Towards a Taxonomy of Approaches for Mining of Source Code Repositories

Huzefa Kagdi, Michael L. Collard, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
{hkagdi, collard, jmaletic}@cs.kent.edu

ABSTRACT
Source code version repositories provide a treasure of information encompassing the changes introduced in the system throughout its evolution. These repositories are typically managed by tools such as CVS. However, these tools identify and express changes in terms of physical attributes i.e., file and line numbers. Recently, to help support the mining of software repositories (MSR), researchers have proposed methods to derive and express changes from source code repositories in a more source-code "aware" manner (i.e., syntax and semantic). Here, we discuss these MSR techniques in light of what changes are identified, how they are expressed, the adopted methodology, evaluation, and results. This work forms the basis for a taxonomic description of MSR approaches.

Categories and Subject Descriptors
D.2.7. [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *documentation, enhancement, extensibility, version control*

General Terms
Management, Experimentation

Keywords
Mining Software Repositories, Taxonomy, Survey

1. INTRODUCTION
Software version history repositories are currently being extensively investigated under the umbrella term *Mining of Software Repositories* (MSR). Many of the repositories being examined are managed by CVS (Concurrent Versions System). In addition to storing difference information across document(s) versions, CVS annotates code commits, saves user-ids, timestamps, and other similar information. However, the differences between documents are expressed in terms of physical entities (file and line numbers). Moreover, CVS does not identify/maintain/provide any change-control information such as grouping several changes in multiple files as a single logical change. Neither does it provide high-level semantics of the nature of corrective maintenance (e.g., bug-fixes).

Researchers have identified the need to discover and/or uncover relationships and trends at a syntactic-entity level of granularity and further associate high-level semantics from the information available in the repositories. Recently, a wide array of approaches emerged to extract pertinent information from the repositories, analyze this information, and derive conclusions within the context of a particular interest.

Here, we present our analyses showing the similarities and variations among six recently published works on MSR techniques. These examples represent a wide spectrum of current MSR approaches. Our focus in on comparing these works with regards to the following three dimensions:

- Entity type and granularity
- How changes are expressed and defined
- Type of MSR question.

Further, we define notation to describe MSR in an attempt to facilitate a taxonomic description of MSR approaches. Finally, we outline the MSR process in terms of the underlying entities, changes, and information required to answer a high-level MSR question. We believe this work provides a better insight of the current research in the MSR community and provides groundwork for future direction in building efficient and effective MSR tools.

The remainder of the paper is organized as follows: section 2 discusses the various MSR approaches, section 3 gives a formal definition of MSR, section 4 outlines the MSR process and requirements, and finally we draw our conclusions.

2. APPROACHES TO MSR
A number of approaches for performing MSR are proposed in the literature. Here, we discuss these techniques with regards to the identified entities, questions addressed, evaluation, and results.

2.1. MSR via CVS Annotations
One approach is to utilize CVS annotation information. In the work presented by Gall et al [2, 3], common semantic (logical and hidden) dependencies between classes on account of addition or modification of a particular class are detected, based on the

version history of the source code. A sequence of release numbers for each class in which it changed are recorded (e.g., class A =<1, 3, 7, 9>). The classes that changed in the same release are compared in order to identify common change patterns based on the author name and time stamp from CVS annotations. Classes that changed with the same time stamp (in a 4 minute window) and author name are inferred to have dependencies. In summary, this work seeks answers to the following representative questions:

- Which classes change together?
- How many times was a particular class changed?
- How many class changes occurred in a subsystem (files in a particular directory)?
- How many class changes occurred across subsystems?

This technique is applied on 28 releases of an industrial system written in Java with half a million LOCS. The authors reported that the logical couplings were revealed with a reasonable recall when verified manually with the subsequent release. The authors suggest that logical coupling can be strengthened by additional information such as the number of lines changed and the CVS comments.

In another study, the file-level changes in mature software (the email client *Evolution*) are studied by German [4]. The CVS annotations are utilized to group subsequent changes into what is termed a modification request (*MR*). Here, the focus is on studying bug-*MR*s and comment-*MR*s to address the following questions:

- Do *MR*s add new functionality or fix different bugs?
- Are *MR*s different in different stages of evolution?
- Do files tend to be modified by the same developer?

Further effort was on investigating the hypotheses that bug-*MR*s involve few files whereas comment-*MR*s involve large number of files.

2.2. MSR via Data Mining

Data mining provides a variety of techniques with potential application to MSR. Association rule mining is one such technique. As an example, the recent work by Zimmerman et al [8] aims to identify co-occurring changes in a software system, For example, when a particular source-code entity (e.g., function with name *A*) is modified what other entities are also modified (e.g., functions with names *B* and *C*). This is akin to market-basket analysis in Data Mining. The presented tool, *ROSE*, parses the (C++, Java, Python) source code to map the line numbers to the syntactic or physical-level entities. These derived entities are represented as a triple (*filename, type, identifier*). The subsequent entity changes in the repository are grouped as a transaction. An association rule mining technique is employed to determine rules of the form $B, C \Rightarrow A$. Examples of deriving association rules such as a particular "*type*" definition change leads to changes in instances of variables of that "*type*" and coupling between interface and implementation is demonstrated. This technique is applied on eight open-source projects with a goal of utilizing earlier versions to predict the changes in the later versions. Although performed at a function and variable granularity, the best precision reported was 26% at the file-level granularity.

In summary, their technique brings forward various capabilities:

- Ability to identify addition, modification, and deletion of syntactic entities without utilizing any other external information (e.g., AST).
- Handles various programming languages and HTML documents.
- Detection of hidden dependencies that cannot be identified by source-code analysis.

2.3. MSR via Heuristics

CVS annotation analysis can be extended by applying heuristics that include information from source code or source-code models. A variety of heuristics, such as developer-based, history-based, call/use/define relation, and code-layout-based (file-based), are proposed and used by Hassan et al [5] to predict the entities that are candidates for a change on account of a given entity being changed. CVS annotations are lexically analyzed to derive the set of changed entities from the source-code repositories. The following assumptions were used: changes in one record are considered related; changes are symmetric; and the order of modification of entities in a change set is unimportant. The authors briefly state that they have developed techniques to map line-based changes to syntactic entities such as functions and variables, but it was not completely clear the extent to which this is automated.

These heuristics are applied to five open-source projects written in C. General maintenance records (e.g., copyright changes, pretty printing, etc) and records that add new entities are discarded. The best *average* precision and recall reported in table 3 of [5] was 12% (file-based) and 87% (history) respectively. The call/use/define heuristics gave a 2% and 42% value for precision and recall respectively while the hybrid heuristics did better.

The research in both [8] and [5] use source-code version history to identify and predict software changes. The questions that they answered are quite interesting with respect to testing and impact analysis.

2.4. MSR via Differencing

Source-code repositories contain differences between versions of source code. Therefore, MSR can be performed by analyzing the actual source-code differences. Such an approach is taken by the tool *Dex*, presented by Raghavan et al [7], for detecting syntactic and semantic changes from a version history of C code. All the changes in a patch are considered to be part of a single higher level change, e.g., bug-fix. Each version is converted to an abstract semantic graph (ASG) representation. A top-down or bottom-up heuristics-based differencing algorithm is applied to each pair of in-memory ASGs specialized with *Datrix* semantics. The differencing algorithm produces an edit script describing the nodes that are added, deleted, modified, or moved in order to achieve one ASG from another. The edit scripts produced for each pair of ASGs are analyzed to answer questions from entity-level changes such as how many functions and function calls are inserted, added or modified to specific changes such as how many *if* statement conditions are changed. *Dex* supports 398 such statistics.

This technique was applied to version histories of GCC and Apache. Only bug-fix patches were considered (deduced from the CVS annotations), 71 for GCC and 39 for Apache respectively.

The differencing algorithm takes polynomial time to the number of nodes. Average time of 60 seconds and 5 minutes per file were reported for Apache and GCC respectively on a 1.8 Ghz Pentium IV Xeon 1GB RAM machine. The six frequently occurring bug-fix changes as a percentage of patches in which they appear are reported. *Dex* reported 378 out of 398 statistics always correct with an average rate of 1.1 incorrect results per patch.

In an approach by Collard et al [1, 6] a syntactic-differencing approach called *meta-differencing* is introduced. The approach allows you to ask syntax-specific questions about differences. This is supported by encoding AST information directly into the source code via an XML format, namely srcML, and then using *diff* to compute the added, deleted, or modified syntactic elements. The types and prevalence of syntactic changes are then easily computed. The approach supports queries such as:

- Are new methods added to an existing class?
- Are there changes to pre-processor directives?
- Was the condition in an if-statement modified?

While no extensive MSR case study has been carried out using meta-differencing, it does support the functionality necessary to address a range of these problems. Additionally, the method is fairly efficient and usable with run times for translation similar to that of compiling and computation of the meta-difference is around five times that of *diff.*

3. A DEFINITION OF MSR

The investigations described in the previous section have a number of common characteristics. They all are working on version release histories (changes), all work at some level of change granularity (software entity), and most of them ask a very similar (MSR) question. We also see that the MSR process is to extract pertinent information from repositories, analyze this information, and derive conclusions within the context of software evolution. From these examples we further define MSR by identifying some fundamental representational issues and defining the terminology so we can contrast the different approaches. However, first we discuss the types of questions asked.

3.1. MSR Questions & Results

What types of questions can be answered by MSR? In the examples described in section 2 we see two basic classes of questions. The first is a type of market-basket question and the other deals with the prevalence, or lack of, a particular type of change. The market-basket[1] type question is formulated as: If *A* happens then what else happens on a regular basis? The answer to such a question is a set of rules or guidelines describing situations of trends or relationships. That is, if *A* happens then *B* and *C* happen *X* amount of the time.

This type of question often addresses finding hidden dependencies or relationships and could be very important for impact analysis. MSR identifies (or attempts to identify) the actual impact set after the fact (i.e., after an actual change). However, MSR oftentimes gives a "best-guess" for the change. The change may not be explicitly documented and as such must sometimes be inferred. This is an interesting trade-off and is reflected in the results described in Hassan et al [5] and Zimmerman et al [8].

The other type of question addressed in the examples discussed concerns the characteristics of common changes. The work by Raghavan et al [7] asks the question: What is the most common type of change in a bug-fix? This also has implications to impact analysis but not directly.

To even begin to answer these types of high-level questions we need to address the practical aspects of extracting facts and information from source-code repositories.

3.2. Underlying Representation

Repositories consist of text documents containing source code (e.g., *routine.h, routine.cpp*). The representation of differences between versions may also contain source code (e.g., output of *diff*). If the mining process uses the source code in its original document form than fact extractors are limited to using a light-weight approach, such as regular expressions as an API to the source code. The source code can also be represented in a data view, such as an AST (Abstract Syntax Tree). The AST view allows an API that is based on the abstract syntax of the source code.

The choice of representation is very important. Using a textual document view allows access to all parts of the document including comments, white space, and particular ordering information. Tools such as *diff* also work on text files. However, this textual view creates difficulty in determining the contents of a particular version. On the other hand, using an AST view of the source code does not easily allow access to white space, comments, etc.

The representation of the differences between source-code documents is an extension of the source-code representation. Textual representations can use tools such as *diff.* Regions of lines that are deleted or added are recorded, along with additional lines of text of the added lines. The ASG tree/graph-based representations of a program allow for changes to be represented as tree/graph changes and can include syntactic information easily.

As information is extracted for the purpose of mining it must be stored. Because of the large amounts of source code involved the extraction result is often chosen to produce as compact a result as possible. For example, if the purpose is to take a single measurement of each source-code document then only this single result is required.

The higher the abstraction of the extraction result the more specific the purpose of the extraction. This makes methods and tools for extracting results unusable for other, even closely related, applications.

Note that the desire to not store all of the original documents is partially based on the source-code representation chosen. An AST representation can be hundreds of times larger than the original document. There is too much to store in memory simultaneously, so an external representation format must be used.

These differences in representation and the level of syntactic information extracted often makes methods and tools for extracting results unusable for other, even closely related, MSR applications.

[1] The term market-basket analysis is widely used in describing data mining problems. The famous example about the analysis of grocery store data is that "people who bought diapers often times bought beer".

3.3. Definitions of Terms

With respect to MSR the basic concepts involve the level of granularity of what type of software entity is being investigated, the changes, and the underlying nature of a change. We present the definitions of these concepts in an attempt to form a terminology for what a change is and how it can be expressed within the context of MSR. If a need arises, these definitions will be refined to accommodate the future MSR approaches as they emerge.

Definition: An *entity*, *e*, is a physical, textual, or syntactic element in software. Example: file, line, function, class, comment, if-statement, white-space, etc.

Definition: A *change*, δ, is a modification, addition, or deletion to, or of, an entity. Additionally, this change defines a mapping from the original entity to the new entity as in $\delta(e) \rightarrow e'$, $\delta(\varnothing) \rightarrow e'$ is addition, and $\delta(e) \rightarrow \varnothing$ is deletion. A change describes which entities are changed and where the change occurs.

Definition: The *syntax* of a change is a concise and specific description of the syntactic changes to the entity. This description is based on the grammar of the language(s) of entities. We classify δ in the context of *e* as having some specific syntactic type (if-statement), change type (add, remove), location, etc. For example: a condition was added to an if-statement; a parameter was renamed; an assignment statement was added inside a loop; etc. The notation for deriving the syntax of a change is as follows: $syntax(e, \delta) = (d_1, d_2, \ldots, d_n)$ where each d_i is some descriptor of the syntax.

Definition: The *semantics* of a change – is a high-level, yet concise, description of the change in the entity's semantics or feature set. This may be the result of multiple syntactic changes that is, $\Delta = \delta_1 \circ \delta_2 \circ \ldots \circ \delta_n$. For example: a class interface change; a bug fix; a new feature was added to a GUI; etc. So we can now define notation for the semantics of a change as: $semantics\ (e, \Delta) = (d_1, d_2, \ldots, d_n)$ where each d_i is some descriptor of the semantics.

4. INFORMATIONAL REQUIREMENTS

Mining of Software Repositories (MSR) is operationalized by the dimensions of the problem and types of information that must be extracted to support the high-level question. We feel the following are key dimensions to categorize MSR approaches:

- Entity type and granularity used (e.g., file, function, statement, etc.);
- How changes are expressed and defined (e.g., modification, Addition, Deletion, location, etc.);
- Type of question (e.g., market-basket, frequency of a type of change, etc.).

We have already addressed the type of questions in section 3.1. We now need to focus on the information necessary to answer these questions. From the discussion in section 3, we see that two types of pertinent information need to be extracted to answer MSR questions, namely entity-level information and information about the nature of change of an entity. We now describe each category and the specific types of fact extraction associated with each.

4.1. Entity-Level Information

The entity-level category addresses which entities changed, the location of the changed entities, and how many were changed. For example if functions represent our entities then we want to be able to answer queries such as:

- Which functions were added?
- Was the function *A* modified?
- How many functions were deleted?

The first query is a discovery or fact extraction activity regarding functions that were added between given source-code versions (e.g., a list of functions [*f1,f2, .. fn*]). Similar questions can be defined for the deletion, modification, or movement of an entity. MSR approaches need this information for addressing questions such as identifying relations between functions that were added i.e., did a addition/deletion of a particular function lead to the addition/deletion of other functions?

The second query regards a particular function of interest. The research discussed in section 2.1 (CVS Annotations) needs this type of support to determine whether a particular class was modified in a given version.

The last query is an aggregate count that is useful for identification of higher level semantic changes such as those in the techniques discussed in section 2.4 (Differencing).

4.2. Change Information

Determining the nature of a change in an entity is the next step in the process. This kind of change can be syntactic or semantic. This specific change information can enhance the research presented in section 2.2 (Data Mining) by enabling the restricted application of the association rules and thus cutting down the list of affected entities that are reported. For example, consider a case where the change to an existing if-statement is only in the condition. The rule *{if-condition change} A* $\Rightarrow$*B, C* would report *B and C* as affected entities only when the precondition *{if-condition change}* in entity *A* is satisfied. Augmenting these rules with the exact nature of change further reduces the number of affected components and applicable association rules; thus avoiding false positives. Also, to determine a semantic change, such as identifying interface changes, this type of knowledge is needed:

- Are the modifications in function A only in if-statements?
- Was the conditional in the 2nd if-statement deleted in function A?

The higher-level semantic information such as identification of conditional bugs addressed by research discussed in section 2.4 (Differencing) needs lower-level facts as reported by the above questions:

- Were only comments changed in function *A*?
- Was the header comment of function *A* modified?
- Is there a change in the code layout in an entity *A*?

These questions enable analysis to utilize or discard such textual changes. The research discussed in section 2.3 (Heuristics) analyzed CVS message annotations to discard header comment changes and proposed heuristics on predicating change propagation based on developer name and code layout. This approach can be augmented with the facts gathered by the above questions.

Table 1. A taxonomy of MSR approaches.

	Entity	Change	Question
Annotation Analysis			
Gall et al	class	syntax and semantic - hidden dependencies	market basket and prevalence
German	file & comment	syntax and semantic – file coupling	market basket and prevalence
Heuristic			
Hassan et al	function & variable	syntax and semantic - dependencies	market basket
Data Mining			
Zimmerman et al	class & method	syntax and semantic - association rules	market basket
Differencing			
Raghavan et al	logical statement	syntax and semantic – move	prevalence
Collard et al	Logical statement	syntax – add, delete, modify	prevalence

5. CONCLUSIONS

In Table 1 we present an overview of the discussed approaches along with their MSR characteristics. We've categorized them generally into four groups (along the left). Then for each, we identify what granularity of entities they deal with, what types of changes they express (as defined in section 3.3), and what general class of question they are trying to address.

There is a large difference in the level to which these approaches understand the programming language syntax. Most of the approaches work with a fairly high-level entity. The two differencing approaches however can work as low as primitive logical programming language statements (if, while, class, or function).

Further investigation is necessary to discern between how changes are expressed. Also, there is very different semantic information being used in the approaches. The notation we defined fits in well here but the domains must be further studied to support a more descriptive taxonomy. It is interesting to notice that both classes of questions are represented in this survey.

6. ACKNOWLEDGEMENTS

This work was supported in part by a grant from the National Science Foundation C-CR 02-04175.

7. REFERENCES

[1] Collard, M. L. Meta-Differencing: An Infrastructure for Source Code Difference Analysis. Kent State University, Kent, Ohio USA, Ph.D. Dissertation Thesis, 2004.

[2] Gall, H., Hajek, K., and Jazayeri, M. Detection of Logical Coupling Based on Product Release History in Proceedings of 14th IEEE International Conference on Software Maintenance (ICSM'98) (Bethesda, Maryland, March 16 - 19, 1998), 190-198.

[3] Gall, H., Jazayeri, M., and Krajewski, J. CVS Release History Data for Detecting Logical Couplings in Proceedings of Sixth International Workshop on Principles of Software Evolution (IWPSE'03) (Helsinki, Finland, September 01 - 02, 2003), 13-23.

[4] German, D. M. An Empirical Study of Fine-Grained Software Modifications in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04) (Chicago, Illinois, September 11 - 14, 2004), 316-325.

[5] Hassan, A. E. and Holt, R. C. Predicting Change Propagation in Software Systems in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04) (Chicago, Illinois, September 11 - 14, 2004), 284-293.

[6] Maletic, J. I. and Collard, M. L. Supporting Source Code Difference Analysis in Proceedings of IEEE International Conference on Software Maintenance (ICSM'04) (Chicago, Illinois, September 11-17, 2004), 210-219.

[7] Raghavan, S., Rohana, R., Podgurski, A., and Augustine, V. Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04) (Chicago, Illinois, September 11 - 14, 2004), 188-197.

[8] Zimmermann, T., Weißgerber, P., Diehl, S., and Zeller, A. Mining Version Histories to Guide Software Changes in Proceedings of 26th International Conference on Software Engineering (ICSE'04) (Edinburgh, Scotland, United Kingdom, May 23 - 28, 2004), 563-572.