Mining Software Repositories – A Comparative Analysis

Sunday O. Olatunji, Syed U. Idrees, , Yasser S. Al-Ghamdi, Jarallah Saleh Ali Al-Ghamdi
King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia

Abstract
Despite of many Mining Software Repositories (MSR) tools in use, it is a relatively new research domain, which forms the basis of classifying various tools and comparing them. In this paper we present a comparative analysis of different tools for MSR, based on some existing and new criteria proposed in this paper. This will assist in determining an appropriate tool that performs the best for a given type of application and to use it directly, instead of relying on the usual trial-and-error approach. This work has several purposes; it acts as a formative evaluation mechanism for tool designers (by quickly understanding and comparing different tools), as an assessment tool for potential tool users (by simply going through the comparative analysis chart to know at a glance, the essential components needed to be incorporated into the intended tool) and as a comparative milestone so that MSR tool researchers can easily differentiate amongst a pool of tools, thereby identifying other new research avenues. The tabular presentation furthers the work by providing a quick index to the reader and a means for quick analysis of the desired tool.
Key word: Mining, Software, Repositories

1. INTRODUCTION

In most projects collaborative documents or artifacts are collected and archived in software repositories: For open source projects, communications between the developers are stored in *mailing lists, newsgroups*, and *personal archives*. Changes to the source code of software are recorded in *version archives* such as CVS. Failures and feature requests are submitted to and discussed in the *issue tracking systems* such as Bugzilla. Explicit knowledge such as documentation and design documents are published on the websites and the likes. Recently a new research area evolved, that mines these software repositories. Several tools have been developed to facilitate mining software repositories. In this paper, we will present a comparative analysis of some MSR tools [AHME 04].
We utilized a comparative analysis criteria derived from a framework for the comparison proposed in [STOR 05] and [DANI 05] for different MSR tools and now suggest additional criteria to compare some new tools as an extension of the earlier cited paper. This comparative analysis is further enhanced by using a tabular presentation to provide a quick glance index to the reader and a means for quick analysis of the desired tools [DANI 05]. The whole of this paper is organized as follows:

- ✓ ***Section 1*** contains the general introduction to the work and the organization of the entire paper into sections.
- ✓ ***Section 2*** provides background information and brief literature survey of researches on MSR.
- ✓ ***Section 3*** consists of the proposed comparative analysis along with the methodology adopted.
- ✓ ***Section 4*** concludes the research work and suggests possible future work.

2. BACKGROUND

Mining Software Repositories is an active research area that utilizes Data Mining techniques to software projects' historical data in order to better understand the software development. This understanding can assist us in guiding and enhancing the software development process and methods.
Software projects accumulate a wealth of information over projects' lives, which can shed light on software engineers' coding habits that would cause defects or indicate a developer's special proficiency. It would allow us to improve change management and locate change patterns throughout source code files.
For software projects, data that is of interest for MSR are collected, either casually in:

- Mailing Lists
- Newsgroups
- Personal Archives, etc.

Or systematically in version archives like:

- Issue Tracking Systems (Examples of such systems are Bugzilla, GNATS for *issue tracking*,

Manuscript received August 5, 2010
Manuscript revised August 20, 2010

and CVS, Rational ClearCase for *source control* and *version tracking*)

Mining Software Repositories' active research areas include [PROC 05]:

- Approaches, applications, and tools for software repository mining
- Quality aspects and guidelines to ensure quality results in mining
- Proposals for exchange formats, meta-models and infrastructure tools to facilitate the sharing of extracted data and to encourage reuse and repeatability
- Models for social and development processes that occur in large software projects
- Search techniques to assist developers in finding suitable components for reuse
- Techniques to model reliability and defect occurrences
- Analysis of change patterns to assist in future development
- Case studies on extracting data from repositories of large long lived projects
- Other interesting and novel applications of mined data

3. METHODOLOGY

We present our chosen criteria for comparing MSR tools followed by the actual comparison [STOR 05] that forms the basis of our research work.

3.1 COMPARISON CRITERIA

Following nine are the criteria, upon which the comparison of our MSR tools would base:

- ***Intent***
- ***Information***
- ***Presentation***
- ***Interaction***
- ***Effectiveness***

The above five criteria have been used to compare some MSR tools in a recent research [STOR 05] while in [DANI 05] the following one is touched along with the extension of the first two criteria of the earlier cited research:

- ***Infrastructure***

However, we enhance the work by not only comparing various *new tools* in the light of the *above six criteria*, but also show a comparative analysis of these new tools on the basis of out newly proposed three more criteria. Together with the usage of different tools and different criteria, our contribution is easily reflected. The three new criteria we proposed are:

- ***Input data required***
- ***Language dependency***
- ***Availability***

In all, these criteria serve to provide a quick glance index to the reader and a means for quickly finding more information about a tool when needed. We discuss these in brief now:

3.1.1 Intent

"Intent" is all about who are the expected users of the tool (Role), time and cognitive support [STOR 05].
(i) Role: This dimension identifies who will use the tool. Roles include managers, developers, testers, maintainers, documenters, reverse engineers, reengineers and researchers.
(ii) Time: Tools may be classified on the basis of time, as to be past, present or future depending on whether it provides information about activities occurring in the distant or near past, present, or future.
(iii) Cognitive Support: Cognitive Support describes how a tool can help improve human cognition [WALE 03]. The questions that various roles can ask about developer activities can be roughly classified into four categories, which are authorship, rationale, chronology, and artifacts.

3.1.2 Information

"Information" describes the specific sources that the tool mines and the type of analysis it makes. This dimension is elaborated in more detail as it is most relevant to MSR [STOR 05].

(i) Change Management: Configuration management tools provide support for building systems by selecting specific versions of software artifacts [GRUN 01]. Version control tools contribute to software projects through software artifact management, change management and team work support [WU 04]. Change management is an important data source because it provides traceability: it records who performed a given change, and when it was performed. The capabilities of the change management system will determine the type of information that can be extracted.

(ii) Program Code: MSR tools are classified into two categories based on how they treat the file (i.e. source code). These are:

a) Programming-Language-Agnostic tools (which treat the file as a unit and make no effort to understand its contents)

b) Programming-Language-Aware tools (these tools attempt to do some fact extraction from the source code). The ***PLA*** tools are further classified based on the language supported, syntactic analysis and semantic analysis as follows:

- ***The Language Supported*:** Given the differences in syntax and grammar, tools that are language-specific can only understand a fixed set of programming languages. Thus MSR tools can be classified based on the language supported
- ***Syntactic Analysis:*** In this type of analysis the extractor does not need to understand what the code does, only its syntax. Examples of this analysis are the removal of comments from the source code (to be able to distinguish if the changes affected actual source code or only its documentation), and extraction of the main entities of the code (such as packages, classes, methods, functions, etc.)
- ***Semantic Analysis:*** This analysis requires an understanding of the intent of the source code and can be done dynamically (by running the software under well defined test-cases) or statically (by processing the source code)

(iii) Defect Tracking: Many larger software projects rely on tracking tools to help with the management of defects and change requests. Such systems often store metadata about who is assigned a task and track the task's completion. In some cases a defect management tool is also used as a way to track activities and changes in requirements.

(iv) Correlated Information: The type of analysis and correlation can be classified into two broad categories:

a) Within the Data Source: This type of analysis uses data from one data source only and attempts to correlate different data entities within it

b) Between the Data Sources: In this type, tools correlate entities from two different data sources.

(v) Informal Communication: Email is undoubtedly the most widely used form of computer-mediated communication, and distributed software development projects rely on it extensively. In the early days of open-source software, a project mailing list used to be one of the first, and often the only, communication and coordination mechanism used by development teams [CUBR 99], but email remains an essential component of distributed development process.

(vi) Local History: Schneider et al. describes how local interaction histories can be mined to support team awareness [SCHN 04]. They proposed that sharing local interactions among team members can benefit the following activities:

- Coordinating team member activities such as undo, identifying refactoring patterns and coordinating refactoring operations
- Mining browsing patterns to identify expertise
- Project management

3.1.3 Presentation

"Presentation" refers to how the tool or the proposed tool presents the extracted and derived information to the various user roles [STOR 05]:

(i) Form: The tool may present awareness information using a combination of ***text***, ***hypertext*** or ***graphics***.

(ii) Kinds of Views: Many tools provide awareness information in the form of ***annotations on existing views*** in a software environment. They may use visual variables or icons to emphasize the owner, state or history of a software artifact.

- ***Statistical Views*** (bar charts, histograms, etc.) provide comparison and analysis of human activity information
- ***Graph Views*** can also be used to display relationships between human and software artifacts
- ***Special Views*** customized; provided by some tools, to provide cognitive support for particular information seeking or understanding tasks (e.g. a special view is a matrix view which may be used to show trends and evolution patterns)
-

(iii) Techniques: Whether the tool provides annotations on existing views or specialized views, they will both use some ***visual variables*** such as color, position, size, transparency and map those to appropriate human activity attributes. ***Animation*** or motion can also be used effectively. Finally, we consider tools which rely on either user or tool-generated ***abstractions*** in communicating awareness information.

3.1.4 Interaction

"Interaction" refers to the interactivity and life of the tools [STOR 05]:

(i) Batch / Live: An important consideration is whether the tool operates ***offline*** or ***online*** [FROE 04]. Some offline tools require that the users write scripts to batch queries on a repository of information. The tool then

displays the queried information using static graphs. Other tools are online and provide updated displays of the information to the users on demand.

(ii) Customization: Effective interaction to suit particular user needs will normally require a high degree of customization. This characteristic addresses whether the available views can be further manipulated and to what extent they can be customized. ***Saving customizations*** and ***sharing customizations*** across team members may also be important.

(iii) Query mechanism: Some tools require special purpose ***languages*** to specify queries. Others allow the user to visually specify the queries through the use of specialized ***filter widgets*** (e.g. double sliders, checkboxes, etc.) or by interacting with the visualization directly (such as selection or brushing).

(iv) View Navigation: How the user navigates the displayed information is important, especially for tools with specialized views. Successful navigation requires that the user maintains orientation so that they know where they are and can decide where to go next. The use of an ***overview*** for ***detailed views*** can be used to provide orientation and to directly support navigation in the information space. Navigation can alternatively be supported by a ***zoom-able user interface*** and ***hypertext***. Another important consideration is that the user may need to compare two views ***side-by-side***. The facility to see multiple views at once provides cognitive support [WALE 03] as it reduces the memory load on the user and redistributes some of the required cognition from the user to the tool. To improve the usefulness of multiple views, views should be ***coupled***.

3.1.5 Effectiveness

"Effectiveness" captures the feasibility of the proposed approach, whether it has been evaluated and whether it has been deployed:

(i) Status: Some researchers propose approaches that have not yet been implemented. This characteristic captures robustness of the tool and checks whether the system has been partially or fully ***implemented***. Tool ***availability*** is also important so that other tool designers and researchers can evaluate it. The ***interoperability*** issue is also very important [FROE 04]. For ***scalability*** we must consider if the tool supports large software projects. If the technique does not appear to scale, it may
be the implementation which does not scale rather than the technique.

(ii) Cost: The adoption of any tool has a cost associated with it. ***Economic Cost*** is a key concern, in addition to other costs such as the cost of ***installing*** the tool, ***learning*** how to use it and the costs incurred during its ***usage***.

(iii) Evaluation: A tool that has been formally evaluated and compared to other approaches will more likely be adopted than one that has not. It is very common for these tools to be evaluated by the designers through informal ***case studies***. The complexity and size of the software in the case study is very important to consider. When a new tool has been evaluated with users other than the tool designers (i.e. in ***user studies***), coincidence in the tool's benefits will be further increased. If the tool has been deployed and subsequently ***adopted***, then the tool has been evaluated through its usage. The rate of adoption can be an important indicator of the usefulness of a tool. However, lack of adoption does not necessarily imply that the tool is not effective as adoption is affected by many forces.

Infrastructure

"Infrastructure" addresses any special needs that the tool has. It addresses the environment needed to support the tool. We further categorize our criteria as [DANI 05]:

(i) Required Infrastructure: This category lists any requirement the tools have, such as a given operating system, an IDE such as Eclipse, a Web server and client, a database management system, etc.

(ii) Online / Offline: These tools can be classified depending upon whether the software repository is required during its operation. For instance, some tools mine a software repository ahead of time while others query the repository as a result of a user request.

(iii) Storage Backend: If the tool operates offline, this category is used to describe how it stores its required data. Examples of backend that are commonly used include SQL backend and XML or a proprietary format.

Input Data Required

This is an important criterion when a particular tool is considered. It indicates to a new user what data he/she must have in order to be able to use the tool [GRIG 07]. Generally the tools require CVS transactions, text documents, etc. as input.

Language Dependency

The theoretical part of each tool is its independence from the language used (either programming or natural). However, a variety of tools are language dependent, restricting the usage, e.g. an English-language dependent

tool cannot be used for a system whose requirements are written in French or Italian [GRIG 07].

Availability
The availability of the MSR tools is a very important issue for new users or the one who tries to compare different tools. Most of the existing ones are publicly available, so that people can use them easily.

3.2 COMPARATIVE ANALYSIS OF MSR TOOLS

We present in this section the actual comparative analysis of different MSR tools using the comparison criteria presented earlier. The tools evaluated here are by no means exhaustive rather the tools presented here are a representative set of available tools.

SOFTCHANGE

Intent: The main goal of SOFTCHANGE is to help programmers, their managers and software evolution researchers in understanding how a software product has evolved since its conception. With respect to the time, SOFTCHANGE concentrates only on the past. In terms of cognitive support, it allows one to query about who made a given change to a software project (authorship), when (chronology) and whenever available, the reason for the change (rationale). The artifacts that SOFTCHANGE tracks are files and some types of entities in the source code (such as functions, classes, and methods) [STOR 05].
Information: SOFTCHANGE extracts and correlates three main sources of information: the version control system (CVS), the defect tracking system (Bugzilla) and the software releases. SOFTCHANGE reconstructs some of the information that is never recorded by CVS (such as recreating commits) and it does syntactic analysis of the source code. The analysis is static and it supports C/C++ and Java. SOFTCHANGE also attempts to correlate information between CVS and Bugzilla using defect numbers [STOR 05].
Presentation: SOFTCHANGE is composed of a hypertext component and a graphical component. The hypertext component allows the users to navigate, search and inspect, for a given change, who made it and when, the files were modified, why the change occurred and when applicable, the defect that was fixed. The graphical component provides two types of views:

(1) It calculates statistics and presents them in histograms where the horizontal axis is usually time, and therefore provides an overview of the evolution of the project
(2) It provides graphs that show files, authors and their interrelationships (such as which files have been modified together, or which authors modify which files). Figure 1 shows this in detail [STOR 05].

(3)

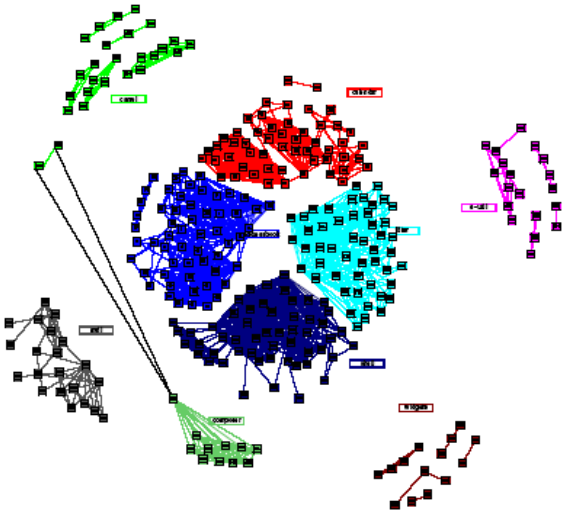

Figure1: A graph created by SOFTCHANGE

Interaction: SOFTCHANGE's hypertext interface allows the user to freely navigate and search the information space. The graphical views in SOFTCHANGE are generated in batch mode and the user is allowed to specify some parameters for their creation [STOR 05].

Effectiveness: SOFTCHANGE has been used by its authors in studies of software evolution and in the analysis of global software development practices in large open source projects. No formal user testing has been performed. It is available on request [STOR 05].

Infrastructure: SOFTCHANGE is an offline tool that uses an SQL database for its storage needs. Its mining is done without any special requirements beyond access to the software repository. Since SOFTCHANGE could retrieve a very large amount of data, it was recommended that it operate on a local copy of the repositories (rather than query the repositories using the Internet, consuming their bandwidth and computer resources). SOFTCHANGE has two different front ends; Web- Based and Java application [STOR 05].

Input Data Required: The input data for SOFTCHANGE is in the form of an extraction from the metadata from CVS and Bugzilla. It then performs the correlation to the collected data [STOR 05].

Language Dependency: SOFTCHANGE is language-dependent, restricting its usage. It is an English-language dependent tool which cannot be used for a system whose

requirements are written in any other language other than English [STOR 05].

Availability: SOFTCHANGE has two different front-ends [URL 1]:

(1) Web-Based
(2) JAVA Application

HIPIKAT

Intent: HIPIKAT can be viewed as a recommender system for software developers, which draws its recommendations from a project's development history [CUBR 04]. The tool is in particular intended to help newcomers to a software project. Therefore, in terms of the time dimension, it is concentrated on the past. Cognitive support is largely limited to answering questions about rationale and artifacts. In terms of user roles, HIPIKAT is targeted almost exclusively at developers and maintainers [DANI 05].

Information: HIPIKAT is designed to draw as many information sources as possible and identify relationships between documents both of same and different types. The information sources that are currently supported by HIPIKAT are Version Control System (CVS), Issue Tracking System (Bugzilla), Newsgroups and Archives of Mailing Lists, and the Project Web Site. All four of these sources are typically present in large open-source software projects. HIPIKAT is programming language-agnostic. The only information that it collects from files in the version control system is versioning data, such as author, time of creation and check-in comment. HIPIKAT correlates information across sources using a set of heuristics, such as matching for bug-id in version check-in comment to link file revisions in CVS and bug reports in Bugzilla. These heuristics are based on observations of development practices in open source projects like Mozilla. Another method that HIPIKAT uses to find documents that are related is by textual similarity [DANI 05].

Presentation: HIPIKAT is a GUI-based tool. Figure 2 shows a snapshot after installation [URL 2].

Interaction: The basic user interface of HIPIKAT is very simple, if it's possible to make a query on something on screen, there will be an option "Query Hipikat" in the right-click menu. E.g. HIPIKAT knows about files in the CVS. Therefore, a right-click on any versioned file in the Navigator, Package Explorer, or CVS Repositories view, even on a revision in the CVS Resource History view can select "Query Hipikat" from the context menu, as can be seen [URL 2].

Effectiveness: HIPIKAT also has a Bugzilla Search tab added to the Eclipse Search pane. The users can enter a bug ID or keywords in the text field and, if desired, limit the search by selecting particular attributes within the middle section of the search pane. Pressing Enter or clicking "Search" proceeds [URL 2].

Infrastructure: Repository mining in HIPIKAT works in offline mode: HIPIKAT periodically checks project repositories for recent changes and updates its model. The model is stored in an SQL database. The front end is an Eclipse plug-in, although in principle it could be implemented for other environments, as long as it follows the communication protocol with the HIPIKAT server [DANI 05].

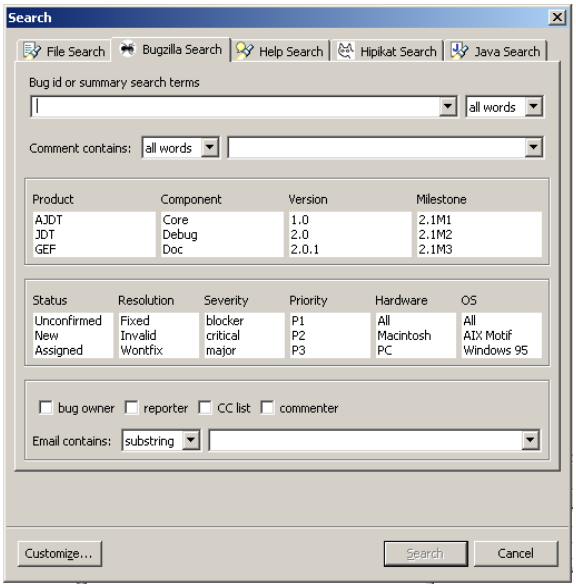

Figure 2: GUI-based HIPIKAT

Input Data Required: The only input data for HIPIKAT is from the files in the CVS through versioning data, such as author, time of creation and check-in comment [DANI 05].

Language Dependency: HIPIKAT is language-dependent, restricting its usage. It is an English-language dependent tool which cannot be used for a system whose requirements are written in any other language other than English [DANI 05].

Availability: HIPIKAT has a front-end in ECLIPSE Web-Based, although in principle, it could be implemented for other environments, too [URL 2].

DYNAMINE

Intent: The name DYNAMINE comes from the combination of Dynamic Analysis and Mining revision histories. DYNAMINE is a tool for discovering application-specific code change patterns and detecting their violations in large software systems where it is proved that violations of coding rules are responsible for a numerous number of errors. Violation of application specific coding rules is called error patterns [BENJ 05].

Information: DYNAMINE can be applied in code revision history such as CVS to find highly correlated method calls as well as common bug fixes in order to automatically discover application-specific coding patterns. Potential patterns discovered through mining are passed to a dynamic analysis tool for validation; finally, the results of dynamic analysis are presented to the user. DYNAMINE adapts dynamic analysis by looking for pattern violations at runtime [BENJ 05].

Presentation: DYNAMINE when applied to new applications involves mining and dynamic program testing steps which are accessible to the user from within custom ECLIPSE views. A diagram representing the architecture of DYNAMINE is shown below in the Figure 3 [BENJ 05].

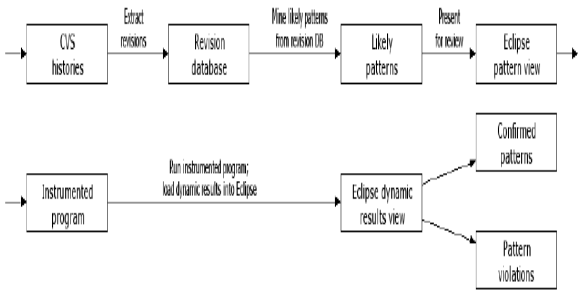

Figure 3: Architecture of DYNAMINE

Interaction: The DYNAMINE, a tool for learning common usage patterns from the revision histories of large software systems interacts through methods and calls by virtue of which any method can learn both simple and complicated patterns, scale to millions of lines of code and can be used to find more than 250 pattern violations [BENJ 05].

Effectiveness: The mining approach of DYNAMINE is effective at finding coding patterns as it is the first tool that combines revision history information with dynamic analysis for the purpose of finding software errors [BENJ 05], specifically using the Eclipse views.

Infrastructure: DYNAMINE can be setup to be used dynamically to discover usage patterns at run time or statically to discover in historical data. In static setup, applying DYNAMINE requires passing through pre-processing step, then an optimization is applied to reduce time and eliminate noise and then actual data mining is applied. DYNAMINE's Eclipse plug-in is used to present mining results to the user [BENJ 05].

Input Data Required: For a given source file revision, a transaction is a set of methods, calls to which are inserted. These together serve as the input to the DYNAMINE [BENJ 05].

Language Dependency: DYNAMINE is language-independent, allowing its wide-spread usage. It can be used for any system whose requirements are written in any other language other than English [BENJ 05].

Availability: The analysis of Eclipse and jEdit, two widely-used, mature, highly extensible applications by mining revision histories, to find dynamic valid patterns shows that the use of DYNAMINE is still available for cross-over projects between the areas of revision history mining and bug detection [BENJ 05].

KENYON

Intent: KENYON objective is to facilitate and speed up the software evolution research by providing a common framework that can be used to extract facts and apply any analysis method on any of the supported Source Control Management (SCM) systems [BEVN 05].

Information: KENYON is designed to support reading data from any SCM system; however it currently supports CVS, Subversion and ClearCase SCM systems. After KENYON automatically retrieves data from an SCM system, data is written to the file system and then fact extraction is done by subclasses specified by the user. These subclasses are the means by which external, analysis-specific, fact extraction tools interface with KENYON [BEVN 05].

Presentation: KENYON allows researchers to store the extracted facts in flexible data structures that can be easily used to compare different research results. Regarding time, KENYON reads historical data form SCM Systems [BEVN 05].

Interaction: KENYON supports data sampling at specific time interval and it can be set to perform sampling periodically. Also, it can be used to process historical data and it supports incremental update for processed data therefore it can keep up with ongoing development [BEVN 05].

Effectiveness: KENYON also reads the database configurations from the Object-Relational Mapping properties' file. KENYON saves the extracted facts into a relational database using an Object Relational Mapping (ORM) system [BEVN 05].

Infrastructure: KENYON works asynchronously, with minimum interaction from the user. The main module, and execution entry point is called the "Data Manager Class" which calls configuration reading, fact execution, and object storage methods. Configuration settings are provided through configuration file which is a text file that indicates the names of the data sources, settings selections and third-party tools to be invoked on each configuration [BEVN 05].

Input Data Required: KENYON supports multiple types of data from different types of systems with varying sizes and domains [BEVN 05].

Language Dependency: KENYON is language-independent, allowing its wide-spread usage. It can be used for any system whose requirements are written in any language like Chinese, Japanese, English, etc. [BENJ 05].

Availability: KENYON is ideally available in a web-based downloadable format to facilitate several common processing platforms and provide effective and helpful documentation o its intended audience [BEVN 05].

CHIANTI

Intent: CHIANTI is a plug-in for the Eclipse environment and intended to assist programmers estimating the effect of code change for Java programs. CHIANTI can be used as a debugging tool by using it to isolate the change that caused a test case to fail [XIAO 05].

Information: A Typical scenario of a CHIANTI session begins with the programmer editing the current project, extracting the latest stable version of this project from CVS repository into the workspace. The programmer then starts the change impact analysis launch configuration, and selects these two projects of interest as well as the test suite associated with these projects [XIAO 05].

Presentation: The CHIANTI shows all the tests in a tree view and each affected test can be expanded to show its set of affecting changes. Each affecting change is an atomic change that can be expanded on demand to show its prerequisite changes. By clicking on an atomic change the Eclipse Java IDE opens the associated program fragment. This quickly provides an idea of the different threads of changes that have occurred [XIAO 05].

Interaction: CHIANTI is intended for interactive use, however instead of comparing the current version with its local history it requires two versions of a program which are saved in two separate Java projects [XIAO 05].

Effectiveness: CHIANTI analyzes two versions of an application and then decomposes the differences as a set of atomic changes and report the change impact in terms of test cases which is affected by the change, also for each affected test CHIANTI also determines a set of affecting changes that were responsible for the test's modified behavior [XIAO 05].

Infrastructure: CHIANTI is designed as an Eclipse plug-in. It can be conceptually divided into three functional parts.
The first part is responsible for deriving the set of atomic changes from the two versions of the Java project.
The second part analyzes the affected tests and their affecting changes by reading "test call graphs" for the original and edited projects.
The third part is responsible of visualizing the results of the change impact analysis. Then analysis results and both atomic change information and call graphs are stored as XML files [XIAO 05].

Input Data Required: CHAINTI requires calls from the program written in Eclipse, a Java Plug-in [XIAO 05].

Language Dependency: CHAINTI is language-dependent as it has been integrated closely with the Eclipse, a Java Plug-in [XIAO 05].

Availability: After the experimentation results were made accurate in 2002, it is also available over the web.

APFEL

Intent: APFEL is an Eclipse plug-in, and the word APFEL means apple in Dutch and short for "A Preprocessing Framework for Eclipse (and CVS)". Researchers and Software Engineers that use tools such as HIPIKAT and eROSE can use the results of the analysis form APFEL for further analysis. APFEL tokenizes source [THOM 06].

Information: APFEL analyzes the changes on token level and represents the syntactic properties of the token such as type, name, context, and instance. Tokens such as method calls, variable usage, exception handling, and important class, can be used to distinguish changes from one version to another [THOM 06].

Presentation: APFEL aims to provide a fine grained analysis of the change in source code stored in CVS archive [THOM 06].

Interaction: APFEL processes one source file at a time [THOM 06].

Effectiveness: Different types of changes are distinguished by APFEL, such as modification / addition / deletion of an element. APFEL processes one source file at a time [THOM 06].

Infrastructure: APFEL stores the results of the analysis in a database, and it works as a plug-in for Eclipse, however, every time Every time APFEL processes a CVS repository it recreates the database. In order to use APFEL it requires Eclipse to be running [THOM 06].

Input Data Required: APFEL requires calls from the program written in Eclipse, a Java Plug-in [THOM 06].

Language Dependency: APFEL is language-dependent as it has been integrated closely with the Eclipse, a Java Plug-in [THOM 06].

Availability: Since APFEL is built upon the Eclipse infrastructure for CVS and Java, it can also be found on the [THOM 06].

3.3 TABULAR PRESENTATION OF COMPARISONS

(Table 1) INDEX OF MSR TOOLS

SC		SOFTCHANGE [DANI, 05]
HP		HIPIKAT [DANI, 05]
DM		DYNAMINE [BENJ, 05]
	INC	Incremental
KY		KENYON [BEVN, 05]
	RD	Relational Databases
	ORM	Object Relational Mapping Systems
	HIB	Hibernate 2.1.6 (KY's current ORM System)
CH		CHIANTI [XIAO, 05]
AP		APFEL [THOM, 06]

(Table 2) LEGENDS

DVL	Developers
MNT	Maintainers
REE	Reverse Engineers & Reengineers
MNG	Managers
TST	Testers
DOC	Documenters
RSR	Researchers
PS	Past
PR	Present
FU	Future
AU	Authorship
RA	Rationale
CN	Chronology
AR	Artifacts
PLAG	Programming Language Agnostic
PLAW	Programming Language Aware
TLS	The Language Supported
SYA	Syntactic Analysis
DY	Dynamically
SY	Statically
SEA	Semantic Analysis
WDS	Within the Data Source
BDS	Between the Data Source
EM	Electronic Mails
ML	Mailing Lists
CVS	Version Control System
ITS	Issue Tracking System (Bugzilla)
SR	Software Releases
NWG	Newsgroups
AML	Archives of Mailing Lists
PWS	The Project Website

(Table 3) COMPARISON OF MSR TOOLS BY INTENT CRITERION

Tools→		SC	HP	DM	KY	CH	AP
Role							
	DVL	✓	✓	✓		✓	✓
	MNT	✓	✓	✓		✓	✓
	REE						
	MNG	✓					
	TST			✓		✓	
	DOC						
	RSR	✓		✓	✓	✓	✓
Time							
	PS	✓	✓	✓	✓	✓	✓
	PR			✓	✓	✓	
	FU				✓		
Cognitive Support							
	AU	✓					
	RA	✓	✓	✓			
	CN	✓				✓	✓
	AR	✓	✓				

(Table 4) COMPARISON OF MSR TOOLS BY INFORMATION CRITERION

Tools →			SC		HP	DM	KY	CH	AP
Change Management						✓	✓	✓	✓
Program Code	PLAG				✓		✓		✓
	PLAW		✓			✓	✓	✓	✓
		TLS	✓			✓		✓	✓
		SYA	DY	✓		✓			
			SY	✓				✓	✓
		SEA	DY			✓			
			SY					✓	

Defect Tracking		✓	✓		✓	✓	
Correlated Information	WDS						
	BDS	✓	✓		✓		
Informal Communication	EM						
	ML						
Local History							
Information Sources Supported	CVS	✓	✓	✓	✓	✓	✓
	ITS	✓	✓		✓		
	SR	✓					
	NWG		✓				
	AML		✓				
	PWS		✓				

(Table 5) COMPARISON OF MSR TOOLS BY PRESENTATION CRITERION

Tools →	SC	HP	DM	KY	CH	AP
Forms	✓	✓	✓		✓	✓
Kinds of Views	✓		✓	✓		
Techniques	✓			✓	✓	✓

(Table 6) COMPARISON OF MSR TOOLS BY INTERACTION CRITERION

Tools ↓	SC	HP	DM	KY	CH	AP
Batch / Live	✓	✓	✓		✓	✓
Customizable			✓	✓		
Queries		✓	✓ INC	✓ INC	✓	✓
Navigation				✓ XML RD ORM Hib	✓ XML	✓ OS-DB

(Table 7) COMPARISON OF MSR TOOLS BY EFFECTIVENESS CRITERION

Tools ↓	SC	HP	XC	DM	KY	CH	AP
Status	✓	✓		✓		✓	✓
Evaluation	✓		✓	✓	✓		

(Table 8) COMPARISON OF MSR TOOLS BY INFRASTRUCTURE CRITERION

Tools ↓	SC	HP	DM	KY	CH	AP
Required infrastructure	✓	✓	✓		✓	✓
Online			✓	✓		
Offline	✓	✓	✓ INC	✓ INC	✓	✓
Storage Backend	✓ SQL			✓ XML RD ORM Hib	✓ XML	✓ OS-DB

(Table 9) COMPARISON OF MSR TOOLS BY INPUT DATA REQUIRED, LANGUAGE DEPENDENCY & AVAILABILITY CRITERA

Tools →	SC	HP	DM	KY	CH	AP
Input Data Required	Extraction of the metadata from CVS & Bugzilla	From the files in the CVS through versioning data	Transaction, a set of methods & calls to be inserted	Multiple types of data from different types of systems	Calls from the program written in Eclipse	Calls from the program written in Eclipse
Language Dependent	YES	YES	NO	NO	YES	YES
Availability	Web-based & Java	Web-based Eclipse	Web-based Eclipse & jEdit	Web-based	Web-based	Web-based Eclipse infrastructure for CVS and Java

4. CONCLUSION

We presented a comparative analysis of different tools for MSR, based on six existing criteria and three new proposed criteria. These three new criteria were the extensions to the previous ones. Tabular presentation has improved the comparative analysis by providing a quick glance index to the reader and a means for swift analysis of the desired tools. This research work is advantageous in that it helps individuals and tool designers to quickly understand and compare different tools and assists users to swiftly assess a potential tool rather than depending on trial-and-error approach. Though the MSR tools evaluated here are by no means exhaustive, yet they are a representative set of available tools. MSR is still a new research area yearning for more research work, particularly as relates to the tools.

REFERENCES

[1] Ahmed E. Hassan, Mining Software Repositories to Assist Developers and Support Managers, PhD. Thesis, School of Computer Science, University of Waterloo, Waterloo, Canada, 2004 aeehassa@plg.uwaterloo.ca

[2] Benjamin Livshits, Thomas Zimmermann, DynaMine: Finding Common Error Patterns by Mining Software Revision Histories, ESEC-FSE, ACM, September 2005

[3] J. Bevan, E. J. White head, Jr., S. Kim, and M. Godfrey, Facilitating Software Evolution with KENYON, In the proceedings of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 177-186, 2005 Mining Software Repositories, http://msr.uwaterloo.ca/msr2006

[4] D. Cubrani'c, G. C. Murphy, J. Singer, and K. S. Booth. Learning from project history: A case study for software development. In proceedings of the ACM Conference on Computer Supported Cooperative Work, pages 82–91, 2004.

[5] D. Cubrani'c and K. S. Booth. Coordinating open-source software development. In 8th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, Pages 61–65, 1999

[6] Daniel M. German, Davor Cubrani'c, Margaret Anne D. Storey. A Framework for Describing and Understanding Mining Tools in Software Development. This framework msr.uwaterloo.ca/msr2005/papers/36.pdf

[7] FROEHLICH, J., AND DOURISH, P. 2004, Unifying artifacts and activities in a visual tool for distributed software development teams. In the Proceedings of the 26th International Conference on Software Engineering (ICSE, 2004), 387-396

[8] D. M. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using SOFTCHANGE. In the Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE 2004), pages 336–341, 2000

[9] Grigoreta Sofia Cojocar, Gabriela Serban. On Some Criteria for Comparing Aspect Mining Techniques, Workshop 2007 March 12-13, 2007 Vancouver, British Columbia, Canada, ACM

[10] J. C. Grundy. Software architecture modeling, analysis and implementation with SoftArch. In the Proceedings of the 25th Hawaii International Conference on System Sciences, Page 9051, 2001

[11] Proceedings of the 2nd International Workshop on Mining Software Repositories, MSR 2005

[12] K. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer's local interaction history. In proceedings of 1st International Workshop on Mining Software Repositories, 2004

[13] M. A. Storey, D. Cubrani'c, D. M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In the Proceedings of the 2nd ACM Symposium on Software Visualization, 2005

[14] Sunghun Kim, Thomas Zimmermann, Miryung Kim, Ahmed Hassan, Audris Mockus, Tudor Girba, Martin Pinzger, E. James Whitehead, Jr., Andreas Zeller, TA-RE: An Exchange Language for Mining Software Repositories. MSR, May 22-23, 2006, Shanghai, China

[15] Fine-grained processing of CVS archives with APFEL, Thomas Zimmermann, Saarland University, Saarbrücken, OOPSLA Workshop on Eclipse Technology eXchange Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange, Portland, Oregon, Pages: 16–20, 2006, ISBN:1-59593-621-1, ACM Press

[16] http://libresoft.urjc.es/Tools/SoftChange

[17] http://www.cs.ubc.ca/labs/spl/projects/hipikat/documentation.html

[18] A. Walenstein. Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. In the Proceedings of the 11th International Workshop on Program Comprehension (IWPC, 2003), pages 185–195

[19] X. Wu, A. Murray, M.-A. Storey, and R. Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In the proceedings of the 11th Working Conference on Reverse Engineering, pages 90–99, 2004

[20] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, Ophelia Chesley. CHIANTI: A Tool for Change Impact Analysis of Java Programs. In the Proceedings of the 27th International Conference on Software Engineering, 2005

[21] Zhang, Dhaval Sheth. Mining Software Repositories for Model-Driven Development. Yuefeng Motorola, January / February 2006, IEEE SOFTWARE