# SCQL: A Formal Model and a Query Language for Source Control Repositories

by

Abram James Hindle

B.Sc, University of Victoria, 2003

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisor: Dr. Daniel M. German

# Abstract

Source Control Repositories are used in most software projects to store revisions to source code files. These repositories operate at the file level and support multiple users. A generalized formal model of source control repositories is described herein. The model is a graph in which the different entities stored in the repository become vertices and their relationships become edges. We then define and implement Source Control Query Language (SCQL), a first order, and temporal logic based query language for source control repositories. We demonstrate how SCQL can be used to specify some questions and then evaluate them using the source control repositories of multiple software projects.

Supervisor: Dr. Daniel M. German (Department of Computer Science)

# Contents

# List of Figures

# List of Tables

# Acknowledgments

I would like to acknowledge the University of Victoria, NSERC, Advanced System Institute of British Columbia and Dr. Daniel German for providing me with the necessary funding and resources which allowed me to undertake this research.

# Dedication

I would like to dedicate this thesis to the Free Software Foundation for their tireless efforts in promoting a community of sharing.

# Chapter 1

# Introduction

Source Control Systems (SCSs) track the modification history of software projects. SCSs record who made the change, where the change occurred, and when the change occurred to a software project. By using this information it is possible to learn how a SCS is used and how its use relates to software evolution.

In recent years we have seen a growing interest in the retrieval of historical information from SCSs, for various purposes. Usually these systems extract information from the Concurrent Versioning System (CVS). CVS is widely used in the free/open source community; as well, several old, mature projects keep their history in CVS repositories. These repositories are available to researchers.

Typically a research project that wants to use this historical information starts with fact extraction. Facts are processed to create new *information*,

such as metrics [MFH02, Ger04b] or predictors [GDL04, HH04]. In some cases, this information is queried or visualized [GHJ04, Wu03].

Some projects store the extracted facts in a relational database ([LS03, GHJ04, FPG03]), and then use SQL queries to analyze that data. Others prefer to use plain text files and create small programs to answer specific questions [MFH02], while others query the CVS repository every time [Wu03].

One of the main disadvantages of these approaches is that it is difficult to query this historical data: a query has to be translated from the CVS history domain into a query on a set of tables that is used to represent the information; or the query has to be translated into a set of subroutines that are then executed on the plain text files. Furthermore, it is difficult to share data between tools as there are no standards for the storage or the querying of the data.

Researchers are not the only group interested in the history of a project. Developers and accompanying management can significantly benefit from improved access to project histories. For instance, a developer may want to know the last developer to contribute to a particular function, if developer A worked in the file that was previously modified by developer B, or which files have been modified at the same time as another file. A skilled user of a SCS may be able to answer each one of these queries with the help of some shell scripts, but it is likely that another SCS shall have a completely different interface, thus a solution for one SCS probably cannot be easily ported to another SCS.

In this thesis we propose a query language, Source Control Query Language (SCQL), that is domain specific to version control histories. This language uses an underlying abstract model to describe version control systems. We then evaluate SCQL on several mature, large projects.

The rationale and motivation for this work includes wanting:

- to ask questions of a SCS using first order logic by extending softChange ;
- to test some or part of Lehman's Laws of Software Evolution [Leh80];
- to ask temporal questions of a SCS;
- to query invariants of a SCS;
- to develop a provably correct system based on the data provided to it;
- to avoid the complexity of asking invariant queries using SQL or XQuery;
- to ask one query across multiple projects;
- to ask existential, universal or aggregated queries;
- to ask queries which could use time both concretely and relationally;
- to compare and contrast results of queries asked across multiple projects;
- to see what useful information can be queried by using a minimal set of facts (changes in a SCS).

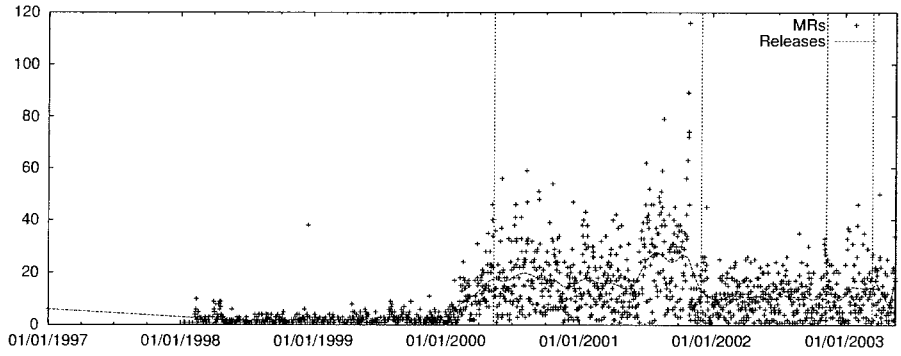


Figure 1.1: Number of Revisions and MRs over time for the Evolution project

A visual example of the data we are querying is provided in figure 1.1, this figure depicts the change history of the GNOME project's email client, Evolution, by the number of MRs per day over the entire life of the project. Note the differing behavior around releases versus normal development time.

## 1.1 Source Control Systems

A SCS is expected to track each change for all files under its control. In this thesis we will use the CVS nomenclature. A developer completes a task, which required her to modify several files. The developer then submits these changes to the SCS, in what we call a *Modification Record* , or MR (this process has also been called a transaction). A MR is atomic (conceptually the MR is atomic, even though it may not be implemented as atomic by the SCS). A change to one or more files is represented in the SCS by a *MR*, which consists of one *revision* per file modified. A MR is, therefore, a set of

one or more file revisions by a single developer. The SCS should allow its users to retrieve any given revision of a file. When given a date a SCS should determine what the most recent revisions were prior to that date for every file under its control.

SCSs vary in complexity and features in regards to how they track change. For example, SubVersion and CMVC (Configuration Management and Version Control from IBM) guarantee that every MR is atomic and that each file revision properly references its corresponding MR. CVS, on the other hand, does not keep track of MRs (some heuristics have been developed to rebuild these MRs, see [Ger04a, ZW04]).

Another important feature of SCSs is *branching*. Branching creates branches of revisions, which are used for parallel development. Branches are further explained in sections 1.1.1 and 1.1.2.

### 1.1.1 Version Naming

Throughout this thesis we will be using CVS nomenclature to refer to revisions stored in the SCS.

- Trunk - This refers to the main branch. The main branch will be considered the primary branch that work is done on or the branch which the SCS labels the Trunk. This is the main work flow of the project.

- HEAD - This refers to the latest revisions to the files on the main

branch in a repository. HEAD is the name of main branch in CVS.

- Branch - This refers to a line of development. A branch can be for only one file or project wide, branches allow parallel development in different version spaces. Branches are often merged back into the main branch (the trunk). A developer can explicitly state that she wants to start a branch off the main development *trunk* or off of another branch at a given point in the development. Any MR is then a part of either the trunk or a branch.

For CVS the first revision is 1.1, the last number is an integer and it increases in increments per each new revision to that file on that branch. If there is a branch, a branch ID is chosen and added to the end of the revision number. 1.1.1 would be a branch where as 1.1.1.1 would be the actual revision which produced that branch. Note this is per file, not for a group of files, groups of files are tracked via their branch names [Fou04b].

### 1.1.2 SCS operations

Operations on SCS specifically related to version control are:

- Checkout - A checkout operation is a request by the user to receive a copy of the files in the repository at a certain time or version. The version checked out is commonly the head of the repository; however, it is possible to checkout older versions of the files or branched versions of the files.

- Commit - A commit operation is a request to add changes to one or more files in the repository. In this thesis we will generally refer to commits as Modification Records. A commit is a set of revisions associated with multiple unique files all changed by the same author and submitted to the repository at the same time. Not all SCSs (such as CVS) record the group of files that were committed at one time.

- Update - An update takes a checked out working copy and updates the files of the working copy to the head, or the requested version, of the current branch. If revisions to a file occurred on the branch that was checked out then that checked out version updated.

- Merge - Merges occur when a branch is joined or rejoined to another branch. CVS does not record merges; merges must be done manually. During an update, CVS will try to merge source code with the checked out modified source code.

- Branching - Branching refers to the creation of a branch. A branch is a line of revisions separate from the main TRUNK. Modifying a branch means that the changes will not be seen on the HEAD of the main TRUNK. Branching usually occurs if the developers want to maintain an older release of the software or they wish to experiment more and use the repository concurrently without disturbing those programmers working on other branches or the TRUNK. Some SCSs, such as Darcs, create new branches per each revision.

- Report - Produces activity information regarding files and their revisions.

### 1.1.3 Entities

There are 4 main entities tracked by repositories: Authors, Files, Revisions and optionally Modification Records.

- **Author**:

  Authors are the creators; they create Modification Records, revisions, and files. Authors usually have varying access rights to the repository. Some have access to the entire repository, whereas, others only have access to certain modules, or read only access. Usually authors are associated with a repository using a unique identifier such as a user id.

- **File**:

  A file is basically a named location to attach revisions to. When an user checks out a copy of the repository it contains files which are composed of the revisions of a branch, up to the point requested. The IEEE standard on Software Configuration Management (SCM) (IEEE 828-1990 [IEE90, IEE98]) recognizes files as configuration items that are identified and named.

- **Revision**:

Revisions are changes to a file. The change can be content addition, content removal, content modification, file addition or file removal. Revisions are the basic building block of the SCS; due to branching they build either linear graphs, tree graphs, or acyclic graphs of revisions, associated with a file. Trees occur if there are branches of the file. Acyclic graphs occur if the branches merge back into a trunk or other branches. Revisions are usually grouped by file and then ordered by date of revision.

Revisions are usually handled as "text diffs". Diffs are patches to a file to produce a new version. This usually entails adding and removing lines. Binary files are generally differenced at the byte level, totally replaced, or differenced using a file-type specific diff (Subversion supports this feature).

- **Modification Record** :

  Modification Records (MRs) are groups of revisions added to the repository during one interval by one author. Some SCSs, such as CVS, do not store MRs; therefore, MRs must be rebuilt from the revision data.

  A CVS commit is considered to an MR.

### 1.1.4 SCSs

There are many SCSs available on the market:

**CVS** is the defacto SCS for Open Source projects. It supports revisions of files and does not track commits. Commits are non-atomic whereas revisions are atomic. CVS uses a centralized repository that can be used both locally and remotely. CVS does not track merges (merges are manual) but allows branching. Many SCSs, such as BitKeeper, export and import to and from CVS repositories. CVS is supported on many platforms. CVS is also used as a means of distribution in the BSD world where entire operating systems and their supporting programs are kept within a single repository. [Fou04a]

**Subversion** is a multi-platform SCS attempting to be the successor of CVS. Subversion is an Open Source SCS project intent on replacing CVS by offering better features while still retaining some of CVS's simplicity. Subversion uses a centralized repository much like CVS. Unlike CVS, Subversion supports renaming of files, tracking merges, and tracking directories. Subversion provides support for revisions of directories and symbolic links. It also claims to support atomic commits. [Col04]

**RCS** is not a full fledged SCS; it supports changes on a per file basis and was used as the basis for CVS. It is multi-platform and is usually only used to handle revisions to a single file or a small personal task. [Fou03]

**Darcs** is an Open Source distributed SCS like Arch or BitKeeper. Darcs supports a decentralized distributed repository which is Peer-2-Peer (P2P) in nature. A P2P-like repository is a repository which merges trees between multiple repositories rather than merging changes into a central repository.

However, it is intentionally kept clear and simple. Every revision is an addition to a tree of branches. Darcs is built upon a formal theory of patches which the implementation tries to adhere to as best as it can. [Rou05]

**Bitkeeper** is a multi-platform SCS that was formerly used in the development of the Linux Kernel [SC03]. Bitkeeper is developed by BitMover. BitKeeper is a P2P-like repository much like Darcs. BitKeeper supports renaming of files, merging, and tracking directories. BitKeeper is similar to other SCSs and supports similar actions to CVS. BitKeeper also supports CVS integration. [Inc04]

**Arch** is meant to be a BitKeeper replacement. It was initially made in protest to the use of Bitkeeper (a proprietary application) for Linux kernel development. At the moment, Arch is UNIX-centric although there are Win32 ports. It supports many of the features of Subversion, but it takes a less centralized approach; rather, it is P2P in nature [Lor04].

**Clearcase** is a SCS from Rational owned by IBM. It is similar to CVS in that it supports revisions to files, but it does not record groups of revisions. However, it supports the versioning of directories. Clearcase supports most of the CVS features, it even supports importing CVS repositories. Clearcase is very adaptable but is generally centralized. Clearcase is commercial software and is multi-platform.

**Perforce** is a multi-platform SCS from Perforce. It consists of a centralized repository like CVS that may be accessed both locally and remotely. It is easily integrated with many IDEs, such as Microsoft Visual Studio, Bor-

land JBuilder and Metroworks. Perforce has the ability to merge 3 branches into one. It is a multi-platform SCS. [Per04b, Per04a]

**Source Safe and Visual Source Safe** are SCSs from Microsoft which do not follow the common route taken by Software Configuration Management Systems (SCMS). Instead of using a revision metaphor, source safe uses a snapshot metaphor. Snapshots refer to the whole project at one point in time as if you had taken a picture of it. Source Safe is normally used locally, but has limited network support (3rd parties provide Source Safe over TCP/IP support). Unfortunately Source Safe is a commercial product currently available for Microsoft Windows only (although there are some 3rd party UNIX tools). It is popular because it is integrated with the Microsoft Visual Studio IDE [Cor04].

## 1.2 Previous Work

There is much previous work in the areas of software evolution, SCS fact extraction, SCS models, and temporal query languages.

Lehman's seminal paper, "Programs, Life Cycles and Laws of Software Evolution" [Leh80], provided much of the inspiration for this avenue of research, particularly those aspects which discover invariants about change throughout time.

### 1.2.1 Mining Software Repositories

This work is primarily related to work done in the Mining Software Repositories [Chu04], Software Evolution and Software Maintenance [Has05] research communities. SCQL was designed to answer questions relating to these research topics.

Mining Software Repositories (MSR) focuses on extracting, analyzing and interpreting facts extracted from a SCS, CMS, other software repositories or collections of releases [FG97]. MSR often deals with correlating the history of projects with current models of development. SCQL has been envisioned to calculate evolution based metrics. Godfrey et al. cover many aspects of software evolution: applying metrics to multiple releases of the Linux kernel [GT00], detecting code clones, extracting the evolution of software architecture and origin analysis [GDKZ04].

Mining repositories or release histories often consists of measuring versions or releases or entities related to those releases. In [FG97], Gall et al. use change rates to describe different behaviors seen in the extracted data. Lopez et al. [LFRMGB04] mined the relationships among developers to produce a social network graph. Xing et al. [XS04] attempted to correlate differences in extracted UML diagrams with the style of a software project.

### 1.2.2 Logics

The systems of logic employed in SCQL were first order and temporal logic. There are various kinds of formal logical systems that are relevant to this research both in formalism and the language itself.

First Order Logic or First-Order Predicate Calculus is a system of logic built from variables, constants, predicates, functions and logical connectives. Functions can be domain specific thus our model is defined with it and our language is designed to look like First Order Logic. And alternative to first order logic would be Second Order Logic, which is the "quantification over subsets of a domain, or functions from the domain into itself, rather than only over individual members of the domain" [Wik05].

Temporal Logic is a system of logic derived from tense logic, its purpose is to reason about entities with respect to time time. It is a logical system where elements of a domain exist in time and time relative questions relative can be asked. Tense logic has modal operators such as *some time before*, *some time after*, *always before*, and *always after*. Temporal Logic is expressible within first order logic.

Linear Temporal Logic (LTL) is a temporal logic that reasons about future events in a linear fashion. These paths can be walks of nodes or states. LTL is often used for reasoning about event traces. It can reason about sequences of events or states. Concurrent temporal logic (CTL) reasons about future events via branching paths which represent possible decisions. CTL has been used in log auditing [BGHS04].

### 1.2.3 Fact Extraction

Fact Extractors extract facts from a working SCS and allow you to either query these facts or place these facts in a more accessible format. Fact extractors are directly related to our work because we rely on a fact extractor (softChange ) to create a database from a repository of a project.

Fisher and Gall have discussed their fact extractor [FPG03], which is furthered refined in softChange by German [Ger04a] and in another extractor by Zimm et al. [ZW04].

The implementation of our work, depends heavily upon softChange [GHJ04]. softChange is a fact extractor for CVS developed by Dr. Daniel German. It attempts to extract MRs from CVS repositories. The MR extracting algorithm is explained in German's paper "Mining CVS repositories, the softChange Experience" [Ger04a], which rebuilds MRs from revisions from a CVS repository.

Kemerer and Slaughter[KS99] discuss methods of fact extraction and data analysis appropriate to MSR. The paper goes into great detail about the various techniques and methods used to study software maintenance and software evolution. Other research goes into detail about source code entities and the ASTs of the actual source code [FSG04].

### 1.2.4 SCS models

SCQL includes a model of SCS that was based upon the models and SCM standards mentioned in this section.

Conradi and Westfechtel provide an overview of SCMS and how they handle versioning [CW98, CW97]. This survey supports the view that revisions form an acyclic graph. This paper is a good overview of what is needed in a general model of SCSs.

Render and Campbell [RC91] proposed an object oriented model of SCS and Software Configuration Management. Their model was defined without a query language and shared many similarities with the model proposed in this thesis; both models are to a certain extent object oriented. The model consisted of many entity types, including composite and aggregated types. This complexity made it difficult to reason about the model. This model was considered when developing the SCS model for SCQL.

The IEEE has provided standards and conventions for SCMs but do not provide much information regarding how revisioning, versioning, patching or taking snapshots should be handled [IEE90, IEE98].

There are many CMS and Version Control models in the literature. Dart [Dar91], discusses the main ideas behind CMSs, which is to identify artifacts and elements of the project and to store these elements. Many of the models focus more on modeling version control rather than modeling CMSs[CW97, Sci94]; there is some focus on the version control of architectural entities such as objects or classes rather than source code [MZY01, BM88].

### 1.2.5 Log Auditing

In areas related to the use of our model and engine, we have found work that uses CTL, such as "Rule-Based Runtime Verification" [BGHS04]. This paper illustrated how temporal logic can be used in auditing system events to flag behavior that could be dangerous. "Log Auditing through Model Checking" also provided an excellent example of using temporal logic for auditing of events [RG01]. This is relevant as logic could can be used to verify that certain behaviors are or are not taking place. Essentially this was one of the aims of this research.

### 1.2.6 Query Languages

SCQL allows user interaction with the model via a query language. The SCQL query language allows temporal and relation queries, some of which was inspired by these query languages listed. Many of the following query languages were looked at either because they queried a similar domain (graphs, time) or because they related to MSR.

Amann and Scholl's [AS92] paper "Gram: a graph data model and query language", describes a query language used to query graphs that model hypertext documents. The query language described is powerful in that it supports recursive queries. The query language is based on relational algebra as it seems to be inspired by SQL. The Gram query language focused on querying walks and paths in graph models. This relates to the present

research because it is a query language specifically built for graphs. Unfortunately, it does not focus on first order logic (it is heavily focused on relational algebra). Also time semantics would have to be hardwired into the graph. The language was inappropriate because of the lack of support for invariants and temporal constraints.

Snodgrass produced a temporal query language named TQuel [Sno87]. TQuel is based on temporal logic and is an extension of the earlier query language Quel. TQuel supports aggregate functions such as summations, average, minimum, maximum, etc.

ATSQL as described in "Querying ATSQL Databases with Temporal Logic" [CTB01] is a temporal query language based on SQL. It is intended to query temporal RDBMS. In this paper the authors describe how temporal logic relations are translated into ATSQL and vice versa. ATSQL works on abstract temporal databases (e.g., tuples with an inferred time) with operators such as contains, meets, overlaps, and precedes defined temporally.

Hipikat [CM03] is an excellent example of a SCS query system. Effectively Hipikat acts as a textual search engine for software trails which are extracted from different sources including SCSs and mailing lists. Hipikat is one of the few query systems that was actually related to mining SCSs.

XPath was evaluated as a possible query language for the model or as a back end. Cassidy [Cas03], suggested extensions to XPath for directed graphs as well as strategies for using XPath with directed graphs. These were used for querying Linguistic Annotations. The structure of the data was

considered more complex than the hierarchical XML model was comfortable handling.

### 1.2.7 Temporal Databases

Temporal Databases were looked at since SCQL can use temporal quantifiers. There has been a lot of research regarding temporal databases over the past 3 decades. Much of the work has focused on databases of tuples, which have either a relative time or a fixed time. This time might be extended by a period over which a state, event, or object exists.

Gadia elaborated on TQuel and discussed TQuels weaknesses in [Gad88]. This paper discusses the application of relational algebra to temporal queries and proposes a data model that works well with temporal queries and relational algebra.

### 1.2.8 Metrics

Metrics are software measurements. These are known algorithms that process some entity or group of entities and produce a measurable quantifiable result. SCQL has been used to define some metrics [GH05], as well it has been used to calculate metrics. Due to the fine grained granularity of SCQL, metrics are very important to describe the entities.

Metrics are heavily used to study software evolution because they allow users to measure and compare releases of projects to each other (as well as

comparing projects against each other). Measurements allow us to compare and contrast entities and projects.

Metrics are used both for measurement and prediction. Examples of metrics used in software evolution include:

- coupling metrics derived from historical commit data [GJK98];
- metrics for predicting or identifying design flaws [Mar04];
- metrics that attempt to predict change [GDL04, Kun04, HH04];

There is much research in the application of metrics to software evolution [LPR+97, MD01b, MD01a]. These metrics range from using metrics to describe differences in releases, to metrics which measure aspects of change in a system.

Metrics that measure the actual changes rather than comparing the system before and after an event, are rather relevant to this research. Metrics which measure the changes (revisions and diffs) themselves have been proposed by both Ball et. al [BAHS97], Draheim [DP03] and German et al. [GH05]. These are metrics which measure and describe fine grain changes rather than just providing a difference of a metric between two versions.

## 1.3 Hypotheses

We are attempting to produce and evaluate a query language and model of SCSs.

Hypothesis 1: Can we produce a query language that allows us to compare multiple projects? That is, one query should work and execute on any project. This hypothesis suggests that the queries will be relative and operate on an abstraction of a SCS for a project.

Hypothesis 2: Can we produce a Model which can represent multiple projects such that we can effectively query multiple projects with the same query? Our model should focus on what hasn't been done that well by the current representations, which is a formal model, and non-relational algebra based model (not SQL tables and queries).

# Chapter 2

# Model

After using softChange it became evident that the relational model used was quite limited. Not only were SQL queries difficult to deal with, the database schema was the model of the SCS itself. However, softChange is very useful for certain queries which use aggregates or string matches. In this section we will discuss a model that affords questions about logical invariants found in SCSs. We will call this model and query language, Source Control Query Language (SCQL).

The purpose of the model is to create a system in which expressive and powerful questions can be stated and evaluated, specifically, questions about invariants in the SCS. In particular, we are interested in a system that supports the ability to ask questions with respect to time. The model must be of reasonable complexity so that interesting invariant based questions may be asked and answered, while being simple enough that questions can be

expressive without excessive complexity.

Questions regarding time are often relative (i.e., did events occur before, after, or during another event, did event $A$ immediately precede event $B$?) We will assume that the concrete time associated with entities is unique and atomic with respect to revisions. There is effectively no "during" for entities of the same type. Entities occur before, after, or at the same time relative to other entities. We need a model where we can use a subset of temporal logic (before and after) easily and efficiently.

## 2.1 Characteristic Graph of a Source Code Repository

We have decided to model an instance of a SCS as a graph. Graph nodes are used to represent the entities (MRs, revisions, files, and Authors) and their edges to represent the interrelationships (including some of the before, and after relationships). Given an instance of a SCS, we can create a directed graph that represents it. Attributes of these entities can be expressed as maps.

We are interested in a process that, given a query on an instance of a SCS, we can translate this query into a graph query. We can then answer the original query by solving the graph query.

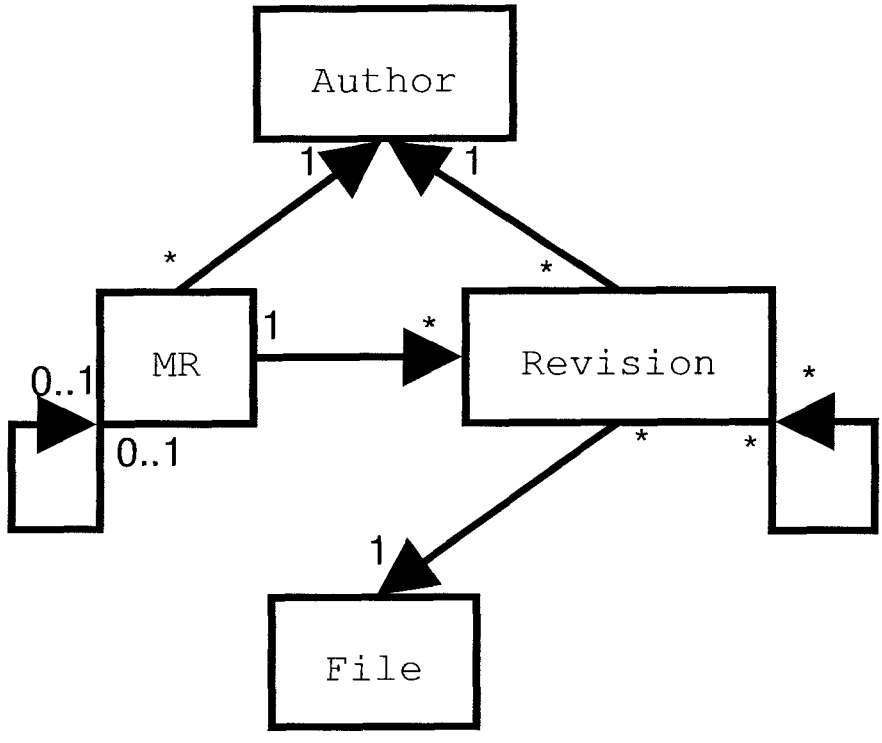


Figure 2.1: Model Node / Edge cardinalities [HG05]

## 2.2 Entities

The data model for SCQL contains four different types of entities: MRs, Revisions, Files and Authors. Figure 2.1, describes the cardinalities of nodes and edges in the graph. Note how both MRs and Revisions link to Authors and how only Revisions reference Files.

- **MRs.** The MR entity models a modification request. MRs have attributes such as log comments and timestamps of their revisions (date, time). In our model, MRs are atomic; therefore, no two MRs have the same timestamp and each MR has a unique ID. We created a partial

relation based on this timestamp: for any given pair of different MRs $(a, b)$, $a$ occurs before $b$, or $b$ occurs before $a$. Thus when one or more MRs exist, there will be one MR which has no MRs preceding it and there will be one MR with no MRs occurring after it (of course, these 2 cases could be the same MR if there was only one MR in the system). The set of all MRs in an instance is called $\mathbb{MR}$. MRs are linked to the next MR in time by an edge. The purpose of time being expressed by an edge is to explicitly encode in the graph structure the importance of this partial relation between MRs; time is the navigable relation between MRs.

Authors are related to MRs in that there is only one author for each MR. The edge extends from the MR to the author. MRs are related to files through their revisions; therefore, an MR might be related to many files. Although an MR's author may be derived from the author of its revisions, an edge extending from an MR to an author allows for greater graph navigability. A single MR cannot contain (by contain we mean an edge extends from the MR to the Revision) more than one revision of the same file.

MRs are often built from revisions as some SCSs like CVS do not track MRs[Ger04b].

- **Revisions.** Each revision corresponds to one and only one file, and each revision can be uniquely identified by a revision identifier and by

the file it corresponds with.

Revisions are much like MRs; they all have unique timestamps. If there exists two revisions with the same time in a repository we assume, due to the atomicity of adding a revision, one revision had to occur before the other. Thus, a unique timestamp is assigned to each revision, even if they are part of the same MR. This guarantees that we can always determine which revision occurred first. Revisions could be interleaved in time with revisions of another MR. Revisions have attributes such as the *diff* of the change, the lines added in the revision, and the lines removed in the revision. The amount of meta-data stored in the SCS varies widely from one implementation to the next. Meta-data also depends on the kind of the file that it modifies. For instance, it might not make any sense to compute the number of lines deleted from a binary file. The set of all revisions in the graph is denoted as $\mathbb{R}evision$.

Revisions are also associated with an author. They are associated with the same author as their MR. As such, all revisions of one MR have the same author. MRs often don't exist in the original SCS, rather, the author is inferred through revisions; that is, revisions comprise the actual data upon which MRs are built. Revisions are not directly linked through time because they have more complex relationships with each other. Surrounding each file is an acyclic graph of revisions that might branch like a tree or merge back together like a stream. Revisions are linked to other branching or consecutive revisions of the same file.

- **Files.** A file is simply a name and location at which revisions are added [IEE90, IEE98]. Each file serves as the "anchor" of an acyclic graph of revisions. A version of a file is the result of applying patches in chronological order from the root of a tree to the requested version. It is important to mention that, even though SCSs track files, they could also track other types of objects (such as functions or classes). Files have a filename attribute, the full path, which is guaranteed to be unique for each file in a project. This filename may be used to derive such attributes as the basename, the extension, and the directory name. Files can be associated with a module if the module is named and identified. For simplicity, we will assume that each file has a unique timestamp. The timestamp is the timestamp of the first revision to that file, and again, due to the atomicity of revisions, file timestamps are guaranteed to be unique over all files. The timestamp can be interpreted as the time in which the file was added to the system. Questions such as "which files were created before this file was created?", could then be asked. The set of all files in the graph is denoted as $\mathbb{F}ile$.

- **Authors.** Authors are simple entities. Their main attribute is their unique userid. Authors may also have many attributes (such as the name of the person or their email). There is only one author associated with one MR and all of the revisions of that MR; however, one author could also be associated with several MRs. Authors are not directly related to files as files can be revised by multiple authors. Authors, like

files, are timestamped with the timestamp of the first revisions they contribute. The set of all authors is denoted as $\mathbb{A}uthor$.

The cardinality of relations could change given new SCSs. For instance the number of authors related to an MR could be more than one if the SCS supports the idea of collaborative work or pair programming.

## 2.3 Formalizing the characteristic graph

Formally we define the characteristics graph G of a SCS as a directed graph:

$$G = (V, E)$$

where:

$$V = \mathbb{MR} \cup \mathbb{F}ile \cup \mathbb{A}uthor \cup \mathbb{R}evision$$

$e = (v_1, v_2) \in E$ if

- $v_1 \in \mathbb{R}evision, v_2 \in \mathbb{F}ile$ iff $v_1$ is a revision of $v_2$, or
- $v_1 \in \mathbb{R}evision, v_2 \in \mathbb{A}uthor$ iff $v_2$ is an author of $v_1$, or
- $v_1 \in \mathbb{MR}, v_2 \in \mathbb{A}uthor$ iff $v_2$ is an author of $v_1$, or
- $v_1 \in \mathbb{MR}, v_2 \in \mathbb{R}evision$ iff $v_1$ contains revision $v_2$, or
- $v_1, v_2 \in \mathbb{MR}$ iff $v_1$ is the MR immediately before $v_2$, or

- $v_1, v_2 \in \mathbb{R}evision$ iff $v_1$ and $v_2$ correspond to the same file $f$ and $v_1$ is a revision of $f$ and a parent revision of $v_2$.

### 2.3.1 Primitives

There are 6 data types in our model:

- Vertices – Entities
- Edges – Relationships
- Sets of Vertices – Sets of entities (abstraction of edges)
- Numbers – Used for numerical questions (aggregate functions and time)
- Strings – Much of the data in the repository is string data and must be represented in the functions and attributes.
- Booleans – Used for invariants and first order logic.

Primitive "isa" functions determine whether a vertice belongs to one of the entity subsets. Assume $\phi$ is an entity; therefore, $\phi \in V$. We define: $isaMR(\phi)$ (is $\phi$ a MR), $isaRevision(\phi)$ (is $\phi$ a Revision), $isaFile(\phi)$ (is $\phi$ a File), and $isaAuthor(\phi)$ (is $\phi$ an Author). We will describe the operations with primitives in detail, as they directly relate to the query language operations described later.

Binary and Unary Boolean Operators used in our model operate on boolean values and produce boolean values. Let $P$ and $Q$ be boolean propositions.

- $P = Q$ – Bijection
- $P \implies Q$ – Implication
- $P \wedge Q$ – Logical And
- $P \vee Q$ – Logical Or
- $\neg P$ – Boolean Not. *Not* was included since we are not focusing on Horne clauses like Prolog does.
- $(P)$ – Parentheses - used for order of operations

Vertex operators operate on vertices and return boolean values. Let $\phi$ and $\theta$ be vertices in $V$.

- $\phi = \theta$ – $\phi$ and $\theta$ are the same vertex.
- $\phi\neg = \theta$ – $\phi$ and $\theta$ are not the same vertex.

Subset ($\mathbb{S} \in \{\mathbb{MR}, \mathbb{A}uthor, \mathbb{F}ile, \mathbb{R}evision\}$) and summation ($\sum$) operators used in the model operate on strict subsets of entities of the same type. Let $\alpha \subseteq \mathbb{S}$, where $\alpha$ is a subset of one of the subsets $\mathbb{MR}$, $\mathbb{A}uthor$, $\mathbb{F}ile$ or $\mathbb{R}evision$. $\alpha$ cannot contain entities of two or more different types. Let $\gamma$ be a numeric expression (a function that maps an entity to a numeric value).

- $\{\phi \in \alpha | P(\phi)\}$ – Produces a subset of $\alpha$ such that $P(\phi)$ evaluates to true for each element in this subset.

- $\sum_{\phi \in \alpha} \gamma(\phi)$ – Produces a summation of the numeric values returned from $\gamma(\phi)$ for each $\phi \in \alpha$. $\gamma(\phi)$ is a numeric function that optionally uses $\phi$ as a parameter.

Existential and universal operators operate on edges, vertices, subsets, and propositions. Let $\phi$ and $\theta$ be vertices.

- $\exists \phi \in \alpha(P(\phi))$ – The existential operator implies that $\phi$ exists in the finite set $\alpha$ ($\phi \in \alpha$) where $P(\phi)$ is true.

- $\forall \phi \in \alpha(P(\phi))$ – The universal operator implies that $P(\phi)$ is true for all elements in the finite set $\alpha$. The universal operator can be derived from $\neg \exists \phi \in \alpha(\neg P(\phi))$

The following operators operate on numeric expressions:

- $x \bullet y$ where $\bullet \in \{+, -, *, /\}$ – These operators return numeric values and use numeric parameters

- $x \bullet y$ where $\bullet \in \{=, \neq, >, <, <=, >=\}$ – These operators return boolean values and use numeric parameters

These are the string primitive functions (let $i, j \in \mathbb{R}$, let $k, l \in \mathbb{Z}$):

- $numberToStr(i) \mapsto \mathbb{S}tring$ – Returns a string representation of the numeric value $i$

- $length(\phi) \mapsto \mathbb{R}$ – Returns the number of characters in the string $\phi$.

- $substr(\phi, k, l) \mapsto \mathbb{S}tring$ – Returns the substring starting at character $k$ (0 indexed) that is $l$ characters long (if $l + k >= length(\phi)$ then the string is truncated to $length(\phi) - i$ characters, if $i >= length(\phi)$ then undefined is returned)

- $eq(\phi, \theta) \mapsto \mathbb{B}$ – Returns true if $\phi$ and $\theta$ are the same string

- $concat(\phi, \theta) \mapsto \mathbb{S}tring$ – Returns a new string that is the in-order concatenation of $\phi$ and $\theta$ (with no delimiter).

- $matches(\phi, \theta) \mapsto \mathbb{B}$ – Returns true if $\theta$ is a substring of $\phi$

These are composite functions, composed of primitives. Let $\mathbb{S}$ be a subset of V, $\phi$, $\theta$ be vertices and let $\gamma$ be a function that maps to a numeric value.

- $isEdge(\phi, \theta) \mapsto \mathbb{B} \iff (\phi, \theta) \in E$ – Is there an edge between $\phi$ and $\theta$ (there are no self referencing links so $isEdge(\phi, \phi)$ is always false)

- $sum(\mathbb{S}, \gamma) \mapsto \mathbb{R} \implies \sum_{\phi \in \mathbb{S}} \gamma(\phi)$ – Summation of the function $\gamma$ applied to all elements of $\mathbb{S}$.

- $count(\mathbb{S}) \mapsto \mathbb{R} \implies sum(\mathbb{S}, f)$ – Counts all the elements of $\mathbb{S}$, where $f(x) = 1$ for all $x$ (this is equivalent to $||\mathbb{S}||$, the number of elements in the subset)

- $avg(\mathbb{S}, \gamma) \mapsto \mathbb{R} \implies sum(\mathbb{S}, \gamma)/count(\mathbb{S})$ – Average of the results of the numeric predicate $\gamma$ applied against all elements of the subset $\mathbb{S}$

- $max(\mathbb{S}, \gamma) \mapsto x | \forall \phi \in \mathbb{S}(x >= \phi)$ – Returns the maximum value of all elements of $\mathbb{S}$ applied to $\gamma$ ($x$ is the maximal numeric value).

- $min(\mathbb{S}, \gamma) \mapsto x | \forall \phi \in \mathbb{S}(x <= \phi)$ – Returns the minimum value of all elements of $\mathbb{S}$ applied to $\gamma$ ($x$ is the minimal numeric value).

For the following functions let $\phi \in \mathbb{MR}$, $\theta, \theta_2 \in \mathbb{R}evision$, $\tau \in \mathbb{F}ile$, and $\psi \in \mathbb{A}uthor$:

- $isAuthorOf(\psi, \phi) \mapsto \mathbb{B} \iff isEdge(\phi, \psi)$ – Is $\psi$ the author of the MR $\phi$?

- $isAuthorOf(\psi, \theta) \mapsto \mathbb{B} \iff isEdge(\theta, \psi)$ – Is $\psi$ the author of the revision $\theta$?

- $isMROf(\phi, \psi) \mapsto \mathbb{B} \iff isEdge(\phi, \psi)$ – Is the MR $\phi$ created by the author $\psi$?

- $isMROf(\phi, \theta) \mapsto \mathbb{B} \iff isEdge(\phi, \theta)$ – Is the MR $\phi$ the MR of revision $\theta$?

- $isRevisionOf(\theta, \psi) \mapsto \mathbb{B} \iff isEdge(\phi, \psi)$ – Is the revision $\theta$ created by the author $\psi$?

- $isRevisionOf(\theta, \phi) \mapsto \mathbb{B} \iff isMROf(\phi, \theta)$ – Is the revision $\theta$ part of the MR $\phi$?

- $isRevisionOf(\theta, \tau) \mapsto \mathbb{B} \iff isEdge(\theta, \tau)$ – Is the revision $\theta$ a revision of the file $\phi$?

- $isFileOf(\tau, \theta) \mapsto \mathbb{B} \iff isEdge(\theta, \tau)$ – Is the file $\tau$ associated with the revision $\theta$?

- $isFileOf(\tau, \phi) \mapsto \mathbb{B} \iff$

$$\exists a \in \mathbb{R}evision \; s.t.$$

$$(isRevisionOf(a, \phi) \wedge isFileof(\tau, a))$$

– Is there a revision of file $\tau$ that is a revision of the MR $\phi$?

- $revBefore(\phi, \theta) \mapsto \mathbb{B} \implies$

$$(\exists \tau \in \mathbb{F}ile(isRevisionOf(\phi, \tau) \wedge isRevisionOf(\theta, \tau) \wedge$$

$$isEdge(\phi, \theta) \vee$$

$$\exists (\psi, \theta) \in E \; s.t. \; (revBefore(\phi, \psi)))$$

– Does revision $\theta$ occur (revision wise) before revision $\theta_2$, and do both $\theta$ and $\theta_2$ modify the same file?

- $revAfter(\theta, \theta_2) \mapsto \mathbb{B} \iff revBefore(\theta_2, \theta)$ – Does revision $\theta$ occur after revision $\theta_2$ and both ,$\theta$ and $\theta_2$, modify the same file?

- $isMROf(\phi, \tau) \mapsto \mathbb{B} \iff \exists isFileOf(\tau, \phi)$ – Is the file $\tau$ a file of a revision of $\phi$?

We implement attributes using maps. Attributes may be subsets, strings, numerics, or booleans. Another assumption is that the output of a mapping is only valid if a node or edge of a correct type is used as an index to the map. All other inputs produce an undefined value. More attributes can be added at any time but these are the expected attributes.

Attributes that return entities return a subset of entities even if this subset only contains one element. The motivation behind this decision is that scope is created each time a new entity is accessed. This makes for consistent access to entities. Since sets are returned we use plural function names. One valuable aspect of returning sets is that empty sets can be returned and handled uniformly via universal and existential scopes. Repeated attribute definitions can be assumed to be combined together using logical ORs.

- $time(\phi) \mapsto \mathbb{R} \implies$ – If $\phi \in V$ return the time attribute of $\phi$ ;
- $\mathbb{MR}$ Attributes
  - $mrID(\phi) \mapsto \mathbb{S}tring$ – Returns the MR Identifier of an MR. Maps $\phi$ to a string if $\phi \in \mathbb{MR}$ return the mrID attribute of $\phi$ otherwise return undefined
  - $logEntry(\phi) \mapsto \mathbb{S}tring$ – Returns the log entry string of an MR if $\phi \in \mathbb{MR}$
  - $authorname(\phi) \mapsto \mathbb{S}tring$ – Returns the author name string of an MR (if $\phi \in \mathbb{MR} \vee \phi \in \mathbb{A}uthor$ returns the name of the author)

- $authors(\phi) \mapsto \mathbb{S}$ – Returns a set containing all the authors (one author) of the MR (if $\phi \in \mathbb{MR}$ returns $\{\theta \in \mathbb{A}uthor | isEdge(\phi, \theta)\}$)
- $revisions(\phi) \mapsto \mathbb{S}$ – Returns a set of Revisions that were associated to the MR ( if $\phi \in \mathbb{MR}$ returns $\{\theta \in \mathbb{R}evision | isRevisionOf(\theta, \phi)\}$)
- $files(\phi) \mapsto \mathbb{S}$ – Returns a set of files of the revisions of the MR ( if $\phi \in \mathbb{MR}$ returns $\{\theta \in \mathbb{F}ile | isFileOf(\theta, \phi)\}$)
- $nextMRs(\phi) \mapsto \mathbb{S}$ – Returns a set of the MRs consisting of the next MR in time (if $\phi \in \mathbb{MR}$ returns $\{\theta \in \mathbb{MR} | isEdge(\phi, \theta)\}$)
- $prevMRs(\phi) \mapsto \mathbb{S} \implies$ – Returns a set of the MRs which were the directly previous MR in time (if $\phi \in \mathbb{MR}$ returns $\{\theta \in \mathbb{MR} | isEdge(\phi, \theta)\}$)

- $\mathbb{R}evision$ Attributes
  - $revisionID(\theta) \mapsto \mathbb{S}tring$ – Returns the revisionID of the revision (if $\theta \in \mathbb{R}evision$ returns the revisionID attribute of $\theta$)
  - $daterev(\theta) \mapsto \mathbb{S}tring$ – Returns the date of revision string (if $\theta \in \mathbb{R}evision$ return the date attribute of $\theta$)
  - $timerev(\theta) \mapsto \mathbb{S}tring$ – Returns the string representation of the time during the day of the revision (if $\theta \in \mathbb{R}evision$ return the time attribute of $\theta$)
  - $linesAdded(\theta) \mapsto \mathbb{R}$ – Returns the number of lines added in this MR (if $\theta \in \mathbb{R}evision$ return the linesadded attribute of $\theta$)

- $linesRemoved(\theta) \mapsto \mathbb{R}$ – Returns the number of lines removed (if $\theta \in \mathbb{R}evision$ return the linesremoved attribute of $\theta$)
- $diff(\theta) \mapsto \mathbb{S}tring$ – Returns a string of the diff contained in the repository between revisions (if $\theta \in \mathbb{R}evision$ return the diff attribute of $\theta$)
- $files(\theta) \mapsto \mathbb{S}$ – Returns a subset containing the File entity the revision modified (if $\theta \in \mathbb{R}evision$ return $\{\phi \in \mathbb{F}ile | isFileOf(\phi, \theta)\}$)
- $mrs(\theta) \mapsto \mathbb{S}$ – Returns a subset containing the MR entity that is related to this revision (if $\theta \in \mathbb{R}evision$ return $\{\phi \in \mathbb{MR} | isMROf(\phi, \theta)\}$)
- $authors(\theta) \mapsto \mathbb{S}$ – Returns a subset containing the author who authored this revision (if $\theta \in \mathbb{R}evision$ return $\{\phi \in \mathbb{A}uthor | isAuthorOf(\phi, \theta)\}$)
- $nextRevisions(\theta) \mapsto \mathbb{S}$ – Returns a subset containing all the child ($revAfter$) revisions of this revision (if $\theta \in \mathbb{R}evision$ returns $\{\phi \in \mathbb{R}evision | revAfter(\theta, \phi)\}$)
- $prevRevisions(\theta) \mapsto \mathbb{S}$ - Returns a subset containing all the parent ($revBefore$) revisions of this revision (if $\theta \in \mathbb{R}evision$ returns $\{\phi \in \mathbb{R}evision | revBefore(\theta, \phi)\}$)

- $\mathbb{F}ile$ Attributes
  - $filename(\tau) \mapsto \mathbb{S}tring$ – Returns the string of the filename of the File (if $\tau \in \mathbb{F}ile$ return the filename attribute of $\tau$)
  - $module(\tau) \mapsto \mathbb{S}tring$ – Returns the string of the module with

which this file is associated (if $\tau \in \mathbb{F}ile$ return the module attribute of $\tau$)

- $mrs(\tau) \mapsto \mathbb{S}$ – Returns the subset of MRs which have a revision of this file (if $\tau \in \mathbb{F}ile$ return $\{\theta \in \mathbb{MR}|isMROf(\theta, \tau)\}$)
- $revisions(\tau) \mapsto \mathbb{S}$ – Returns all the revisions associated with this file (if $\tau \in \mathbb{F}ile$ return $\{\theta \in \mathbb{R}evision|isRevisionOf(\theta, \tau)\}$)
- $filesInModule(\tau) \mapsto \mathbb{S}$ – Returns a subset of files which belong to the same module as this file (if $\tau \in \mathbb{F}ile$ return $\{\theta \in \mathbb{F}ile|eq(module(\tau), module(\theta))\}$)

- $\mathbb{A}uthor$ Attributes
  - $userid(\psi) \mapsto \mathbb{S}tring$ – Returns the string of the author's userid used in the SCS (if $\psi \in \mathbb{A}uthor$ return the userid attribute of $\psi$)
  - $mrs(\psi) \mapsto \mathbb{S}$ – Returns all of the MRs authored by this author (if $\psi \in \mathbb{A}uthor$ return $\{\theta \in \mathbb{MR}|isMROf(\theta, \psi)\}$)
  - $revisions(\psi) \mapsto \mathbb{S}$ – Returns all of the revisions authored by this author (if $\psi \in \mathbb{A}uthor$ return $\{\theta \in \mathbb{R}evision|isRevisionOf(\theta, \psi)\}$)

### 2.3.2 Time

For MRs, if $x, y \in \mathbb{MR}$ and $\exists(x, y) \in E$, then $x$ comes immediately before $y$ and $y$ comes immediately after $x$. Therefore, given two different MRs, $x, y \in \mathbb{MR}$ , $x$ is before $y$ if there exists a walk from $x$ to $y$ along edges

$(u, v) \in \{(\phi, \theta) \in E | \phi, \theta \in \mathbb{MR}\}$. This is recursively expressed using the predicate:

$$before(x, y) \iff isMR(x) \wedge isMR(y) \wedge \exists (a, y) \in E($$
$$isMR(a) \wedge (isEdge(x, y) \vee before(x, a)))$$

After is defined for MRs:

$$after(x, y) \iff isMR(x) \wedge isMR(y) \wedge before(y, x)$$

For other entities and MRs $before(x, y) \iff time(x) < time(y)$ and $after(x, y) \iff before(y, x)$.

Version wise we can traverse the edges between revisions to find the trails of change for a particular file $f$:

$$E_f = \{(u, v) \in E | (isRevisionOf(v, f) \wedge isRevisionOf(v, f))\}$$

With edges in $E_f$ we can traverse edges for a particular file. If there is a walk using the edges of $E_f$, for all those vertices in that walk, each vertex belongs to $\mathbb{R}evision$ and each vertex links to the same vertex $x$ in $\mathbb{F}ile$. Our graph creation rules dictate that files are only related in time to their first revision thus this invariant will be true for any properly made graph:

$\forall \phi, \theta \in \mathbb{F}ile(\phi \neq \theta \implies time(\phi) \neq time(\theta))$

Authors have no time properties associated with them other than the time of revisions and MRs they produce. We have decided to give an author a time of their first MR. This allows us to compare when authors join a given project:

$$\forall \phi, \theta \in \mathbb{A}uthor(\phi \neq \theta \implies time(\phi) \neq time(\theta))$$

## 2.4 Extraction and Creation

The general algorithm for extracting and creating a graph from a SCS is (as described in the paper by Hindle et al. [HG05]):

- Each file becomes a vertex in $\mathbb{F}ile$.
- Each author becomes a vertex in $\mathbb{A}uthor$.
- Each revision becomes a vertex in $\mathbb{R}evision$. Assign revisions unique timestamps and connect each revision its corresponding author and file.
- Create vertices for each MR. The MR inherits the timestamp from its first file revision. Associate the MR to its author.
- Each MR is then connected to the next MR (according to their timestamp), if it exists.

- For each file, connect each revision to the next revision of the file, version-wise. If branching is taken into account, only revisions in the same branch are connected in this manner, and then branching and merging points are connected.

### 2.4.1 Detailed Graph Generation

The following is an overview of how to create an instance of the equivalent graph of a CVS repository.

- Extract all the revisions from the repository. Each revision becomes a vertex.
- For each revision, create a file vertex for the file the revision is associated with only if a vertex for such a file does not already exist.
- For each revision, create a revision vertex. Create an author vertex of the revision's author if one does not already exist. Create an edge from the revision vertex to the author vertex.
- Create an edge from the revision vertex to the file vertex of the files with which it is associated.
- Run the MR extractor algorithm [Ger04b] on the revisions. For each MR, create an MR vertex with a unique time based on the earliest revision with which it is associated.

- For each MR, create an edge from the MR vertex to the revision vertices it is associated with.
- For each MR vertex, create an edge from the MR vertex to the author vertex who is the author of the MR and all of the MR's revisions.
- For each MR vertex create an edge from that MR vertex to the next MR vertex in time only if there exists an MR vertex with a later timestamp than this MR vertex.
- For each file, for each of its revisions ($x$):
  - If a revision $x$ is a parent revision of revision $y$, create an edge from $x$ to $y$. Parents can be determined by the SCS's versioning system or the patches. In CVS it may be the case that version 1.3 is the parent of to 1.3.2.1 and 1.4 (but 1.3.2.1 might not be a parent of 1.4 (if it was, the branch merge would have to be detected)). This case covers both branches and branch merges.

When this algorithm terminates, the result is a characteristic graph of the instance of the SCS.

Some repositories such as CVS do not record branch merges. Branch merge identification can be done using techniques discussed in "Populating a Release History Database from Version Control and Bug Tracking Systems" [FPG03]. Branch merge identification is not 100% accurate; therefore, by integrating branch merge data, the graph is more of an interpretation of the

SCS rather than an exact representation of the SCS. MRs are derived; they are not necessarily accurate representations of commits either.

An example of the produced graph is depicted in figure 2.2. An example of how revisions are structured with respect to each other is depicted in figure 2.3.

### 2.4.2 Formal Graph Generator

For the formal extraction we assume that we have a list of tuples. (Filename,RevisionID,Userid,Time,...) These tuples are extracted revisions from a CVS repository. All these tuples are stored in the array $R$. Let there be a function $timesort(\alpha)$ which sorts an array of tuples by the Time column in the tuple. $germanMRExtractor()$ is the implementation of algorithm that produces MRs from revisions [Ger04b]. Assume $germanMRExtractor()$ adds edges from the MRs to the revisions. $addNode()$ sets all the data in map. Branch Merge detection is done after this algorithm is run. (The source code is in pseudo-SML notation)

```
let stretchtime list =
  let countdups curr last = function
      [] -> 0
    | x::xs -> if ((timeof x) = last) then
                  max (curr+1) (countdups (curr+1) (timeof x) xs)
               else
```

```
                countdups 0 (timeof x) xs
  in
  let dedup last count = function
      [] -> []
    | x::xs -> let t = timeof x in
               if (last = t) then
                  (chgtime x (t  + count + 1))::(dedup t (count+1) xs)
               else
                  x::dedup (timeof x) 0 xs
  in
  let c = 1 + countdups 0 0 list in
  dedup 0 0 (List.map (fun x -> chgtime x ((timeof x) * c)) list)
;;

let  extractor R =
        let c := -1
 #atomicize the revisions if there are conflicts
        let R = stretchtime R
        #now all tuples have unique time
        (Revision,MR,Author,File,V,E) = (Emptyset,Emptyset,Emptyset,
                                         Emptyset,Emptyset,Emptyset)
        iter (fun r ->
          let r  = addNode(r,Revision) in
```

```
  let f  = addNode(filename(r),File) in
  let e  = addEdge(r,f,E) in
  let a  = addAuthor(author(r)) in
  let ea = addEdge(r,a,E) in
) R
let MR = germanMRExtractor(Revision,Author,File,E) in
iter (fun m ->
  let m = setTime(m,fold (fun old r -> min old time(r))
            maxtime revisions(m))
  let e = addEdge(m,author(first(revisions(m))),E)
) MR
for i in 1 .. |MR|
  let m = MR[i]
  let p = MR[i-1]
  addEdge(p,m,E)
iter (fun f ->
  let revs = revisions f in
  iter (fun r1 ->
    iter (fun r2 ->
      if (revLt(r1,r2) && previous(r1,2)) then
        addEdge(r1,r2,E)
    ) revs
  ) revs
```

```
) File

(Revision,MR,File,Author,V,E)
```

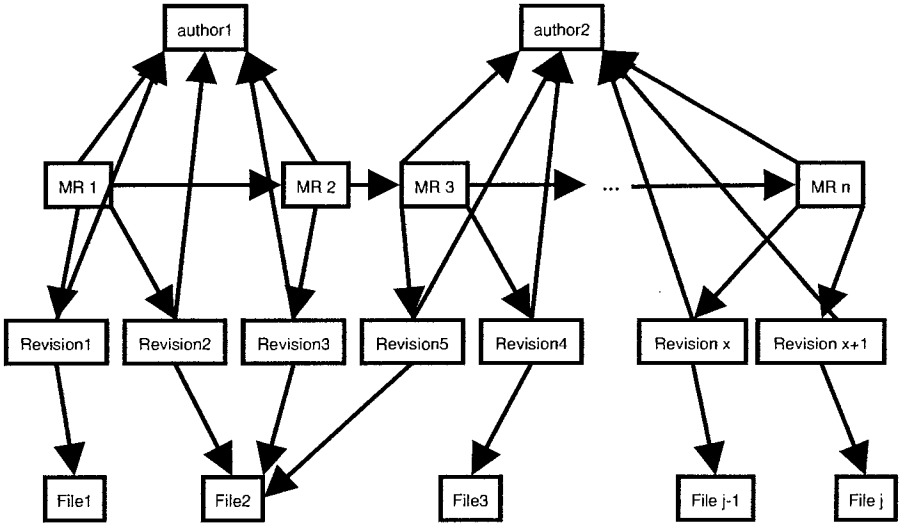


Figure 2.2: Example Model Subgraph

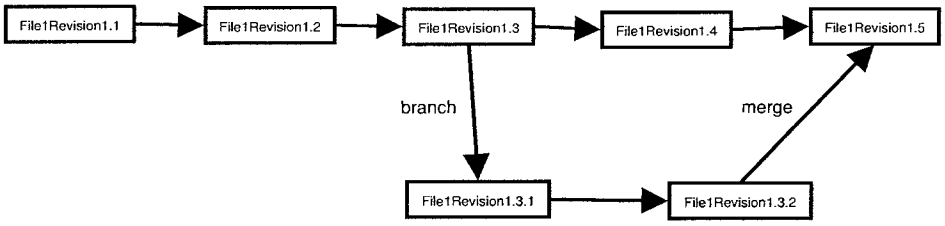


Figure 2.3: Example Revision Subgraph

# Chapter 3

# Query Language

The present model is defined by using a graph theoretic language combined with restrictions described using first order logic and sets. We want to produce a language that is close to the model and the theory so that we can make proofs regarding the correctness of the answers to the queries. Our language should also support queries that query time both relatively and concretely. Concrete time refers to exact times and dates. Relative time refers to time relative to an event or time attribute of an object. Before and after are examples of relative times where as December 1st 2005 is an example of a concrete time.

The language should allow for edge traversal and iterating over vertices. We want to be able to easily ask questions both globally and locally in relation to the entities in the model.

## 3.1 Basis

The basis for our language will be First Order Predicate Logic (FOPL). FOPL can handle the temporal aspects of temporal logic [AHV96], while still being powerful enough to handle reasonable queries.

The language allows for queries that have not only boolean answers but string, numeric and set based answers as well. Also print statements are provided to allow for debugging of queries and calculating metrics per entity.

General features of the language are:

- A multi type system to provide answers of different types to queries
- Graph access (via entities and attributes of entities)
- Edge Traversal (via subsets of entities)
- Extrapolation of partial relations (via subsets of entities)
- Multiple scopes to iterate and test subsets
- Arithmetic operations
- String operations

## 3.2 Motivation

We were motivated to produce a new language because other languages required too much effort to traverse typed graphs like our model. By effort

we mean the act of translating a question into the query language. We will assume that models which do not map well to our mental or formal model of the domain are harder to query than those models which are more similar to our model. It is harder to ask graph questions with a relational model of a graph model than to ask questions of the graph model itself. Conversely, it is hard to ask relational algebra related questions of a graph model. We found that XQuery was hard to use for this purpose since it was not well suited to graphs. XQuery required that the data be hierarchical in nature (XML). Without hierarchical structures, the XQuery language requires references through partial relations. This makes any sort of query acting in the reverse direction of the hierarchy or across multiple hierarchical tree nodes very difficult to express.

SQL based querying suffers from mapping from one domain, relation algebra, to another, first order logic and graphs. SQL can use relational algebra to solve the queries but it isn't semantically built for graphs. It is meant to query and combine tables of values, not deal with somewhat hierarchical and path based structures. Finding paths in graphs or iterating through the related elements of a graph in SQL is possible but is not an easy or elegant way to express a query.

Some aspects of temporal logic and first order logic are represented in the SCQL query language. Aspects of temporal logic were used which fit better into a system of multiple types.

## 3.3 Language

The language consists of a few atomic predicates: boolean values (True, False), strings ("string"), numbers (1, 1.0). We can combine these atomic predicates into expressions using operators. The language is composed of many operators and quantifiers:

- Binary operators: $\rightarrow$ , $==$ , $\leq$ , $\geq$ , $>$ , $<$ , $!=$ , $+$ , $-$ , $/$ , $*$ , $\&\&$ , $||$
- Unary operators: $\neg$
- Universal Quantifiers: $\forall_{before}$ , $\forall_{after}$ , $\forall$
- Existential Quantifiers: $\exists_{before}$ , $\exists_{after}$ , $\exists$
- Predefined Functions: $function(x)$
- Attributes: $x.Attribute$
- Identifiers : $x, node, y$

Variables store instances of entities such as MRs, Authors, Revisions or Files. Each entity usually has a host of attributes accessible using the notation $x.attribute$. Variables are only created when scope is created; thus there is no real assignment other than selection based scope (anchors). The universal, existential, subset, and selection quantifiers produce scope; they bind a variable to both an entity type and an instance of a entity. This variable will be used to iterate over the subset provided. We will refer to the

quantifiers as scope producers or scopes. A Universal scope produces a new scope that will have its predicate evaluated for all elements of the subset that the scope is iterating over. Universal and Existential scopes iterate over an entire domain or a subset of that domain. Selection based scope (anchors) find matching elements and iterate over them.

Attributes are of different types, as described in the model. For the purposes of the language, we will abstract edges into sets of entities. Edges can be explicitly queried using the $isEdge$ function.

There are functions that return sets, strings, numbers, or boolean values. These functions can accept entities, sets, strings, numbers, or boolean values as parameters.

Existential scopes imply that the proposition will be tested throughout the entire domain until the proposition is found to be true; while universal scopes will test the proposition till an element evaluates as false. If the preposition is false for all of the elements in the subset, an existential scope returns false. If the preposition is true for any element in the subset, then an existential returns true. For empty domains, the existential scope returns false, but the universal scope returns true.

Existential and Universal scopes are extended by the temporal modifiers "before" and "after" ($\forall_{before}$, $\forall_{after}$, $\exists_{before}$, $\exists_{after}$). These modifiers take an extra parameter of an entity or time. This entity is used to relatively test other entities if they come before or after it. If the tested entities match (either they are before or after the parameter entity) they are tested via the

proposition supplied to the scope. It is analogous to testing $before(\phi) \implies P(\phi)$ or $after(\phi) \wedge P(\phi)$ (depending on if a universal or existential scope is used) instead of just $P(\phi)$, where $P$ is the predicate.

Subset based scopes iterate through all the elements in a subset of entities (such as $\mathbb{MR}$, $\mathbb{R}evision$, $\mathbb{F}ile$ , $\mathbb{A}uthor$). They produce a subset by selecting entities that cause the predicate to evaluate as true. A subset can only be the same size or smaller than the set it is selected from. Subsets in the language can only be subsets of one kind of entity ($\mathbb{MR}$, $\mathbb{R}evision$, $\mathbb{F}ile$ , or $\mathbb{A}uthor$)

## 3.4 Mapping the Model To The Language

Let the function $\ell$ be the translation function where given a preposition $P(a)$, where $a$ is the parameter, it will convert $P(a)$ into its representation in the SCQL query language. This will be used heavily in table 3.4 to convert prepositions. This function exists to allow a clear translation. Let the notation $2^{\alpha}$ map to the power set of $\alpha$ (since $P$ is already used for propositions).

Table 3.4 describes various elements of the language but uses the following definitions. The rest of this section will use following definitions unless otherwise stated. Let `MR` $= \mathbb{MR}$, `Author` $= \mathbb{A}uthor$, `Revision` $= \mathbb{R}evision$, and `File` $= \mathbb{F}ile$ . Let $\alpha \in \{\mathbb{MR}, \mathbb{A}uthor, \mathbb{R}evision, \mathbb{F}ile\}$. Let $\alpha$ be called a domain and subsets of $\alpha$ be a sub-domain. Thus domains and sub-domains are subsets of entities of the same type. A sub-domain can be a domain. Let $\exists\delta \in (2^{\mathbb{MR}} \cup 2^{\mathbb{R}evision} \cup 2^{\mathbb{F}ile} \cup 2^{\mathbb{A}uthor})$ . Let $\delta$ be a subset of vertices that is a

subset of either $\mathbb{MR}, \mathbb{R}evision, \mathbb{F}ile, \mathbb{A}uthor$. Let $\exists\zeta \in \{$ set of attribute functions $\}$. Let $\exists\gamma \in \{$ set of functions $\}$ . Let $\exists\phi, \theta \in \{$ set of all variable names$\}$. Let $P(\phi)$ be a proposition on $\phi$. P returns a boolean value. Let $E_{MRTime}$ be a subset of edges such that $\forall(a,b) \in E_{MRTime} isMR(a) \wedge isMR(b) \wedge before(a,b)$. Let $I$ and $J$ be numeric expressions. Let $P$ and $Q$ be boolean propositions. Let $S$ and $T$ be string expressions. Let $N(\phi)$ be an expression that produces a numeric value. Let $T(\phi)$ be an expression that produces a string value. Let $U(\phi)$ be an expression that produces a subset value.

The anchor selection is a short cut scope that attempts to use logarithmic to constant time access to query one entity. It is similar to an arbitrary assignment but is meant for fast access of entities.

Scopes iterating through MRs generally start at the first MR in time and follow through to the next MR using the edges in $E_{MRTime}$. In the case of an existential operator, once a certain MR satisfies the proposition, no further MRs need to be evaluated. If no MR satisfied the proposition of the existential operator, the existential operator is evaluated as false. In the case of the universal operator, once an MR does not satisfy the proposition, no further MRs need to be evaluated as the expression is evaluated as false.

Scopes were made explicit to illustrate the possible complexity of a query, where first order logic query $\exists a, b, c, d \in V(P(a,b,c,d))$ does not look like it would take $O(n^4)$ but it could if executed naively. This is much more obvious in scoped form:

| Name | Type | Language | Model Equivalent |
|---|---|---|---|
| MR | $\mathbb{S}$ | MR | $\mathbb{MR}$ |
| Revision | $\mathbb{S}$ | Revision | $\mathbb{R}evision$ |
| Author | $\mathbb{S}$ | Author | $\mathbb{A}uthor$ |
| File | $\mathbb{S}$ | File | $\mathbb{F}ile$ |
| Not | $\mathbb{B}$ | $!\ell(P)$ | $\neg P$ |
| Implies | $\mathbb{B}$ | $\ell(P) -> \ell(Q)$ | $P \implies Q$ |
| Bijection | $\mathbb{B}$ | $\ell(P) <=> \ell(Q)$ | $P \Leftrightarrow Q$ |
| Logical And | $\mathbb{B}$ | $\ell(P)\&\&\ell(Q)$ | $P \wedge Q$ |
| Logical Or | $\mathbb{B}$ | $\ell(P)\|\|\ell(Q)$ | $P \vee Q$ |
| Equals | $\mathbb{B}$ | $\ell(I) == \ell(J)$ | $I = J$ |
| Not Equals | $\mathbb{B}$ | $\ell(I) != \ell(J)$ | $I \neq J$ |
| Less than | $\mathbb{B}$ | $\ell(I) < \ell(J)$ | $I < J$ |
| Less than or Equal | $\mathbb{B}$ | $\ell(I) <= \ell(J)$ | $I \leq J$ |
| Greater than | $\mathbb{B}$ | $\ell(I) > \ell(J)$ | $I > J$ |
| Greater than or Equal | $\mathbb{B}$ | $\ell(I) >= \ell(J)$ | $I \geq J$ |
| Addition | $\mathbb{R}$ | $\ell(I) + \ell(J)$ | $I + J$ |
| Subtraction | $\mathbb{R}$ | $\ell(I) - \ell(J)$ | $I - J$ |
| Division | $\mathbb{R}$ | $\ell(I)/\ell(J)$ | $I/J$ |
| Multiplication | $\mathbb{R}$ | $\ell(I) * \ell(J)$ | $I \bullet J$ |
| String Equality | $\mathbb{B}$ | $eq(\ell(S), \ell(T))$ | $eq(S, T)$ |
| Universal (general) | $\mathbb{B}$ | $A(\phi, \delta)\{\ell(P(\phi))\}$ | $\forall \phi \in \delta(P(\phi))$ |
| Universal (eg.) | $\mathbb{B}$ | $A(a, MR)\{\ell(P(a))\}$ | $\forall a \in \mathbb{MR}(P(a))$ |
| Existential (general) | $\mathbb{B}$ | $E(\phi, \delta)\{\ell(P(\phi))\}$ | $\exists \phi \in \delta(P(\phi))$ |
| Existential (eg.) | $\mathbb{B}$ | $E(a, MR)\{\ell(P(a))\}$ | $\exists a \in \mathbb{MR}(P(a))$ |
| Attribute | $\mathbb{A}ny$ | $\phi.\zeta$ | $\zeta(\phi)$ |
| Attribute (eg.) | $\mathbb{A}ny$ | $a.time$ | $time(a)$ |
| Function | $\mathbb{A}ny$ | $\gamma(\ell(P))$ | $\gamma(P)$ |
| Function (eg.) | $\mathbb{A}ny$ | $length(\ell(P))$ | $length(P)$ |
| Universal Before | $\mathbb{B}$ | $Abefore(\phi, \delta, \theta)\{\ell(P(\phi, \theta))\}$ | $\forall \phi \in \delta(before(\phi, \theta) \implies P(\phi, \theta))$ |
| Universal Before (eg.) | $\mathbb{B}$ | $Abefore(a, MR, b))\{\ell(P(a, b))\}$ | $\forall a \in \mathbb{MR}(before(a, b) \implies P(a))$ |
| Universal After | $\mathbb{B}$ | $Aafter(\phi, \delta, \theta)\{\ell(P(\phi, \theta))\}$ | $\forall \phi \in \delta(after(\phi, \theta) \implies P(\phi, \theta))$ |
| Universal After (eg.) | $\mathbb{B}$ | $Aafter(a, MR, b))\{\ell(P(a, b))\}$ | $\forall a \in \mathbb{MR}(after(a, b) \implies P(a))$ |
| Existential Before | $\mathbb{B}$ | $Ebefore(\phi, \delta, \theta)\{\ell(P(\phi, \theta))\}$ | $\exists \phi \in \delta(before(\phi, \theta) \wedge P(\phi, \theta))$ |
| Existential Before (eg.) | $\mathbb{B}$ | $Ebefore(a, MR, b))\{\ell(P(a, b))\}$ | $\exists a \in \mathbb{MR}(before(a, b) \wedge P(a))$ |
| Existential After | $\mathbb{B}$ | $Eafter(\phi, \delta, \theta)\{\ell(P(\phi, \theta))\}$ | $\exists \phi \in \delta(after(\phi, \theta) \wedge P(\phi, \theta))$ |
| Existential After (eg.) | $\mathbb{B}$ | $Eafter(a, MR, b))\{\ell(P(a, b))\}$ | $\exists a \in \mathbb{MR}(after(a, b) \wedge P(a))$ |
| Subset | $\mathbb{S}$ | $S(\phi, \delta)\{\ell(P(\phi))\}$ | $\{\phi \in \delta \| P(\phi)\}$ |
| Subset (eg.) | $\mathbb{S}$ | $S(a, MR)\{\ell(P(a))\}$ | $\{a \in \mathbb{MR} \| P(a)\}$ |
| Anchor Select | $\mathbb{B}$ | $Anchor(a, MR, "mrid")\ell(P(a))$ | $\exists a \in \mathbb{MR}(eq(id(a), "mrid") \wedge P(a))$ |
| count | $\mathbb{R}$ | $count(\delta)$ | $count(\delta)$ |
| count (eg.) | $\mathbb{R}$ | $count(\mathbb{MR})$ | $count(\mathbb{MR})$ |
| Sum | $\mathbb{R}$ | $Sum(\phi, \delta)\{\ell(P(\phi))\}$ | $sum(\phi \in \delta, P(\phi))$ |
| Sum (eg.) | $\mathbb{R}$ | $Sum(a, MR)\{\ell(P(a))\}$ | $sum(\mathbb{MR}, P(a))$ |
| Average | $\mathbb{R}$ | $Avg(\phi, \delta)\{\ell(P(\phi))\}$ | $avg(\phi \in \delta, P(\phi))$ |
| Average (eg.) | $\mathbb{R}$ | $Avg(a, MR)\{\ell(P(a))\}$ | $avg(\mathbb{MR}, P(a))$ |

Table 3.1: Language to Model Mappings of SCQL

```
E(a,V) {
  E(b,V){
    E(c,V){
      E(d,V){
        P(a,b,c,d)
      }
    }
  }
}
```

Regular expressions were not used because some queries cannot be represented by regular expressions. Regular expressions do not allow for subexpressions. For instance, if a subexpression referenced an entity from an enclosing scope it would suggest that one would have to re-evaluate the entities. There are cases where finite state machines or regular expressions would work. Given this query:

```
E(a,MR) {
  P(a) &&
  Eafter(b,MR,a) {
    Q(b)
  }
}
```

we could go through and mark nodes if they return true for both functions $P$ and $Q$. Then we could use the Regex $P.*Q$ as the query. Instead of naively taking $O(n^2)$ time we would only take $O(n)$. However the query:

```
E(a,MR) {
  Eafter(b,MR,a) {
    Q(a,b)
  }
}
```

cannot be executed effectively with a finite state machine. We would have to do $O(n^2)$ operations before we could begin. It would no longer be a stream of entities, rather, it would be a matrix of entities. Thus, for queries of a reasonable complexity, there is minimal benefit in producing finite state machines when the naive approach will often work just as well.

## 3.5 Functions

Functions such as $revAfter$, $revBefore$, $isFileOf$, $isMROf$, $isRevisionOf$, $isAuthorOf$, $isEdge$, , $after$, $before$, $isaMR$, $isaFile$, $isaRevision$, $isaAuthor$, $eq$, $length$, $matches$ are imported from the model. Other functions are provided as a convenience:

- $isFirstRevision(\phi \in \mathbb{R}evision) \mapsto \mathbb{B} \implies \neg\exists\theta \in \mathbb{R}evision(isEdge(\theta, \phi))$

  - is this revision $\phi$ the first revision of the file the revision is related to?

- $print(x \in \mathbb{S}tring) \mapsto \mathbb{B} \implies true$ - as a side effect, this function prints the string; it is provided for debugging purposes

- $endMatch(x \in \mathbb{S}tring, y \in \mathbb{S}tring) \mapsto \mathbb{B} \implies length(y) <= length(x) \wedge eq(substr(x, length(x) - length(y), length(y)), y)$- returns true if $y$ is a suffix of $x$

- $boolean(x \in \mathbb{R}) \mapsto \mathbb{B} \implies \neg(x = 0)$ - returns false if x is zero, otherwise it is true ($x$ ends in $y$)

- $boolean(x \in \mathbb{S}tring) \mapsto \mathbb{B} \implies (\neg eq(x, ""))$ - returns false if x is an empty string, otherwise it is true

- $neq(x \in \mathbb{S}tring, y \in \mathbb{S}tring) \mapsto \mathbb{B} \implies \neg eq(x, y)$ - are strings $x$ and $y$ not equal?

- $days(x \in \mathbb{R}) \mapsto \mathbb{R} \implies$ the number of days since the Unix epoch that this time represents, an integer value rounded down

## 3.6 Domains and Sub-domains

Domains and sub-domains exist to allow for navigation between entities that share a relationship with each other. Domains are subsets of entities of the same type and sub-domains are subsets of those domains. Sub-domains often fill the role of traversing edges from one entity to another (a set of end points of directed edges). In the query language, sub-domains are used means to

traverse edges and partial relations that exist in the model. For example, sub-domains could be the revisions of an MR accessed by *mr.revisions*, the MRs of an author, *author.mrs*, or the revisions of a file ,*file.revisions*. See tables 3.2, 3.3, 3.4 and 3.5 for a more explicit explanation and definition of the sub-domains used.

Sub-domains exist for single edges (MR to Author) as well, but we use plural names for consistency. This is because the only assignment in the language is done with scope operators; thus, this enforces uniform access of entities via sets and subsets. There might be the case where single edges in the model are not necessarily accurate, for instance if a SCS supported pair programming, that is if it allowed multiple programmers to associated with a change, more than one author would be associated with the revisions and MRs.

Some sub-domains are expansions of partial relations such as time. Other sub-domains are extra edges which are shortcuts to other entities, these shortcuts allow queries to reduce their complexity (e.g. $f.authors$ would be a set of author who made revisions to the file $f$). Additionally, sub-domains are groups of entities that are related. Expressing partial relations as sub-domains allows us to iterate through entities as though we were iterating through edges in the graph. Sub-domains provide a quick abstraction of edges in our model.

| Entity | Sub-domain Attribute | Type | Example |
|---|---|---|---|
| MR | authors | Author | $A(a, MR)\{A(b, a.authors)\{$ |
| | A sub-domain consisting of the author of the MR . $\{\theta \in \mathbb{A}uthor \mid isAuthorOf(\theta, a)\}$ | | |
| MR | revisions | Revision | $A(a, MR)\{A(b, a.revision)\{$ |
| | A sub-domain consisting of the revisions of the MR. $\{\theta \in \mathbb{R}evision \mid isRevisionOf(\theta, a)\}$ | | |
| MR | files | File | $A(a, MR)\{A(b, a.files)\{$ |
| | A sub-domain consisting of the files that the MR has revisions of. $\{\theta \in \mathbb{F}ile \mid isFileOf(\theta, a)\}$ | | |
| MR | nextmrs | MR | $A(a, MR)\{A(b, a.nextmrs)\{$ |
| | A sub-domain consisting of the next MR in time after the current MR. $\{\theta \in nextMRs(a)\}$ | | |
| MR | prevmrs | MR | $A(a, MR)\{A(b, a.prevmrs)\{$ |
| | A sub-domain consisting of the last MR in time before the current MR. $\{\theta \in prevMRs(a)\}$ | | |

Table 3.2: Sub-domains of MRs

| Entity | Sub-domain Attribute | Type | Example |
|---|---|---|---|
| Revision | authors | Author | $A(a, Revision)\{A(b, a.authors)\{$ |
| | A sub-domain consisting of the author associated with the Revision. $\{\theta \in \mathbb{A}uthor \mid isAuthorOf(\theta, a)\}$ | | |
| Revision | mrs | MR | $A(a, Revision)\{A(b, a.mrs)\{$ |
| | A sub-domain consisting of the MR that the Revision is associated with. $\{\theta \in \mathbb{MR} \mid isMROf(\theta, a)\}$ | | |
| Revision | files | File | $A(a, Revision)\{A(b, a.files)\{$ |
| | A sub-domain consisting of the file that the Revision is associated with. $\{\theta \in \mathbb{F}ile \mid isFileOf(\theta, a)\}$ | | |
| Revision | nextrev | Revision | $A(a, Revision)\{A(b, a.nextrev)\{$ |
| | A sub-domain consisting of the first Revision, revision-wise after the current Revision. $\{\theta \in \mathbb{R}evision \mid \theta \in nextRevisions(a)\}$ | | |
| Revision | prevrev | Revision | $A(a, Revision)\{A(b, a.prevrev)\{$ |
| | A sub-domain consisting of the last Revision, revision-wise before the current Revision. $\{\theta \in \mathbb{R}evision \mid \theta \in prevRevisions(a)\}$ | | |

Table 3.3: Sub-domains of Revisions

| Entity | Sub-domain Attribute | Type | Example |
|---|---|---|---|
| Author | mrs | MR | $A(a, Author)\{A(b, a.mrs)\{$ |
| | A sub-domain consisting of the MRs that the author has created. $\{\theta \in \mathbb{MR} \mid isMROf(\theta, a)\}$ | | |
| Author | revisions | Revision | $A(a, Author)\{A(b, a.revisions)\{$ |
| | A sub-domain consisting of the revisions of the MRs that the author has created. $\{\theta \in \mathbb{R}evision \mid isRevisionOf(\theta, a)\}$ | | |
| Author | files | File | $A(a, Author)\{A(b, a.files)\{$ |
| | A sub-domain consisting of the files of the revisions of the MRs that the author has created. $\{\theta \in \mathbb{F}ile \mid \exists\tau \in \mathbb{R}evision(isRevisionOf(a, \theta) \wedge isFileOf(\theta, \tau))\}$ | | |
| Author | prevauth | Author | $A(a, Author)\{A(b, a.prevauth)\{$ |
| | A sub-domain consisting of the first Author in time after the current Author (time is based on the first revision of each author). $\{\theta \in \mathbb{A}uthor \mid \forall\tau \in \mathbb{A}uthor((before(\tau, a) \implies (\theta = \tau \vee after(\theta, \tau)))\}$ | | |
| Author | nextauth | Author | $A(a, Author)\{A(b, a.nextauth)\{$ |
| | A sub-domain consisting of the last Author in time before the current Author (time is based on the first revision of each author). $\{\theta \in \mathbb{A}uthor \mid \forall\tau \in \mathbb{A}uthor((after(\tau, a) \implies (\theta = \tau \vee before(\theta, \tau)))\}$ | | |

Table 3.4: Sub-domains of Authors

| Entity | Sub-domain Attribute | Type | Example |
|---|---|---|---|
| File | mrs | MR | $A(a, File)\{A(b, a.mrs)\{$ |
| | A sub-domain consisting of the MRs that contain revisions of this file. $\{\theta \in \mathbb{MR} \mid isMROf(\theta, a)\}$ | | |
| File | revisions | Revision | $A(a, File)\{A(b, a.revisions)\{$ |
| | A sub-domain consisting of the Revisions of this file. $\{\theta \in \mathbb{R}evision \mid isRevisionOf(\theta, a)\}$ | | |
| File | authors | Author | $A(a, File)\{A(b, a.authors)\{$ |
| | A sub-domain consisting of all the Authors who created revisions of this file. $\{\theta \in \mathbb{A}uthor \mid \exists \tau \in revisions(a)(isAuthorOf(\theta, \tau))\}$ | | |
| File | modulefiles | File | $A(a, File)\{A(b, a.modulefiles)\{$ |
| | A sub-domain consisting of all the files in the same module as the current file including the current file. $\{\theta \in \mathbb{F}ile \mid eq(module(a), module(\theta)\}$ | | |
| File | dirfiles | File | $A(a, File)\{A(b, a.modulefiles)\{$ |
| | A sub-domain consisting of all the files in the same directory as the current file including the current file. $\{\theta \in \mathbb{F}ile \mid eq(directory(a), directory(\theta)\}$ | | |
| File | nextfile | File | $A(a, File)\{A(b, a.nextfile)\{$ |
| | A sub-domain consisting of the first File in time after the current File (time is based on the first revision of each file). Given $a$, $\{\theta \in \mathbb{F}ile \mid \forall \tau \in \mathbb{F}ile((after(\tau, a) \implies (\theta = \tau \vee before(\theta, \tau)))\}$ | | |
| File | prevfile | File | $A(a, File)\{A(b, a.nextfile)\{$ |
| | A sub-domain consisting of the last File in time before the current File (time is based on the first revision of each file). Given $a$ , $\{\theta \in \mathbb{F}ile \mid \forall \tau \in \mathbb{F}ile((before(\tau, a) \implies (\theta = \tau \vee after(\theta, \tau)))\}$ | | |

Table 3.5: Sub-domains of Files

## 3.7 Constants

There are 2 main kinds of constants, strings and numbers:

- Strings: `"Example String"`, `""`, `"2005-04-04"` - Strings are characters between double quotes, slashes are used to escape double quotes.
- Integers and Floats: `1`, `1.0`, `-1`, `1.0e0` - Numbers are weakly typed as numbers, that is numbers can act as both floating point numbers and integers.

## 3.8 Examples of Queries

We will show how to compose the query for the question, "does there exist an author, who only modifies files that are previously modified by another author?" This can be expressed in first order logic as:

$$\exists a \in Author($$

$$\exists b \in Author($$

$$\forall r \in \{x \in Revision | isAuthorOf(a, x)\}($$

$$\forall f \in \{x \in Files | isFileOf(x, r)\}($$

$$\exists r_2 \in \{x \in Revision | isFileOf(f, x)\}($$

$$before(r_2, r) \wedge isAuthorOf(b, r_2)$$

$$) \quad ) \quad ) \quad ) \quad )$$

To convert this first order logic query to the SCQL language, we first find the two authors, $a$ and $b$ ( `E(a,Author){E(b,Author){` ), then we go through all the revisions of author $a$ ( `A(r,a.revisions){` ). We are using *a.revisions* to get the subset of all revisions that belong to author $a$. Then for that file of that revision ( `A(f,r.files){` ) we check if all of the revisions before revision $r$ ( `Ebefore(r2,f.revisions,r){` ) are from the same author ( `isAuthorOf(b,r2)` ).

```
E(a,Author) {
  E(b,Author) {
    a != b &&
    A(r,a.revisions) {
      A(f,r.files) {
        Ebefore(r2,f.revisions,r) {
          isAuthorOf(b,r2)
        }
      }
    }
  }
}
```

## 3.9 Halting

Does every query asked in our query language halt? To answer this question, we must first be aware of some intentional design characteristics of the model and the query language. Sets created in the model are subsets of other sets. New entities are never created or added in anyway. All the entities are finite: there are a finite number of nodes modeled and there are a finite number of relations.

All of the primitives of the system are indeed haltable; the composition of these primitives also halts. All the *isa* functions (*isaMR* , *isaRevision*

, $isaFile$ , $isaAuthor$) are checks for set inclusion, which are in polynomial time for finite domains. All the boolean operators such as $=, \implies, \wedge, \vee, \neg$ are simple boolean logic expressions and obviously halt.

Existential operators halt if their propositions halt and their domain of values is finite. Existential operators in the model work on only the finite vertices and edges. Conversely since $(\neg\exists\phi \in \alpha \neg P(\phi)) = (\forall\phi \in \alpha P(\phi))$ we know that the universal operators halt if its proposition halts (because the primitives used halt as well).

The subset $S(\phi, \alpha)\{P(\phi)\}$ operator used in the model and the query language halts if its proposition halts. It acts in a similar manner to the universal operator but acts on the whole domain it is querying without short circuiting. Since all domains are finite, if the proposition for determining members of the subset halts then the subset operator halts.

For all the numeric operators whether they return boolean values or numeric values such as $P \bullet Q s.t. \bullet \in \{+, -, *, / =, ! =, >, <, <=, >=\}$ we know that given propositions $P$ and $Q$ halt because these operators can be evaluated in polynomial time.

Attributes will always halt since they are maps from one value (an entity) to another (string, subset, numeric, boolean). Maps have been shown to be accessed in constant time. Attributes that return subsets will return subsets of entities. They will not make any new entities – there is no way to create new entities, only query them. Given that restriction, universal and existential will ultimately halt if they are iterating over a subset that was

returned from an attribute function (and if their proposition halts).

Therefore the functions formed of previously defined primitives and primitive functions will all halt unless they are broken recursive functions. There are only two recursively defined functions, $revBefore(\phi, \theta)$ and $revAfter(\phi, \theta)$. $revAfter(\phi, \theta)$ is derived from $revBefore$. We will show that $revBefore(\phi, \theta)$ halts:

Let $P(\phi, \theta) \implies isaRevision(\phi) \wedge isaRevision(\theta) \wedge (\exists \tau \in \mathbb{F}ile(isEdge(\phi, \tau) \wedge isEdge(\theta, \tau)))$.

Let $Q(\phi, \theta) \implies ((\phi, \theta) \in E \vee \exists (\tau, \theta) \in E(revBefore(\phi, \tau)))$.

Therefore $revBefore(\phi, \theta) \implies P(\phi, \theta) \wedge Q(\phi, \theta)$ $P(\phi, \theta)$ halts because it is composed of non-recursively halting primitives. $Q(\phi, \theta)$ returns immediately if there is a direct edge from revision $\phi$ to revision $\theta$. If there is not a direct edge then all edges that go to entity $\theta$ are tested to see if they are on the same file and if they are revision edges. For those edges that meet those requirements the same test is repeated. We know that the subgraph of revisions on one file is acyclic and since there are a finite number of nodes there are a finite number of paths from $\phi$ to $\theta$, if there exists such a path. If there is not a path from $\phi$ to $\theta$, effectively all possible paths to $\theta$ will be exhausted. This can be evaluated in $O(n^2)$ time as each previous element might have to evaluate the proposition for almost all the revisions in the model.

Since $revBefore(\phi, \theta)$ has been proven to halt, $revAfter(\phi, \theta)$ halts it is a composition of halting operators and primitives as well as uses $revBefore(\theta, \phi)$.

Therefore since all primitives halt and it has been shown that even the most complex composites provided halt, any query generated using these primitives will halt. This is mainly due to the fact that SCQL iterates over finite sets, and our functions and operators do not allow for arbitrary looping or recursion.

# Chapter 4

# Engine

The purpose of the engine is to allow us to ask our queries and execute them on real SCSs. As described in figure 4.1, when a query is to be evaluated, it must be first parsed into an abstract syntax tree (AST); this AST is passed off to the interpreter, which executes the query upon the model. The model holds the graph skeleton and calls upon the data layer to access data from the softChange database when it needs it. softChange extracts and processes the data from a CVS repository and puts it into a relational database.

The implementation consists of 4 modules:

- Query Language Interface and Parser: The query language interface reads and interprets the query, by converting it to an Abstract Syntax Tree (AST). The interface also handles reporting the results back to the user once the interpreter has executed the AST.

- Interpreter: The interpreter runs the ASTs against the graph model.

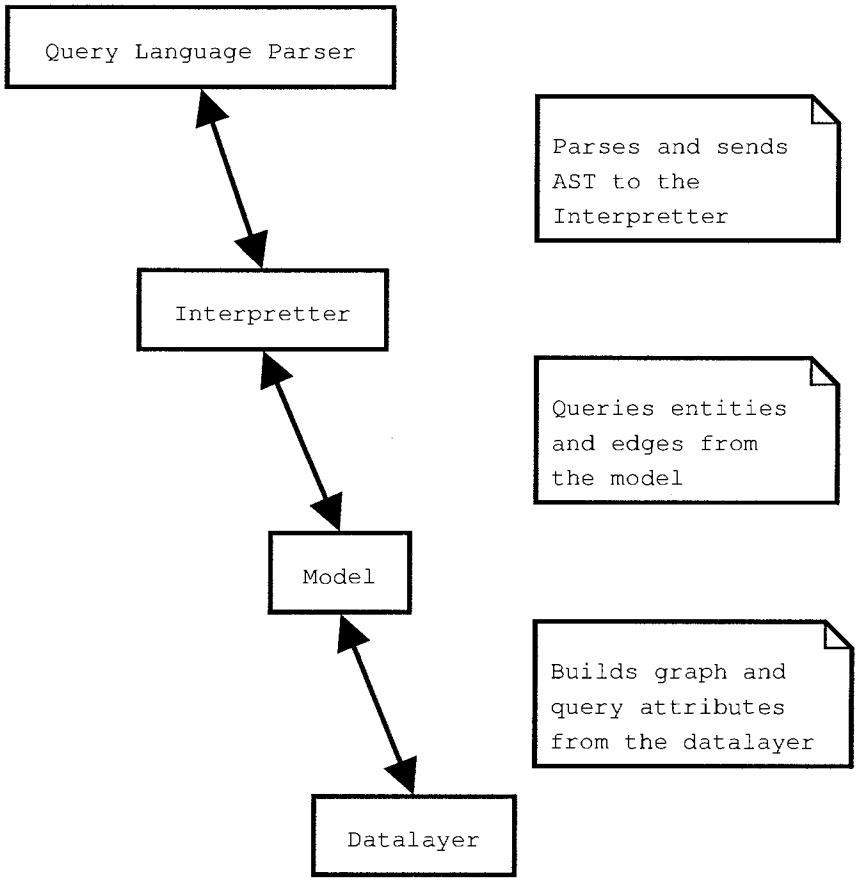


Figure 4.1: SCQL Implementation Architecture

The purpose of the interpreter is to keep track of symbols, iterate across sets and execute all the expressions needed in a query. The interpreter calls upon the model to access entities and edges.

- Model Layer: The model maintains a skeletal graph of the system and provides the primary means of interacting with the entities and their edges. The model layer overlays a graph layout upon what is stored in the data layer. The model layer also abstracts data layer access from

layers above it.

- Data Layer: The data layer queries, caches and pre-fetches entities from the softChange relational database. The data layer manages the caching and prefetching of data in memory to speed up queries and reduce interprocess communication costs. Since some instances of the model may contain upwards of a million elements one has to be careful about the amount of data stored in memory, especially if one wants to keep the skeleton of the model in memory.

Usually a query is provided by the user, it is then parsed into an AST and given to the interpreter to execute. The interpreter then runs the query by iterating over the scopes, maintaining the symbols and executing the expressions. The interpreter calls upon the model to provide the entities and the sets of entities that are needed. The model layer maintains a skeletal structure of the graph (flyweight entities and their relations). It gets the full implementation of the entities from the data layer. The data layer will cache and query data from softChange .

## 4.1 Implementation

When the engine starts, it extracts the vertices and the edges of the database (not the attributes) and produces a skeletal graph. This graph is often traversed by the scope operators. Usually this is much faster than accessing a database repeatedly. The database is only called when the attributes of an

entity are accessed. When an entity's attribute is accessed on a flyweight object, the full heavyweight entity is pulled from the cache or the database. The current implementation relies on pre-caching and pre-fetching to avoid much of the overhead of inter-process communication (IPC) when accessing the database.

The current implementation is written in Perl. A dynamically typed language such as Perl was ideal for dealing with the multiple types of return values. One disadvantage was that data takes up a lot memory due to the Perl interpreter.

To extract data from SCS we use softChange , developed by Dr. Daniel German. softChange takes a CVS repository, extracts the revisions, regenerates the MRs, and stores it all in a relational database. We use this relational database as the data source for SCQL.

The data in the relational databases maps well enough to the entities in the model, but querying the database can prove quite difficult. Some entities such as revisions are spread across multiple tables and authors are just a column in the MR table. This makes it difficult to write queries using SQL. In section 3.2, some difficulties of dealing with SQL are discussed. Iterating through records and grabbing records related to the current element is quite costly as it often requires large database queries.

There is not always enough memory to store the entire database in memory, especially when dealing with multiple projects. The size of the database is partially the reason a SQL backend is still used. Unfortunately the sizes

of some domains such as revisions, are so large that any query more than $O(n)$ in complexity could take a rather long time to compute on a large project. The runtime of large queries can be somewhat alleviated by using time relational scopes and short circuiting logic.

We implemented a prefetching and caching strategy. The cache could hold many entities but once there was a cache miss, a specified number of entities would be pre-fetched along with the entity requested. The entities are usually pre-fetched in order. The cache is a large hash table of entity IDs so a cache hit takes constant time while a non-cache hit depends on the implementation of the database ($O(log(n))$) and the cost of prefetching.

Prefetching was more useful than caching because we were iterating through the entire set of entities in order. This means we would linearly walk through entities rather than accessing them again. Caching helped performance when the sets of entities were small enough to fit in the cache. Revisions are usually the most numerous entities; as a result, one should cache as many as possible. There are usually fewer files and authors than revisions or MRs, thus they can be permanently cached, although there are limitations. For instance, the number of files in the repository of the Mozilla project (an open source web browser) we used, was near 35000 files versus near 500000 revisions (as of December 2003).

To iterate over the sets of entities, the Iterator design pattern was used [GHJV95]. The Flyweight design pattern [GHJV95] was also heavily used. The Flyweight pattern allows us to store lightweight entities as a graph skele-

ton and then query the heavy weight implementation from the database when we actually needed it. Moreover it allows us to use delete and garbage collect old heavy weight implementations without much trouble.

If for either existential or universal operators, the operator is a "before" operator, the first MR used is the earliest MR (the MR that has no edges in $E_{MRTime}$ that point to it). If it is an "after" operator the first MR is used is the immediately next MR. The MRs are evaluated in order from earliest to latest.

"Before" and "after" work differently for the other domains, that is, instead of following directed edges, the entities are ordered by a partial relation (time).

Authors, Files, and Revisions are iterated through by their partial time relations, not unlike MRs, but each is ordered by time since there are no time based edges for these entities.

# Chapter 5

# Applications

There are many applications for SCQL. They range from the evaluation and verification of theories of software evolution, to SCS access control, and to automatic invariant discovery. One important aspect of this research is that it can be used for multiple purposes such as historically based access controls, verifying laws of evolution and metrics.

## 5.1 Verifying Lehman's Laws

One of the original motivations of this research was to empirically verify Lehman's Laws of Software Engineering [Leh80] on a per project basis. That is, can we rate a project on how closely it follows Lehman's laws? We required a system which could answer questions about the lack of change in a repository of change. We had to create a system in which invariants of

change regarding a project could be queried.

Throughout this research it has become apparent that to verify many of these laws we need to model the architecture of the software project as well as the repository. While some measurements of software evolution need information about the software architecture, not all measurements need it. Furthermore, there can be multiple interpretations of information about software architecture. One view of a project architecture is not necessarily the designers or implementors view either.

Which of Lehman's laws [Leh80] can we attempt to test, check, or verify?

- The law of increasing complexity: " As time flows forwards entropy increases. That is, as a program evolves its structure will become more complex. Just as in physics this effect can, through great cost, be negated in the short term." [Leh80]

  To properly test this law we would need more architectural information. Simple information such as how many methods were added or removed can provide us with a metric for complexity. Luckily softChange provides some support for extracting this information. If we provide these statistics to SCQL, it can help us by providing us with the average differences between methods for each MR. If the average MR adds 0 or more methods this may suggest that the system slowly grows more complex over time. Of course, the size of this average may also suggest the rate of change in complexity.

- The law of large program evolution. "Program evolution is a self-regulating process and measurements of system attributes such as size, time between releases, number of reported errors, etc., reveals statistically significant trends and invariances." [Leh80]

  This law has provided a lot of the motivation for SCQL. SCQL already provides many possible ways of asking about system wide invariants. More importantly SCQL can be used to calculate metrics for the entities and attributes of a software project.

- The law of organizational stability. "Over the lifetime of a program, the rate of development of that program is approximately constant and independent of the resources devoted to system development." [Leh80]

  By using per MR and per revision measurements we may be able to test if a project has consistent properties of change. For example, the standard deviation of rate can be used to test if the rate of change is variable or not.

- The law of conservation of familiarity. " Over the lifetime of a system, the incremental system change in each release is approximately constant." [Leh80]

  As explained before SCQL provides some support for verifying incremental system change.

## 5.2 CVS Access Control / Auditing

By using a system such as SCQL we can create a very elaborate system of access controls and rules for policy verification and enforcement. Other research available suggests that log auditing and rule-based verification are effective [BGHS04, RG01] for verification and enforcement. A manager or project lead who controls the software process could attempt to use SCQL queries to audit the repository. These audits could be attempts to enforce certain policies in regards to how changes are made to a repository or to the source code. These changes could be audited based on the history of the actual project. This implies that policies can be based upon data that is actually in the repository and the history of that data.

System constraints could be built upon the historical behavior of a developer. Decisions based on historical information can be both complex and powerful. For example, a manager could be notified every time a programmer modified source files that they have never modified before. An example of this query could be:

```
S(a,MR) {
  E(b,a.revisions) {
    Abefore(c,Revisions,b) {
      eq(a.author,c.author) -> neq(b.filename,c.filename)
    }
  }
}
```

Another potential policy could dictate that programmers are only allowed to work in modules where they have already modified files or created files.

```
A(a,Author) {
  A(r1,a.revisions) {
    (!isFirstRevision(r1)) ->
    Ebefore(r2,a.revisions,r1) {
      A(f1,r1.files) {
        A(f2,r2.files) {
          eq(f1.module,f2.module)
        }
      }
    }
  }
}
```

Another example would be a policy where programmers can only change

code which they changed previously or added to the repository.

```
A(a,author) {
  A(r1,a.revisions) {
    (!isFirstRevision(r1)) ->
    Ebefore(r2,a.revisions,r1) {
      (r1 != r2) ->
      A(f1,r1.file) {
        isRevisionOf(r2,f1)
      }
    }
  }
}
```

Perhaps a manager would like to enforce a commit style. They might want to be notified each day of programmer infractions against this commit policy. The example query checks if all MRs during the past 24 hours have followed a commit style of always including a Changelog, modifying no more than 15 files, updating the Changelog and enforcing that all files are within the same module (this query could be run nightly):

```
A(a,MR) {
  ((days(today) - days(a)) < 1) ->
  count(a.revisions) > 1 &&
  count(a.revisions) <= 15 &&
  E(d,a.files) {
    eq(b.filename,"Changelog")
  } &&
  A(b,a.files){
    A(c,a.files) {
      eq(b.module,c.module)
    }
  }
}
```

Policies could be enforced at the commit time or by batch jobs during unused times (e.g. nightly). It is probably more important to identify problematic entities than to attempt to prevent their addition to the SCS. The performance of the queries is not necessarily good enough to run during a commit. If these queries were to be run at commit time they might make it difficult for a programmer to figure why their commit is not going through. Thus due to performance and possible headaches for the SCS users, it would be beneficial to run queries during off times.

Once SCQL is integrated with a metrics suite, managers can ask even more complex questions.

CVS servers often support external authentication. It is possible to integrate SCQL into an authentication module such that real commits could be authenticated. Caching the previous results of a query to allow for speedups with regards to access control. Since changes are only increment the old results can be combined with an evaluation of the new incremental changes.

## 5.3 Asking questions about entities

One of the more useful features of having a system that deals with time is that one can choose an MR and ask about its history as well as the future changes related to a MR. For instance if one were looking for MRs in which a programmer caused a bug, one might look at the MR where the bug is diagnosed and fixed. Then one could look for a MR that revised most of the lines that were changed in the bug fix MR. Using the history of previous couplings between files, one could attempt to predict change propagation [HH04].

Queries that compute metrics can be used with machine learning techniques to classify, cluster, and partition MRs. What makes many of these queries useful is that they produce boolean answers, thus enabling the creation of decision trees to partition entities. By using multiple queries, one can find subsets of entities that share a partial relation.

By partitioning entities we can attempt to cluster and classify such entities. Classifying entities of change is quite valuable as it can give users

a quick overview of changes to repositories. Classification of changes in a SCS [Ger04b, Ger04a] allows for many useful global visualizations as well providing potentially useful subsets of entities to query.

These classifications can be joined together to provide metrics for change or even quality attributes of a software project. Even more importantly a subset of queries could provide information on the health of a software project. If a developer can be given possible indicators of a project's health, measures can be taken to circumvent foreseeable project difficulties or improve the quality of the software project itself.

Possible metrics SCQL can provide:

- Count of how many unique authors previously modified the files which the current MR modified. In this query we want to get results per MR, so we'll use the print function to print the MR identifier and the number of unique previous authors. To count the previous authors we check each author if they were responsible for immediately previous revisions to the same file. This is a coarse grained measurement which suggests how many authors might be interested or affected by this change (MR). If the MR has a large number of previous unique authors it might suggest that the MR is a bug fix MR (or that there is not a lot of code ownership), where as only one unique author may suggest that an author was previously working on these files.

  Note how `r.prevrev` is used to access the immediately previous revisions to the revisions of the current MR.

```
A(m,MR) {
  print(concat(m.mrid,
    count(S(a,Author) {
      E(r,m.revisions) {
        E(r2,r.prevrev) {
          isAuthorOf(a,r2)
        }
      }
    })
  ))
}
```

- Per author, print the percentage of revisions for the file made by the current author. Again, this would be a possible code ownership metric.

```
A(a,Author) {
  print(a.userid,
      numberToStr(
        100 *
        count(S(s,Revision) {
          isAuthorOf(a,s) && isRevisionOf(s,f)
        }) /
        count(S(s,Revision) { isRevisionOf(s,f) })
      )
  )
}
```

Measures of quality of a software project which can be derived from the repository could be:

- Percentage of MRs that span modules. A large percentage of MRs spanning multiple modules could indicate that the code is highly coupled. There has already been much research suggesting that source files changed at the same time are possibly coupled [GH05]:

```
count(
 S(a,MR) {
   E(e,a.files) {
     E(f,a.files) {
       neq(e.modules,f.modules)
     }
   }
 }
) /  count(MR)
```

- The ratio of the number of files of a specific module that a specific author has modified divided by the total number of all the files that the author has modified. This could be used to rank authors by how coupled they were to one module. It might suggest that an author is focusing on a specific module. In this example we are using "module-name" and "userid" as constants to specify both a specific author and a specific module.

```
count(S(f,File) {
  eq(f.module,"modulename") &&
  E(f.authors) {
    eq(a.userid,"userid")
  }
}) /
count(S(f,File) {
  E(f.authors) {
    eq(a.userid,"userid")
  }
})
```

## 5.4 Legal Questions And Responsibility

### 5.4.1 SCO Case

Often there are issues such as the SCO vs IBM [Tay03] case regarding who was responsible for adding code to a product, or the extent to which a programmer made changes to the system. By being able to generate queries about who changed what and the extent of a user's changes, one would be able to provide factual evidence in court.

> SCO refused to show me the revision history of the Unix file. I pointed out this made it impossible to judge the order of deriva-

> tion; SCO agreed, and said it was a matter of discovery for the court case. SCO said it is confident the code had not appeared in BSD and was developed internally at AT&T and successors. – Ian Lance Taylor on Linux Journal [Tay03].

Initially SCO made claims that IBM had submitted SCO code into the Linux kernel. If the was tainted code it would be very important to find all changes to the tainted files, or files which were modified at the same time as to determine the propagation of code taint. Using propagation metrics [HH04] SCQL could help determine the extent of the propagation of misappropriated code. Possibly propagation could be determined by all those files that were modified together with the misappropriated code.

### 5.4.2 Malicious Linux Code

During 2003, a malicious attempt was made to insert a root exploit in the Linux kernel source code [And03]. The attempt was caught because it was a change to a CVS repository and at the time the Linux kernel used BitKeeper. CVS was still used because some developers didn't want to use BitKeeper and BitKeeper could export to CVS repositories. The malicious change was luckily caught by BitKeeper during an update of the CVS tree; some checksums did not match and thus the maintainer was notified.

It seemed that the server that the CVS repository was hosted on was compromised and a change was made to the CVS repository holding the

Linux kernel.

How could have SCQL helped? SCQL could have been used to ask the Question, "is this programmer modifying something which they have never touched before, or is a programmer modifying something relatively stable?". Let `N` be a threshold for the number of days before something is considered stable.

```
Eafter(a,Revision,yesterday()) {
  A(f,a.files) {
    Ebefore(b,f.revisions,a) {
      days(a.time) - days(b.time) < N
    }
  }
}
```

## 5.5 Invariant Testing

Invariant testing is testing of invariants on software projects. SCQL was built explicitly to do this. To test an invariant in SCQL one translates the query to the SCQL query language and then runs that query against a project.

There are two main kinds of invariants, project wide and software wide. Project wide invariants are invariants which are true for the whole project at all times. Software wide invariants are invariants which for all projects are

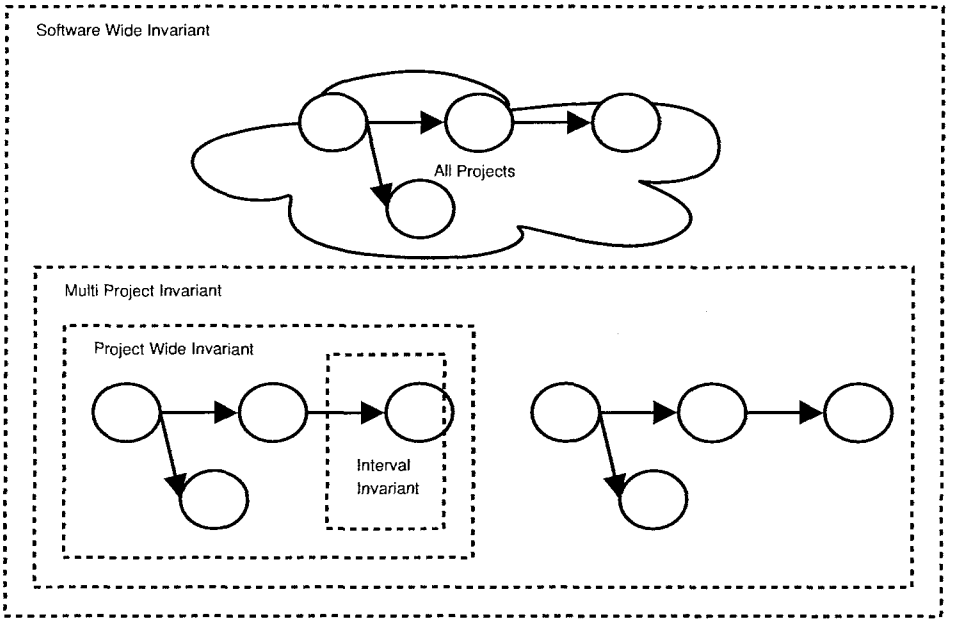


Figure 5.1: Invariant Hierarchy

true. A set of invariants on one project may not be invariants on another project.

Invariant testing can be used to test for strict rules regarding the evolution of a project or other software in general. They can suggest a strictness or rigidity of development.

## 5.6 Invariant Discovery

Invariant discovery is when one or more projects are queried, either manually or automatically, for an invariant. Invariants can exist over time intervals (an invariant could exist during the initial development but not be true during maintenance), over entire project histories (project invariants),

across projects or for all projects (software invariants). Figure 5.1 visually describes the hierarchy of the various kinds of invariants. Invariants which are project invariants but not global software invariants are good candidates for computing metrics which can be used to compare projects. The fact that it is not a software invariant suggests that it can be used to compare projects. The metric could be used as a feature in a feature set used for clustering, or for determining project similarities. Therefore, even project wide invariant an might be useful when used with other metrics.

For manual invariant discovery one needs to have an interface through can ask one query of multiple projects. For automatic discovery one needs a query generator and a multiplexing interface. With such a system we can search for invariants automatically and verify those invariants across multiple projects. Essentially a query generator would generate queries. Possible invariants could be tested across multiple projects. If an invariant was true across many of the projects it could be recorded for further evaluation. Automatic invariant discovery could be used to find good queries to act as features which describe a project.

Issues facing automatic invariant discovery are of the following:

- the search space of queries for invariants is infinite in size;
- there are too many predicates to test ;
- randomly generated invariants could be very naive and thus have a large time complexity. For example, a naive query could easily take

$O(n^2)$ time:

```
A(a,MR) {
  A(b,MR) {
    eq(b.mrid,b.mrid)
  }
}
```

- for a large data set of nearly a million entities this could take a long time .

We have created a query generator and learned about a few of the issues that face automatic query generation. One issue was that generating semantically correct queries (appropriate types used in functions). Even more important is generating queries which make sense to us. Queries that don't make sense would include queries which compare attributes which are unrelated (revisionID vs mrID).

There are many attributes that can be operated on, but many are not related. Thus if they were used together, e.g. `eq(a.timerev,a.mrid)`, it would be meaningless to evaluate. Such predicates could be trivially true, trivially false, or random.

The parameters of a function, scope, or operator must be chosen according to their type in order to produce semantically correct queries. Not all semantically correct queries are useful. Therefore, we found it beneficial to

codify the relationships between attributes since attributes are the base data that is queried.

By creating a table of relations between attributes of the same type one can probabilistically add weight to attributes that are related to each other. This way, when a query is randomly generated, there is a higher probability that the attributes used as parameters in a function are semantically relevant to each other, while still allowing for all possible valid choices.

Since the search space is so large, randomization is used to recursively evaluate possible tree paths for generation of queries. This allows us to take a sample of the search space rather than exhaustively iterating through the search space. Instead of a linear traversal recursively through the query space, we take random recursive paths. The choices are not totally random, if one random choice fails, the next will be chosen until all possibilities are exhausted.

Queries can be generated very fast. In a few minutes, 100000 queries or more can be generated. Evaluation of each of these queries can take a much longer time so we want a system to rank these queries for usefulness or value. By value we mean how a user would rank the query: would they think it was redundant? Would it be useful? Were there better ways to express the same query? Such a system would be much more difficult to create as it would have to encode a value system and rank the aspects of a valuable query.

Examples of queries that contain worthless parts could be sub queries such as `a || !a` or `eq(a.b,a.b)` . Other traits of bad queries would be

repeating the same predicate, e.g., `P(a) || P(a)` . Another bad trait is a useless scope, for example:

```
A(a,Author) {
  A(b,author) {
    P(a)
  }
}
```

If scope is created a variable will iterate over a domain. If that variable is not accessed or used with the predicate that is inside the scope, it is a wasteful query which may increase the complexity of the query by an order of magnitude or more.

## 5.7 Metrics

We have shown many examples of SCQL queries being used to compute metrics; thus we suspect that SCQL would be most useful in querying and calculating metrics from existing projects, as well as discovering new useful metrics. We would like to see existing metrics tools integrated into our engine. By providing functional access to metrics, queries could become even more powerful without much trouble. One method of metric integration is to cache the evaluated metrics in a separate table in the softChange database. These results can thus be easily queried and used as attributes in SCQL or just as

functions.

```
Std(a,MR) { a.methodsAdded }
```

This example query, the Standard Deviation of methods added per MR, might be useful for determining how variable the changes that occur to a system are to the system over time. Metrics of incremental change are especially relevant to a system dedicated to query repositories of change.

Examples of evolution appropriate metrics come from research by German and Hindle [GH05]. These metrics include coupling metrics which measure the proportion of total changes that the two entities are changed together, strength of couplings between authors and entities such as files or modules, and many measurements of changes such as revisions and MRs. German et al. also covers both SCS entities and architectural entities of the code. The paper specifically covers metrics of change (such as an MR) which are measurements of the change itself rather than just the difference.

# Chapter 6

# Evaluation

In this section we provide sample queries created using SCQL, additionally, we evaluate these queries against various open source projects. Many of the sample queries can be used as metrics for describing entities or projects.

We evaluated these queries on the extracted data from various open source projects: Gnumeric, a spreadsheet program for the GNOME project; Evolution, an email client similar to Microsoft Outlook for the GNOME project; Postgresql, a RDBMS; Xerces, an XML parser from the Apache project; OpenSSL, a secure socket layer implementation; Samba, Windows network file system support for UNIX; and mod_perl, an Apache module for running Perl web applications.

These projects were chosen by their availability of their repositories, their popularity, random choice, and some, such as Evolution, and Postgresql were chosen because they were common benchmarks used in the MSR community.

The next 3 groups of queries were run across different sets of projects due to time constraints for larger queries for larger projects, as well as to allow for more projects to be evaluated. This section is a demonstration of queries being run across multiple projects.

## 6.1 Sample Queries

These queries were run against the multiple projects. Some queries produce the same results across many projects, which suggests that it may be a multiple project invariant.

**Example 1**: Is there an author $a$ who only modifies files that author $b$ has already modified? This query can be formally expressed as:

$$
\begin{aligned}
&\exists a, b \in \mathbb{A}uthor \quad \text{s.t.} \\
&\quad a \neq b \wedge \\
&\quad \forall r \in revisions(a) \quad \text{s.t.} \implies \\
&\quad\quad \forall f \in files(r) \quad \text{s.t.} \implies \\
&\quad\quad \exists r_b \in revisions(f) \quad \text{s.t.} \; before(r_b, r) \wedge \\
&\quad\quad\quad isAuthorOf(b, r_b)
\end{aligned}
$$

We have used this query before in section 3.8 but now we are going to further explain and evaluate it. We are trying to find two different authors

such that for all revisions of one author, there exists a previous revision (by the second author) to the same file. The SCQL query first finds two authors and makes sure each is different. It then iterates through all the revisions of author $a$. For each revision, it checks if the file of that revision has another previous revision that belongs to author $b$. `a.revisions` obtains all the revisions related to the author $a$ while `isAuthorOf(b ,r2)` tests if $b$ is the author of the revision of the file $f$.

```
E(a, Author) {
  E(b, Author) {
    a!=b &&
    A(r, a.revisions) {
      A(f, r.file) {
        Ebefore( r2, f.revisions, r) {
          isAuthorOf( b, r2)
        }
      }
    }
  }
}
```

**Example 2**: Compute the proportion of MRs that have a unique set of files that have never appeared as part of another MR before. With this query we want to find out how variable are the sets of files modified in MRs. We

hypothesize that an old, stable project will have a small proportion, while a project that is still growing, and continues to have structural changes, will have a larger proportion. This query can be expressed directly in SCQL as:

```
1 - (Count(mr,MR) {
      Ebefore(a,MR,mr)) {
         A(f,mr.files) {
           isFileOf(f,a)
         }
       }
    } / count(MR)
```

It iterates over the set of all MRs, counting only those that have a previous MR that modifies all its files. It then counts all MRs and computes the desired proportion.

**Example 3**: Is there an Author whose changes stay within one directory?

$$\begin{aligned} &\exists a \in \mathbb{A}uthor \quad \text{s.t.} \\ &\quad \forall f \in files(a) \quad \text{s.t.} \\ &\quad \forall f_2 \in files(a) \quad \text{s.t.} \\ &\qquad eq(directory(f), directory(f_2)) \end{aligned}$$

In this case we want to know if there exists an author such that for all pairs of files modified by this author, they are both in the same directory. This query can be written in SCQL as:

```
E(a, Author) {
  A(f, a.files) {
    A(f2, a.files) {
      eq(f.directory, f2.directory)
    }
  }
}
```

## 6.2 Evaluation of Sample Queries

We have built an implementation of SCQL. In order to demonstrate the effectiveness of SCQL we ran the 3 example queries against five different projects: Evolution (an Email Application), Gnumeric (a spreadsheet), OpenSSL (A Secure Socket Layer library), Samba (Linux support for Win32 network file systems), and modperl (a module for Apache that acts like a Perl Application server). The table 6.1 provides the output of the 3 example queries for each of these projects. We include the size of the $\mathbb{MR}$ set (number of MRs) and the $\mathbb{F}ile$ set too.

## 6.3 Example Queries

The following queries were evaluated using the extracted data from: Gnumeric, Evolution, Postgresql, Xerces.

Table 6.1: Evaluation of the 3 example queries

| | evolution | gnumeric | openssl | samba | modperl |
|---|---|---|---|---|---|
| Ex 1 | true | true | false | false | true |
| Ex 2 | 0.002 | 0.004 | 0.003 | 0.002 | 0.015 |
| Ex 3 | false | false | false | false | true |
| $\|\mathbb{F}ile\|$ | 4748 | 3685 | 3698 | 4246 | 300 |
| $\|\mathbb{MR}\|$ | 18573 | 11337 | 10847 | 27413 | 1398 |

**Similar Changes Interval**: During the past 30 days has there been a repeated change to the repository. A repeated change is an MR that has revisions of the same subset of files as a previous MR in that interval.

Using the `days()` function we check the day of the current MR and the MR before it. By testing the filenames modified by the revision, we shortcut accessing the file.

```
E(a,MR) {
  Ebefore(b,MR,a) {
    { days(a) - days(b) <= 30 &&
      A(x,a.revisions) {
        E(y,b.revisions) { eq(x.filename,y.filename) }
      }
    }
  }
}
```

**One Revision**: There exists a file that has only 1 revision.

$$\exists f \in \mathbb{F}ile1 = |\{r \in Revision | isFileOf(r, f)\}|$$

This can be expressed using `f.revisions`:

```
E(f,File) {
   count(a,f.revisions) { true } = 1
}
```

**Small Committers:** The number of authors who only commit less than five files per MR. To get the number of authors we want to produce a subset of authors and the count the number of authors in the subset.

```
count(
S(a,Author) {
  A(b,a.mrs) {
    count(b.revisions) < 5
  }
})
```

**Total Line Diff:** The sum of all the lines added and removed.

```
Sum(a,Revision) { a.linesadd - a.linesrm }
```

**Quiet Authors:** For a project do all the authors submit MRs with no log comment?

```
A(a,Author) {
  E(b,a.mrs) {
    length(b.log) < 1
  }
}
```

**Log Commenters:** For a project do all the authors submit log comments on average of less than 100 characters?

```
A(a,Author) {
  100 > Avg(b,a.mrs) {
    length(b.log)
  }
}
```

**Module Stick:** Does there exist an author who has at least two revisions in each module that the author has modified?

```
E(a,Author) {
  A(b,a.revisions) {
    E(c,a.revisions) {
      b != c && A(d,b.file) {
        A(e,c.file) {
          eq(d.module,e.module)
        }
      }
    }
  }
}
```

**C Files:** How many C source files are in the repository?

```
count(S(a,File) {
  endMatch(a.filename,".c")
})
```

**Java Files:** How many Java source files are in the repository?

```
count(S(a,File) {
  endMatch(a.filename,".java")
})
```

**.H Files:** How many C .h source files are in the repository?

| Name | | Evolution | Gnumeric | Postgresql | Xerces |
|---|---|---|---|---|---|
| Total Code Ownership | | false | false | false | false |
| Subset Changes | | true | true | true | true |
| One Revision | | true | true | true | true |
| Small Committers | | 37 | 28 | 0 | 2 |
| Total Line Diff | | 12403454 | 16563594 | 6394136 | 2495942 |
| Quiet Authors | | false | false | false | false |
| Log Commenters | | false | false | false | false |
| Module Stick | | true | true | true | true |
| C Files | | 1195 | 547 | 1072 | 46 |
| Java Files | | 0 | 0 | 273 | 1767 |
| H Files | | 1063 | 473 | 1012 | 90 |
| C++ Files | | 0 | 0 | 0 | 1020 |

Table 6.2: Comparison of queries on various projects

```
count(S(a,File) {
  endMatch(a.filename,".h")
})
```

**C++ Files:** How many C++ source files are in the repository?

```
count(S(a,File) {
  endMatch(a.filename,".cpp")
})
```

Table 6.2 compares the results of all these queries.

## 6.4 Even More Example Queries

This section exists to further demonstrate SCQL and to show that the enforced invariants that were used to create the graph hold true (e.g. cardinalities of edges).

These queries are provided as an example of what else can be asked of SCQL. These queries were evaluated on Gnumeric, mod_perl, OpenSSL, Rsync (a file synchronizer) and Xerces. The results of this evaluation can be found in table 6.3. The index column correlates to the numbered query in this section. Sub indexes are used for examples that consisted of more than one related query (the sub indexes are assigned in the order than the example query appears).

1. There exists a last revision. This is a check of the data to make sure there are end points in the revision graph.

```
E(a,Revision) {
  count(
    S(b,a.nextrev){
      true
    }) == 0
)
```

2. How many authors only made 1 single commit.

| Index | Gnumeric | mod_perl | OpenSSL | Rsync | Xerces |
|---|---|---|---|---|---|
| 1 | true | true | true | true | true |
| 2 | 30.00 | 1.00 | 2.00 | 1.00 | 4.00 |
| 3.1 | 31504.00 | 2857.00 | 19657.00 | 2260.00 | 6636.00 |
| 3.2 | 462.00 | 251.20 | 2976.81 | 555.55 | 955.84 |
| 3.3 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 4.1 | 31966.00 | 751.00 | 4201.00 | 5530.00 | 52213.00 |
| 4.2 | 125.60 | 7.38 | 21.51 | 19.57 | 36.10 |
| 4.3 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| 4.4 | 9052448.00 | 27806.00 | 1024538.00 | 119605.00 | 1552636.00 |
| 5.1 | 305.00 | 106.00 | 933.00 | 49.00 | 872.00 |
| 5.2 | 6.36 | 2.70 | 4.39 | 1.86 | 3.94 |
| 5.3 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 6 | true | true | true | true | true |
| 7 | true | true | true | true | true |
| 8 | true | true | true | true | true |
| 9.1 | true | true | true | true | true |
| 9.2 | false | true | false | true | false |
| 10 | false | true | false | false | false |
| 11 | 4.34 | 2.85 | 3.19 | 5.96 | 1.97 |
| 12 | true | true | true | true | true |
| 13 | false | true | false | false | false |
| 14 | 0.84 | 0.84 | 0.79 | 0.86 | 0.68 |
| 15 | 0.17 | 0.31 | 0.19 | 0.46 | 0.23 |
| 16 | true | true | true | true | true |
| 17 | true | true | true | true | true |
| 18 | true | true | true | true | true |
| 19 | true | true | true | true | true |
| 20 | true | true | true | true | true |
| 21 | true | true | true | true | true |

Table 6.3: Results of running queries from section 6.4 on Gnumeric, mod_perl, OpenSSL, Rsync and Xerces

```
count(S(a,Author) {
  count(S(b,a.mrs) {
    true
  }) == 1
})
```

3. Maximum, average, and minimum number of revisions per author.

```
Max(a,Author) { count(S(b,a.revisions) {true}) }
Avg(a,Author) { count(S(b,a.revisions) {true}) }
Min(a,Author) { count(S(b,a.revisions) {true}) }
```

4. Maximum, average, minimum, and summation of the number of lines per revision.

```
Max(a,Revision) { a.linesadd }
Avg(a,Revision) { a.linesadd }
Min(a,Revision) { a.linesadd }
Sum(a,Revision) { a.linesadd }
```

5. Maximum, average, and minimum numbers of revisions per author.

```
Max(a,MR) { count(S(b,a.revisions){true}) }
Avg(a,MR) { count(S(b,a.revisions){true}) }
Min(a,MR) { count(S(b,a.revisions){true}) }
```

6. Check to see that every MR has only 1 author.

```
A(a,MR) { count(S(b,a.authors){true}) == 1 }
```

7. All authors have changed more than 1 file.

```
E(a,Author) {
  count(S(b,a.files){1}) > 0
}
```

8. All the files belonging to an author have more than one revision.

```
E(a,Author) {
  A(b,a.files) {
    count(S(c,b.revisions){1}) > 0
  }
}
```

9. For all revisions related to an author, that author is the author of the revision.

```
E(a,Author) {
  E(c,a.revisions) {
    eq(a.userid,c.author)
  }
}

E(a,Author) {
  A(b,a.files) {
    E(c,b.revisions) {
      eq(a.userid,c.author)
    }
  }
}
```

10. There exists an author who only modified files that were previously modified.

```
E(a,Author) {
  A(b,a.revisions) {
    Ebefore(c,Revision,b) {
      neq(c.author,a.userid) &&
      eq(b.filename,c.filename)
    }
  }
}
```

11. The average number of revisions per file of an author.

```
Avg(a,Author) {
  Avg(y,a.files) {
    count(S(x,y.revisions) {
      eq(x.author,a.userid)
    })
  }
}
```

12. There exists a file such that for all MRs that include revisions of this file stay within the same directory, the other revisions are in the same directory as well.

```
E(a,File) {
  A(mr,a.mrs) {
    A(file,mr.files) {
      (file != a) ->
      eq(file.directory,a.directory)
    }
  }
}
```

13. There exists an author who for all files they have modified, those files reside in the same directory.

```
E(author,Author) {
  A(filea,author.files) {
    A(fileb,author.files) {
      eq(fileb.directory,filea.directory)
    }
  }
}
```

14. The average per revision of all the revision authors modify the file before.

```
Avg(a,Revision) {
  int( A(author,a.authors) {
    Ebefore(rev,author.revisions,a) {
      eq(a.filename,rev.filename)
    }
  })
}
```

15. Average per MR over the average number of MRs before the current MR that has revisions of files in the same directory as the current MR.

```
Avg(mr,MR) {
  count(S(a,MR) {
    before(a,mr) &&
    A(f,a.files) {
      E(f2,mr.files) {
        eq(f.directory,f2.directory)
      }
    }
  }) /
  count(
  S(a,MR) {
    before(a,mr)
  })
}
```

16. Check to see that every revision has only 1 author.

```
A(a,Revision) { count(S(b,a.authors){1}) == 1 }
```

17. Check to see that every revision has only 1 file.

```
A(a,Revision) { count(S(b,a.files){1}) == 1 }
```

18. Check to see that every revision has only 1 MR.

```
A(a,Revision) {  count(S(b,a.mrs){1}) == 1 }
```

19. All authors have revisions.

```
A(a,Author) { count(S(b,a.revisions){true}) > 0 }
```

20. All authors have MRs.

```
A(a,Author) { count(S(b,a.mrs){true}) > 0 }
```

21. There exists both revisions with no revisions before or no revisions after.

```
E(a,Revision) { count(S(b,a.prevrev){true}) == 0 } &&
E(a,Revision) { count(S(b,a.nextrev){true}) == 0 }
```

# Chapter 7

# Future Work

In this section, we discuss query optimization, extensions to both the model and the language, improved integration with merge point identifiers, and the use of SCQL to generate features of entities for machine learning.

## 7.1 Query Optimization

One of the downsides of implementing an engine such as this is that huge issues such as query optimization must be handled by us. This suggests that naive queries can easily run in $O(n^2)$ or greater time such as `A(a,MR) { E(b,MR) { after(a,b) } }` .

To reduce query run time, one should use a query optimizer. Simple optimizations, such as balancing expressions, could take advantage of short circuiting boolean logic.

Other optimizations include caching results or creating temporary tables of partial results, that may be used to speed up repetitive queries. A simple optimization could include identifying variables that are not being used; and thus, ignoring their scope. Some optimizations may seem trivial; however, if one is running automatically generated queries, avoiding wasted computation is helpful.

We did not use systems such as Prolog because of issues regarding how to integrate a Prolog system with a conventional SQL database. Other issues that concerned us was the need to have over 500 000 data elements, could our Prolog implementations handle that much? Many of our queries are iterative, that is they are a linear iteration through subsets, thus the benefits of backtracking are questionable.

Both XQuery and SQL, although very powerful, require identifiers or primary keys to work with the graph model. The edges of the model cause much difficulty when querying the data through SQL or XQuery. We could have modeled the system using a hierarchical structure but we would have run into problems with entities that are not in the same hierarchy, or are in super hierarchies. Both XQuery and SQL were easy to use from a hierarchical, top-down approach, such as MR (which contains revisions), but this structure did not afford the same amount of navigability in the opposite or different directions (e.g., MRs of a file, or revisions of a file).

Future work could be done in automatically translating SCQL queries into other query languages (like XQuery, SQL, Prolog) in order to gain the

performance benefits of those systems. Reducing the SCQL implementation to a tree transformation engine would reduce complexity and offload maintenance and performance issues onto the underlying data store.

## 7.2 Model Extension

To extend the model, future work could focus on the expansion of either the formal or the instantiated model of a source code repository to include the entities of source code such as classes, methods, and associations, functions, etc.

If we were to add architectural entities to the model, we would have to add elements of architectural change. Some of the architectural modeling might be based on work done by Render and Campbell with their object oriented model of a SCS. [RC91].

Some aspects of architecture are easily extracted and do not require human help to identify. These are the most valuable facts as they are true; such facts would be method definitions, parameters and other code level facts. Fact extraction on dynamically typed systems poses a challenge. By focusing on facts based on real entities in the code we can guarantee answers which are as accurate as the fact extractors used.

The extension to the SCQL model would include architectural revisions. That means for any architectural entity, it could be added, removed or modified.

Once the architecture of a program has been modeled and linked against the evolution of the program (the changes to the program that occur through the source code repository) we can further validate and verify some of Lehman's laws that relate to complexity and incremental change.

Non-architectural additions could be included as well. These could be external sources, such as emails on mailing lists related to the entities, or perhaps Bugzilla entries. By adding non-architectural entities we enable our queries to ask questions regarding the level of developers activity at the time particular entities where created. Questions could be inquiring about the amount of activity that is recorded inside and outside the repository when a bug is being fixed. The bug would be identified by its Bugzilla ID, allowing queries to anchor around the identifiers of already identified entities.

The model could include better support for intervals and units of time. A time domain could be created that is partitioned into months, weeks, and days. An example of the improved expressiveness interval wise could be

```
E(d,Day) {
  A(mr,d.mrs) {
    P(mr)
  }
}
```

instead of

```
E(mr,MR) {
  Aafter(mr2,MR,mr) {
    day(mr2) - day(mr) <= 1 ->
    P(mr2)
  }
}
```

## 7.3 Query Language Extension

If the model is extended, the query language should be extended as well. The query language would have to the cover the new domains such as classes, methods, functions, attributes, and associations. As well, domains would have to include revisions of these new elements.

At the functional level, the addition of more statistically oriented functions and maps would be quite useful. Functions related to statistical distributions would be very useful. A Fourier transform could be used to extract frequency data, this makes it easier to reasoning about the frequency of events.

User defined functions could be useful as they would provide even more expressive power to the language (unfortunately, the query language could no longer provably halt unless there was no recursion and the predicates were created from the query language itself). Lambdas could allow users to make and abstract parameterized queries.

A possible improvement for iterating through the graphs would be a

XPath kind of syntax: `E(a,'Revision/Files/Author') { P(a) }` . This example could be translated to `E(r,Revision) { E(f,r.files) { E(a,f.authors) { P(a) }}}` . This kind of query would enable a globally relationally path based query. An added benefit over XQuery would be that the paths specified would be edge traversal in a non-hierarchical model, whereas XQuery and XPath require hierarchical XML data structures.

## 7.4 Branch Merge Points

We have made an assumption that branch merge points can be identified. The extractor used should be modified to include the latest research regarding the identification of branch merge points from CVS.

We can only act on the information we have and how we choose to annotate it. Whether CVS branch merges should be identified is an issue of whether we want queries based on our interpretation of the project (a merge point detection algorithm could be wrong), or on the actual data that was recorded (of course, we assume that our MR detection algorithms are accurate). A pleasant side-effect of improving the fact extractor is that, in our model, our queries would become more expressive and accurate.

## 7.5 Machine Learning

By using the results from the various queries evaluated in SCQL, we can attempt to partition, group, and cluster projects and their entities. Further investigation is needed to find and choose queries that can be used to create a feature vector. This feature vector is a point in n-dimensional space that can be used to partition and cluster the projects that the points represent.

One could then manually evaluate the projects that were grouped together to determine if they had similar development styles, project sizes, or commit styles. Projects such as Mozilla and Apache differ from each other as the Apache project requires the developers to vote on patches, whereas Mozilla gives developers CVS access and they are free to make changes without committee approval.

It could be valuable if, by extracting a feature vector and clustering or by similarity testing, we could make the claim that project A is more related to Apache than to Mozilla in development style and developer behavior than project B.

Future work could include applying machine learning techniques to classify the development style of projects. For instance a classifier could attempt to determine if one project follows Apache CVS style commit conventions versus Mozilla style commit conventions.

Of course, the results of the partitioning would have be validated manually. Initially some projects would have to be studied and manually classified.

Thus, using their feature vectors and machine learning techniques, future projects could be evaluated without the manual labor of the initial projects.

Machine learning techniques such as cluster analysis have already been used in software evolution research, Ball et al [BAHS97] derives the probability that two classes are modified at the same time using cluster analysis.

We have already used machine learning techniques with SCQL. Using SCQL we calculated metrics for all the revisions in various projects like PostgreSQL. Using this data we asked the question "Can we identify the MRs which occur around a release, even when we don't know when the release actually occurred?", we asked this question of a minimum amount of data, the data and relations stored within a SCS. We had no source code level metrics. Unfortunately machine learning did not produce reliable or positive results.

Figure 7.1 shows the average MRs per day of Postgresql correlated with the releases (the vertical lines), the marks at the bottom of the graph indicate an MR which were identified as release MRs by the Bayesian Network. Note the virtual lack of correlation between releases and predicted releases. The training data used was correlated release data from many other projects. At least these experiments indicate that there probably isn't enough information to allow classifiers to operate with a reasonable amount of accuracy.

See table 7.1 for a summary of the effectiveness of each machine learning algorithms with the Postgresql project. The classifiers were from the WEKA project [IH99]. "Pre." is precision as in Precision and Recall, the higher

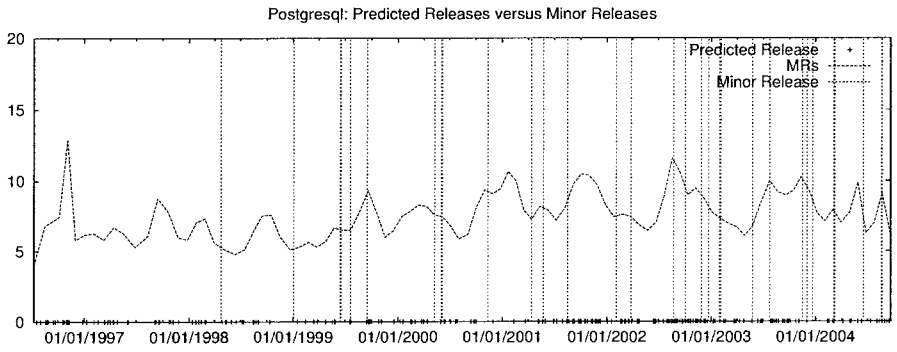


Figure 7.1: MRs classified as releases correlated with releases

| Classifier | Correct | Work Pre. | Work Recall | Release Pre. | Release Recall |
|---|---|---|---|---|---|
| BayesNet | 77% | 0.862 | 0.876 | 0.170 | 0.153 |
| Naive Bayes | 85% | 0.857 | 0.981 | 0.120 | 0.016 |
| $K*$ | 82% | 0.859 | 0.946 | 0.163 | 0.063 |
| 1-NN | 70% | 0.855 | 0.788 | 0.131 | 0.193 |
| Perceptron | 81% | 0.855 | 0.946 | 0.096 | 0.034 |

Table 7.1: Results of various machine learning classifiers on classifying Postgresql aggregate MRs

these numbers the better the results. In general less than 10% of the MRs were release MRs.

# Chapter 8

# Summary

This research is important as it attempts to enable provably correct empirical studies of software evolution based on the actual data that was recorded when the software was developed. This is especially important as the analysis of data extracted from industry and OSS projects is invaluable for bridging the gap between theory and practice; this allows researchers to generate new theories about software evolution based upon real data from real software projects.

This research attempted to use a minimal amount of information (the information in the SCS rather than the source code) to answer various queries and formulate new ones. Throughout the research the limitations of the fine grained level of granularity and the lack of architectural information became apparent, especially for testing Lehman's Laws of Software Evolution. Many of Lehman's Laws required architectural information. This project focused

on invariant querying and testing with regards to incremental changes in a SCS.

The research consists of a theoretical model of SCS – formally defined to allow for answers to queries upon the model to be proved correct. Built upon this model is a formally defined query language built from primitives to allow for an easily composable and provably correct query language.

A query engine, based on both the model and query language, that answers queries based on the extracted data from real CVS repositories, was created. The engine was used to query data from the extracted repositories of such OSS projects as Evolution and Xerces.

Most importantly was that an effort was made to make domain specific tools (a query language and engine) for mining software repositories and the data they hold.

There is much in the way of future research, whether it be further invariant discovery manually or automatically, query optimization, model extension, architectural modeling, machine learning or metrics of fine grained changes.

# Bibliography

[AHV96] Serge Abiteboul, Laurent Herr, and Jan Van den Bussche. Temporal versus first-order logic to query temporal databases. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 49–57, 1996.

[And03] Jeremy Andrews. Linux: Kernel "Back Door" Attempt. `http://kerneltrap.org/node/view/1584`, November 2003. Accessed July 2005.

[AS92] Bernd Amann and Michel Scholl. Gram: a graph data model and query languages. In *Proceedings of the ACM conference on Hypertext*, pages 201–211. ACM Press, 1992.

[BAHS97] Thomas Ball, Jung-Min Kim Adam, A. Porter Harvey, and P. Siy. If your version control system could talk. In *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.

[BGHS04] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of Fifth Internationals Conference on Verification,Model Checking and Abstract Interpretation (VMCAI 04)*, 2004.

[BM88] David Beech and Brom Mahbod. Generalized version control in an object-oriented database. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 14–22, Washington, DC, USA, 1988. IEEE Computer Society.

[Cas03] Steve Cassidy. Generalizing XPath for directed graphs. In *Proceedings of the Extreme Markup Languages Conference*, 2003.

[Chu04] Derek Church. A survey of techniques for the recovery and observation of software evolution. Unpublished, 2004.

[CM03] Davor Cubranic and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 2003 International Conference on Software Engineering*, pages 408–418, Portland, May 2003. Association for Computing Machinery.

[Col04] CollabNet. Subversion FAQ. `http://subversion.tigris.org/faq.html`, 2004. Accessed July 2005.

[Cor04] Microsoft Corporation. Microsoft Visual Source Safe. `http://msdn.microsoft.com/vstudio/previous/ssafe/productinfo/default.as%px`, 2004. Accessed July 2005.

[CTB01] Jan Chomicki, David Toman, and Michael H. Bhlen. Querying ATSQL databases with temporal logic. *ACM Transactions on Database Systems*, 26(2):145–178, 2001.

[CW97] Reidar Conradi and Bernhard Westfechtel. Towards a uniform version model for software configuration management. In *International Conference of Software Engineering '97: Proceedings of the SCM-7 Workshop on System Configuration Management*, pages 1–17, London, UK, 1997. Springer-Verlag.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[Dar91] Susan Dart. Concepts in configuration management systems. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 1–18, New York, NY, USA, 1991. ACM Press.

[DBL04] *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*. IEEE Computer Society, 2004.

[DP03] Dirk Draheim and Lukasz Pekacki. Process-centric analytical processing of version control data. In *Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 131–136. IEEE, 2003.

[FG97] Michael Fischer and Harald Gall. Software Evolution Observations Based on Product Release History. In *Proceedings of the International Conference on Software Maintenance (ICSM ' 97)*, September 1997.

[Fou03] Free Software Foundation. The Revision Control System (RCS). `http://www.gnu.org/software/rcs/rcs.html`, 08 2003. Accessed June 2005.

[Fou04a] Free Software Foundation. CVS. `http://www.gnu.org/software/cvs/`, 2004. Accessed July 2005.

[Fou04b] Free Software Foundation. CVS Info Manual. info cvs (GNU Info Page), 2004. Accessed July 2005.

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, September 2003.

[FSG04] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting facts from open source software. In DBLP:conf/icsm/FerencSG04 [DBL04], pages 60–69.

[Gad88] Shashi K. Gadia. A homogeneous relational model and query languages for temporal databases. *ACM Transactions of Database Systems*, 13(4):418–448, 1988.

[GDKZ04] M. Godfrey, X. Dong, C. Kapser, and L. Zou. Four Interesting Ways in Which History Can Teach Us About Software. In *2004 International Workshop on Mining Software Repositories (MSR-04)*, May 2004.

[GDL04] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In DBLP:conf/icsm/2004 [DBL04], pages 40–49.

[Ger04a] D. M. German. Mining CVS repositories, the softChange experience. In *1st International Workshop on Mining Software Repositories*, pages 17–21, May 2004.

[Ger04b] D.M. German. An empirical study of fine-grained software modifications. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, Sept 2004.

[GH05] Daniel M. German and Abram Hindle. Measuring fine-grained change in software: towards modification-aware change metrics. In *Proceedings of 11th International Software Metrics Symposium (Metrics 2005)*, 2005. To be presented.

[GHJ04] Daniel M. German, Abram Hindle, and Norman Jordan. Visualizing the evolution of software using softchange. In *Proceedings SEKE 2004 The 16th Internation Conference on Software Engineering and Knowledge Engineering*, pages 336–341, 3420 Main St. Skokie IL 60076, USA, June 2004. Knowledge Systems Institute.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, 1995.

[GJK98] Harald Gall, Mehdi Jazayeri, , and Jacek Krajewski. CVS Release History Data for Detecting Logical Couplings. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, September 1998.

[GT00] Michael W. Godfrey and Qiang Tu. Evolution in Open Source Software: A Case Study. In *Proceedings of International Conference on Software Maintenance*, pages 131–142, 2000.

[Has05] Ahmed Hassan. Mining software repositories to guide software development. `http://plg.uwaterloo.ca/~aeehassa/home/pubs.html`, 2005. Accessed July.

[HG05] Abram Hindle and Daniel M. German. SCQL: A formal model and a query language for source control repositories. In *2nd International Workshop on Mining Software Repositories*, 2005.

[HH04] Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *Proceedings of ICSM 2004: International Conference on Software Maintenance*, pages 284–293, September 2004.

[IEE90] IEEE. *IEEE Standard 828-1990: Standard for software configuration management plans.* IEEE, New York, NY, 1990.

[IEE98] IEEE. *IEEE Standard 828-1998: Standard for software configuration management plans.* IEEE, New York, NY, 1998.

[IH99] Eibe Frank Ian H.Witten. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor, October 1999.

[Inc04] BitMover Inc. Do you need BitKeeper? `http://www.bitkeeper.com/section4.html`, 2004. Accessed July 2005.

[KS99] Chris F. Kemerer and Sandra Slaughter. An Empirical Approach to Studying Software Evolution. *IEEE Transactions on Softwware Engineering*, 25(4):493–509, 1999.

[Kun04] Hsiang-Jui Kung. Quantitative method to determine software maintenance life cycle. In DBLP:conf/icsm/2004 [DBL04], pages 232–241.

[Leh80] Meir M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

[LFRMGB04] L. Lopez-Fernandez, G. Robles-Martinez, and J. M. Gonzales-Barahona. Applying social network analysis to the information in CVS repositories. In *1st International Workshop on Mining Software Repositories*, May 2004.

[Lor04] Tom Lord. GNU Arch. `http://www.gnu.org/software/gnu-arch/`, 2004. Accessed July 2005.

[LPR+97] M.M. Lehman, D.E. Perry, J.F. Ramil, W.M. Turski, and P.D. Wernick. Metrics and laws of software evolution-the nineties view. In *Proceedings of Metrics Symposium 1997*, Nov 1997.

[LS03] Y. Liu and E. Stroulia. Reverse Engineering the Process of Small Novice Software Teams. In *Proc. 10th Working Con-*

*ference on Reverse Engineering*, pages 102–112. IEEE Press, November 2003.

[Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In DBLP:conf/icsm/2004 [DBL04], pages 350–359.

[MD01a] Tom Mens and Serge Demeyer. Evolution metrics. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, New York, NY, USA, 2001. ACM Press.

[MD01b] Tom Mens and Serge Demeyer. Future trends in software evolution metrics. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 83–86, New York, NY, USA, 2001. ACM Press.

[MFH02] Audris Mockus, Roy T. Fielding, and James Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[MZY01] Hong Mei, Lu Zhang, and Fuqing Yang. A software configuration management model for supporting component-based software development. *SIGSOFT Software Engineering Notes*, 26(2):53–58, 2001.

[Per04a] Perforce. Merge tools. `http://www.perforce.com/perforce/products/merge.html`, 2004. Accessed July 2005.

[Per04b] Perforce. Perforce SCM System. http://www.perforce.com/-perforce/products.html, 2004. Accessed July 2005.

[RC91] Hal Render and Roy Campbell. An object-oriented model of software configuration management. In Peter H. Feiler, editor, *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 127–139, 1991.

[RG01] Muriel Roger and Jean Goubault-Larrecq. Log auditing through model checking. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 220–236. IEEE Computer Society Press, June 2001.

[Rou05] David Roundy. Darcs manual. http://abridgegame.org/-darcs/manual, 2005. Accessed July 2005.

[SC03] Maha Shaikh and Tony Cornford. Version management tools: CVS to BK in the linux kernel. In *Proceedings of 3rd Workshop on Open Source Software Engineering*, pages 127–131, May 2003.

[Sci94] Edward Sciore. Versioning and configuration management in an object-oriented data model. *The VLDB Journal*, 3(1):77–106, 1994.

[Sno87] Richard Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, 1987.

[Tay03] Ian Lance Taylor. Linux Buzz: My Visit to SCO. `http://www.linuxjournal.com/article.php?sid=6956`, June 2003. Accessed July 2005.

[Wik05] Wikipedia. Second order logic. `http://en.wikipedia.org/wiki/Second\_Order\_logic`, June 2005. Accessed June 2005.

[Wu03] Xiaomin Wu. Visualization of Version Control Information. Master's thesis, University of Victoria, 2003.

[XS04] Zhenchang Xing and Eleni Stroulia. Understanding phases and styles of object-oriented systems' evolution. In DBLP:conf/icsm/2004 [DBL04], pages 242–251.

[ZW04] Thomas Zimmermann and Peter Weisgerber. Preprocessing CVS data for fine-grained analysis. In *1st International Workshop on Mining Software Repositories*, May 2004.