RMINER: An integrated model for repository mining using Rascal

A feasibility study

Waruzjan Shahbazian

August 31, 2010

Master Software Engineering

Thesis Supervisor : Jurgen Vinju

Internship Supervisor: Jurgen Vinju

Institute: Centrum voor Wiskunde en Informatica

Availability: Public Domain

University of Amsterdam

Abstract

In this thesis the feasibility of an integrated model for repository mining using Rascal is examined. To this end the SCM data sources CVS, SVN and Git are integrated into one integrated repository model named Rminer, which can be used in the data extraction and analysis phases of MSR research.

First the requirements for Rminer are analysed by examining the various MSR research being done and analysing the commonalities and differences between the three SCM systems to be supported. After the design and implementation of the tool, a case study is performed to evaluate the feasibility of an integrated model for repository mining in Rascal.

We found out that while it is possible to create an integrated repository model, by unifying the commonalities and making the differences between the SCM systems explicit, MSR research still needs to be careful when using the unified data in the model, due to the possible differences in the semantics of the SCM systems.

1 Introduction

Mining software repositories (MSR) is the process of analysing the data stored in software repositories, about a software system, usually to improve the future evolution of the software system. This data describes the evolution of a software system and can be mined for different purposes: from modelling the software development process [11] to predicting bugs in the software parts yet to be written [7]. Data sources used in this process include among others: software configuration management (SCM) systems, bug-tracking systems and various communication archives (e.g. mailing lists).

In this thesis the feasibility of an integrated model for repository mining using Rascal, is examined. To this end Rminer is designed and implemented, which facilitates the data extraction and analysis phases of MSR research.

Rminer is a tool, written in Rascal [8] and Java, that facilitates repository mining research by abstracting the various repository data models to an integrated model. This integrated model supports the following SCM systems: CVS[1], SVN[2] and Git[3]. The reasons for the choice of this SCM systems are:

- CVS was one of the most popular SCM systems, thus there are a lot of software projects archived in CVS repositories which can be subject for MSR research.
- SVN has taken over the CVS user base and is the most popular SCM system used.
- Git is the new kid in town and is getting more popular over time.
- By supporting both distributed (Git) and centralized (CVS/SVN) SCM systems, we can address the differences between the two models.

1.1 Motivation

Many MSR projects start their research with the extraction of the needed facts from software repositories between a range of dates or revisions. Depending on the needs of the analysis, the facts extracted in this preprocessing phase differ, from meta-data (e.g. author, commit date) to fine-grained entity information about the source code (e.g. classes, methods, fields). The extracted facts are often stored and used in further analysis.

This, non-trivial, task of extracting and storing of the necessary facts, is often reimplemented for each software evolution research performed. Reimplementing the fact extraction tools costs a lot of time and therefore often a single source of data (e.g. SCM system) is supported. For example, often CVS is supported, since many open source projects use that SCM. This means that the research systems cannot use the evolution data of the software projects archived in other

[1] http://www.nongnu.org/cvs/
[2] http://subversion.apache.org/
[3] http://git-SCM.com/

SCM systems which is a possible threat to the validity of the research results. There might, for example, be some similarities in the evolution of the open source projects archived in CVS, while commercial projects archived on other SCM systems might result in other evaluation results.

Since each research uses its own ad-hoc fact extractor tool and data format, it is difficult to share the research results or even (re)use the raw facts extracted from the software repositories.

To conclude, below are the main motivations for the creation of Rminer:

- Researchers spend a lot of time in the creation of the tools needed for MSR research. Thus not having to create the MSR tools every time, can reduce the startup time of a MSR research a lot.
- Creating ad-hoc MSR tools for every research often means that the data produced or consumed by those tools are not easily interchangeable. So the results of one MSR research can not easily be compared with the results of another or be used for future research.
- Often, due to time constraints, a single source of data (e.g. SCM system) is supported by the adhoc MSR tools, which means that the researchers cannot analyse the evolution data from other data sources. This can be a threat to the validity of the research results, since there is a possibility that a particular data source is often used in a particular way. For example, it is not inconceivable that there are some similarities in the evolution of the projects archived in CVS, while other projects archived in other SCM systems might have a very different evolution history.

1.2 Research question

The main question of this research is:

RQ1 Can we design and implement an integrated repository model using Rascal, that facilitates repository independent MSR research? (Section 7)

To determine the feasibility of such an integrated model, the following sub-questions need to be answered:

RQ1.1 What kind of MSR research is being done? (Section 2.1.5)

RQ1.2 What are the commonalities and differences between the repository models of various SCM systems? (Section 3.3)

RQ1.3 How can the various repository models be integrated into one model, where the commonalities between the individual models are shared and the differences are preserved (Section 4).

In order to support as much MSR research as possible, we need to preserve as much data from the individual models as possible. Therefore, when unifying the commonalities between (some of) the individual models, no data may get lost.

RQ1.4 Is the integrated repository model for Rascal a good solution for repository independent MSR research? (Section 7)

Once Rminer is designed and implemented, we have to determine to what extent it is the solution to the problem we are trying to solve (section 1.1).

1.3 Research method

The research is divided into three phases: requirements analysis, design & implementation and testing.

1.3.1 Requirements analysis

The goal of the first phase of the research is to determine the requirements of Rminer. To this end we will first analyse the MSR domain (section 2.1), then examine the related work (section 2.2) and finally compare the different SCM systems (section 3) to find out the differences and commonalities between the repository models.

Using a literary survey on MSR research and MSR tools that facilitate the research, we can determine what uses cases Rminer should support and if/how the MSR tools facilitate MSR research. RQ1.1 will be answered by categorizing the different MSR projects based on task they are meant to fulfill.

Then, since Rminer is aimed to support different SCM systems, the commonalities and differences of the various repository models will be examined. Since each SCM system uses it's own terminology, a common set of terms will be defined (section 1.4). RQ1.2 will be answered by modelling the commonalities and the differences between the meta-data provided by the SCM systems in a Venn diagram[4] (figuur 4 in Section 3.3).

Having answers to the questions above, the requirements of Rminer can be determined (section 4.1).

1.3.2 Design & Implementation

The third phase of the research is designing and implementing Rminer, according to the requirements (section 4). The datamodel of Rminer will be designed in Rascal, while the underlying communication with the various SCM systems may be implemented in Java. RQ1.3 can be answered with the design of the tool in Rascal (listings 2, 3, 4, 1)

1.3.3 Testing

The last phase of the research is aimed to answer RQ1.4: Is the integrated repository model for Rascal a good solution for repository independent MSR research?. To this end, a case study will be performed to (partially) validate Rminer with the requirements (section 4.1) and evaluate the feasibility of an integrated repository model in Rascal as the solution to the problems formed in Section 1.1. More detailed description of the study can be found in Section 5. The results of the study will be discussed in Section 6 and conclusions will be drawn in Section 7.

[4] http://www.combinatorics.org/Surveys/ds5/VennEJC.html

1.4 Definitions

Atomic commit An atomic commit is the operation in which a set of changes are committed to the SCM system, as a single operation. Furthermore, when during the operation something goes wrong, the system will roll-back an none of the changes will be committed to the SCM system. This way, the SCM system is always left in a consistent state.

Branch Branches are alternative development lines stored on the SCM, used when the development team needs to work on two different copies of a project at the same time.

CVCS Centralized Version Control System is an SCM System that uses the client-server architecture pattern, where all the history of software changes is stored on the server and the clients have to communicate with the server to get the changes.

Changeset A changeset is a group of changes, made to the software stored on a SCM system, that may or may not be committed together in an atomic way.

Checking out Checking out is the process of getting the software stored on the SCM system, as it was at a given time (revision).

Committing Committing is the process of storing changes to the software from the working copy to the SCM system.

DVCS Distributed Version Control System or a Decentralized Version Control System is an SCM System. that uses a peer-to-peer approach, where all the clients contain all the history of the software and can push and pull changes to each others repositories.

Merging Merging is the process of integrating the changes from one branch into another.

MSR Mining Software Repositories is the process of analysing the data from software repositories.

MSR script MSR script is software that can be used to mine a repository.

Pulling changeset Pulling changesets is the process of getting the changes from a remote repository into the local repository.

Pushing changeset Pushing changesets is the process of committing the changes from the local repository into a remote repository.

Revision Revision identifies a version of a data stored on a SCM system.

SCM Software Configuration Management is the process of controlling changes to software.

SCM System SCM System, Software Repository or a Version Control System, is a tool/system that controls changes to software.

Tag Tags are short-cuts to a state in the repository.

Working copy contains a local copy of the software stored on the SCM system, at a given time.

2 Mining Software Repositories

MSR research analyses the data available in software repositories to uncover interesting information about the (evolution of the) software systems archived on the repositories. This information is usually used to improve the future evolution of the software.

As mentioned in the introduction, MSR research can have many different purposes. The first step in the requirements analysis of RMINER is to find out what kind of MSR research is being done and with what purpose. Once we know this, we can look at the tools that facilitate MSR research. By analysing the strong points and shortcomings of those tools, the kind of MSR research being done and the purposes of those MSR research, we can determine the requirements of RMINER.

2.1 Taxonomy

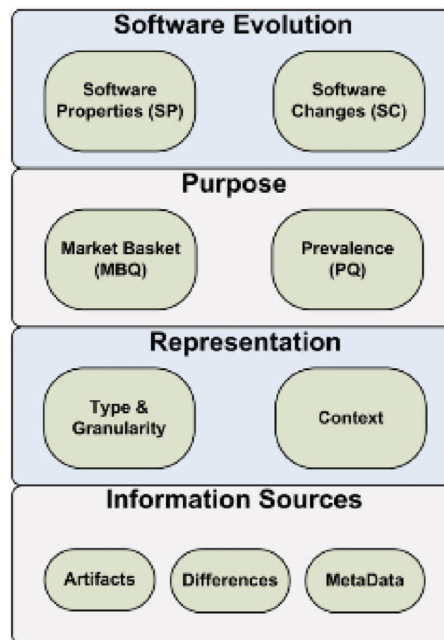

Figure 1: Four-layer taxonomy of MSR approaches [6]

Kagdi [6] has devised a four-layer taxonomy (figure 1) based on literature survey on approaches for mining software repositories. The MSR approaches studying high-level properties of a system are separated from the approaches studying more detailed changes

in the actual artefacts. Furthermore, the approaches can have different purposes, consider different representations of the information and use different information sources.

In the taxonomy, approaches studying "Software Changes" are aimed at the more fine-grained changes in the artefacts, while the approaches studying "Software Properties" are aimed at the properties of the software stored on the SCM systems. The survey has found 11 MSR approaches studying the artefacts, 14 MSR approaches studying the properties and 23 MSR approaches studying both aspects of software projects.

MSR Tools (related work, Section 2.2) is compared and the requirements of RMINER (section 4.1) are defined according to the taxonomy described above.

2.1.1 Software Evolution

The goal of MSR is to learn more about the evolution of a software project. Thus a MSR research can be interested in the change characteristics of the high-level software properties of the software system:

> "For example, metrics for software complexity, defect density, or maintainability can be computed for two versions of a system taken from CVS and the quality of the evolved system assessed. In this approach the interest is in the changes of high-level or global properties of a software system under evolution. We refer to this group of MSR investigations as interested in changes to properties." *[6, page 84]*

An MSR research can also be aimed at the more detailed changes in the actual artefacts of the software system:

> "The second perspective represents investigations that study the actual mechanisms or facts that take a software system from one version to the next. Here the focus is on the specific differences between versions. These types of approaches use the difference data supplied by CVS or other tools. We refer to this group of MSR investigations as interested in changes to artefacts. There can be a significant level of variation with respect to the granularity and type of source code change. Types of source code entities include physical, syntactic, documentary, etc. Likewise, one can examine changes to a file, class, or function. These differences are reflected in how sophisticated the tools used in the investigation are with respect to such things as programming-language knowledge." *[6, page 84]*

2.1.2 Purpose

Changes in the properties or artefacts of a software system can provide answers to different types of questions. The taxonomy divides those questions into two classes: "market basket questions" and "prevalence type questions". Explanation of the two classes by Kagdi:

> "The first is the market-basket question (MBQ) formulated as: if A occurs then what else occurs on a regular basis? The answer is a set of rules or guidelines describing situations of trends or relationships. For example, if A occurs then B and C happen X amount of the time.
>
> 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'." *[6, page 82]*

> "The second type of MSR purpose relates to prevalence questions (PQ). Instances include metric and boolean queries. For example, was a particular function added/deleted/modified? Or how many and

> which of the functions are reused? The questions asked indicate the purpose of the mining approach.” *[6, page 82]*

2.1.3 Representation

The data mines by the MSR research can be of different types and granularity. The third layer of the taxonomy deals with the representation of the data mined by the MSR research.

> “Layer 3 (representation) refers to the type (e.g., physical), granularity (e.g., system, files, classes), and expression of the artefacts and their differences. Source code repositories are typically limited to the physical-level representation of source code (i.e., file and line numbers). As such, the answers to MSR questions can be further extended to more fine-grain representations of artefacts and differences. Therefore, the representation of the information in the repositories can be refined based on the syntax and semantics of the underlying programming language(s).” *[6, page 85]*

2.1.4 Information Sources

The main information sources used by MSR research can be found in the fourth level of the taxonomy. Explanation about the choice of the information sources to include in the taxonomy, by Kagdi:

> “However, information sources depicting high-level abstractions such as design models and architecture models are typically not directly available in the software repositories. They may need to be reverse engineered or computed to support the corresponding MSR questions. Therefore, layer 4 represents the information sources that are readily available in the software repositories and those that need to be made available to support the MSR investigation.” *[6, page 85]*

2.1.5 Research Question

RQ1.1 What kind of MSR research is being done?

Kagdi[6] has devised a four-layer taxonomy (figure 1 in Section 2.1) based on literature survey on approaches for mining software repositories.

48 MSR projects are studied, and divided into categories based on the evolution task being accomplished [6, table IV, page 91]:

- Evolution coupling/patterns
- Change classification/representation
- Change comprehension
- Defect classification and analysis
- Source code differencing
- Origin analysis and refactoring
- Software reuse
- Development process and communication
- Contribution analysis
- Evolution metrics

2.2 Tools

In this section we will look at the tools that facilitate MSR research. While we are more interested in the tools that can be used for different kinds of MSR research, we will also discuss some of the tools that have a specific goal or are made for a specific MSR research. The tools described in this section are related to RMINER, so we will compare the tools with RMINER, to determine what the differences are and why we need RMINER to solve the problem described in Section 1.1.

While there are many ad-hoc[5] implementations of repository mining tools, only a few are generic enough to be used by different researchers with different interests. SoftChange [5], HipiCat [3], APFEL [13], Minero [10] and Bloof [4] are all MSR research tools with only SCM support of CVS, while the first two tools also support non-SCM (e.g. bug-tracker) resources for their analysis.

Kenyon is [1] is a tool that facilitates software evolution research by providing access to various SCM systems through one interface. Also, the data extracted from the SCM systems can be saved and reused for analysis purposes. None of the other tools mentioned above gives the user access to the extracted raw facts so that different kinds of analysis can be performed.

One commonality between the tools mentioned above is the fact that all of them use a database as a backend. Also, most of the tools are aimed at the content of the revisions of the files stored on the SCM. It is often not possible to gather the meta-information of the revisions and only request the actual content of the files if necessary.

2.3 Kenyon

Kenyon is a tool that does a similar job as RMINER. The main differences between Kenyon and RMINER are described bellow.

2.3.1 No option for only meta-information

Kenyon is aimed at the fact extraction from the files checked out from SCM systems and provides some of the meta information about the revisions checked out. Kenyon has no option to retrieve only the meta information and do the checkout later on if necessary. This means that even if the evolution research does not need the actual contents of the files, they will have to be checked out, which can unnecessary take lots of time on large projects with many revisions, especially if the SCM is not locally available.

[5] Ad-hoc repository mining tools are designed for, and used by, one specific MSR research

2.3.2 Too few meta-information

Kenyon provides too few of the meta information stored on the SCM, which makes it impossible to perform evolution research that use the information not accessible through Kenyon. The following is a list of meta information that is not accessible through Kenyon:

- A list of affected resources and their status (modified/added/removed/copied/renamed) per changeset
- Origin (file name + revision) of the resources that are renamed or copied
- Tags information
- Merging information
- Number of lines added/removed
- Access to the history (meta-information of previous revisions) of a particular resource

Some of the items above are not (easily) accessible from some of the SCM systems investigated in this thesis, but most of them are. Although Kenyon is an uniform tool that supports multiple SCM systems, the SCM specific meta information should also be accessible when that specific SCM is used.

2.3.3 Fixed transaction recovery algorithm

Kenyon uses the sliding-window transaction recovery algorithm to regroup the files on CVS into a changeset to checkout. The variables used in this process of transaction recovery (e.g. maximum time between start and end of a transaction) cannot be changed by the user and therefore can be wrong for some kind of projects (where for example the latency is very high and many files are committed together). Since the user cannot directly call the checkout process, he cannot create his own changesets, with another algorithm, and perform a checkout.

2.3.4 Library dependencies

Although there is a “nodb” configuration mode in Kenyon which is used when no database back-end is needed, Kenyon requires all the library dependencies required by the ORM library (Hibernate) used. This means that even if the user does not want to store the data on a database, more than fifteen dependencies have to be kept up-to-date. The last version of the project has been released on 2005-04-18 and the website is taken off-line in 2007. Kenyon does not work with the new versions of Hibernate and RDBMS systems.

2.3.5 GraphSchema

Often research results (facts) need to be saved and reused later on. Kenyon facilitates fine-grained facts persistence by providing a mechanism of GraphSchema and ConfigGraph that holds to the GraphSchema. This mechanism makes it possible to store, reuse and compare fine-grained analysis results inbetween revisions and resources. For example, a “C” GraphSchema could describe the structure of a “C” program, while an instance of a ConfigGraph could hold the actual nodes and edges parsed from a given “C” program. By storing the ConfigGraph instances in a database, one can compare the edges and the nodes of a “C” program at various revisions.

2.4 APFEL

APFEL is an eclipse plugin that collects fine-grained changes (Java) from version archives (CVS) in a database. By using the Eclipse infrastructure (CVS plugin and JDT parser), APFEL parses the various versions of the Java software projects, archived in CVS, and stores the extracted fine-grained changes in an database. Using a relational databse makes it easy for the user to find and use the necessary facts from the database and perform evolution research.

2.5 Gitdm

Gitdm is an ad-hoc repository mining tool used in the research on the Linux Kernel Development [9]. Gitdm is aimed at the mining of Git repositories and makes heavy use of unix command line tools. The fact extraction and analysis code are not always separated, which makes it hard to reuse Gitdm for different purposes, without having to rewrite it altogether.

The ad-hocness of Gitdm has however its advantages. Since the various processes are integrated with each other, Gitdm can be very efficient. It can for example determine which facts have to be extracted to meet the requirements of the analysis, so that no unnecessary data has to be handled.

2.6 Conclusion

The related work is compared (table 1) with Rminer according to the information sources of the taxonomy from the introduction (figure 1) and multi SCM support.

The results show that all the tools provide the user with the Artifacts stored on the SCM systems. The meta-data provided by Kenyon is however limited and one can get more meta-data with Gitdm by scripting the parser, while Rminer provides much more meta-data out of the box.

File differences are only supported by Gitdm, since Gitdm is the only tool that communicates directly with the repository. That makes it possible to fetch file differences, while Kenyon and Rminer do not have this feature. In order to achief that in those tools, one has to checkout both the versions of a file and do a diff comparison manually.

Regarding “Multi SCM Support”, both Kenyon and Rminer support multiple SCM systems. Please do note that all the SCM systems supported by Kenyon have a“Client-server” model (CVCS), while Rminer also has support for a “Distributed” (DVCS) SCM system: Git.

	Kagni Information Sources			Multi SCM Support
	Artifacts	**Differences**	**Metadata**	**SCM support**
Kenyon	OTB	NA	OTB	CVS,SVN & ClearCase
Gitdm	S	S	S	Git
Rminer	OTB	NA	OTB+	CVS, SVN & Git

Table 1: Related work comparison: OTB = Out of the box, S = Scriptable, NA = Not Available

3 SCM Systems

The second part of the requirements analysis is aimed at the commonalities and differences between the various SCM systems and their datamodels. This way we can determine which data the integrated repository model has to support.

Software configuration management systems track and control changes to the documents, software source code and other data stored on the system. Besides the actual content of the files, SCM systems keep track of various kinds of meta-information. This information can vary from the author name to the sha1 hash[6] of the content of a particular version of a file or the origin of a copy operation.

In order to design a common meta model that supports multiple SCM systems, the similarities and differences between the SCM systems are examined. Please note that only the SCM concepts important for MSR research are presented in this chapter.

3.1 Similarities

Most of the SCM systems share some common concepts, which are presented in Figure 2. In an UML class diagram the components mostly involved in the process of MSR research are presented.

[6] http://http://book.git-SCM.com/1_the_git_object_model.html

3.1.1 Revision

Each resource stored on the SCM system is being version controlled. This means that each version of a resource can be identified and requested, the identification of a version of a resource is done with a revision.

Please note that a revision can be used as an identifier of a changeset as well as of a single resource in a changeset (more on revision identifiers in the section Differences).

Revisions can also have references to one or more parents. The definition of a parent differs when used on a changeset or a resource:

- changeset revision parents are the revisions of the changesets, this changeset is based on. In case of a merge operation, a changeset can have multiple parents.
- resource revision parents refer to the revision of the previous version of the resource. In case of a move/copy operation, the parent refers to the revision of the origin this resource is moved/copied from. In case of modification, the parent refers to the revision of the previous version of the resource. In case of a new file (add operation) or a removal of a file (delete operation) no parent revision exist.

3.1.2 Info

Info contains information provided by the author during a commit. Commit is the process of submitting the changes of the resources to the SCM system.

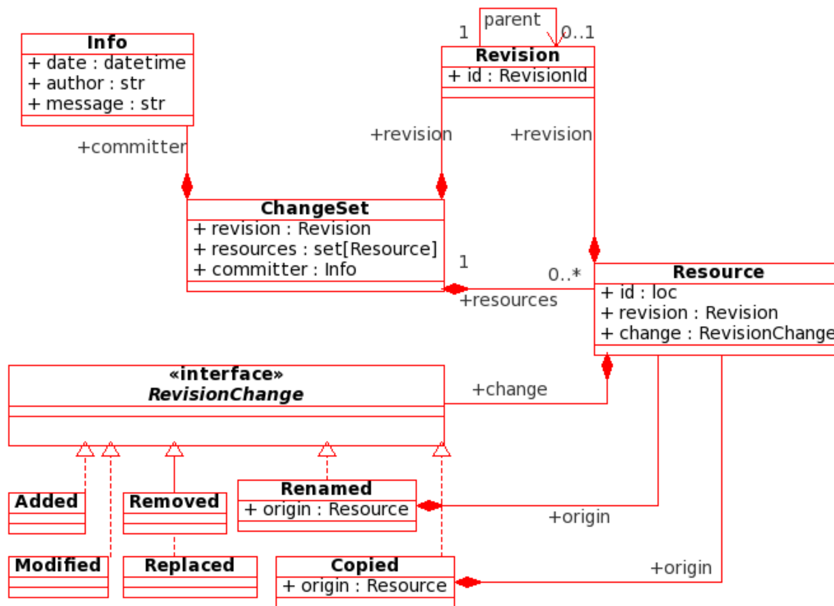

Figure 2: UML2 class diagram of common concepts in SCM systems

Most of the SCM systems keep track of the date of the commit, the name of the committer and a commit message describing the commit.

3.1.3 Changeset

Changeset identifies the set of changes made in a single commit operation. As described earlier, a changeset uses a revision as identifier. Please do note that not all the SCM systems investigated support changesets (see Section 3.2.1), but those that do, have a similar concept of changesets.

3.1.4 RevisionChange

RevisionChange represents the kind of the change performed in a particular revision to a single resource. Mostly the following kinds of changes are stored:

- "added" when this is the first revision of a resource
- "removed" when this is the last revision of a resource
- "renamed" when the resource is renamed (moved) from an old location. In this case the old location is stored in the "origin" field
- "copied" when the resource is copied from an old location. In this case the old location is stored in the "origin" field
- "replaced" when a new resource is added in place of the old resource within the same changeset. In that case it's not a modification because the new file on the same location has no relation with the old file (except the filename)
- "modified" when the content of the resource is modified

3.1.5 Resource

A resource is a file or a folder that is under version control on the SCM system. Some SCM systems (e.g.

CVS, Git) only keep track of the revisions of files, while others (e.g. SVN) also store the revisions of folders.

3.2 Differences

There are also many differences in the semantics and the kind of meta-information that can be tracked with the various SCM systems. This makes it hard for researchers to do a SCM system independent evolution research. In this section the differences between the three SCM systems, related to MSR research, will be described.

3.2.1 Atomic operations

A system supports atomic operations if it is left in a consistent state, even if an operation is interrupted. The commit operation of an SCM system is atomic when no single change, from the set of changes (changeset) of the commit, makes to the system, unless all of the changes are committed succesfully.

Git and SVN support atomic operations, while CVS does not. Since most of the MSR research analyse the various versions of the system, instead of individual files, support for atomic operation is important. To overcome this shortcoming of CVS, several algorithms [12] (e.g. sliding-window "transaction recovery") are invented to recover the transaction information not recorded by CVS.

3.2.2 Revision identification

All the three SCM systems use different kinds of revision identifiers.

CVS does not have any changesets, since no atomic operations are supported.

> "Each version of a file has a unique revision number. Revision numbers look like '1.1', '1.2', '1.3.2.2' or even '1.3.2.2.4.5'. A revision number always has an even number of period-separated decimal integers. By default revision 1.1 is the first revision of a file. Each successive revision is given a new number by increasing the rightmost number by one. It is also possible to end up with numbers containing more than one period, for example '1.3.2.2'. Such revisions represent revisions on branches." [7]

Since each file has it's own unique set of revision numbers and no changesets are tracked, it's not possible to compare revision numbers and for example determine if a file "A.txt" with revision number "1.4" is older then the file "B.txt" with revision number 1.3.

SVN has the "easiest" revision identification system of the three SCM systems investigated. Each changeset is identified with an unique natural number, one greater then the number of the previous revision, started with 0. Each revision number represents the state at which the filesystem was at the moment that the changeset was committed. Due to the incremental numbers of the revisions, its easy to see which revision comes next and how many revision are in total. Please note that since the revision numbers are global in the whole repository, they also get incremented when a commit in a branch occures. This means that the last revision number, does not represent the amount of changesets in a specific branch, but the whole repository.

> "When a Subversion user talks about revision 5 of foo.c, they really mean foo.c as it appears in revision 5." [2]

GIT uses sha1 hashcodes as identifiers for the revisions of changesets and individual files. Each changeset in Git has, similar to SVN, an unique identifier.

[7] http://www.thathost.com/wincvs-howto/cvsdoc/cvs_4.html

[2] http://svnbook.red-bean.com/en/1.0/ch02s03.html#svn-ch-2-dia-7

The difference is that Git uses hashcodes that describe a particular commit, which makes it impossible to determine if some revision is older then another one.

The individual resources in a changeset have their own sha1 hashcodes, which, in contrast to SVN, have no relation to the revision number of the changeset they are in to. Git computes the sha1 hashcode of the content (blob) of a file and uses it as the revision identifier of that version of the file.

> "A blob generally stores the contents of a file.
>
> 5b1d3..
>
> blob | size
>
> #ifndef REVISION_H
> #define REVISION_H
>
> #include "parse-options.h"
>
> #define SEEN (1u<<0)
> #define UNINTERESTING (1u
> #define TREESAME (1u<<2)
>
> The "commit" object links a physical state of a tree with a description of how we got there and why.
>
> ae668..
>
> commit | size
>
> tree | c4ec5
> parent | a149e
> author | Scott
> committer | Scott
>
> my commit message goes here and it is really, really cool" *Git Book* [2]

3.2.3 Tags, Branches and Merges

All the three SCM systems have support for tags, branches and merging between the branches. There are however some differences between the SCM systems, in the way tags/branches are represented and the amount of information kept about the merges.

[2] http://book.git-SCM.com/1_the_git_object_model.html

CVS has two kinds of tags: branches and "release tags". Each revision of a file is associated with zero or more tags and multiple files with different revisions can have the same tags associated to them. This makes it possible to "save" the state of the repository, current versions of all the current files, and refer to it with the tag.

With a branch tag its possible to commit changes, while this commits will still be "tagged" with the name of the branch. This makes it possible to have multiple versions of a file, in the same branch.

While it is possible to get branching information per file in the history of a project, CVS does not keep track of the merges between the branches.

SVN has an unique view on branches and tags, different from both CVS and GIT. To create a new branch or a tag (in SVN there is no difference between those two concepts), one has to perform a copy operation. Since SVN tracks the copy and move actions, the copied branch or tag can be compared and merged with the other branch, on the original location. It is also very easy to determine which branch a commit has affected, since the pathname of the files changed contain the branchname.

Since SVN 1.5, it is possible to track the merges performed on the repository. This means that MSR tools can, unlike in CVS and like Git, know when and which revisions where merged.

Git has a very similar, to CVS, notion of tags and branches. There are however some differences, mainly due to the implementation of branches, that are important to know when it comes to repository mining. Figure 3 shows a scenario where two branches are used in parallel and are merged two times. Each command executed on the git repository is shown in Table 2, in the reverse order. The first two columns show which commands are executed, while the rest of the columns show the state of the repository after the command is executed. Since Git does not keep track of branches a commit was made one, it's not always possible to determine on which branch

Time	Action	Commit	Parent 1	Parent 2	Branch
9	checkout master, merge feature: Fast-forward	h	g		master
8	commit h	h	g		feature
7	checkout feature, commit (merge): g	g	e	f	h → feature h → master
6	checkout master, commit f	f	d		g → master
5	checkout feature, commit e	e	b		g → feature
4	commit (merge): d	d	c	b	f → master
3	checkout master, commit c	c	a		d → master
2	checkout feature, commit b	b	a		e → feature
1	checkout master, commit a	a			c → master b → feature

Table 2: Git commits separated after the merges. Branch detection based on the parent relation (→)

	Changeset revision	Resource revision	Tags	Branches	Merges
CVS	NA	Numbers (e.g. 1.4)	Revision based	Revision based	Not traceable
SVN	Natural numbers (e.g. 13)	Same as the Changeset	Path based	Path based	Traceable
Git	Sha1 hash of the changeset	Sha1 hash of the content	Revision based	Revision based	Good traceable

Table 3: Differences between the implementation of the common concepts by the SCM systems

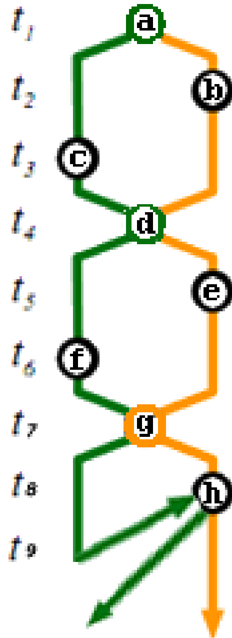

Figure 3: Git commits [2]

a commit was made on [2]. More on this in the following subsections.

Merges in Git are represented as commits with two or more parents. This makes it very easy to track which commits (the parents) and what resources (conflict resolutions) are merged.

Branch line detection in Git is much harder then in CVS. Git, unlike CVS, does not store the branch information as part of the history of a file and it's not always possible to tell which branch a commit was made on. For example, a merge of two commits will be described by the "git log" as an "commit sha" with two "parent sha"'s. The order of this parent sha's depend on the order of the arguments provided to the "git merge" command. This makes it hard to follow the history of a branch back (from the HEAD) to the ancestor commits to know which commit is made on which branch. Information about the order of the parents is not necessary lost, since Git does keep a private "log" file of the interactions

with the repository. However, this file can (accidentally) be cleaned by running the "gc –aggresive" command. Also, when initializing a git repository, often by cloning from some other repository, the content of this (private log) file will not get copied.

Fast-forward merges (t9 in Table 2) can happen when only the second branch (with which the merge will happen) contains changes (and the current branch has none). In that case, Git redirects the HEAD tag of the current branch, to the HEAD of the second branch. In the example git makes look like the commit "h" (and it's parent "g") where directly made on the master branch commit "f", while before the merge the HEAD of the master branch was "f" and "g" was never committed on the master branch. It is therefore important to keep in mind that this kind of merges don't create a "merge" commit and can be a threat to the validity of a research which uses that information.

Pulling commits from remote repositories can be very hard to track. It is for example hard to tell if a branch pulled from repository A is originated in A or is itself pulled from a repository B. Analysis that use information about pulls between repositories should be performed very carefully, because of this issue.

3.3 Conclusion

In conclusion, to answer RQ1.2, the three SCM systems investigated have the following similarities:

- Revision identifiers are used to identify the unique resources stored on the SCM system
- Tags and Branches are supported
- Committer information (e.g. name, date) per revision is saved
- Changesets, if available, contain the set of changes in a commit
- RevisionChange represents the kind of a change performed in a particular revision to a single resource
- Resources are physycal files (section 2.1.3) or folders archived on the SCM system. Please note that SVN is the only SCM system, from the three examined, that handles folders as a resource.

Differences between the SCM systems are shown in Table 3, where the following concepts are compared:

- Changeset revision: how the changesets are identified
- Resource revision: how a version of a resource is identified
- Tags: how the tags are supported
- Branches: how the branches are supported
- Merges: if the merges are traceable and how much information is available (e.g. which changesets or files are merged)

While in the discussion above all the important commonalities and differences between the repository models are mentioned, Figure 4 shows only the *kinds* of meta-data provided by the SCM systems (and not the specific representation differences). For example, while the three SCM system use different kind of revision identifiers (e.g. numbers, sha1 hash), Figure 4 just shows "Revision" as a meta-data which is provided by all the SCM systems. The same holds true for the "branches" and "tags": those information can be obtained from all the three SCM systems, but they all differ from each other in the way "branches" and "tags" are supported (e.g. naming conventions in SVN vs native support in CVS).

In addition to the differences mentioned in this section, various differences in the semantics of the repository models might exist. See evaluation (section 5) for more info.

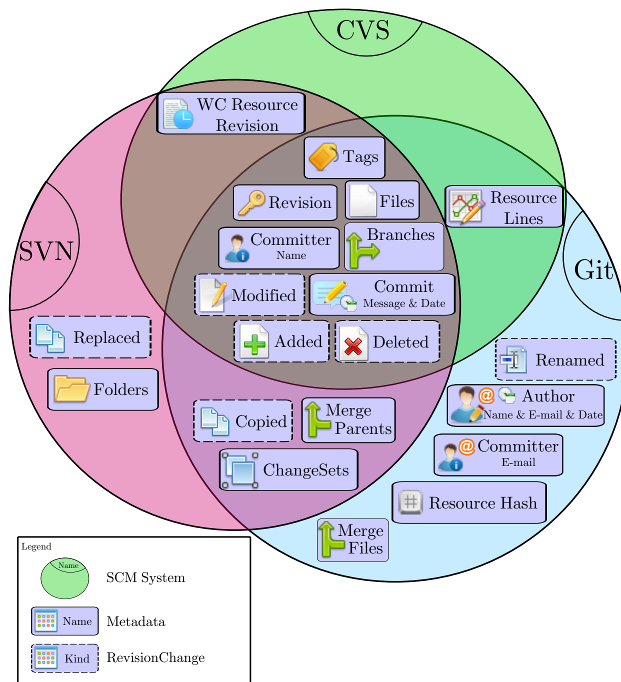

Figure 4: Venn diagram of the kinds of meta-data provided by the SCM systems

4 Rminer

> "Rascal is a domain-specific language that takes away most of this boilerplate by integrating source code analysis and manipulation at the conceptual, syntactic, semantic and technical level." *[8]*

As stated above, Rascal can be used to analyse source code. Integrating the Rascal environment with SCM systems would make it possible to perform source code analysis on multiple versions of the source code and analyse the differences.

The rich set of datastructures and collection manipulation mechanisms (e.g. list comprehension, visitors) in Rascal should make it possible to write compact MSR scripts and perform fine-grained syntactic source code analysis as well.

Rminer is designed in Rascal and is the main tool this research is about. Before performing a feasibility study, we will discuss the design of the tool and answer RQ1.3 mentioned in Section 1.2.

4.1 Requirements

Many software evolution research start with the extraction of facts from the software repositories, before doing some kind of analysis. Once the facts are extracted, various analysis (e.g. metrics) can be computed, sometimes without the need to communicate with the software repositories. In this section the requirements are described that are aimed at this process of MSR projects.

By studying the MSR tools (related work) (section 2.2) and various SCM systems (section 3), the following set of requirements are determined that should be met by the meta model and the tool supporting it.

4.1.1 Support for multiple SCM systems

The tool should support different Software Configuration Management systems in a generic way. That means that a MSR script can be written, that works regardless of the SCM system being used.

Motivation for this requirement is the main motivation of this project and is described in section 1.1.

4.1.2 Partially history

It is not always necessary to process the whole revision history (e.g. we are only interested in the changes performed since last year) of a software project. It can also be the case that only some directories in the repository need to be mined. For those cases, it should be possible to configure the tool and specify history limitation.

4.1.3 Common Meta-data

The meta-data about the revisions of resources stored on the SCM are an important source of data for "Software Properties" [6, page 85] research projects. The following meta-data must be supported:

- Committer information
- Commit date
- Commit message
- List of changed resources (changeset) and the kind of the change (modified, added, removed, etc)

Motivation for this requirement is the amount of MSR projects using meta-data, listed in [6, table II]

4.1.4 SCM specific Meta-data

The meta-data that is not listed in the previous requirement, but are available by the SCM system being used, must be accessible too. The following SCM system specific meta-data should be available in the integrated meta model.

- Tags/branches information per revision (Git/CVS)
- Merging information (Git/SVN)
- Copy/move/rename origin filename + revision (Git/SVN)
- Copy/move/rename origin percentage (Git) Git
- Author information (Git)
- File and changeset sha1 hashes (Git)
- Number of lines added/removed at each revision (Git/CVS)

Motivation for this requirement is to support as much MSR research as possible. Even though scripts using this feature would not work when used on repository data from SCM systems not providing the used meta-data, it can still be useful to use it on the SCM systems that do provide the meta-data. Also, it is not inconceivable that the SCM systems can compensate the lack of some meta-data with the presence of one ore more other meta-data and get the same results.

4.1.5 File based history access

When performing an analysis on a particular file, it should be possible to have access to the meta information about the previous versions of the file.

Motivation for this requirement is to provide more flexibility in the kinds of "Software Properties" [6, table II] research projects that is supported.

4.1.6 Reuse

It should be possible to exchange the meta information extracted and use it for further analysis without having access to the SCM system used to extract the data (assuming no file contents are needed).

Once the facts are extracted, it should be possible to exchange them with other research so they don't have to extract the facts themself. This can be very handy when "closed source" software repositories aren't publicly accessible, but the copyright owners do want to provide (anonymized) facts extracted from the software repositories for analysis purposes.

4.1.7 File management

Checkout It should be possible to checkout a state of the repository based on the "Revision", "Commit date" or a "Tag" to the working copy.

List checkout files Support for listing of all the files (and their revision information, if available [8]) in the current working copy;

4.2 Design

RMINER facilitates MSR research by providing an API consisted of an integrated repository model and a way to communicate with the SCM systems. By writing Rascal scripts that utilize the RMINER API, we can do MSR research.

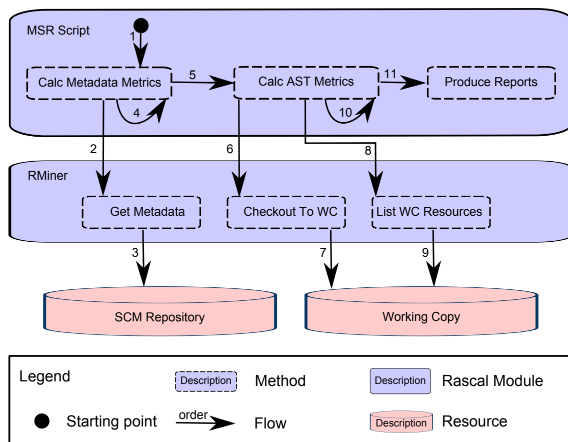

Figure 5: High-level control flow architecture of a MSR script using RMINER. The numbers on the arrows indicate the processing order.

Figure 5 shows the high level control flow of a MSR script using RMINER. First, metrics are calculated (step 4) using the meta-data from the repository (step 3). Steps 5 to 10 are used to fetch the actual resources from the repository and calculate metrics based on the content of this resources.

Figure 6 shows the various components of RMINER and their size in lines of code. Please note that the Java component of RMINER doesn't show all source files, due to lack of space. Scm.rsc is the unified repository model, while CVS.rsc, SVN.rsc and Git.rsc

[8]revision information of the files in the working copy is not available in GIT

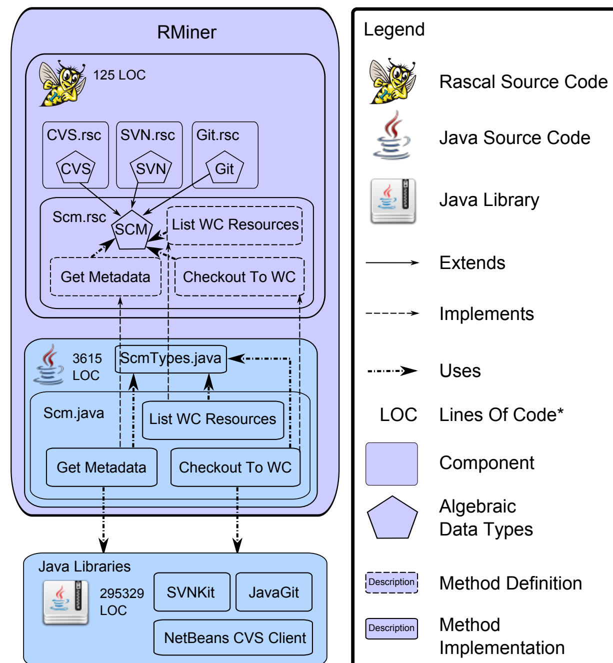

Figure 6: RMINER components. *LOC is the amount of lines (including empty and documentation) of the files with the following extension: .rsc and .java

are extensions on the model and contain repository specific meta-data. Algebraic Data Types (ADT) in Rascal are used to define the repository data model (see for more section 4.2.2).

4.2.1 Raw facts

By providing the raw evolution data extracted from the SCM systems, RMINER tries to be as broad as possible regarding the kinds of MSR research supported (section 4.1.4). This means that RMINER will not modify or extend the evolution data, to unify the repository model.

For example: since CVS does not support atomic changesets, we could try to recover the changesets and provide the extracted data as a changeset to the user. However, in this process we might lose data that is important to the MSR research (in the exam-

ple the datetimes of the individual commits will get lost, since a changeset has one common datetime). Instead, RMINER will just return the data from the CVS repository in a structure that does not require any modification to the original data.

4.2.2 Integrated model

RMINER tries to unify the commonalities between the various repository models, so the same data should be accessed the same way, regardless of the SCM system being mined. For example:

SVN and Git both track the changesets that are merged. While both SCM systems contain the revision's of the merged changesets (parents), Git additionally keeps track of the resources being merged. Below the implementation of this variation point with Rascal ADT's.:

```
1 data MergeDetail = mergeParent(Revision
      parent);
```

SVN.rsc module defines the data type called "MergeDetail", which can be constructed with a constructor called "mergeParent". This constructor requires one field, "parent" of type "Revision", thus a MergeDetail object created with this constructor will only contain the parent information.

```
1 data MergeDetail = mergeResources(Revision
      parent, rel[Resource resource,
      RevisionChange change] resources);
```

Git.rsc however, defines the same data type with a constructor that has an extra field "resources", in addition to the "parent" field.

Defining the same data type multiple times, as we had done above, results in a merge of the constructors, so that the different constructors will actually construct an object of the same type:

```
1 data MergeDetail = mergeParent(Revision
      parent) | mergeResources(Revision parent,
       rel[Resource resource, RevisionChange
      change] resources);
```

Having variation points in the data model, allows us to write MSR scripts that use the fixed field regardless of the existence of the variable fields. So, in our example, the merge information provided by the two SCM systems are not the same, but if a researcher is only interested in the part of the merge information that is supported by both system (the parent field), then an "SVN/Git" independent script can be written.

```
1 MergeDetail merge = ....;
2 print(merge.parent);
```

The "...." above can be replaced with a constructor call "mergeParent(..)" or "mergeResources(.., ..)".

4.2.3 Abstraction

RMINER abstracts the differences between the repository models at the levels that are common between the models. This way, MSR research scripts, that only deal with the data at a certain abstraction level, can be more repository independent. For example:

Revision information is accessible through multiple abstraction layers.

```
1 data Revision = revision(RevisionId id) |
      revision(RevisionId id, Revision parent);
```

```
1 data RevisionId = number(str number); //CVS
2 data RevisionId = id(int id);          //SVN
3 data RevisionId = hash(Sha sha);       //Git
```

```
1 data Sha = blob(str sha) | commit(str sha);
```

Depending on the needs of the MSR research, scripts can use the appropriate level of abstraction. When the Revision data type is used, the script is repository independent, while checking on the actual implementation of the RevisionId (e.g. number or hash) would make the script less repository independent.

Please note that the three definitions of "RevisionId" type above are actually divided into multiple Rascal modules (listings 1, 2, 3 and 4). This way, each module makes the differences with the unified model,

which defines the commonalities between the various repository models, explicit.

4.2.4 Annotations

In addition to the ADT constructors, annotations can be used to add more variation points to the data types. Below an example of an annotation:

```
1 data Revision = revision(RevisionId id) |
      revision(RevisionId id, Revision parent);
2 anno list[MergeDetail] Revision@mergeDetails;
```

The annotation called "mergeDetails" adds an extra variation point to the Revision data structure. The same behaviour could have been achieved by adding a new constructor with an extra field "mergeDetails" of type "list[MergeDetail]". However, since we already have two constructors this would mean that two additional constructors are needed (each with the old fields an the new "mergeDetails" field).

The advantage of an annotation is that it affects all the constructors of the type being annotated. So, regardless of the constructor used to create a Revision object we can annotate it with mergedetails.

4.3 Recovery

As stated earlier, RMINER's datamodel represents the unmodified data from the various repositories. However, MSR research might require data that is not directly provided by the SCM system used. So, in order to do such MSR research we need to recover the missing information.

Changeset recovery is a good example where various algorithms[12] can be used to recover the changesets in CVS.

4.4 Enhancements

Besides the data recovery, many enhancements can be made on the raw data extracted from the repository. This enhancements can vary from data/noise cleaning to more detailed data correction. Data enhancements are important part of MSR research, because omitting them can result in erroneous results.

Data correction can include the detection of "renames" from the "copies". SVN tracks the copy operations on files, but does not distinguish between renames and actual copies where the old file is removed. In order to do a correct MSR research where rename operations are tracked, one has to enhance the data and distinguish "renames" from "copies".

Data/noise cleaning is another enhancement point which can be used to detect and remove large transactions, which often are result of infrastructure changes, and merge changesets. Merge changesets are often irrelevant for MSR research, because the changes they contain are already processed and counting those changes twice can influence the research results.

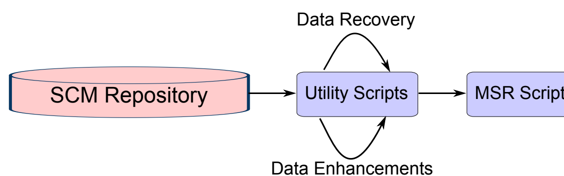

Figure 7: Data recovery and enhancement. Arrow indicates the flow of the data while the blue boxes are Rascal scripts.

4.5 Research question

RQ1.3 How can the various repository models be integrated into one model, where the commonalities between the individual models are shared and the differences are preserved.

The answer to this question is provided by the RMINER API in the code listings 1, 2, 3 and 4. Most of the data types of the integrated model can directly be mapped back to the venn diagram (figure 4 in Section 3.3). Therefore, the model is an explicit documentation of RMINER and the repository models unified: it documents which types of meta-data is supported and what the differences are between the SCM systems.

Listing 1: Rminer unified model (Scm.rsc)

```
1 module experiments::scm::Scm
2
3 data Project = project(Repository configuration, list[ChangeSet] changesets);
4
5 data Repository;
6 data Connection = fs(str url);
7 data LogOption = startUnit(CheckoutUnit unit) | endUnit(CheckoutUnit unit);
8
9 data ChangeSet;
10 data RevisionChange = added(Revision revision) | modified(Revision revision) | removed(Revision
     revision);
11 data Revision = revision(RevisionId id) | revision(RevisionId id, Revision parent);
12 data RevisionId;
13 data Info = none(datetime date) | author(datetime date, str name) | message(datetime date, str
    message) | message(datetime date, str name, str message);
14 data Resource = file(loc id) | folder(loc id) | folder(loc id, set[Resource] resources);
15 data WcResource;
16
17 data CheckoutUnit;
18 data Tag = label(str name) | branch(str name);
19
20 anno set[Tag] Revision@tags;
21
22 @javaClass{org.rascalmpl.library.experiments.scm.Scm}
23 public list[ChangeSet] java getChangesets(Repository repository);
24 @javaClass{org.rascalmpl.library.experiments.scm.Scm}
25 public void java getChangesets(Repository repository, ChangeSet (ChangeSet) callBack);
26 @javaClass{org.rascalmpl.library.experiments.scm.Scm}
27 public void java checkoutResources(CheckoutUnit unit, Repository repository);
28 @javaClass{org.rascalmpl.library.experiments.scm.Scm}
29 public set[WcResource] java getResources(Repository repository);
30 @javaClass{org.rascalmpl.library.experiments.scm.Scm}
31 public map[Resource, int] java linesCount(set[Resource] files);
32 @javaClass{org.rascalmpl.library.experiments.scm.Scm}
33 public set[Resource] java buildResourceTree(set[Resource] files);
```

Listing 2: Rminer CVS extension (Cvs.rsc)

```
1 module experiments::scm::cvs::Cvs
2 import experiments::scm::Scm;
3
4 data Repository = cvs(Connection conn, str mod, loc workspace, set[LogOption] options);
5 data Connection = pserver(str url, str repname, str host, str username, str password);
6
7 data ChangeSet = resource(Resource resource, rel[RevisionChange change, Info committer]
    revisions, rel[Revision revision, Tag symname] revTags);
8 data RevisionId = number(str number);
9
10 data WcResource = wcResourceRevisionInfo(Resource resource, Revision revision, Info info);
11 data CheckoutUnit = cunit(datetime date);
12
13 anno loc Connection@logFile;
14 anno int RevisionChange@linesAdded;
15 anno int RevisionChange@linesRemoved;
```

Listing 3: RMINER SVN extension (Svn.rsc)

```
1 module experiments::scm::svn::Svn
2 import experiments::scm::Scm;
3
4 data Repository = svn(Connection conn, str mod, loc workspace, set[LogOption] options);
5 data Connection = ssh(str url, str username, str password) | ssh(str url, str username, str
     password, loc privateKey);
6 data LogOption = mergeDetails() | fileDetails();
7
8 data ChangeSet = changeset(Revision revision, rel[Resource resource, RevisionChange change]
     resources, Info committer);
9 data RevisionChange = added(Revision revision, Resource origin) | replaced(Revision revision) |
      replaced(Revision revision, Resource origin);
10 data RevisionId = id(int id);
11
12 data MergeDetail = mergeParent(Revision parent);
13 data WcResource = wcResourceRevisionInfo(Resource resource, Revision revision, Info info);
14 data CheckoutUnit = cunit(datetime date) | cunit(Revision revision);
15
16 anno list[MergeDetail] Revision@mergeDetails;
```

Listing 4: RMINER Git extension (Git.rsc)

```
1 module experiments::scm::git::Git
2 import experiments::scm::Scm;
3
4 data Repository = git(Connection conn, str mod, set[LogOption] options);
5 data LogOption = mergeDetails() | fileDetails() | symdiff(CheckoutUnit from, CheckoutUnit to) |
      onlyMerges() | noMerges() | reverse() | allBranches();
6
7 data ChangeSet  = changeset(Revision revision, rel[Resource resource, RevisionChange change]
     resources, Info committer);
8 data RevisionChange = renamed(Revision revision, Resource origin) | copied(Revision revision,
     Resource origin);
9 data RevisionId = hash(Sha sha);
10 data Sha = blob(str sha) | commit(str sha);
11
12 data MergeDetail = mergeResources(Revision parent, rel[Resource resource, RevisionChange change
     ] resources);
13 data WcResource = wcResource(Resource resource);
14 data CheckoutUnit = cunit(Revision revision) | cunit(Tag symname);
15
16 anno loc Connection@logFile;
17 anno Info ChangeSet@author;
18 anno int RevisionChange@originPercent;
19 anno int RevisionChange@linesAdded;
20 anno int RevisionChange@linesRemoved;
21 anno list[MergeDetail] Revision@mergeDetails;
```

5 Case study

The goal of the case study is to evaluate the advantages and disadvantages of RMINER. By performing a MSR research, we can actually use RMINER to mine repositories, discuss the results and answer RQ1.4: Is the integrated repository model for Rascal a good solution for repository independent MSR research?

Another purpose of the case study is to (partially) validate the design and the implementation of RMINER. By doing a case study, we can check if the model is complete, at least for the particular MSR research we are performing.

More important, with the case study we can check if the semantics of the data modelled in the integrated repository model are well understood. The various SCM systems could contain data and functionality that, at first sight, might seem similar and are therefore unified in RMINER. However, subtle differences exist, even for data fields in the repository models that are called the same and seem to contain the same data according to the documentation.

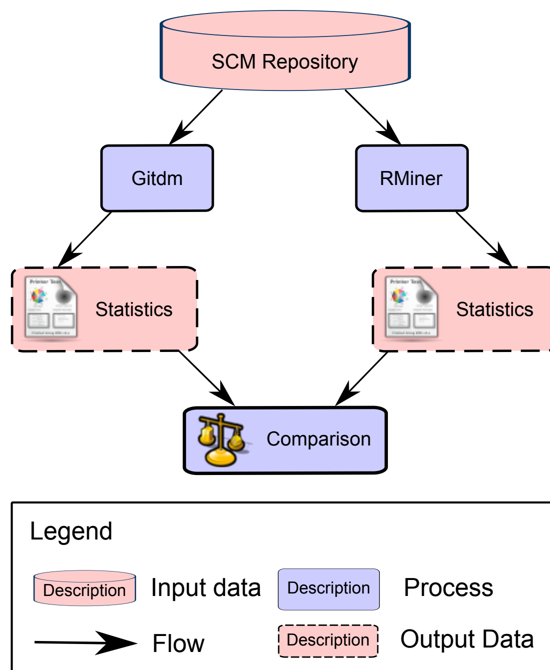

Figure 8: Case Study Method Outline

5.1 Method

Figure 8 contains the method outline of the case study. “Linux Kernel Development” [9] is a repository mining research which calculates statistics about the development of the Linux kernel. We will write MSR scripts with RMINER to calculate those statistics. Then we can compare the results of the research and the implementations of the MSR scripts. By comparing the results we can validate RMINER as described earlier and the implementations of the MSR scripts will be compared based on the criteria described in Section 5.3.

5.2 Statistics

Four kinds of information will be mined from the Linux kernel repository. Each category contains one or more questions that result in statistics about the releases of the Linux kernel between versions 2.6.12 and 2.6.21. Below are those statistics:

1. Frequency of release

   What are the time frames between the releases? What is the average time between releases?

2. Rate of Change

   What are the amount of changesets that have resulted to the particular releases. What is the average amount of changesets applied per hour?

3. Kernel Source Size

   What is the total amount of files in a release and how many lines of “code” is that? How many new lines are added per hour per release? Note that we count every line of every file in a release, since thats what the original research has done and as the author says “someone creates those files, and are worthy of being mentioned”.

4. Where the Change is Happening

   The Linux kernel source tree can be divided into multiple categories: "core", "drivers", "architecture","network", "filesystems" and "miscellaneous". For each of this categories, the following questions have to be answerd:

   - What is the amount of files and which percentage is that of the total amount of files?
   - What is the amount of lines of code and which percentage is that of the the total lines of code?

   Please note that in the original research[9] the author answers these questions only for the release version 2.6.21, so only the results of that release can be compared.

5. Who is Doing the Work and Who is Sponsoring it?

   Since many developers have contributed code to the linux kernel, the following information will be mined:

   - Which individual Kernel developers are in the top of contributors and how much have they contributed? What percentage is this amount of the total?
   - Which companies are in the top of contributors and how much have they contributed? What percentage is this amount of the total?

     Many companies are supporting the Linux kernel development, by having employees working on it. It is therefore interesting to know how many changesets a company has contributed and which percentage that is of the total amount of changesets.

Please do note that the original research had included the merge commits in the calculation of all the statistics except the "Who is doing the work" and "Who is sponsoring it". To be able to compare our results, we will handle the merges the same way.

5.3 Evaluation

We will discuss the pros and cons of Rminer based on the following In order to determine the feasibility of Rminer, the following evaluations will be performed.

5.3.1 Script reusability

The main goal of Rminer is to provide an environment where MSR scripts can be written which take as little as possible account with the repository that will be mined. To determine if this goal has been met, we will evaluate the reusability of the Rminer scripts.

The reusability is determined by counting all the decision points in the code (e.g. if statements) that are used to distinguish between the repositories, the input data is extracted from. In other words, if different code fragments are executed depending on the repository used, then the reusability of the script is lower then when the same code is executed all the time.

5.3.2 Data reusability

Data reusability is determined by whether or not the data extracted from the repositories and stored in the Rminer model, can be reused for different purposes (e.g. calculate different statistics).

5.3.3 Correctness

The statistics calculated with Rminer will be compared with the original statistics. In case of different statistics, the differences should be explained.

5.3.4 Speed

It is interesting to know how Rminer performs and how it compares with the tool used by the original research. Therefore, we will measure the amount of time the scripts need to calculate the statistics.

Knowing the performance of Rminer will help to answer the RQ1.4, where the question is being asked of whether or not Rminer is a good solution to the problem.

5.4 Results

Below the statistics calculated during the case study are presented. Discussion of the results and the MSR scripts can be found in Section 6.

For each statistic calculated the following information will be provided:

- method call, showing the initialization data required (see section 5.4.1)
- line number of the method called, in the code Listing 21 in Appendix B.
- calculated statistics in a table
- differences, if any, between the calculated statistics and Gitdm's results

5.4.1 Initialization

Before the actual statistics are calculated, Rminer fetches the evolution history from the repository as changesets. This changesets are used to initialize a set of mappings, which can be re-used during the calculation of multiple statistics.

```
1 public alias InitVars = tuple[list[Tag]
      releases, Repository repo, rel[str cat,
      str dir] catDirs, list[ChangeSet]
      changesets];
2 public alias MappingVars = tuple[rel[
      RevisionId child, RevisionId parent]
      childParents,
3   map[Tag version, ChangeSet changeset]
        tagChangeset,
4   map[RevisionId revId, ChangeSet cs]
        revChangeset,
5   rel[Tag version, RevisionId revId]
        versionRevisions,
6   rel[Tag version, ChangeSet cs]
        versionNoMergesChangesets];
7 InitVars initVars = ....
8 MappingVars maps = ....
```

Listing 5: Global variables

Listing 5 shows the data that will be initialized, and passed to the statistics calculation scripts. This way, the data mappings that can be reused, will not to be recalculated for each statistics. More on this in Section 6.1.1.

5.4.2 Frequency of release

```
1 statsOne(maps.tagChangeset, initVars.releases
      );
```

Listing 6: Calculating the frequency of release

The statistics calculated with Listing 6 are shown in Table 4. Implementation of the “statsOne” method can be found at line 52 of Listing 21.

Please note that the development days of all versions are higher, with one day, in the original research then presented in this thesis. Since we don't have the exact script used to calculate those numbers, we can only speculate that the difference lies in the calculation of differences between the start and end dates (inclusion of the end date or not). Furthermore, the amount of days of development of v2.6.19 and v2.6.20 calculated by Gitdm are 72 and 68 respectivaly. This means that, in addition to the normale one day offset, our statistics miss one day. The reason for this is that the original research does not count the hours of the release dates and the Rminer does. Therefore, those two versions miss a couple of hours and the amount of days are round down by the Rminer script.

5.4.3 Rate of Change

```
1 statsTwo(versionRevisions, releases);
```

Listing 7: Calculating the Rate of Change

The statistics calculated with Listing 7 are shown in Table 5. “statsTwo” can be found at line 65 of Listing 21 .

Kernel Version	Days of Development
v2.6.13	72
v2.6.14	60
v2.6.15	67
v2.6.16	76
v2.6.17	89
v2.6.18	94
v2.6.19	70
v2.6.20	66
v2.6.21	80

Table 4: Frequency of release with an average of 2 months and 2 weeks. Minor differences with the original results.

Kernel Version	Changes per Release
v2.6.13	4174
v2.6.14	3931
v2.6.15	5410
v2.6.16	5734
v2.6.17	6113
v2.6.18	6791
v2.6.19	7073
v2.6.20	4983
v2.6.21	5349

Table 5: Amount of changesets per release with an average of 3 changesets per hour. Reproducing the original results.

5.4.4 Kernel Source Size

```
1 statsThree(repo, releases);
```

Listing 8: Calculating the Kernel Source Size

The statistics calculated with Listing 8 are shown in Table 6. “statsThree” can be found at line 73 of Listing 21 .

Please note that very minor differences in the amount of files and lines are encountered during the comparison of the results with the original research. For example: in the original paper v2.6.12 has 17361 files and 6777860 lines, while we have encountered 17360 files and 6777945 lines. Different tools are used to calculate the amount of files and lines (of eventually binary files), which explains the differences.

Kernel Version	Files	Lines
v2.6.12	17360	6777945
v2.6.13	18090	6988886
v2.6.14	18433	7143289
v2.6.15	18810	7290126
v2.6.16	19250	7480116
v2.6.17	19552	7588067
v2.6.18	20207	7752891
v2.6.19	20935	7976266
v2.6.20	21279	8102576
v2.6.21	21612	8246463

Table 6: Kernel source size with an average of 91 lines added per hour. Minor differences with the original results.

5.4.5 Where the Change is Happening

```
1 statsFour(repo,getResources(repo).resources);
```

Listing 9: Calculating Where the Change is Happening

The statistics calculated with Listing 9 are shown in Table 7 and 8 . “statsFour” can be found at line 100 of Listing 21.

The only difference of Table 7 with the original research is the amount of files in the category “driver”. The original research has counted 6537 files, while RMINER script counts 6536 files.

Category	Files	% of kernel
core	1371	6%
drivers	6536	30%
architecture	10235	47%
network	1095	5%
filesystems	1299	6%
miscellaneous	1068	4%

Table 7: Amount of files at each category of v2.6.21

Category	Lines of Code	% of kernel
core	330637	4%
drivers	4304860	52%
architecture	2127155	25%
network	506966	6%
filesystems	702914	8%
miscellaneous	263936	3%

Table 8: Amount of lines at each category of v2.6.21 with very minor differences with the original results

5.4.6 Who is doing the work and Who is sponsoring it

```
1 statsFive(versionNoMergesChangesets);
```

Listing 10: Calculating who is doing the work and Who is sponsoring it

The statistics calculated with Listing 10 are shown in Table 9 and 10. “statsFive” can be found at line 160 of Listing 21.

Name	Number of Changes	%
Ralf Baechle	134	2%
Eric W. Biederman	111	2%
Adrian Bunk	83	1%
Al Viro	79	1%
Andrew Morton	72	1%
Takashi Iwai	67	1%
Bob Moore	66	1%
Jeff Dike	64	1%
David Brownell	59	1%
David S. Miller	59	1%
Robert P. J. Day	58	1%

Table 9: Who is doing the work in v2.6.21, reproducing the original results

Company	Number of Changes	%
Unknown	2761	55%
Novell	275	5%
IBM	271	5%
Intel	237	4%
LinuxFoundation	155	3%
RedHat	141	2%
Oracle	106	2%
SGI	69	1%
MontaVista	59	1%
linutronix	59	1%
Toshiba	39	0%

Table 10: Who is sponsoring the work in v2.6.21, with very minor differences with the original results

6 Discussion

In order to calculate the statistics presented in previous section, the meta-data stored on the repositories are needed. This meta-data can be obtained from the repository in a ”textual” format and has to bo parsed to be meaningful for any analysis. The parsing and analysing processes of a MSR tool can be scheduled with different approaches:

1. Parse the whole data beforehand in changesets and then call the analysis script;
2. Parse the data as changesets and call back to the analysis script after a changeset is parsed;
3. Parse the data the way that the analysis script needs it. In this approach the analysis script interferes with the parsing process and thus can be more specific about how (and which) data has to be parsed;

Each of the appraches described above has it’s pros and cons on various aspects of the analysis scripts and data extracted from the repositories. In the rest of this section, Rminer will be compared with Gitdm and Kenyon framework on the following aspects:

Script Reusability describes the reusability of the scripts and the reusability of the data extracted from the repositories. The reusability of the scripts are determined by the fact that the scripts can be used with different data (e.g. data from Git instead SVN).

Please do note that most of the statistics make use of the ChangeSet as defined in Section 3.1. ChangeSets extracted from CVS are a little different, since they contain all the changes of a single resource, instead of all the changes of a commit. So in order to use the data from CVS, one has first to transform it into "SVN/Git" ChangeSets, before actually trying to call the scripts (see also Section 4.3). So, the reusability of the script will not be affected by this point.

Data Reusability is evaluated by the extent to which the data, created/used by the scripts, can (and is) reused for different purposes (e.g. to calculate different statistics).

Correctness of the RMINER script is very high, since it results in mostly the same statistics as the Gitdm tool, used by the original research. There are however some differences in the way the two tools calculate some of the statistics. For example, there are some differences in the "Frequence of releases" statistics, where the original research does not count the hours of the release dates and the Rascal script does. The result is that some releases have one day offset, because they miss a couple of hours and round down the day count. But overall the statistics are the same, thus we can conclude that the script does work correctly.

Speed describes the execution time of the scripts. While a fair speed comparison between the tools cannot be done, it is interesting to know if RMINER is at least usable. To that end we will measure the time needed to calculate each statistic, by calculating the difference between timestamps taken before and after the execution of parts of the scripts.

6.1 Global design

Some of the tools examined have an initialization phase, before the actual statistics are calculated. In this section this first phase of the tools will be discussed. Furthermore, the global design of the tools will be compared on the aspects described in the previous section, before the actual statistics calculation codes are evaluated.

6.1.1 Rminer

RMINER script fetches, conform to the first scheduling approach, all the changesets from the repository (figure 9), between the release versions provided, before doing any kind of analysis. Below the pros and cons of the RMINER script:

- Fetching a large amount of changesets can take a lot of time, depending on the configuration of RMINER and the amount of changesets and resource changes. There are some configuration points in RMINER, for example to exclude "merge changesets" or include "detailed information about the resources", which can influence the process of fetching the changesets.
- Since the changesets are parsed in a predefined data structure, before any specific analysis is done, it is possible to cache and reuse them at a later time. This eliminates the need of communicating with (and parsing the output of) the repository again.

Please do note that RMINER also supports the second scheduling approach, where the script gets called back when a changeset is processed, but since most of the analysis in the case study require all the changesets to be present, there is no reason to be notified before all the changesets are processed.

The initialization phase of the RMINER script can be divided into two procedures, shown in Figure 9.

```
1 public void initChangesets() {
2   releases = [label("v2.6.<i>") | i <-
        [12..21]];
3   repo = git(fs("/tmp/linux-2.6"), "", {
        startUnit(cunit(releases[0])), endUnit(
        cunit(releases[size(releases)-1]))});
4   changesets = getChangesets(repo);
5 }
```

Listing 11: Fetching the changesets in Rascal

Init Changeset fetches all the changesets, between the releases "2.6.12" and "2.6.21", from the repository (listing 11).

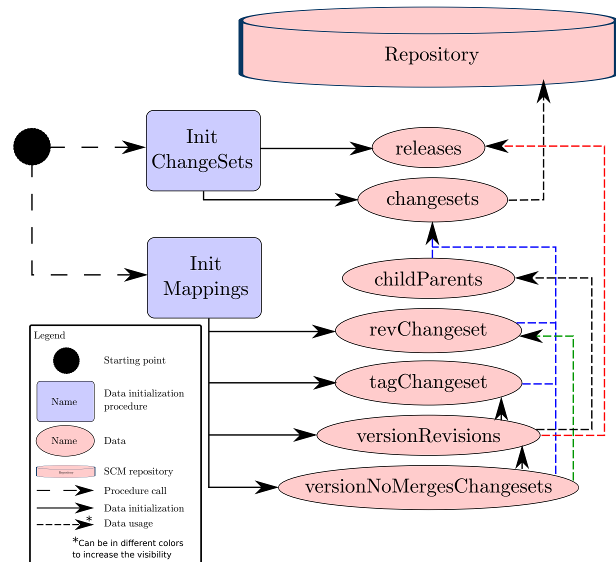

Figure 9: Data initialization in RMINER

Init Mappings initializes various data mappings to be used during the calculations of various statistics. With dotted lines Figure 9 shows which data mappings are used in the initialization phase of which data field. For example: in order to initialize "versionNoMergesChangesets", we need "tagChangeset" and "changesets", while we don't need any data to initialize "releases" since the list of release version's are provided by the user.

- childParents contains the relations between the revision identifiers of the changesets and their ancestors
- revChangeset contains the relations between the revision identifiers and the changeset they identify
- tagChanget contains the relations between the tag identifiers and the changeset they refer to
- versionRevisions contains the relations between the version tags and the revision identifier of all the changesets that have been committed since the previous version
- versionNoMergesChangesets contains the relations between the version tags and the changeset committed since since the previous version. Please do note that this relation excludes the changesets that represent a merge operation, while the rest of the mappings do contain the merge changesets. This is in line with the requirements of the statistics that will be using this data.

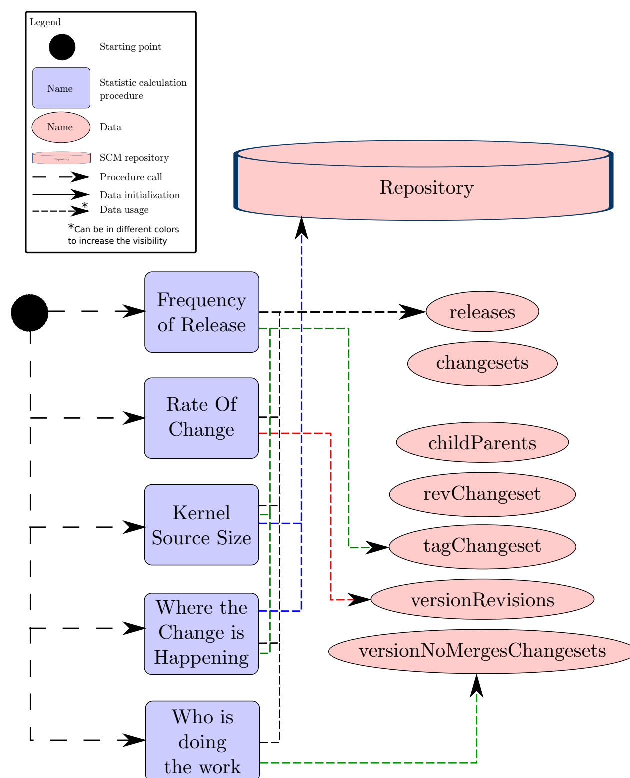

Figure 10: Data usage in RMINER when mining all the versions of Linux.

Data usage by the statistics calculation scripts is shown in Figure 10. The diagram shows which data is

used in the calculation of which statistics.We can see that three of the seven data mappings are only (directly) used in the initialization phase. However, this mappings where used to initializze the other mappings which are used by the calculation scripts.

Round trips to the repository are limited to: one during the initialization phase plus the amount of checkout commands made by the statistics calculation scripts where the actual resources stored on the repository is needed.

6.1.2 Gitdm

Gitdm (Figure 11) uses the third scheduling approach, where the analysis scripts interefere with the parsing process. Below the pros and cons of Gitdm:

- The analysis script is integrated with the parsing process and decides how and which lines of the repository log has to be parsed. This has as consequence that the parsing process can take much less time since only the data used by the analysis script needs to be parsed.
- Gitdm keeps track of various data mappings, so that data can be reused to calculate different statistics. However, as can be seen in Figure 11, the data mappings are not used by more then one statistic script. This is due the fact that the other statistics are calculated with additional ad-hoc bash or perl scripts, which have their own temprory data structures and connect to the repository on their own.
- Gitdm cannot be used with data from other repositories then Git.

Round trips to the repository is required for each release version of Linux we want to mine. Gitdm requests the history of each release separatly from the repository, which means that all the scripts need to be executed all over again for each revision being mined.

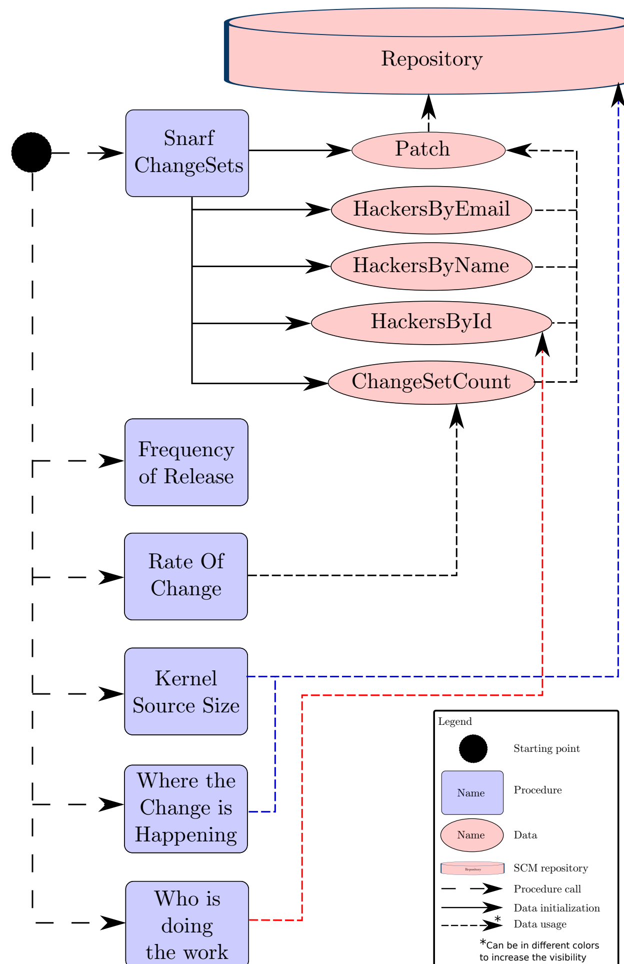

Figure 11: Data initialization and usage in Gitdm. To be executed for each Linux release mined.

6.1.3 Kenyon

Kenyon programs use the second scheduling approach, where the analysis script is called each time a changeset is parsed. Below the pros and cons of Kenyon:

- Kenyon checks out the actual contents of the revisions, before the analysis script is being called back. That means that in an initialization phase it has little value to fetch all the changesets be-

forehand. Besides the fact that most of our statistics do not use the actual contents of the files, Kenyon would overwrite the checked out files when fetching a new version. This means that it would take unnecessary much time to gather all the changesets metainformation from the repository in the initialization phase.

- The parsed meta-data is reusable because it is stored in a predefined data structure and is not altered by the analysis scripts.

6.2 Frequency of release

6.2.1 Rminer

RMINER scripts are able to calculate the time frame between the releases by using the “Tag to Changeset” mapping. This results in 7 lines procedure shown in listing 12.

```
1 public int statsOne(map[Tag version,
     ChangeSet cs] tagChangeset, list[Tag]
     releases) {
2   int i = 0;
3   for (version <- releases, version in
       tagChangeset) {
4       if (i > 0) {
5   prev = releases[i-1];
6   print("<version.name> - <daysDiff(
       tagChangeset[prev].committer.date,
       tagChangeset[version].committer.date)>"
       );
7       }
8       i += 1;
9   }
10  return printStopTimer("statsOne");
11 }
```

Listing 12: Calculating release frequency in Rascal

Reusability of this script is very high, since no decision points are used to distinguish based on the repository used.

6.2.2 Gitdm

Gitdm does not provide any scripts that calculates the time frame between releases. Judging on the design of Gitdm, a solution would be very much similar to the Rascal solution. Gitdm stores necessary data during the parsing process in various maps, and uses it during the reporting process.

6.2.3 Kenyon

Kenyon makes it impossible to calculate the time frame between releases becaus no tag information is available. Furthermore, because of the Kenyon's callback mechanism (schedule approach 2) only one changeset is loaded into the memory at a time. This means that adding the tag information to the changesets will not be enough, unless the changesets can be loaded from the database on basis of the tag information (currently it is only possible to load a changeset on basis of a datetime).

6.2.4 Rate of Change

An important aspect of this statistics is that calculating the “Changes per release” is not as simple as counting all the changesets between two release dates. It is for example possible for an changeset to be committed in-between the commit dates of release A and B, but not being included in release B. In order to know which changesets have contributed to a particular release, the (merge) parents of a release need to be followed back to the initial commit and include all the changesets that are being passed by.

Rascal scripts solve this problem by following the history of a release through it's changeset's (merge) parents back to the initial commit and including all the changesets passed by. The“versionRevisions” mapping is used to print the amount of changesets in listing 14.

```
1 public int statsTwo(rel[Tag version,
     RevisionId revision] versionRevisions,
     list[Tag] releases) {
```

```
2   for (version <- releases, version in
        versionRevisions.version) {
3     print("<version> - <size(versionRevisions
          [version])>");
4   }
5 }
```

Listing 13: Calculating Rate of Change in Rascal

Reusability of this script is not so high, since the getVersionRevisions(), used by the initialization script to initialize “versionRevisions”, checks whether the Git repository is used.

```
1   if (gitRepo) {
2       reachable = reach(childParents, {
            tagChangeset[version].revision.id})
            ;
3     } else {
4       reachable = reach(childParents, {
            tagChangeset[version].revision.id},
             {tagChangeset[ver].revision.id|
            ver <- tagChangeset.version, ver !=
             version});
5     }
6 }
```

Listing 14: Getting the changesets resulted to each version in Rascal

Gitdm solves this problem by commanding Git to only provide the “textual log” of the changesets resulted to a particular release (e.g. git log v2.6.20..v.2.6.21). This approach has the following implications:

- This approach is the fastest one, since Git can use it’s intern structure to efficiently select the changesets to output. Since the “textual log” will only contain the needed changesets, no unnecessary parsing will happen.
- However, if many releases are analysed, this approach could be inefficient, since for each release the whole history needs to be requested from Git and parsed. Those release histories often contain common changesets, which needlessly will be reparsed.

Kenyon can’t solve this problem, due to i’ts lack of knowledge of tags and support for Git. Even if Tags and Git would be supported, it would be needlessy slow to count the amount of changesets, because each changeset needs to be checked out first.

6.2.5 Kernel Source Size

Rascal script counts the lines of code of the files in Java with the LineNumberReader[9] library. Listing 15 shows this process of: checking out the files, gathering them into a “wcResources” and counting the amount of lines.

```
1 public int statsThree(Repository repo, map[
      Tag version, ChangeSet cs] tagChangeset,
      list[Tag] releases) {
2   for (version <- releases) {
3     checkoutVersion(repo, tagChangeset,
          version);
4     set[WcResource] wcResources =
          getResources(repo);
5     set[Resource resource] resources = {r.
          resource | r <- wcResources};
6     map[Resource file, int lines] fileLines =
           linesCount(resources);
7     int totalLines = 0;
8     for(f <- fileLines.file) {
9       totalLines += fileLines[f];
10    }
11    print("<version.name> - <size(fileLines.
          file)> - <totalLines>");
12  }
13 }
```

Listing 15: Calculating Kernel Size in Rascal

```
1 public void checkoutVersion(Repository repo,
      map[Tag version, ChangeSet cs]
      tagChangeset, Tag version) {
2   CheckoutUnit cu;
3   //little workaround for gits lack of
        checkout by date
4   if (git(_,_,_) := repo) {
5     cu = cunit(tagChangeset[version].revision
          );
6   } else {
7     cu = cunit(tagChangeset[version].
          committer.date);
8   }
```

[9] http://download-llnw.oracle.com/javase/6/docs/api/java/io/LineNumberReader.html

```
9    checkoutResources(cu, repo);
10 }
```

Listing 16: Checking out a version from the repository

Gitdm Gitdm does not provide the script used to calculate this statistics, but since the next statistic is very similar to this one, one can imagine how the author had calculated the amount of files and lines. More on this in the next section.

Kenyon checks out the files of each revision, and further calculation of the file statistics can be done in Java similar to the Rascal approach.

6.2.6 Where the Change is Happening

Rascal scripts calculate the size of the linux source tree categories, by dividing the source files into the categories and performing the analysis on the resources in each category. Listing 17 contains the method that given a set of "Resources" and a relation of (top) directories to the appropriate category, returns a relation of category and it's Resources.

```
1  public rel[str cat, Resource file]
       resourcesByCategory(set[Resource
       resource] resources, rel[str dir, str
       cat] dirCategories) {
2    rel[str cat, Resource resource]
         catResources = {};
3    for (r <- resources) {
4      for(dir <- domain(dirCategories)) {
5        if (startsWith(r.id.path, dir)) {
6          for(cat <- dirCategories[dir]) {
7            catResources += {<cat, r>};
8          }
9        }
10     }
11   }
12   return catResources;
13 }
```

Listing 17: Categorizing Resources in Rascal

Gitdm calculates this statistics with the bash commands "find", "wc", "cut" and "grep". See 19 for the full bash script used by Gitdm.

Kenyon provides the Java mining programs with a "File" object referring to the root directory where the files are checked out. This means that the programs are free to calculate the statistics the way they wish. It is hoewever obvious to use the "File" and "LineNumber" API, just like the Rascal library does under the hood, to calculate the metrics. The file categories can be specified with "FileFilters".

6.2.7 Who is doing the work and Who is sponsoring it

Rascal parses the author's information with regular expressions, as shown in listing 18. E-mail aliasses and misspelled names are detected with the "solveBrokenRelation" method.

```
1  public rel[Tag version, str action, str
       devverName, str email, ChangeSet cs]
       calcDevelopers(Tag version, str action,
2    rel[Info info, ChangeSet cs] devvers,
         bool parseCsMessage) {
3    rel[Tag version, str action, str
         devverName, str email, ChangeSet cs]
         results = {};
4    for (Info info <- devvers.info) {
5      str msg = (parseCsMessage ? (info.
           message ? "") : "") + " " + action
           + "-by: " + info.name;
6      for (/\s*<action:[^\s]*>-by:\s*<name
           :.*>/ := msg) {
7        if (/\s*"?<fname:[^\<"]+>"?\s\<<mail
             :[^\>]+>\>/ := name) {
8          results += {<version, action, fname
               , toLowerCase(mail), cs> | cs
               <- devvers[info]};
9        } else if (/\<<mail:[^\>]+>\>/ :=
             name){
10         results += {<version, action, "",
               toLowerCase(mail), cs> | cs <-
               devvers[info]};
11       } else {
12         results += {<version, action, name,
                "", cs> | cs <- devvers[info
               ]};
13       }
14     }
```

```
15      }
16      return results;
17    }
```

Listing 18: Parsing author information in Rascal

Gitdm uses regular expressions too to parse the author information. Please note that due to the adhocness of Gitdm, this information is directly parsed as it is outputed from the Git repository. This is a difference with Rascal, where the parsing process only parses the common fields and properties and no specific comit message parsing is performed. Another difference with the Rascal approach is the detection of e-mail aliasses and misspelled names. This process is also integrated into the parsing process, where a map of predefined email aliasses are used to detect aliasses.

Kenyon provides the Java programs with the Author name of a changeset. It is therefore possible to parse this information the same way, the Rascal scripts have done it. Due to it's callback nature, the caller has to save the parsed information and do the e-mail alias detection at the end.

6.3 Speed

In this section the execution times of Rminer and Gitdm are presented, which can help to see which statistics need the most time and if Rminer is reasonably fast enough.

Initialization phase of Rminer took place in 262s and 346ms, 15s and 168ms of which where spend on fetching the changesets from the repository.

Statistics calculated with Rminer took in total 213s and 903ms. Below the measured times per statistic:

1. Frequency of release: 23ms
2. Rate of Change: 168ms
3. Kernel Source Size: 162s 834ms
4. Where the Change is Happening: 21s 366ms
5. Who is Doing the Work and Who is sponsoring it: 29s 512ms

Gitdm has a different architecture then Rminer, which makes it hard to measure the times per statistic. Individual scripts are used to calculate some of the statistics, while the rest of the statistics are calculated by one python script.

The statistics "Rate of Change" and "Who is Doing the Work and Who is sponsoring it" are calculated in a total time of 194s and 325ms. "Where the Change is Happening" is calculated in 46s and 331ms. Since no scripts are provided to calculate the "Frequency of release" and "Kernel Source Size" statistics, we can't measure the times of those statistics.

Summary of the execution time of the statistics measured on both scripts are as follows: Rminer: 213,903+0,168+21,366+29,512= 264s 949ms and Gitdm: 194.325+46.331 = 240s and 656ms. We can thus say that, at least for this case study, Rminer is not too slow and is almost as fast as the ad-hoc tool (Gitdm).

6.4 Evaluation

Before concluding the research in next section, by answering the main research question, we will evaluate the case study in this section.

Performing the case study, we have encountered that mistakes can be made very easily. Especially in cases where the various repositories seem to be similar, but are not. Below an example of a case where a MSR research could fail because of semantic differences between Git and other SCM systems.

Most of the SCM systems allow the users to set limitation on the history extraction process based on a

"begin" and "end" revisions. In SVN we can get all the changesets from A until B by using the "-rA:B" filter. Git's "since..until" filter seems to provide the same behaviour: return the history of the repository started from "since" and ended by "until". However, Git might exclude some of the changesets committed after "since" and before "until" in the resulted history. In fact, the filter makes sure that only the changesets that where an ancestors of "until", but where not an ancestor of "since" are returned.

This differences between the SCM systems have been made explicit in RMINER by creating an extra filter that is only supported by Git. So when a MSR script uses the common filter to specify the "since" and "until" ranges, the expected behaviour of processing all the changesets between "since" and "until" will be encountered. However, the extra filter in the Git module can be used if the "ancestors exclusion" behaviour is preferred.

Nevertheless, future research needs to be done on the usage of various SCM systems to make the more subtle differences between the repositories explicit. Bird et al. has investigated the perils and promises of mining Git"[2] and similar studies for other SCM systems should be done, in order to further improve RMINER.

6.5 Validation

In this research we have investigated three SCM systems as data sources for RMINER. The question arises: are those three systems representative for the domain of SCM systems?

During the research we have encountered many (often small) differences between the datamodels and the feature sets of the three SCM systems. However, the high-level SCM concepts are shared between the systems (section 3.1). Furthermore, we have examined two different types of SCM systems (centralized, CVS/SVN, and decentralized: Git). This makes the model more representative, because the two different types have their own concepts (e.g. hashing of resources in decentralized systems) and supporting them means supporting (part of) the models of other SCM systems of the same type.

Therefore, we expect that the concepts introduced by the three SCM systems will be shared by other SCM systems (e.g. there is always a repository, a revision or a resource change kind). So, the answer to the question is: Yes, the three SCM systems examined are representative for the domain of SCM systems.

A second question that remains to be answered is: can the integrated model easily be extended so that more SCM systems are supported?

Due to RMINER's design it is possible to easilty add new data types and variation points to the integrated repository model. By hiding the differences between the repository models behind various abstraction levels, RMINER allows us to extend the integrated repository model without changing the models of the SCM systems already supported. This means that the probability that MSR scripts already written need to be changed is very low.

It is worth mentioning that some MSR research use additional kinds of data sources (e.g. bug-tracking systems), which we have not investigated in our research. RMINER is designed with only SCM systems in mind which means that adding other kinds of data sources to the integrated model can be potentially more difficult then expected.

Furthermore, we have performed only one case study, and as stated earlier more case studies have to be performed to fully validate the model. One of the reason for this is the fact that various SCM systems can be used on different ways, so it is not always possible to write repository independent MSR scripts and expect it to give the correct results when used with different SCM systems.

7 Conclusion

In this thesis the feasibility of an integrated model for repository mining using Rascal is examined. Following are the contributions of this research:

- we have summarized research in MSR domain and tools available for it.
- we have summarized the commonalities and differences between three SCM systems (CVS/SVN and Git), on concepts important for MSR research
- we have designed and implemented an integrated model for repository mining with Rascal
- we have performed a case study to evaluate the feasibility of the model

RQ1 Feasibility: Can we design and implement an integrated repository model using Rascal, that facilitates repository independent MSR research?

In order to support a broad range of MSR research, we have chosen to include as much of the meta-data provided by the various software repositories as possible, in the (extensions of the) integrated model. This, ofcourse, means that MSR research using repository specific meta-data are not repository independent. Furthermore, every SCM system has it's own set of filters and options that can be used to configure the software history extraction process. Since most of the MSR research require the history from the repository, care should be taken when using those filters and options, otherwise the extraction process will not be repository independent.

So, the answer to the question is: partially. If the MSR scripts make use of the generic parts of the model, that are supported by all the SCM systems, then it is indeed possible to perform repository independent MSR research. Otherwise, when SCM specific data or filters/options are used, the integrated model and its extensions should be checked to see which SCM systems are supported. Nevertheless, Rminer unifies as much common meta-data from the repository models as possible (section 4.2.2) , while providing multiple abstraction levels for different needs (section 4.2.3).

RQ1.4 Is the integrated repository model for Rascal a good solution for repository independent MSR research?

It is a step in the right direction. Various SCM systems might contain data, and/or functionality, that at first sight might seem to be similar but has slightly different meaning. Therefore, it is important to make the differences between the models explicit, when unifying the models. Rminer does this, as described in section 4.

It is important to know if concepts unified between the models, are indeed in practice used the same way. If that is not the case, MSR research using those parts of the unified model might get erroneous results when performed on various SCM systems. Partially, this problem can be solved by cleaning the data as described in section 4.4. However, improving the unified model by making the different usage patterns more explicit is a better solution.

To conclude this research we can say that while Rminer unifies the commonalities between various repository models and makes the differences explicit, it is not the silver bullet.

References

[1] Jennifer Bevan, E. James Whitehead, Jr., Sunghun Kim, and Michael Godfrey. Facilitating software evolution research with kenyon. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 177–186, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: http://doi.acm.org/10.1145/1081706.1081736.

[2] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3493-0. doi: http://dx.doi.org/10.1109/MSR.2009.5069475.

[3] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: a project memory for software development. 31(6):446–465, 2005. doi: 10.1109/TSE.2005.71.

[4] D. Draheim and L. Pekacki. Process-centric analytical processing of version control data. In *Proc. Sixth Int Software Evolution Workshop Principles of*, pages 131–136, 2003. doi: 10.1109/IWPSE.2003.1231220.

[5] Daniel M. German, Davor Čubranić, and Margaret-Anne D. Storey. A framework for describing and understanding mining tools in software development. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM. ISBN 1-59593-123-6. doi: http://doi.acm.org/10.1145/1083142.1083160.

[6] Collard M.L. Maletic J.I. Kagdi, H. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, Vol 19, No 2:77–131, 2007.

[7] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. Automatic identification of bug-introducing changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2. doi: http://dx.doi.org/10.1109/ASE.2006.23.

[8] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *SCAM '09: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3793-1. doi: http://dx.doi.org/10.1109/SCAM.2009.28.

[9] Greg Kroah-Hartman. Linux kernel development. volume One, pages 239–244, 2007.

[10] Premkumar T. Devanbu Omar Alonso and Michael Gertz. Database techniques for the analysis and exploration of software repositories. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR 2004).*, 2004.

[11] Vladimir Rubin, Christian W. Günther, Wil M. P. Van Der Aalst, Ekkart Kindler, Boudewijn F. Van Dongen, and Wilhelm Schäfer. Process mining framework for software processes. In *ICSP'07: Proceedings of the 2007 international conference on Software process*, pages 169–181, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-72425-4.

[12] Thomas Zimmermann. Preprocessing cvs data for fine-grained analysis. 2004.

[13] Thomas Zimmermann. Fine-grained processing of cvs archives with apfel. In *eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 16–20, New York, NY, USA, 2006. ACM. ISBN 1-59593-621-1. doi: http://doi.acm.org/10.1145/1188835.1188839.

A Appendix A

Listing 19: Gitdm kernel source size script (stats.sh)

```
1 #!/bin/bash
2
3 CORE="init/ block/ ipc/ kernel/ lib/ mm/ include/linux/ include/keys/"
4 DRIVERS="crypto/ drivers/ sound/ security/ include/acpi include/crypto include/media include/
    mtd include/pcmcia include/rdma include/rxrpc include/scsi/ include/sound/ include/video/"
5 ARCH="arch/ include/asm-* include/math-emu/ include/xen"
6 NET="net/ include/net/"
7 FS="fs/"
8 MISC="Documentation/ scripts/ usr/"
9
10
11 echo `ketchup -m`
12 echo "files in whole tree   `find . -type f | wc -l`"
13 echo ""
14 echo "files in core         `find $CORE -type f | wc -l`"
15 echo "files in drivers      `find $DRIVERS -type f | wc -l`"
16 echo "files in architecture `find $ARCH -type f | wc -l`"
17 echo "files in network      `find $NET -type f | wc -l`"
18 echo "files in filesystems  `find $FS -type f | wc -l`"
19 echo "files in miscellaneus `find $MISC -type f | wc -l`"
20
21 echo ""
22
23 echo "lines in whole tree   `find . -type f | xargs wc -l | grep total | cut -f 1 -d 't'`"
24
25
26 CORE_LINES=`find $CORE -type f | xargs wc -l | grep total | cut -f 1 -d 't'`
27 echo "lines in core         $CORE_LINES"
28
29 DRIVERS_LINES=`find $DRIVERS -type f | xargs wc -l | grep total | cut -f 1 -d 't'`
30 echo "lines in drivers      $DRIVERS_LINES"
31
32 ARCH_LINES=`find $ARCH -type f | xargs wc -l | grep total | cut -f 1 -d 't'`
33 echo "lines in architecture $ARCH_LINES"
34
35 NET_LINES=`find $NET -type f | xargs wc -l | grep total | cut -f 1 -d 't'`
36 echo "lines in network      $NET_LINES"
37
38 FS_LINES=`find $FS -type f | xargs wc -l | grep total | cut -f 1 -d 't'`
39 echo "lines in filesystems  $FS_LINES"
40
41 MISC_LINES=`find $MISC -type f | xargs wc -l | grep total | cut -f 1 -d 't'`
42 echo "lines in miscellaneus $MISC_LINES"
```

B Appendix B

Listing 20: RMINER Linux stats configuration (Git.rsc)

```
module Git

import Utilities;
import Statistics;
import experiments::scm::Scm;
import experiments::scm::git::Git;

import DateTime;
import List;
import Map;
import Set;
import Relation;
import Graph;
import String;
import Real;
import Node;

public tuple[list[int], InitVars, MappingVars] gitStats() {
  gitConfig = getGitConfig();
  initVars = <gitConfig.releases, gitConfig.repo, gitConfig.catDirs, getChanges(gitConfig.repo)
      >;
  MappingVars maps = getMappings(initVars, ());
  return <stats(initVars, maps), initVars, maps>;
}

public tuple[list[Tag] releases, Repository repo, rel[str cat, str dir] catDirs] getGitConfig()
     {
  repo = git(fs("/export/scratch1/shabazi/linux-2.6"), "", {});
  releases = [label("v2.6.<i>") | i <- [12..21]];

  rootDir = repo.conn.url;
  rel[str cat, str dir] catDirs = {};
  catDirs += {<"core", "<rootDir>/<d>"> | d <- ["init", "block", "ipc", "kernel", "lib", "mm",
      "include/linux", "include/keys"]};
  catDirs += {<"drivers", "<rootDir>/<d>"> | d <- ["crypto", "drivers", "sound", "security", "
      include/acpi", "include/crypto",
    "include/media", "include/mtd", "include/pcmcia", "include/rdma", "include/rxrpc", "include
        /scsi", "include/sound", "include/video"]};
  catDirs += {<"architecture", "<rootDir>/<d>"> | d <- ["arch", "include/asm-", "include/math-
      emu", "include/x"]};
  catDirs += {<"network", "<rootDir>/<d>"> | d <- ["net", "include/net"]};
  catDirs += {<"filesystems", "<rootDir>/<d>"> | d <- ["fs"]};
  catDirs += {<"miscellaneous", "<rootDir>/<d>"> | d <- ["Documentation", "scripts", "usr"]};

  return <releases, repo, catDirs>;
}
```

Listing 21: RMINER stats (Statistics.rsc)

```
1 module Statistics
2
3 import Utilities;
4 import ValueIO;
5 import DateTime;
6 import IO;
7 import experiments::scm::Scm;
8 import experiments::scm::cvs::Cvs;
9 import experiments::scm::svn::Svn;
10 import experiments::scm::git::Git;
11 import experiments::scm::Timer;
12 import DateTime;
13 import List;
14 import Map;
15 import Set;
16 import Relation;
17 import Graph;
18 import String;
19 import Real;
20 import Node;
21
22 public alias InitVars = tuple[list[Tag] releases, Repository repo, rel[str cat, str dir]
     catDirs, list[ChangeSet] changesets];
23 public alias MappingVars = tuple[rel[RevisionId child, RevisionId parent] childParents,
24   map[Tag version, ChangeSet changeset] tagChangeset,
25   map[RevisionId revId, ChangeSet cs] revChangeset,
26   rel[Tag version, RevisionId revId] versionRevisions,
27   rel[Tag version, ChangeSet cs] versionNoMergesChangesets];
28
29 public list[int] stats(InitVars initVars, MappingVars maps) {
30   domainMap = readTextValueFile(#(map[str domain, str company]), |file:///export/scratch1/
       shabazi/domain-map.txt|);
31   list[int] durations = [];
32   Tag lastVersion = initVars.releases[size(initVars.releases) - 1];
33   print("------------------STATS 1------------------");
34   durations += statsOne(maps.tagChangeset, initVars.releases);
35   print("------------------STATS 2------------------");
36   durations += statsTwo(maps.versionRevisions, initVars.releases);
37   print("------------------STATS 3------------------");
38   durations += statsThree(initVars.repo, maps.tagChangeset, initVars.releases);
39   print("------------------STATS 4------------------");
40   durations += statsFour(initVars.repo, maps.tagChangeset, [lastVersion], initVars.catDirs);
41   print("------------------STATS 5------------------");
42   durations += statsFive(maps.versionNoMergesChangesets, domainMap, [lastVersion]);
43
44   int totalDuration = 0;
45   for(d <- durations) {
46     totalDuration += d;
47   }
48   print("Duration:<totalDuration>");
49   return durations;
50 }
51
52 public int statsOne(map[Tag version, ChangeSet cs] tagChangeset, list[Tag] releases) {
53   printStartTimer("statsOne");
54   int i = 0;
```

```
55   for (version <- releases, version in tagChangeset) {
56     if (i > 0) {
57       prev = releases[i-1];
58       print("<version.name> - <daysDiff(tagChangeset[prev].committer.date, tagChangeset[version
           ].committer.date)>");
59     }
60     i += 1;
61   }
62   return printStopTimer("statsOne");
63 }
64
65 public int statsTwo(rel[Tag version, RevisionId revision] versionRevisions, list[Tag] releases)
      {
66   printStartTimer("statsTwo");
67   for (version <- releases, version in versionRevisions.version) {
68     print("<version> - <size(versionRevisions[version])>");
69   }
70   return printStopTimer("statsTwo");
71 }
72
73 public int statsThree(Repository repo, map[Tag version, ChangeSet cs] tagChangeset, list[Tag]
     releases) {
74   printStartTimer("statsThree");
75   for (version <- releases) {
76     checkoutVersion(repo, tagChangeset, version);
77     set[WcResource] wcResources = getResources(repo);
78     set[Resource resource] resources = {r.resource | r <- wcResources};
79     map[Resource file, int lines] fileLines = linesCount(resources);
80     int totalLines = 0;
81     for(f <- fileLines.file) {
82       totalLines += fileLines[f];
83     }
84     print("<version.name> - <size(fileLines.file)> - <totalLines>");
85   }
86   return printStopTimer("statsThree");
87 }
88
89 public void checkoutVersion(Repository repo, map[Tag version, ChangeSet cs] tagChangeset, Tag
     version) {
90   CheckoutUnit cu;
91   //little workaround for gits lack of checkout by date
92   if (git(_,_,_) := repo) {
93     cu = cunit(version);
94   } else {
95     cu = cunit(tagChangeset[version].committer.date);
96   }
97   checkoutResources(cu, repo);
98 }
99
100 public int statsFour(Repository repo, map[Tag version, ChangeSet cs] tagChangeset, list[Tag]
     releases, rel[str cat, str dir] catDirs) {
101   int duration = 0;
102   for (version <- releases) {
103     printStartTimer("checkout <version>");
104     checkoutVersion(repo, tagChangeset, version);
105     duration += printRestartTimer("getResources <version>");
106     set[WcResource] wcResources = getResources(repo);
```

```
107     duration += printStopTimer("getResources <version>");
108     set[Resource resource] resources = {r.resource | r <- wcResources};
109     duration += statsFour(repo, resources, catDirs);
110     print("statsFour - <version> - <duration>ms");
111   }
112   return duration;
113 }
114 
115 public int statsFour(Repository repo, set[Resource resource] resources, rel[str cat, str dir]
        catDirs) {
116   int duration = 0;
117 
118   printStartTimer("resourcesByCategory");
119   rel[str cat, Resource file] filesByCat = resourcesByCategory(resources, catDirs<1,0>);
120   duration += printStopTimer("resourcesByCategory");
121   print("Category - Files - % of kernel");
122   int totalFiles = size(filesByCat.file);
123   for (c <- filesByCat.cat) {
124     print("<c> - <size(filesByCat[c])> - <size(filesByCat[c])*100/totalFiles>% ");
125   }
126 
127   print("Category - Lines of Code - % of kernel");
128   printStartTimer("resourcesByCategory");
129   map[Resource file, int lines] fileLines = linesCount(filesByCat.file);
130   duration += printStopTimer("resourcesByCategory");
131   int totalLines = 0;
132   for(f <- fileLines.file) {
133     totalLines += fileLines[f];
134   }
135   for (c <- filesByCat.cat) {
136     int catLines = 0;
137     for (f <- filesByCat[c], file(_) := f) {
138        catLines += fileLines[f];
139     }
140     print("<c> - <catLines> - <catLines*100/totalLines>%");
141   }
142   return duration;
143 }
144 
145 public rel[str cat, Resource file] resourcesByCategory(set[Resource resource] resources, rel[
        str dir, str cat] dirCategories) {
146   rel[str cat, Resource resource] catResources = {};
147   for (r <- resources) {
148     for(dir <- domain(dirCategories)) {
149       if (startsWith(r.id.path, dir)) {
150         for(cat <- dirCategories[dir]) {
151           catResources += {<cat, r>};
152         }
153       }
154     }
155   }
156   return catResources;
157 }
158 
159 
160 public int statsFive(rel[Tag version, ChangeSet cs] versionChangesets, map[str, str] domainMap,
         list[Tag] releases) {
```

```
161   int duration = 0;
162   rel[Tag version, str email, str devverName, ChangeSet cs] result = {};
163   printStartTimer("calcDevelopers");
164   for (version <- releases, version in versionChangesets.version) {
165     result += calcDevelopers(version, "Author", {<cs@author ? cs.committer, cs>| cs <-
          versionChangesets[version]}, false)<0,2,3,4>;
166   }
167
168   duration += printRestartTimer("calcDevelopers");
169   for (version <- releases, version in result.version) {
170     map[set[str name] user, set[ChangeSet] cs] userChanges = getUserChangeSets(result[version])
          ;
171     map[set[str name] user, int count] userChangesCount = (usr : size(userChanges[usr]) | usr
          <- domain(userChanges));
172     duration += printRestartTimer("<version.name> - <size(userChanges.user)> users");
173     printMapOrderedOnRange(userChangesCount, 10);
174   }
175   startTimer();//quietly restart the timer
176   map[str mail, str company] emailCompany = (addr : domainMap[dom] | dom <- domain(domainMap),
        addr <- result.email, endsWith(addr, dom) );
177   duration += printRestartTimer("emailCompany");
178   rel[Tag version, str company, ChangeSet cs] versionCompanyChangesets =
179     {<version, emailCompany[email] ? "Unknown", cs> | <Tag version, str email, str devverName,
          ChangeSet cs> <- result};
180   duration += printRestartTimer("versionCompanyChangesets");
181   for (version <- releases, version in versionCompanyChangesets.version) {
182     rel[str company, ChangeSet cs] companyChangesets = versionCompanyChangesets[version];
183     map[str company, int changes] companyChangesCount = (company : size(companyChangesets[
          company]) | company <- domain(companyChangesets));
184     duration += printRestartTimer("\n<version.name> & <size(companyChangesets.company)>
          companies");
185     printMapOrderedOnRange(companyChangesCount, 10);
186   }
187   duration += printStopTimer("versionCompanyChangesets");
188   return duration;
189 }
190
191 public map[set[str name] user, set[ChangeSet] cs] getUserChangeSets(rel[str email, str
      devverName, ChangeSet cs] input) {
192   rel[str name, str email] devMail = {<user, email>  | email <- input.email, email != "", user
        <- input[email]<0>, user != ""};
193   rel[str name, ChangeSet cs] userCs = input[_];
194   rel[str email, ChangeSet cs] mailCs = input<0,2>;
195
196   devMail = solveBrokenRelations(devMail);
197
198   mailDev = devMail<1,0>;
199   map[str mail, set[str] userNames] mailUserNames = (email : mailDev[email] | email <- range(
        devMail));
200   map[set[str] userNames, set[str] mailAdresses] users = invert(mailUserNames);
201
202   map[set[str name] user, set[ChangeSet] cs] result =
203     (userNames : domainR(userCs, userNames)<1> + domainR(mailCs, users[userNames])<1> | set[str
          ] userNames <- users.userNames);
204
205   set[str] processedUsernames = {userName | userName <- users.userNames};
206   set[str] usersWithoutMail = domainX(userCs, processedUsernames)<0>;
```

```
207   result += ({user} : userCs[user]| user <- usersWithoutMail);
208
209   return result;
210 }
211
212 /**
213 *   Makes sure that each name has a relation with any email known for the same user,
214 * even if alternative usernames are used
215 * For example, if the tuple <Linus Torvalds, linus@linux.com> exists in the input along with
      two
216 * other tuples: <Linus Torvalds, linux@linux.com> and <Torvalds, linux@linux.com>, then the
      resulted set will
217 * have the additional tuple: <Torvalds, linus@linux.com>.
218 */
219 public rel[str name, str email] solveBrokenRelations(rel[str name, str email] input) {
220   r = input;
221   solve(r) {
222     r = r o invert(r) o r;
223   }
224   return r;
225 }
226
227 /*
228 * Reachability from set of start nodes with exclusion of certain nodes.
229 * Another implementation then the one in Graph.rsc, since the later uses
230 * the transitive closure and gets out of memory by large amount of data.
231 */
232 public set[&T] reach(Graph[&T] G, set[&T] Start, set[&T] Excl) {
233     set[&T] R = Start;
234   solve (R) {
235     R = R + G[R] - Excl;
236     }
237   return R;
238 }
239
240 public void printMapOrderedOnRange(map[value,int] content, int topMax) {
241   int totalCount = 0;
242   for(v <- domain(content)) {
243     totalCount += content[v];
244   }
245   for (int v <- reverse(quickSort(range(content))), d <- rangeR(content, {v}), topMax >= 0) {
246     topMax -= 1;
247     print("<d> - <v> - <v*100/totalCount>%");
248   }
249 }
250
251 public rel[Tag version, str action, str email, str devverName, ChangeSet cs] calcDevelopers(Tag
       version, str action,
252   rel[Info info, ChangeSet cs] devvers, bool parseCsMessage) {
253   rel[Tag version, str action, str email, str devverName, ChangeSet cs] results = {};
254   for (Info info <- devvers.info) {
255     str msg = (parseCsMessage ? (info.message ? "") : "") + " " + action + "-by: " + info.name;
256     for (/\s*<action:[^\s]*>-by:\s*<name:.*>/ := msg) {
257       if (/\s*"?<fname:[^\<"]+>"?\s\<<mail:[^\>]+>\>/ := name) {
258         results += {<version, action, toLowerCase(mail), fname, cs> | cs <- devvers[info]};
259       } else if (/\<<mail:[^\>]+>\>/ := name){
260         results += {<version, action, toLowerCase(mail), "", cs> | cs <- devvers[info]};
```

```
261       } else {
262         results += {<version, action, "", name, cs> | cs <- devvers[info]};
263       }
264     }
265   }
266   return results;
267 }
268 
269 //gets the changesets and measures the time
270 public list[ChangeSet] getChanges(Repository repo) {
271   printStartTimer("initChangesets");
272   changesets = getChangesets(repo);
273   printStopTimer("initChangesets");
274   return changesets;
275 }
276 
277 public MappingVars getMappings(InitVars initVars, map[Tag version, CheckoutUnit cunit]
      manualReleases) {
278   changesets = initVars.changesets;
279   repo = initVars.repo;
280   releases = initVars.releases;
281 
282   int total = 0;
283   printStartTimer("childParents");
284   childParents = {<cs.revision.id, m.parent.id> | cs <-changesets, revision(_, _) := cs.
        revision, m <- (cs.revision@mergeDetails ? {mergeParent(cs.revision.parent)})};
285   total += printRestartTimer("childParents");
286   revChangeset = (cs.revision.id : cs |cs <- changesets);
287   total += printRestartTimer("revChangeset");
288   tagChangeset = (t : cs | cs <- changesets, t <- (cs.revision@tags ? {}));
289   extraTags = (t : revChangeset[rev.id] | t <- manualReleases, cunit(Revision rev) :=
        manualReleases[t]);
290   //TODO we currently only support tags, but we might implement support for other checkoutunit
        types (e.g. date)
291   tagChangeset += extraTags;
292   if (size(extraTags) < size(manualReleases)) {
293     tagChangeset += (t : tagChangeset[symName] | t <- manualReleases, cunit(Tag symName) :=
          manualReleases[t]);
294   }
295   total += printRestartTimer("tagChangeset");
296 
297   totalVersionRevisions = getVersionRevisions(repo, childParents, tagChangeset, releases);
298   total += totalVersionRevisions[0];
299   versionRevisions = getOnlyUniqueRevisions(totalVersionRevisions[1], releases);
300   total += printRestartTimer("uniqueVersionRevisions");
301   rel[Tag version, ChangeSet cs] versionChangesets = versionRevisions o toRel(revChangeset);
302   total += printRestartTimer("versionChangesets");
303   versionNoMergesChangesets = {<version, cs> | version <- versionChangesets.version, cs <-
        versionChangesets[version], "mergeDetails" notin getAnnotations(cs.revision)};
304   total += printStopTimer("versionNoMergesChangesets");
305   print("Total duration:<total>");
306 
307   mappingVars = <childParents, tagChangeset, revChangeset, versionRevisions,
        versionNoMergesChangesets>;
308   return mappingVars;
309 }
310 
```

```
311
312 public tuple[int total, rel[Tag version, RevisionId revision] versionRevisions]
      getVersionRevisions(
313   Repository repo, rel[RevisionId child, RevisionId parent] childParents, map[Tag version,
        ChangeSet cs] tagChangeset, list[Tag] releases) {
314   int total = 0;
315
316   rel[Tag version, RevisionId revision] versionRevisions = {};
317   for (version <- releases, version in tagChangeset) {
318     set[RevisionId] reachable = {};
319     if (git(_,_,_) := repo) {
320       reachable = reach(childParents, {tagChangeset[version].revision.id});
321     } else {
322       reachable = reach(childParents, {tagChangeset[version].revision.id}, {tagChangeset[ver].
            revision.id| ver <- tagChangeset.version, ver != version});
323     }
324     versionRevisions += {<version, reaching> | reaching <- reachable};
325     total += printRestartTimer("versionRevision - <version> - <size(reachable)>");
326   }
327   return <total, versionRevisions>;
328 }
329 /**
330 * Makes sure that a revisionId is only referenced by one version. So if revision A is part of
      release 12 and 13, release 13 will
331 * no longer reference to it in the returned relation.
332 */
333 public rel[Tag version, RevisionId revision] getOnlyUniqueRevisions(rel[Tag version, RevisionId
       revision] versionRevisions, list[Tag] releases) {
334   rel[Tag version, RevisionId revision] results = {};
335
336   int i = 0;
337   for (version <- releases, version in versionRevisions.version) {
338     if (i == 0) {
339       results += {<version, rev> | rev <- versionRevisions[version]};
340     } else {
341       prev = versionRevisions[releases[i-1]];
342       results += {<version, rev> | rev <- versionRevisions[version], rev notin prev};
343     }
344     i += 1;
345   }
346   return results;
347 }
```

Listing 22: RMINER utilities (Utilities.rsc)

```
module Utilities

import Set;
import List;
import IO;
import experiments::scm::Timer;

//Timer functions
public datetime printStartTimer(str msg) {
  sTime = startTimer();
  print("started at <sTime> \t[<msg>]");
  return sTime;
}
public int printStopTimer(str msg) {
  dur = stopTimer();
  print("duration <dur> ms \t[<msg>]");
  return dur;
}
public int printRestartTimer(str msg) {
  dur = stopTimer();
  print("duration <dur> ms \t[<msg>], restarted <startTimer()>");
  return dur;
}

//Utility functions
public list[&T] quickSort(set[&T] st) {
  return quickSort(toList(st));
}
public list[&T] quickSort(list[&T] lst)
{
  if(size(lst) <= 1){
    return lst;
  }

  list[&T] less = [];
  list[&T] greater = [];
  &T pivot = lst[0];

  <pivot, lst> = takeOneFrom(lst);

  for(&T elm <- lst){
     if(elm <= pivot){
       less = [elm] + less;
     } else {
       greater = [elm] + greater;
     }
  }

  return quickSort(less) + pivot + quickSort(greater);
}
```