Anwendungen der Softwarearchäologie Einführung und Grundlagen

Bastian Venthur

Freie Universität Berlin – Institut für Informatik

venthur@inf.fu-berlin.de

2005-06-11

Diese Ausarbeitung ist eine Einführung in die Softwarearcheologie (SA). SA kann prinzipiell aufgrund der rapiden Entwicklung der Freien/Open Source Software (F/OSS) in den letzen Jahren gut betrieben werden, doch leider fehlen die technischen Möglichkeiten und Werkzeuge um die Unmengen der Quellen effektiv auszuschöpfen.

Diese Ausarbeitung beschreibt im ersten Teil ganz Allgemein vier verschiedene Lösungsansätze für dieses Problem, und im Zweiten ganz konkret einen fünften Ansatz durch die Anfragesprache SCQL.

Inhaltsverzeichnis

1 Einführung 2

2 Einführung in SCS 3

2.1 Was gibt es? . . . 3

2.2 Wichtige Begriffe und Grundlegende Operationen . . . 3

3 Verschiedene Ansätze für MSR 4

3.1 MSR via SCS-Annotations . . . 4

3.2 MSR via Data Mining . . . 6

3.3 MSR via Heuristics . . . 6

3.4 MSR via Differencing . . . 6

4 Einführung in SCQL **8**
4.1 Motivation 8
4.2 Das formale Modell 8
4.3 Extraktion der Daten und Erstellung des Graphen 9
4.4 SCQL 11
4.5 Beispiele und Auswertung 12

5 Zusammenfassung **13**

1 Einführung

Die traditionellen Probleme der empirischen Forschung an Daten zu kommen wurden durch die rapide Entwicklung der *Freien/Open Source Software* (F/OSS) -Bewegung zum Teil ins Gegenteil verkehrt. Allein auf sourceforge – einem der populärsten Sammelpunkte für F/OSS-Projekte – werden zu diesem Zeitpunkt über 100.000 Projekte betreut.

Die repositories sind frei zugänglich und bieten jedem Interessierten die Möglichkeit sie zu durchstöbern. Zusätzlich zu den repositories gehören zu den meisten Projekten eine Mailingliste (oder Forum) und ein bugtracking-System.

Insgesamt existiert also eine riesige Menge Material, die für Analysen und Auswertungen zur Verfügung steht. Eine Aufgabe, die sich *zu Fuß* kaum bewältigen ließe, ohne dabei deutliche Einschränkungen der Quellen in Kauf nehmen zu müssen. Gesucht wird also eine Möglichkeit diese Analysen und Auswertungen automatisch zu erledigen.

Ein wichtiger Aspekt ist das Problem, das man später in der Lage sein möchte die Daten auf gewisse Fragen hin zu untersuchen. Einige (und längst nicht alle) Beispiele wären:

- Welche Klassen/Methoden/Pakete bilden ein funktionales Konzept?
- Welche Art von Bugs/Defekten tritt am häufigsten auf?
- Gibt es Metriken, mit denen man die Wahrscheinlichkeit von Defekten in bestimmten Code-Teilen vorhersagen kann?
- Welcher Autor hat welches Spezialwissen?
- In welcher Phase steckt das Projekt?

Solche Fragen könnten ähnlich wie Abfragen an eine Datenbank aussehen, doch diese

Möglichkeit bieten Source Control Systeme[1] (SCS) sogut wie garnicht.

Auf sourceforge wird zur Revisionskontrolle CVS eingesetzt. Dies bietet grundsätzlich eine gute Grundlage für die automatische Auswertung der Entwicklung des Projektes, doch muss bedacht werden, dass CVS für derartige Analysen nicht geschaffen wurde. CVS gibt zwar bereitwillig Auskunft über die Veränderungen zwischen jeder Revision, doch einen gesamten Überblick über die Entwicklung des Projektes kann man nur aus der Summe aller Revisionen erhalten.

2 Einführung in SCS

2.1 Was gibt es?

Als erstes sei darauf hingewiesen, dass es für Versionskontrollsysteme verschiedene Begriffe und Abkürzungen gebräuchlich sind, die allesamt synonym zu betrachten sind: source control system (SCS), revision control system (RCS), version control system (VCS) und sicher noch einige Andere. In diesem Text wird source control system (SCS) verwendet.

Wir unterscheiden prinzipiell zwei Arten von SCS: die zentraliesierten- und die dezentralisierten-SCS. Bekannte Vertreter beider Typen sind CVS und SVN für die zentalisierten- sowie Bitkeeper und GNU-Arch für die dezentralen SCS. Obwohl sich beide Systeme stark in der Art der Operationen und Speicherung der Daten unterscheiden, beobachten beide Systeme Dateien und ihre Revisionen [4].

Einen sehr ausführlichen Vergleich in verschiedenen Kategorien der bekanntesten SCS bietet [7].

2.2 Wichtige Begriffe und Grundlegende Operationen

Das zentrale Archiv wird als *Repository* (engl. Behälter, Aufbewahrungsort) bezeichnet. Die meisten Systeme verwenden hierfür ein eigenes Dateiformat (oder eine Datenbank). Die Versionsverwaltungssoftware speichert dabei üblicherweise nur die Unterschiede zwischen zwei Versionen um Speicherplatz zu sparen. Dadurch kann eine große Zahl von Versionen archiviert werden.

Damit die in der Softwareentwicklung eingesetzten Programme wie z.B. Compiler mit den im Repository abgelegten Dateien arbeiten können ist es erforderlich, dass sich jeder

[1] Eine genaue Erklärung was SCS sind, folgt im nächsten Abschnitt

Entwickler den aktuellen (oder einen älteren Stand) des Projektes in Form eines Verzeichnisbaumes aus herkömmlichen Dateien erzeugen kann. Ein solcher Verzeichnisbaum wird als *Arbeitskopie* bezeichnet.

Ein wichtiger Teil des Versionsverwaltungssystems ist ein Programm, das in der Lage ist, diese Arbeitskopie mit den Daten des Repositories zu synchronisieren. Das Übertragen einer Version aus dem Repository in die Arbeitskopie wird als *Checkout* oder *Aktualisieren* bezeichnet, während die umgekehrte Übertragung als *Checkin* oder *Commit* genannt wird [6]. Bei einem Checkin hat man meist die Möglichkeit die eigenen Änderungen in Form eines Textes zu beschreiben. Auf diese Weise entsteht bei kontinuierlicher Anwendung ein Log, der es erlaubt – auch ohne den Code direkt zu betrachten – die Entwicklung einer Datei über die Versionen hinweg zu verstehen.

Über diese Funktionen hinaus bieten die meisten SCS noch eine Möglichkeit verschiedene Revisionen einer Datei zu vergleichen. Traditionell wird dieser Vergleich als *diff* [2] ausgegeben.

3 Verschiedene Ansätze für MSR

Im folgenden untersuchen wir die verschiedenen Ansätze und die möglichen Fragen, die durch den jeweiligen Ansatz beantwortet werden können. Von den vielen Quellen die uns offen stehen, beschränken wir uns (vorerst) auf die SCS und führen den Begriff *Mining of Software Repositories* (MSR) ein [1].

3.1 MSR via SCS-Annotations

Eine Annotation bezeichnet die komplette, zeilenweise Ausgabe einer Datei. Zusätzlich zu ihrem Inhalt erhält man zu jeder Zeile die Revisionsnummer, die angibt wann diese Zeile das letze Mal geändert wurde, den Autor der für diese Änderung verantwortlich ist und das Datum der Änderung.

Man beachte, dass man die Änderung an sich nicht sieht, diese kann über Annotations nur herausgefunden werden, wenn man sich die Annotation der entsprechenden Revision $r - 1$ ansieht und mit der Annotation von Revision r vergleicht.

Ein gekürztes Beispiel einer Annotation könnte etwa so aussehen:

[2] diff ist ein Unix-Werkzeug, dass die Differenz zweier Dateien in einer standardisierten Form ausgibt. Diese Ausgabe wird als Diff oder oft auch als Patch bezeichnet und ist die komplette Information, die man benötigt um Datei A auf Datei B zu ändern und umgekehrt.

```
bventhur@phoibe:~/robocup/venthur/observer/src/decoder$ svn blame decoder.cpp
  ...
  2283    venthur
  2283    venthur     Frame* frame = new Frame();
  2283    venthur     size_t offset=0;
  2283    venthur     float posx,posy;
  2504    venthur     int timestamp;
  2504    venthur     memcpy(&timestamp,buffer+offset,sizeof(int));
  2504    venthur     offset += sizeof(int);
  2283    venthur     memcpy(&posx,buffer+offset,sizeof(float));
  2283    venthur     offset += sizeof(float);
  2283    venthur     memcpy(&posy,buffer+offset,sizeof(float));
  2283    venthur     offset += sizeof(float);
  2283    venthur
  2283    venthur     frame->addBall( Ball(posx, posy) );
  2370    venthur     frame->setTimeStamp(timestamp);
  ...
```

MSR mit Annotations könnte folgendermaßen Praktiziert werden: Man untersucht alle Revisionen, die in einer Klasse sichbar sind (im Beispiel die Revisionen: 2283, 2370, 2504). Wenn man dies bei allen Klassen eines Projektes macht und die Revisionen in den jeweiligen Klassen vergleicht, so erhält man die Information, welche Klassen in der gleichen Revision geändert wurden. Wenn man dieses System verfeinert und die Zeitstempel einbezieht, kann man diese Definition etwas aufweichen und alle Revisionen die innerhalb eines bestimmten Zeitfensters (z.B. 5min) als *eine* Revision betrachten.

Auch lassen sich beziehungen zwischen Klassen und Autoren herstellen, je nachdem wer am häufigsten welche Dateien verändert hat.

Zusammengefasst können folgende Fragen von dieser Art des MSR beantwortet werden:

- Welche Klassen ändern sich zusammen?
- Wie oft wurde eine bestimmte Klasse geändert?
- Wieviele Änderungen traten in einem Subsystem (z.B. Verzeichnis) auf?
- Wieviele Änderungen traten zwischen Subsystemen auf?
- Welcher Zusammenhang besteht zwischen einem Autor und einer Klasse?

3.2 MSR via Data Mining

Data Mining bietet eine Vielzahl an Möglichkeiten die vorhandenen Daten auszuwerten. Bevor dies geschehen kann müssen die Rohdaten zuerst eine für das Werkzeug brauchbare Form gebracht werden. Anschließend können DM-Algorithmen *Assoziationsregeln* erstellen, also Regeln wie folgende: Wenn Funktion A und B geändert wurden, wird auch C geändert usw. Die Granularität kann von der Klassen- bis hinunter zur Variablenebene gehen.

Die MSR durch Datamining wurde auf Datei-, Funktions- und Variablenebene getestet, doch die beste erreichte Genauigkeit[3] war im besten Fall 26% auf der Dateiebene.

3.3 MSR via Heuristics

Grundidee sind bestimmte Heuristiken (Faustregeln basierend auf Erfahrung). Eine SCS-Annotation-Analyse kann mit Heuristiken ausgeweitet werden indem man SCS-Annotations lexigraphisch analysiert um die geänderten Entitäten der im Repostitory abzuleiten.

Welche Heuristiken zum Einsatz kamen wurde nicht erwähnt, aber die Ergebnisse sind ähnlich wie beim MSR durch Data Mining: 12% Genauigkeit auf Dateiebene bei 87% recall[4] und 2% Genauigkeit und 42% recall bei call/use/define-Heuristiken.

3.4 MSR via Differencing

Bei dieser Methode werden die *diffs/patches* der Verschiedenen Versionen im Repository analysiert.

```
bventhur@phoibe:~/robocup/observer/src/decoder$ svn diff decoder.cpp -r 2370
Index: decoder.cpp
===================================================================
--- decoder.cpp (Revision 2370)
+++ decoder.cpp (Arbeitskopie)
@@ -70,9 +70,9 @@
     Frame* frame = new Frame();
```

[3] Precision/Genauigkeit: beschreibt die Genauigkeit eines Suchergebnisses. Sie ist definiert als der Anteil relevanter Dokumente von allen bei einer Suche gefundenen Dokumenten. [8].

[4] Recall: Der Recall beschreibt die Vollständigkeit eines Suchergebnisses. Er ist definiert als der Anteil bei einer Suche gefundenen relevanten Dokumente (bzw. Datensätzen) an den relevanten Dokumenten der Grundgesamtheit.

```
     size_t offset=0;
     float posx,posy;
-    float timestamp;
-    memcpy(&timestamp,buffer+offset,sizeof(float));
-    offset += sizeof(float);
+    int timestamp;
+    memcpy(&timestamp,buffer+offset,sizeof(int));
+    offset += sizeof(int);
     memcpy(&posx,buffer+offset,sizeof(float));
     offset += sizeof(float);
     memcpy(&posy,buffer+offset,sizeof(float));
```

Jede Version wird zu einem *abstract syntax graph* (ASG)[5] umgewandelt. An jedem Paar von ASG wird ein spezieller differencing-Algorithmus angewendet, welcher als Ausgabe eine Beschreibung liefert welche Knoten (Funktionsaufrufe, Variablen, usw.) im Graph enfernt wurde, neu hinzu kamen, geändert oder verschoben wurden.

Diese Ausgaben können nun auf verschiedene Fragen hin analysiert werden.

Möchte man spezielle Fragen beantworten, wie zum Beispiel welche Art von Bug-Fixes am häufigsten auftregen, müssen anhand von SCS-Kommentaren die Versionen herausgesucht werden, die Bugfixes enthalten/repräsentieren.

Für diese Aufgabe wurde ein spezielles Tool *Dex* entwickelt, was SCS-Versionen auf 398 Fragen hin beantworten kann. Die Statistiken sind sind sehr beeindruckend: von den 398 Statistiken waren 378 immer korrekt, d.h. Dex hat eine Fehlerquote von lediglich 1,1% pro patch.

Typische Fragen an Dex wären:

- Wieviele Funktionen oder Funktionsaufrufe wurden eingefügt?
- Wieviele if-Statements wurden verändert?
- Und viele andere gleichartige Fragen…

[5] ASG: Allgemeinere Form eines Syntaxbaumes

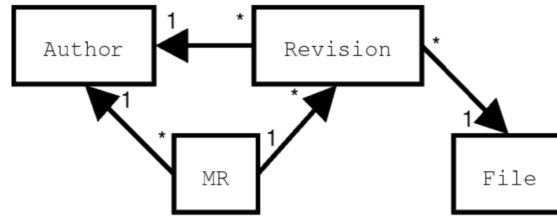

Abbildung 1: Kardinalitäten und Richtungen der Knoten im Model

4 Einführung in SCQL

4.1 Motivation

Ein SCS verfolgt die Änderungshistorie eines Softwareprojektes. Es speichert einen Eintrag darüber, wer welchen Teil es Systems wann geändert hat und was diese Änderung genau war. Ein Werkzeug was diese historischen Informtionen benutzen möchte muss irgendeine Form der Extrahierung von Fakten betreiben. Diese Fakten werden verarbeitet um neuen Informationen zu erlangen oder Vorraussagen für die Zukunft zu treffen.

Einige Werkzeuge benutzen Relationale Datenbanken und benutzen SQL-Abfragen um die Daten analysieren; Andere bevorzugen Textdateien für die Speicherung der Daten und erstellen kleine Programme um speziefische Fragen zu beantworten. Jede dieser Vorgehensweise hat das Problem, dass es auf diese Art schwer ist *historische* Zusammenhänge zu erfragen bzw. Anfragen müssen mühsam in das Modell oder Schema der gespeicherten Daten übersetzt werden.

Des weiteren gibt es keine standardisierte Form zu Speicherung und Abfrage dieser Daten. Dieses Problem versucht SCQL zu beheben [4].

4.2 Das formale Modell

Um eine Sprache zu erstellen die Anfragen an ein SCS stellen kann benötigt man zuerst ein Modell welches die Daten beschreibt. Das Modell soll uniform für die verschiedenen SCS sein und es soll die Darstellung der zeitlichen Zusammenhänge wie *vorher* oder *danach* ermöglichen.

Bei SCQL wurde als Modell der gerichtete Graph gewählt. Knoten dieses Graphen sind Objekte und die Kanten sind die Beziehungen zwischen den Objekten. Abbidlung 1 zeigt die grundsätzliche Verknüpfung der einzelnen Elemente des Graphen.

Hier werden vier verschiedene Objekttypen unterschieden:

MRs (model modification requests) haben Attribute wie: Kommentare, Zeitstempel und eine eindeutige ID. Es wird vorraussgesetzt, das IDs eineindeutig- und MRs atomare Operationen sind. Von jedem MR geht eine Kante zum (zeitlich) nächsten MR aus. Ausserdem geht eine Kante zu einem Autor und mindestens einer Revision ab. Ein MR hat mehrere Revisionen, wenn verschiedene Dateien betroffen sind und diese jeweils unterschiedliche Revisionen haben; ein MR kann nicht zu mehreren Revisionen einer Datei gehören.

Revisions Revisionen haben einzigartige Zeitstempel und IDs. Sie enthalten Attribute wie *diffs* und die Veränderten Zeilen. Von Revisionen geht eine Kante zum Author und eine zur Datei aus. Ausserdem geht von jeder Revision eine Kante zu allen direkten nachfolgenden Revisionen aus. Eine Revision kann also mehrere Kinder (branches) haben. Wenn eine Revion zwei branches *merged* zeigen beide Revisionen auf die entstandene Revision.

Files Dateien haben Attribute wie: Pfad, Dateiname, Verzeichnis und einen eindeutigen vollen Pfadnamen. Zeitlich gesehen haben Dateien einen eindeutigen Zeitstempel. Dateien sind durch Revisionen verbunden.

Authors Autoren haben Attribute wie user ID, name und email. Zeitlich gesehen sind Autoren mit ihrer ersten Revision assoziiert. Es gibt nur einen Autor pro MR und Revision.

Formal ist das Modell also als gerichteter Graph G eines SCS mit $G = (V,E)$ definiert. Wobei die Kanten und Knoten folgendermaßen definiert sind:

$$V = MR \cup File \cup Author \cup Revision \quad (1)$$
$$E = (v_1 \in MR, v_2 \in MR) \cup (v_1 \in MR, v_2 \in Revision) \quad (2)$$
$$\cup (v_1 \in MR, v_2 \in Author) \cup (v_1 \in Revision, v_2 \in Revision) \quad (3)$$
$$\cup (v_1 \in Revision, v_2 \in Author) \cup (v_1 \in Revision, v_2 \in File) \quad (4)$$

4.3 Extraktion der Daten und Erstellung des Graphen

Um ein SCS in einen Graphen zu übersetzen wird ein Algorithmus benötigt. Dieser sieht folgendermaßen aus:

1. Jede Datei wird ein Knoten der Menge File

2. Jeder Autor wird ein Knoten der Menge Author

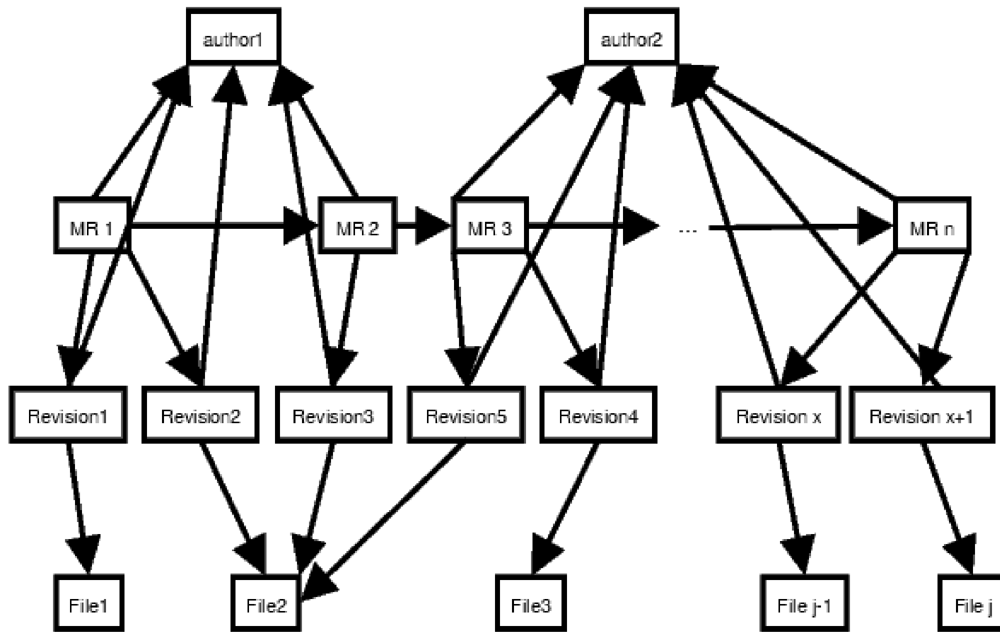

Abbildung 2: Model Untergraph

3. Jede Revision wird ein Knoten der Menge Revision. Ordne Revisionen eineindeutige Zeitstempel zu und verbinde jede Revision zu ihrer entsprechenden Datei und ihrem Autor

4. Erstelle Knoten für jedes MR. Das MR erbt den Zeitstempel seiner seiner ersten Dateirevision. Verknüpfe MR zum entsprechenden Autor (Vermutlich Fehler im Paper: Man muss noch die MRs mit den Revisionen verbinden)

5. Verbinde alle MRs zu ihren jeweils nächsten MRs (nacht Zeitstempel), sofern sie existieren.

6. Für jede Datei: Verbinde jede Revision dieser Datei zur jeweils nächsten dieser Datei. Wenn gebranched wurde, werden nur Revsionen einer Brach auf diese Weise verbunden. Branch- und Mergepoints werden verbunden.

Wenn dieser Algorithmus terminiert hat, liefert er den Charakteristischen Graphen dieser SCS. Beispielgraphen sieht man in den Abbildungen 2 und 3. Diese Beiden Graphen sind Untergraphfen des gleichen Graphen und wurden getrennt angegeben um dem Leser das Verständniss zu erleichtern.

Eine Anmerkung zum branching: Da CVS keine merges als solche unterstützt und auch keine MRs kennt müssen diese mit entsprechenden Heuristiken erkannt werden. Dafür existieren Algorithmen, die aber nicht sehr genau sind, von daher muss der charakteris-

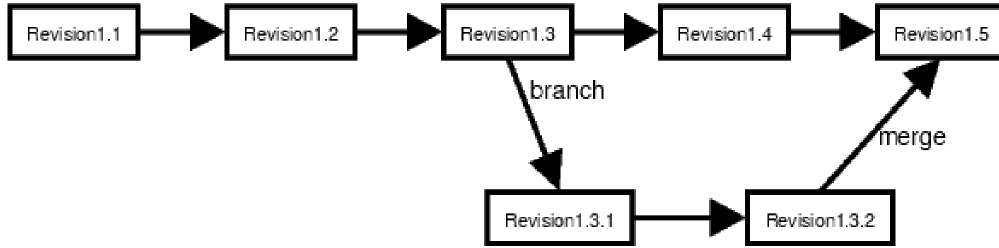

Abbildung 3: Revisions Untergraph

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.

Abbildung 4: Sprachbeschreibung von SCQL

tische Graph eher als eine Interpretaion eines SCS gesehen werden anstelle einer exakten Repräsentation. Die Tabelle in Abbildung 4 gibt eine Übersicht über die wichtigsten Sprachelemente von SCQL.

4.4 SCQL

Nachdem die Basis für die Sprache mit dem charakteristischen Graphen gelegt wurde, wenden wir uns der Sprache selbst zu. Für SCQL wurde die Prädikatenlogik erster Stufe[6] als Grundlage genutzt. Diese Sprache ist also keine allgemeine Programmiersprache sondern eine Anfragensprache an das zugrunde liegende Modell.

[6]Prädikatenlogik erster Stufe: Erweiterung der Aussagenlogik um All- und Existenzquantor

4.5 Beispiele und Auswertung

Im folgenden Betrachten wir einige Anfragen, die man mittels SCQL an das SCS stellen kann:

Beispiel 1 Gibt es einen Author a welcher nur Dateien ändert die von Author b verändert wurden?

Diese Anfrage würde in SCQL so aussehen:

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

Beispiel 2 Berechne das Verhältniss der MRs welche eine eindeutige Menge von Dateien haben, die niemals zuvor als Teil eines anderen MR aufgetreten sind:

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

Wir nehmen an, das dieses Verhältniss bei alten, stabilen Projekten kleiner ist, als bei Projekten die noch wachsen und ständigen strukturellen Änderungen unterworfen sind.

Beispiel 3 Gibt es einen Autor, dessen Änderungen nur in einem Verzeichniss stattfinden?

	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

Abbildung 5: Auswertung der drei Beispielanfragen

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

Diese drei Beispiele wurden auf fünf verschiedene Projekte angewendet: Evolution (Emailclient), Gnumeric (Spreadsheet), OpenSSL, Samba und modperl. Die Ergebnisse sind in der Tabelle auf Abbildung 5 zusammengefasst, eignen sich aber kaum für eine Analyse, da die Fragen nur zur beispielhaften Erläuterung der Sprache dienen sollten und nicht zu einer echten Analyse von Projekten.

5 Zusammenfassung

In der Ausarbeitung wurden fünf Möglichkeiten vorgestellt, wie Software Archäologie mit Hilfe von Analyse eines SCS betrieben werden kann. Eine wichtige Aussage dieser Ausarbeitung ist, dass verschiedene Fragen verschiedene Lösungsansätze bevorzugen und es keinen Königsweg für alle Fragen gibt.

Ein weiterer wichtiger Punkt ist, dass bereits mit der Einschränkung auf die Analyse von SCS, andere wichtige Informationsquellen ausgeschlossen wurden, denn zu fast jedem Projekt gibt es Mailinglisten und ein Bugtrackingsystem. Diese Quellen sind in der Regel nicht so leicht automatisch auszuwerten, da ihre Informaionen in gesprochener Sprache vorliegen. Trotzdem kann davon ausgegangen werden, das gerade die Mailinglisten im Zusammenhang mit dem Bugtrackingsystem eine wesentlich ergiebigere Quelle sein dürfte, wenn es um die Analyse von Fehlern im Code geht.

Literatur

[1] H. Kagdi, M. L. Collard, J. I. Maletic, Towards a Taxonomy of Approaches for Mining of Source Code Repositories

[2] D. M. German, D. Cubranic, M. D. Storey, A Framework for Describing an Understanding Mining Tools in Software Development

[3] L. Gasser, G. Ripoche, R. J. Sandusky, Research Infrastructure for Empirical Science of F/OSS

[4] A. Hindle, D. M. German, SCQL: A formal model and a query language for souce control repositories

[5] L. Pekacki, Entwurf und Implementierung einer Wergzeugumgebung für die Analyse von Softwareprojektdaten

[6] http://de.wikipedia.org/wiki/Versionsverwaltung, 2005-06-11

[7] http://better-scm.berlios.de/comparison/comparison.html, 2005-06-11

[8] K. Zacharov, B. Venthur, Beschreibung des Inputs und Formen des Outputs