SCQL: A formal model and a query language for source control repositories

Abram Hindle
Software Engineering Group
Department of Computer Science
University of Victoria
abez@uvic.ca

Daniel M. German
Software Engineering Group
Department of Computer Science
University of Victoria
dmg@uvic.ca

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 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 five different large software projects.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version Control*; D.2.8 [**Software Engineering**]: Metrics—*Process metrics*

1. INTRODUCTION

A configuration management system, and more specifically, a source control system (SCS) keeps track of the modification history of a software project. A SCS keeps a record of who modifies what part of the system, when and what the change was.

Typically a tool that wants to use this historical information starts by doing some type of fact extraction. These facts are processed in order to create new *information* such as metrics [9, 3] or predictors of future events [6, 7]. In some cases, this information is queried or visualized [5, 10]. Some projects store the extracted facts into a relational database ([8, 5, 2]), and then use SQL queries to analyze the data. Others prefer to use plain text files, and create small programs to answer specific questions [9], or query the SCS repository every time [10]. One of the main disadvantages of these approaches is that querying this history becomes difficult. A query has to be translated from the domain of the SCS history to the data model or schema used to stored this information. Also, questions regarding the temporal aspects of the data are difficult to express. Furthermore, there is no standard for the storage or the querying of the data, making it difficult for a project to share its data or its analysis methods with another one.

When a developer completes a task it usually means that she has modified one or more files. The developer then submits these changes to the SCS, in what we call a *modification request*, or MR (this process has also been called a transaction). A MR is, therefore, atomic (conceptually the MR is atomic, even though it might not be implemented as such by the SCS system). Once the change is accepted by the SCS, it creates a new *revision* for each file present in the MR. Thus an MR is a set of one or more file revisions, committed by one developer. The SCS allows its users to retrieve any given revision of a file, or for a given date, determine what is the latest revision for every file under its control.

There are many SCSs available on the market. They can be divided into two types: centralized repositories (like CVS) and Peer-to-Peer repositories (such as BitKeeper, Darcs, Arch). Even though they differ strongly in the way they operate and store the tracked changes, they all track files and their revisions. We will focus on CVS because there is a large number of CVS repositories available to researchers. We will, therefore, use the CVS nomenclature in this paper. It is important to mention that our model and SCQL can be applied to any SCS.

This paper is divided as follow: first we present an abstract model to describe version control systems; second, we define a query language, called SCQL, we end demonstrating how it can be used to pose questions related to the source control history in several mature, large projects.

2. MODEL

In order to create a language for the querying of a SCS we first need to be able to describe its data model. This data model will be used to formally describe the data available in the SCS and to provide a uniform representation of the information available across multiple SCSs. One of the requirements of this model that is "time aware" and it is able to represent the temporal relationships ("before", "after") of the different entities stored in the SCS.

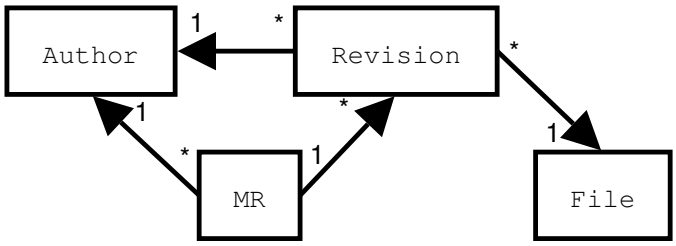

Figure 1: Cardinality and Directions of Edges in the Model

2.1 Characteristic Graph of a Source Code Repository

We represent an instance of a SCS as a directed graph. Entities such as MRs, Revisions, Files and Authors are vertices, while their relationships are represented by edges. It is important that for any given instance of a SCS, there exists a corresponding characteristic graph, and that given a query, this query can be translated into an equivalent graph query on its characteristic graph. As a consequence, the original query will be answered by solving the graph query.

2.2 Entities

The model for SCQL contains four different types of entities: MRs, Revisions, Files and Authors. See figure 1.

MRs model modification requests and correspond to the set $\mathbb{MR}$ in the graph instance. MRs have attributes such as log comments, timestamp, and a unique ID. We assume that the timestamp of an MR is unique (derived from its earliest revision), and that an MR is an atomic operation. There exists an edge from each MR to the next MR in time (if one exists). One edge extends from the MR to the author of its revisions, and one edge is also created from the MR to each of its revisions (an MR is not connected to more than one revision of the same file).

Revisions correspond to the set of file revisions and are denoted by $\mathbb{R}evision$. Revisions are atomic in time with respect to other revisions, thus they have unique timestamps and they are assigned unique identifiers. They have attributes such as the *diff* of the change, and the lines added and removed. An edge extends from the revision to its author, and another one to the corresponding file. Revisions are also connected to each other. An edge is created from any given revision to each of its successor (the revision which modified it), thus one revision can have multiple children (or branches). Revision subgraphs are characterized as acyclic stream-like graph which springs up from a single node. If a revision merges a branch from another branch (or the main development trunk), an edge will be created from the "predecessor" revisions on both branch to the merged revision.

Files are represented as the subset of vertices $\mathbb{F}ile$ in the graph. Files are the springs from which streams of revisions flow. Files have attributes such as path, filename, directory, and a unique full path name. Time-wise, files have unique timestamps associated with the first revision made of a file (this records the moment the file first appears in the graph). Files are connected to by revisions as described above.

Table 1: Model Primitives

$isaMR(\phi)$	is ϕ an MR?
$isaRevision(\phi)$	is ϕ a Revision?
$isaFile(\phi)$	is ϕ a File?
$isaAuthor(\phi)$	is ϕ an Author?
$numberToStr(i)$	Represent i as a string.
$length(\phi)$	Length of the string ϕ
$substr\phi, k, l)$	Return a substring of ϕ of length l at k.
$eq(\phi, \theta)$	are ϕ and θ equivalent strings?
$match(\phi, \theta)$	is θ is a substring of ϕ?
$isEdge(\phi, \theta)$	is there an edge from ϕ to θ?
$count(\mathbb{S})$	counts the elements in a subset.
$isAuthorOf(\psi, \phi)$	is ψ an author of ϕ?
$isFileOf(\tau, \phi)$	is τ an File of ϕ?
$ifMROf(\phi, \phi)$	is ϕ is an mr of ϕ?
$isRevisionOf(\theta, \phi)$	is θ is a revision of ϕ?
$revBefore(\theta, \theta_2)$	is there is a revision path from θ to θ_2?
$revAfter(\theta, \theta_2)$	is there is a revision path from θ_2 to θ?

Authors are represented by the subset $\mathbb{A}uthor$ in the graph. Authors have attributes such as user ID, name and email. Time wise authors are associated to their first revision implying their entry into the project. There is only one author per MR and per Revision.

2.3 Formalizing the characteristic graph

Formally we define the characteristic graph G of a SCS as a directed graph of $G = (V, E)$ where

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

$$E = (v_1 \in \mathbb{MR}, v_2 \in \mathbb{MR}) \cup (v_1 \in \mathbb{MR}, v_2 \in \mathbb{R}evision) \cup (v_1 \in \mathbb{MR}, v_2 \in \mathbb{A}uthor) \cup (v_1 \in \mathbb{R}evision, v_2 \in \mathbb{R}evision) \cup (v_1 \in \mathbb{R}evision, v_2 \in \mathbb{A}uthor) \cup (v_1 \in \mathbb{R}evision, v_2 \in \mathbb{F}ile)$$

There are 6 data types in our model: Vertices representing entities; edges representing relationships; sets of entities which abstract edges; numbers used for numerical questions; strings are needed since much of the data in the repository is string data; and Booleans which are necessary to prove invariants exist. Table 1 provides a description of some of the primitives that operate on these types.

We implement attributes using maps. Attributes can map from entities to 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. More attributes can be added at any time but the attributed mentioned in section 2.2 are the expected attributes. Attributes which are expected to return one entity still return a subset. The motivation is to maintain uniform access to entities while providing a method of abstracting edge traversal. Since sets are returned we use plural function names. Attributes that are subsets of entities (edge traversals) are described in table 2.

2.4 Extraction and Creation

The general algorithm for extracting and creating a graph from a SCS is:

Table 2: Sub-domain Attributes

$authors(\phi \in \mathbb{MR})$	the author of the MR
$revisions(\phi \in \mathbb{MR})$	the revisions of the MR
$files(\phi \in \mathbb{MR})$	the files of the revisions of the MR
$nextMRs(\phi \in \mathbb{MR})$	next MR in time
$prevMRs(\phi \in \mathbb{MR})$	previous MR in time
$mrs(\theta \in \mathbb{R}evision)$	MR related of the Revision
$authors(\theta \in \mathbb{R}evision)$	the author of the revision.
$files(\theta \in \mathbb{R}evision)$	the files of a the revision
$nextRevs(\theta \in \mathbb{R}evision)$	Next revisions version-wise.
$prevRevs(\theta \in \mathbb{R}evision)$	Previous revisions version-wise.
$mrs(\tau \in \mathbb{F}ile)$	MRs of the Revisions of the file
$revisions(\tau \in \mathbb{F}ile)$	Revisions of the file
$authors(\tau \in \mathbb{F}ile)$	Authors of the revisions of the file.
$mrs(\psi \in \mathbb{A}uthor)$	MRs of the author.
$revisions(\psi \in \mathbb{A}uthor)$	Revisions of the author
$files(\psi \in \mathbb{A}uthor)$	Files of the revisions of the author

- Each file becomes a vertex in $\mathbb{F}ile$.
- Each author becomes a vertex in s $\mathbb{A}uthor$.
- 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 MR to its author MR.
- 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.

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

CVS does not record branch merges or modification requests, but some heuristics have been developed to recover both [2, 4, 11]. Branch-merge and MR recovery in CVS are not accurate, and therefore the extracted SCS graph is an interpretation rather than an exact representation of the SCS.

An example of the SCS graph is depicted in figures 2 and 3. The vertices corresponding to the revisions in 2 and 3 are the same and they are shown in two figures to avoid clutter.

3. QUERY LANGUAGE

The rationale for our model is to provide a basis for a query language for a SCS. We are interested in a language that has the following properties:

- It is based on primitives that correspond to the actual data and relationships stored in a SCS. We want a language that directly models files, authors, revisions, etc.

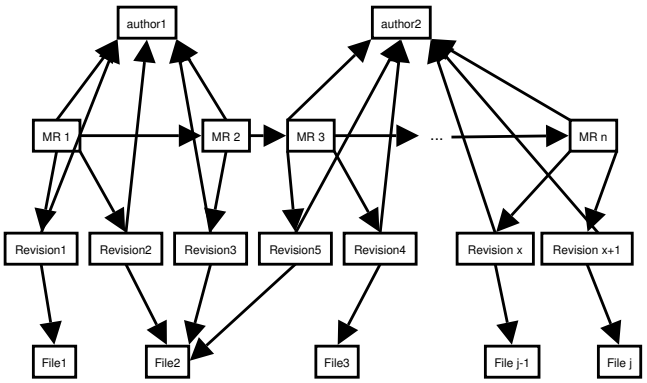

Figure 2: Example Model Subgraph

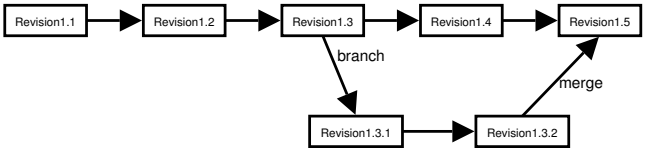

Figure 3: Example Revision Subgraph

- It has the ability to take advantage of the time dimension. We want to able to pose questions that include predicates such as "previous", "after, "last time", "always", "never". For example, "has this file *always* been modified by this author?", "find all MRs do not include the following file", "find the file revision *after* this other one", "find the *last* revisions by a given author", etc.
- It is computable. We need confidence that if a query is posed, it can be evaluated.
- It is expressive. We are interested in a language that is able to express a wide range of queries.

The characteristic graph of a source code repository is the basis for this language. Thus our language is built such that any query expressed in it can be translated to a query of the characteristic graph.

First order predicate logic will serve as a basis for our query language, as it can handle both graph semantics and "before and after" aspects of temporal logic [1]. The language is designed to query the model, not to provide a general purpose programming language. We have focused in evaluating decision queries with this language (those which answer is either yes or not), but we also support other types of queries that return other types of data (such as the id of an author, the number of files modified, or a set of files).

The language has a rich syntax, but due to a lack of space we only summarize its main features in table 3.

Identifiers are unbound variables that reference entities. Using a variable, one can access the attributes of the referenced

entities ($x.attribute$). Identifiers are only created by a scoping operator such as an Anchor, Universal Quantifiers, Existential Quantifiers or Selection Scope. These scopes iterate over elements in a subset by applying a predicate to each element.

Existential and Universal scopes iterate through an entire subset until a preposition returns either true or false. For empty subsets universal scopes return true and existential scopes return false.

Subset/Select based scopes effectively iterate through all the elements in set of entities such as $\mathbb{MR}$, $\mathbb{R}evision$, $\mathbb{F}ile$, $\mathbb{A}uthor$ selecting entities to form a subset. A subset can only be the same size or smaller than the set it is testing. These subsets may only have 1 type of entity. Anchor scopes are like select- based scopes, but are meant to access a single element in constant time. Scope operators that are "before" or "after" scopes iterate through their respective subsets in sequential order from first to last.

3.1 Example Queries

We now present three different queries and show how they are expressed in SCQL.

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

$$\exists a, b \in \mathbb{A}uthor \ s.t. \ a \neq b \wedge$$
$$\forall r \in Revision \ s.t. \ isAuthor(a, r) \implies$$
$$\exists r_b \in Revision \ s.t. \ before(r_b, r) \wedge$$
$$isAuthor(b, r_b) \wedge r.file = r_b.file$$

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 they are different. Then it iterates through all the revisions of author a. Per each revision, it checks if the file of that revision has another previous revision that belongs to author b. `a.revisions` gets 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 which have never appeared as part of another MR before. With this query we are 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 easily 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 too. Then it counts all MRs, and computes the desired proportion.

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

$$\exists a \in \mathbb{A}uthor \ s.t.$$
$$\forall f \in \mathbb{F}ile \ s.t. \ isAuthorOf(a, f) \implies$$
$$\forall f_2 \in \mathbb{F}iles.t.isAuthorOf(a, f_2) \implies$$
$$directory(f) = directory(f_2)$$

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, author.files) {
    A(f2, author.files) {
      eq(f.directory, f2.directory)
    }
  }
}
```

4. EVALUATION

We have built an implementation for 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 4 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.

Table 4: 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
$\lvert\mathbb{F}ile\rvert$	4748	3685	3698	4246	300
$\lvert\mathbb{MR}\rvert$	18573	11337	10847	27413	1398

Table 3: Language Description

Name	Language	Explanation
$\mathbb{MR}$	MR	Set of Modification Requests
$\mathbb{R}evision$	Revision	Set of Revisions
$\mathbb{A}uthor$	Author	Set of Authors
$\mathbb{F}ile$	File	Set of Files
Universal	$A(\phi, \delta)\{P(\phi)\}$	For all ϕ in the set δ is the predicate $P(\phi)$ true?
Existential	$E(\phi, \delta)\{P(\phi)\}$	Does ϕ exist in set δ where predicate $P(\phi)$ is true?
Attribute	$\phi.\zeta$	Given an entity ϕ return its attribute ζ
Function	$\gamma(P)$	Evaluate the function γ with P as the parameter
Universal Before	$Abefore(\phi, \delta, \theta)\{P(\phi, \theta)\}$	For all ϕ in δ before θ is the binary predicate $P(\phi, \theta)$ true?
Universal After	$Aafter(\phi, \delta, \theta)\{P(\phi, \theta)\}$	For all ϕ in δ after θ is $P(\phi, \theta)$ true?
Existential Before	$Ebefore(\phi, \delta, \theta)\{P(\phi, \theta)\}$	Does ϕ exist in δ before θ where $P(\phi, \theta)$ is true?
Existential After	$Eafter(\phi, \delta, \theta)\{P(\phi, \theta)\}$	Does ϕ exist in δ after θ where $P(\phi, \theta)$ is true?
Subset	$S(\phi, \delta)\{P(\phi)\}$	Create a subset of δ, such that for each element ϕ in that subset, $P(\phi)$ is true.
Universal From Subset	$A(\theta, S(\phi, \delta)\{P(\phi)\})\{Q(\theta)\}$	For each elements θ in the set δ for which $P(\phi)$ is true, $Q(\theta)$ is also true
Anchor Select	$Anchor(\phi, MR,'' mrid'')P(\phi)$	Evaluate $P(\phi)$ on the entity of type $\mathbb{MR}$ with id "mrid"
count	$count(\delta)$	Count the number of elements of the subsets δ
Sum	$Sum(\phi, \delta)\{P(\phi)\}$	Summate the predicate $P(\phi)$ for all ϕ in δ
Average	$Avg(\phi, \delta)\{P(\phi)\}$	Get the average of the predicate $P(\phi)$ for all ϕ in δ
Count	$Count(\phi, \delta)\{P(\phi)\}$	Count the number of elements ϕ in δ where $P(\phi)$ is true.

5. SUMMARY

This paper presents a formal model to describe SCSs. This model is then used to define a query language, SCQL, that can be used to pose queries on the SCSs. The objective of SCQL is to be domain specific and to support temporal logic operators in those queries. We have demonstrated the use of SCQL with example queries, and demonstrated their effectiveness by running those queries against the SCS of 5 different large, mature software projects.

While it is possible to use other query languages to investigate SCSs (such as SQL and XQuery) we believe that SCQL has 2 important properties that these languages are do not. First, it is domain specific: the queries refer to entities in the repository, and second, it supports temporal logic operators. While it is possible to implement temporal logic operations in SQL or XQuery, it might result in overly complex expressions.

We expect to use SCQL in the exploration of the evolution of software and to help us compute metrics on SCS repositories.

6. REFERENCES

[1] S. Abiteboul, L. Herr, and J. Van den Bussche. Temporal versus first-order logic to query temporal databases. pages 49–57, 1996.

[2] M. Fischer, M. Pinzger, and H. 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, Sept. 2003.

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

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

[5] D. M. German, A. Hindle, and N. 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.

[6] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, Sept 2004.

[7] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. pages 284–293, September 2004.

[8] Y. Liu and E. Stroulia. Reverse Engineering the Process of Small Novice Software Teams. In *Proc. 10th Working Conference on Reverse Engineering*, pages 102–112. IEEE Press, November 2003.

[9] A. Mockus, R. T. Fielding, and J. 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.

[10] X. Wu. Visualization of version control information. Master's thesis, University of Victoria, 2003.

[11] T. Zimmermann and P. Weisgerber. Preprocessing cvs data for fine-grained analysis. In *1st International Workshop on Mining Software Repositories*, May 2004.