were no relationship between whether or not the set of files are changed together (called *expected frequencies*). A statistically significant difference between the two frequencies indicates, with a degree of confidence, that the fact that the files changed together are correlated.

When we applied the correlated set algorithm (computing correlated sets of cardinality two) to the change data of the target systems, few change patterns were generated, indicating that the algorithm is not applicable on the data. We observed that the expected frequencies are much smaller than the observed frequencies because the total number of possible files in the system is much larger than the number of times any two files changed together. Data with such a distribution does not work well with the chi-squared test used in correlated set mining, which requires expected frequencies to not be too small. One way to improve this situation is to partition the transactions in a way that it would not dramatically weaken the correlations and to apply the correlated set mining algorithm to each partition so that the number of files in the system is closer to the number of times any two files changed together. Dividing the system into architectural modules may be one such meaningful way to partition the system.

6 Related Work

We first focus on the comparison of our approach to others that rely upon development history (Section 6.1) to help determine relevant code. We then focus on approaches that rely on impact analysis to determine the scope of a modification task (Section 6.2). Finally, we describe other work that uses data mining approaches to software engineering problems (Section 6.3).

6.1 Using Development Histories to Identify Relevant Code

Zimmermann et al., independently from us, have developed an approach that also uses association rule mining on CVS data to recommend source code that is potentially relevant to a given fragment of source code [29]. The rules determined by their approach can describe change associations between files or methods. Their approach differs from ours in that they use a particular form of association rule mining in which rules determined must satisfy some support and confidence. Frequent pattern mining, the algorithm that we use, uses only support to determine the association. The reason that we chose not to use the confidence value is because it can give misleading association rules in cases where some files have changed significantly more often than others [7]. It was also part of our motivation for considering correlation rule mining, which takes into account how often both files are changing together as well as separately. In their implementation, they use an online approach, where only the patterns that match the current query are computed, whereas we compute all the patterns and use them for multiple queries. Both Zimmermann et al.'s approach and ours produce similar quantitative results: Precision and recall that measure the predictability of the recommendations are similar in value when one single starting file is in the query. Zimmermann et al. also did similar experiments with a different number of starting files. The qualitative analyses differ. They present some change associations that were generated from their approach and argue that these associations are of interest. In contrast, we analyzed the recommendations provided in the context of completed modification tasks, emphasizing when the results would be of value to a developer. We assessed the value of the recommendations using the interestingness criteria that we developed.

Shirabad et al. also address a similar question to that addressed in this paper: When a programmer is looking at a piece of code, they want to determine which other files or routines are relevant [25]. They proposed an approach that predicts the relevance of any pair of files based on whether the files have been looked at or changed together. Information about pairs of relevant files is used to learn concepts by building decision trees on attributes, such as the length of the common filename prefix and the number of shared routines. Similar to our approach, their approach can apply across languages and platforms if the attributes do not depend on programming constructs. Our approach differs in that we find files that change together *repeatedly* instead of only changing at least once. Their results show that the error rate of classifying a file, given another file, to one of the three levels of relevance is 27 percent on average, which is better than the error rate of 67 percent when the relevance value is assigned randomly. In contrast, our notion of recall and precision is based on whether the recommendations are correct with respect to a given modification.

Hipikat is a tool that provides recommendations about project information a developer should consider during a modification task [9]. Hipikat draws its recommended information from a number of different sources, including the source code versions, modification task reports, newsgroup messages, email messages, and documentation. In contrast to our approach, Hipikat uses a broader set of information sources. This broad base allows Hipikat to be used in several different contexts for recommending different artifacts for a change task. When a recommendation is requested based on a description of a modification task at hand, Hipikat recommends similar modifications completed in the past, with their associated file changes. Our approach is complementary to Hipikat because our approach can be used when Hipikat does not apply. First, Hipikat requires a similar bug to have been fixed in the past to provide a specific recommendation about which code is of interest. Our approach may apply even when a similar bug cannot be determined. In addition, Hipikat's analysis requires a board source of artifacts, including the modification tasks reports. Our approach applies to projects where information on modification tasks is not available or where revisions of the system cannot be associated with modification task descriptions.

Some techniques that leverage development history data are proposed to identify groups of relevant code. Mockus and Weiss proposed a method to identify groups of related source code, called *chunks*, for independent development [21]. Their approach of identifying chunks involves iteratively finding a better grouping by optimizing some quantitative measures based on development history, such as the number of modification tasks that involve source code entities within a chunk. Thus, the result is that a source code entity within a chunk is often changed together with another entity in the same chunk, but not with an entity in a different chunk. Fischer et al. have used development history data for identifying a software system's *features*, which are referred to as an observable and relatively closed behavior or characteristic of a software part, such as HTTP [10]. Similar to Mockus and Weiss's work and Fischer et al.'s work, our approach is also based on the notion of files being changed together in the development history; however, the recommendations given by our approach are task-specific.

6.2 Other Approaches in Detecting Related Code

Impact analysis approaches (e.g., [4]) attempt to determine, given a point in the code base involved in a modification task, all other points in the code base that are transitively dependent upon the seed point. This information may help a developer determine what parts of the code base are involved in the modification task. Many of these approaches are based on static slicing (e.g., [12]) and dynamic slicing (e.g., [1]). Static slicing identifies all of the statements in a program that might affect the value of a variable at a given point in the program by analyzing the data-flow and control-flow of the source code. Dynamic slicing finds all parts of source code that affect a variable in an execution for some given input of the program, rather than for all inputs as in static slicing. In contrast to these approaches, our data mining approach can work over code written in multiple languages and platforms and scales to use on large systems. In addition, dynamic slicing relies on an executable program and on the availability of appropriate inputs of the program, whereas our approach can work with code that is nonexecutable, or with code that consists of components running on different platforms. On the other hand, slicing approaches can provide finer-grained information about code related to a modification task, without relying on the code having been changed repeatedly in the past. Our approach complements slicing approaches.

Code clone detection attempts to determine code that is the same or similar. This information may help a developer determine different instances of the clones when one instance of the clone changes in a modification task. Many of these approaches defined clones based on various program aspects such as software metrics (e.g., [17]), the program's text (e.g., [5]), abstract syntax trees (e.g., [6]), and program dependency graphs (e.g., [14]). In contrast, our approach based on frequency of code changed in the past is not limited to identifying only instances of code that are the same or similar to each other.

6.3 Other Data Mining Approaches for Software Engineering Tasks

Zimmermann et al. have also applied association rule mining to a different problem than that described in Section 6.1: determining evolutionary dependencies among program entities for the purpose of determining and justifying a system's architecture [28]. This involves determining the degree of modularity of a system based on analyzing the density of evolutionary dependencies between entities in the source as well as the proportion of inter versus intra-entity evolutionary coupling.

Association mining has been used for suggesting structurally related code. Michail used such an approach to find library reuse patterns to aid a developer in building applications with a particular library [18], [19]. The extracted patterns summarize usage information on the program structure of the program, such as explicating that application classes which inherit from a particular library class often override certain member functions. Tjortjis et al. apply association rule mining to help recover a structure for a program [26]. They find association rules that describe related source code attributes such as variables, data types, and method invocation. They consider source code fragments that contain many attributes involved in such rules as a collection of structurally related fragments. Sartipi et al. use both association rule mining and clustering to identify structurally related fragments in the architecture recovery process [24]. These three approaches differ in both intent and form. In addition, we are applying the data mining to version data, rather than to a single version as in Michail's, Tjortjis et al.'s, and Sartipi et al.'s work.

7 Conclusion

In this paper, we have described our approach of mining change history to help a developer identify pertinent source code for a change task at hand. We have validated our hypothesis that our approach can provide useful recommendations by applying the approach to two open-source systems, Eclipse and Mozilla, and then evaluating the results based on the predictability and likely interestingness to a developer. Although the precision and recall are not high, recommendations can reveal valuable dependencies that may not be apparent from other existing analyses. In addition to providing evidence for our hypothesis, we have developed a set of interestingness criteria for assessing the utility of recommendations; these criteria can be used in qualitative analyses of source code recommendations provided by other systems.

Acknowledgments

This research was funded by an NSERC postgraduate scholarship, an NSERC research grant, and IBM. The authors would like to thank Davor Cubranic for his help on the Hipikat database and Brian de Alwis, Robert O'Callahan, Martin Robillard, and the anonymous reviewers for insightful comments on earlier drafts.

References

[1] H. Agrawal and J.R. Horgan, "Dynamic Program Slicing," *Proc. Conf. Programming Language Design and Implementation,* pp. 246-256, June 1990.

[2] R. Agrawal, T. Imielinski, and A.N. Swami, "Mining Association Rules between Sets of Items in Large Databases," *Proc. Int'l Conf. Management of Data,* pp. 207-216, 1993.

[3] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. Int'l Conf. Very Large Data Bases,* pp. 487-499, 1994.

[4] R. Arnold and S. Bohner, *Software Change Impact Analysis.* IEEE CS Press, 1996.

[5] B.S. Baker, "A Program for Identifying Duplicated Code," *Computing Science and Statistics,* vol. 24, pp. 49-57, 1992.

[6] I.D. Baxter, A. Yahin, L.M.D. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," *Proc. Int'l Conf. Software Maintenance,* pp. 368-377, 1998.

[7] S. Brin, R. Motwani, and C. Silverstein, "Beyond Market Baskets: Generalizing Association Rules to Correlations," *Proc. Int'l Conf. Management of Data,* pp. 265-276, 1997.

[8] W. Cheung and O. Zaines, "Incremental Mining of Frequent Patterns without Candidate Generation or Support Constraint," *Proc. Int'l Database Eng. and Applications Symp.,* pp. 111-116, 2003.

[9] D. Cubranic and G.C. Murphy, "Hipikat: Recommending Pertinent Software Development Artifacts," *Proc. Int'l Conf. Software Eng.,* pp. 408-418, 2003.

[10] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and Relating Bug Report Data for Feature Tracking," *Proc. Working Conf. Reverse Eng.,* pp. 90-99, 2003.

[11] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," *Proc. Int'l Conf. Software Maintenance,* pp. 23-33, 2003.

[12] K. Gallagher and J. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Trans. Software Eng.,* vol. 17, no. 8, pp. 751-761, Aug. 1991.

[13] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," *Proc. Int'l Conf. Management of Data,* W. Chen et al., eds., pp. 1-12, 2000.

[14] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," *Proc. Working Conf. Reverse Eng.,* pp. 301-309, 2001.

[15] D. LeBlang, *The CM Challenge: Configuration Management that Works.* John Wiley & Sons, 1994.

[16] B. Magnusson and U. Asklund, "Fine Grained Version Control of Configurations in Coop/Orm," *Proc. Int'l Synp. System Configuration Management,* pp. 31-48, 1996.

[17] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics," *Proc. Int'l Conf. Software Maintenance,* pp. 244-254, 1996.

[18] A. Michail, "Data Mining Library Reuse Patterns in User-Selected Applications," *Proc. Int'l Conf. Automated Software Eng. ,* pp. 24-33, 1999.

[19] A. Michail, "Data Mining Library Reuse Patterns Using Generalized Association Rules," *Proc. Int'l Conf. Software Eng.,* pp. 167-176, 2000.

[20] A. Mockus, R.T. Fielding, and J. Herbsleb, "Two Case Studies of Open Source Software Development: Apache and Mozilla," *ACM Trans. Software Eng. and Methodology,* vol. 11, no. 3, pp. 1-38, 2002.

[21] A. Mockus and D.M. Weiss, "Globalization by Chunking: A Quantitative Approach," *IEEE Software,* vol. 18, no. 2, pp. 30-37, 2001.

[22] J.S. Park, M.-S. Chen, and P.S. Yu, "Using a Hash-Based Method with Transaction Trimming for Mining Association Rules," *Trans. Knowledge and Data Eng.,* pp. 813-825, 1997.

[23] D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Comm. ACM,* pp. 1053-1058, 1972.

[24] K. Sartipi, K. Kontogiannis, and F. Mavaddat, "Architectural Design Recovery Using Data Mining Techniques," *Proc. European Conf. Software Maintenance and Reeng.*, pp. 129-140, 2000.

[25] J.S. Shirabad, T.C. Lethbridge, and S. Matwin, "Supporting Maintenance of Legacy Software with Data Mining Techniques," *Proc. Conf. the Centre for Advanced Studies on Collaborative Research*, 2000.

[26] C. Tjortjis, L. Sinos, and P. Layzell, "Facilitating Program Comprehension by Mining Association Rules from Source Code," *Proc. Int'l Workshop Program Comprehension*, pp. 125-133, 2003.

[27] M. Weiser, "Program Slicing," *Trans. Software Eng.*, vol. 10, no. 7, pp. 352-357, July 1984.

[28] T. Zimmermann, S. Diehl, and A. Zeller, "How History Justifies System Architecture (or Not)," *Proc. Int'l Workshop Principles of Software Evolution*, 2003.

[29] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *Proc. Int'l Conf. Software Eng.*, pp. 563-572, 2004.

Annie T.T. Ying received the BSc (hons) and the MSc degrees in computer science from the University of British Columbia in 2001 and 2003, respectively. Her MSc thesis involved mining past changes for recommendating relevant changes to a current modification to a software system, as presented in this paper. She is currently a software engineer on a configuration management team at the IBM T.J. Watson Research Center.

Gail C. Murphy received the BSc degree in computing science from the University of Alberta in 1987 and the MS and PhD degrees in computer science and engineering from the University of Washington in 1994 and 1996, respectively. From 1987 to 1992, she worked as a software designer in industry. She is currently an associate professor in the Department of Computer Science at the University of British Columbia. Her research interests are in software evolution, software design, and source code analysis. She is a member of the IEEE Computer Society.

Raymond Ng received the BSc (hons) degree in computer science from the University of British Columbia in 1984, the MMath degree in computer science from the University of Waterloo in 1986, and the PhD degree in computer science from the University of Maryland, College Park, in 1992. He is now a full professor at the University of British Columbia. His areas of research include data mining, bioinformatics, image databases, and multimedia systems. He has published numerous conference and journal papers on these topics. He is one of the associate editors for the *IEEE Transactions on Knowledge and Data Engineering*, and the *VLDB Journal*. He is a program cochair of the 2002 SIGKDD Conference and has served as a member of program committees for many premier conferences, including ACM SIGMOD, VLDB, SIGKDD and ACM PODS, and ACM PODS.

Mark C. Chu-Carroll received the BA degree in computer science from Rutgers University in 1990 and the PhD degree in computer and information sciences from the University of Delaware in 1996. Since graduation, he has worked for the IBM T.J. Watson Research Center on a variety of projects focused on helping make software development more productive and more fun for developers. He is currently the project lead on the open-source Stellation project, which is focused on providing a platform for building collaborative software development tools.