Enhancing History-Based Concern Mining With Fine-Grained Change Analysis

Masatomo Hashimoto and Akira Mori
AIST Tokyo Waterfront
2-3-26 Aomi, Koto-ku
Tokyo 135-0064, Japan
e-mail: {m.hashimoto,a-mori}@aist.go.jp

***Abstract*—Maintenance of large software projects is often hindered by cross-cutting concerns scattered over multiple modules. History-based mining techniques have been proposed to mitigate the difficultly by examining changes related to methods/functions in development history to suggest potential concerns. However, the techniques do not cope well with renamed entities and may lead to irrelevant information about concerns. The intricate procedures of the methods also make the results difficult for others to reproduce, utilize or improve.**

In this paper, we reinforce history-based concern mining techniques with fine-grained change analysis based on tree differencing on abstract syntax trees. Source code changes are recorded as facts over source code regions according to the RDF (Resource Description Framework) data model so that the analysis can be performed in terms of factbase queries.

To show the capability of the method, we report on an experiment that emulates the state-of-the-art concern mining technique called COMMIT using our own change analysis tool called Diff/TS. A comparative case study on several open source projects written in C and Java shows that our technique improves results and overcomes the language barrier in the analysis.

I. Introduction

Maintenance of large software projects is often hindered by cross-cutting concerns scattered over multiple modules such as caching, logging, and authentication. History-based mining techniques have been proposed to mitigate the difficultly by examining changes related to methods/functions in development history to suggest potential concerns. However, the techniques do not carry out detailed change analysis to track renamed source code entities across versions and may report irrelevant methods/functions as part of candidate concerns.

On the other hand, the intricate procedures of history-based methods make the results difficult for others to reproduce, utilize or improve. One has to collect change histories from source code management systems such as CVS and SVN and process source files to extract changes related to methods/ functions before applying his/her own technique for concern mining. It is a demanding task considering the volume of changes involved in a large project and the precision needed for analyzing the source code. The difficulty lead to limited use and segregation of methods since preparing tools needed for target projects and programming languages takes too much effort from developers and maintainers. In this regards, we believe that generation and analysis of facts must be separated in a way that third-party users can access the database of facts (called the *factbase*) with reasonable costs of initial learning.

In this paper, we propose a method for recording source code changes as facts over textual regions according to the RDF (Resource Description Framework) data model to allow analysis to be performed in terms of factbase queries. We apply the method to history-based concern mining by reinforcing the techniques with a fine-grained change analysis based on tree differencing on abstract syntax trees. To show the capability of the method, we report on an experiment that emulates the state-of-the-art concern mining technique called COMMIT [1] using our own change analysis tool called Diff/TS [2]. A comparative case study on several open source projects written in C and Java shows that our technique improves the results and overcomes the language barrier in the analysis.

To summarize, our contribution is twofold:

- allowing history-based analysis to be performed at the level of factbase queries, independent of particular programming languages and models of dependencies, and
- improving history-based concern mining techniques by integrating a fine-grained change analysis into the factbase.

The rest of the paper is organized as follows. Section II explains the method of fined-grained change analysis and construction of factbase before overviewing the mining technique of COMMIT in Section III. The emulation of COMMIT in our setting including the enhancement in change analysis is presented in Section IV and the result of an experiment is reported in Section V. After related work is reviewed in Section VI, we conclude in Section VII with future plans.

II. Fine-Grained Change Analysis and Factbase

Our aim is to facilitate collaborative efforts in software evolution analysis in a way that is independent of specific tools, platforms, protocols and languages. This lead us to an idea of representing facts about source code as properties/relations on textual regions in source files. This is probably the most universal and independent way of sharing and exchanging information about source code since no tools can not point the code regions of their interests and users normally comprehend source code by browsing its texts.

```
   0    5    10   15   20   25
1  class F {
2    static int compute(int n){
3      int r;
4      if(n == 0)
5        r = 1;
6      else
7        r = n * compute(n - 1);
8      return r;
9    }
10 }
```

Fig. 1. Program P_0

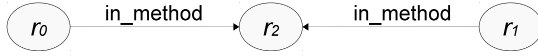

Fig. 2. A Fact Graph

A. Facts About Source Code

we define several basic notions to materialize the idea. A *source code entity*, or simply *entity*, is specified by a *code region* denoted by $\langle \mathit{fid}, p_s, p_e \rangle$, where fid denotes a file identifier and p_s and p_e denote a start position and an end position in the file, respectively. We employ throughout the rest of the paper hash values to identify files and pairs of line and column numbers to indicate positions in the file. For example, a boxed method invocation in a program P_0 shown in Fig. 1 is specified by

$$\langle \mathrm{cac5bc1f87b89c835aebbae4b7c6e2d7}, (7, 14), (7, 27) \rangle,$$

where MD5 (Message-Digest algorithm 5)[1] is used to compute the hash of the source file containing P_0. Note that we can use any hash algorithm as long as it produces no hash collision among source files to be analyzed and that the same algorithm is used in the whole analysis.

We define *facts* based on the RDF data model. A fact is denoted by a triple (s, p, o), where s, p, and o denote a *subject*, a *predicate* (or a *property*), and an *object*, respectively. We may specify the tool that provides a subject, a predicate, an object, or the whole fact by a subscript symbol to them.

Example 1: Suppose that a tool `t` provides a fact stating that the `if`-statement in Fig. 1 is in the method `compute`. The fact is represented by

$$(\langle \mathit{fid}, (4, 4), (7, 28) \rangle, \mathsf{in_method}, \langle \mathit{fid}, (2, 2), (9, 2) \rangle)_t,$$

where $\mathit{fid} = \mathrm{cac5bc1f87b89c835aebbae4b7c6e2d7}$.
According to the RDF data model, a set of facts forms a directed graph, where each triple is represented by a (sub-) graph $s \xrightarrow{p} o$. Fig. 2 depicts an example of a fact graph, where r_0, r_1, and r_2 denote code regions.

If we integrate facts generated by different tools, we must consider alignment of code regions in order to increase opportunities to combine facts. This is necessary because an entity is not always uniquely specified by a single code region. It is possible that different tools refer to the same code entity by

[1] http://www.rfc-editor.org/rfc/rfc1321.txt

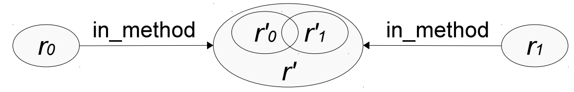

Fig. 3. Alignment of Regions

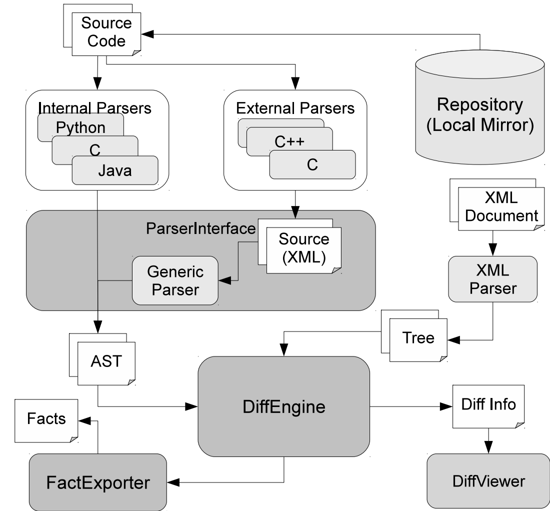

Fig. 4. Diff/TS System

presenting different code regions. In other words, identification of code regions and/or models of code structures vary from tool to tool.

Suppose that we integrate $f_1 = (r_0, \mathsf{in_method}, r'_0)_{t_0}$ and $f_2 = (r_1, \mathsf{in_method}, r'_1)_{t_1}$, where $t_0 \neq t_1$. If r'_0 and r'_1 are exactly the same code region, say r_2, then f_1 and f_2 form a combined graph, which means that r_0 and r_1 are in the same method as shown in Fig. 2. However, we cannot combine f_1 and f_2 when r'_0 and r'_1 are disjoint since it is not adequate at all to regard r'_0 and r'_1 as representing the same source code entity in this case. Our region alignment tries to find an *approximated* region r' containing both r'_0 and r'_1 when they intersect as shown in Fig. 3. Since this paper only concerns a single tool Diff/TS, there is no need for aligning code regions and the technical details are omitted. They will be published elsewhere.

B. Diff/TS: A Fine-Grained Change Analysis Tool

Diff/TS is a fine-grained differencing system for tree structures, specifically tailored for abstract syntax trees (ASTs) of source code. Diff/TS consists of four components: ParserInterface, DiffEngine, FactExporter, and DiffViewer. An overview of Diff/TS is depicted in Fig. 4. Explanations of each component will follow.

ParserInterface mediates parsers and DiffEngine. Since we accept multiple programming languages, multiple parsers are present in Diff/TS. There are two types of parsers: internal and external. We have built our own internal parsers for Python and Java and have adapted the C parser of Coccinelle[2] program matching and transformation engine. All internal parsers are

[2] http://coccinelle.lip6.fr/

able to cooperate with DiffEngine without having to serialize AST data. Internal representations are passed to DiffEngine directly. External parsers are required to produce ASTs in an XML format. Currently, Diff/TS works together with a C/C++ parser built on libraries developed in CDT (Eclipse C/C++ Development Tooling) project[3] as well as a C parser developed in CIL (Infrastructure for C Program Analysis and Transformation) project[4]. Diff/TS uses XML parsers internally for processing ASTs provided by external parsers and native XML documents.

DiffEngine lies at the core of the system and receives a pair of labeled ordered trees and calculates an *edit sequence* of *edit operations*, namely, delete, insert, rename, and move that transforms one tree into another. We illustrate an example of differencing by a small Java code shown in Fig. 5, due to Kawrykow and Robillard [3]. In Fig. 6, a set of tree edit operations between (ASTs of) versions N and $N+1$ is shown by shaded ellipses, as computed by Diff/TS. Note that only changes between shaded areas in Fig. 5 are depicted. The nodes that are unchanged or renamed are marked *mapped* and the node groups that are moved, inserted, or deleted are marked accordingly. It is notable that Diff/TS is able to detect a move of the method invocation `size()` accompanied by a rename of `l` to `list`. Diff/TS achieves the detection of move by elaborated post-processing of edit sequences [2]. Precise tree differencing is a major tool in our approach.

```
// Version N                    // Version N+1
Object field = ...              Object m_field = …

  void sample() {                 void sample() {
1.  List l = ...                  java.util.List list = ...
2.  l.add(this.field);            list.add(m_field);
3.  m(l.size());                  int size = list.size();
4.  return;                       m(size);
  }                             }
```

Fig. 5. A Small Java Code (due to Kawrykow and Robillard [3])

FactExporter exports facts about pieces of source code and changes between them. *Code facts* of a piece of source code are extracted by a parser for the language in which the source code is written. Examples of code facts will be given in Section IV. *Change facts* are derived from low-level edit sequences computed by DiffEngine. We give several basic change facts with short descriptions. Let R be the set of code regions.

$(s, \mathsf{added_to}, o)$ $(s \in R, o \in R)$
  $\Rightarrow$ s is added to a code region to which o is mapped.
$(s, \mathsf{removed_from}, o)$ $(s \in R, o \in R)$
  $\Rightarrow$ s is removed from a code region that is mapped to o.
$(s, \mathsf{renamed}, o)$ $(s \in R, o \in R)$
  $\Rightarrow$ s is renamed to be o.
$(s, \mathsf{moved_to}, o)$ $(s \in R, o \in R)$
  $\Rightarrow$ s is moved to o.

[3]http://www.eclipse.org/cdt/
[4]http://cil.sourceforge.net/

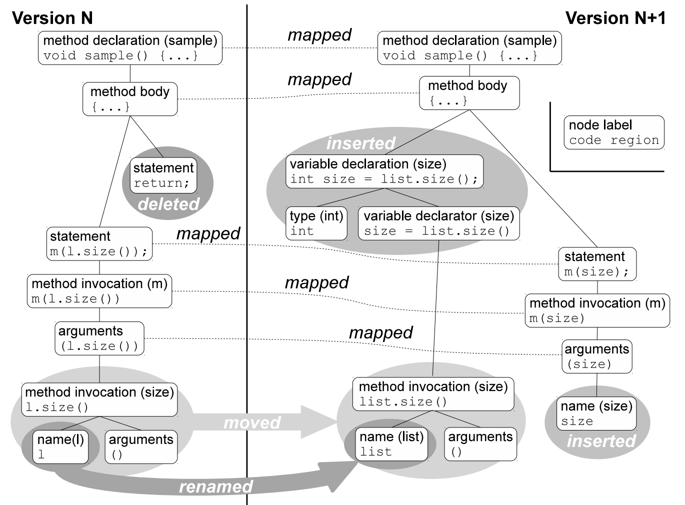

Fig. 6. Edit Operations Generated by Diff/TS on ASTs (Partial)

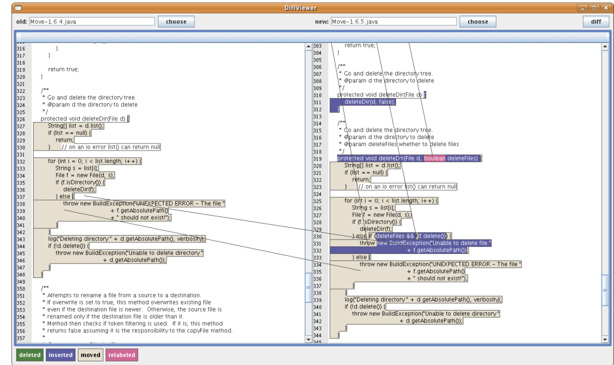

Fig. 7. DiffViewer

$(s, \mathsf{modified}, o)$ $(s \in R, o \in R)$
  $\Rightarrow$ s is modified and corresponds to o.

Note that the object in $(s, \mathsf{added_to}, o)$ mean mapped code regions. The reason why we describe mapped code regions for the fact is that we can express two pieces of information in a single fact: s is added to a code region and o is mapped to the region. Another kind of fact $(s, \mathsf{removed_from}, o)$ is interpreted similarly.

Edit sequences are visualized by DiffViewer. DiffViewer understands DiffEngine's output format which describes relations between ASTs and source code to overlay edit operations on source code texts. Change related texts are colored to highlight syntactic layout. For operations rename and move, "sticky" lines are drawn to keep connections traceable between original and modified parts. Fig. 7 has a screen image of DiffViewer.

III. History-Based Concern Mining

In certain kind of software systems, common functionality that spans different modules typically supports operations such as caching, exception management, logging, and tracing. Such functionality is generally described as *cross-cutting concerns*. Developers need to know the locations of these concerns. For example, if the code that writes to the logs is scattered

throughout a number of modules, and the requirements related to these concerns change, they may have to update the relevant code throughout the entire system. Concern mining techniques support the identification of concerns in software systems. Static techniques analyze source code, dynamic techniques analyze execution traces, and history-based techniques analyze changes in the source code repository. The techniques generate concern *seeds*, which are sets of program entities that possibly contribute to the implementation of a concern.

COMMIT (COncern Mining using Mutual Information over Time) is a history-based concern mining technique proposed recently. COMMIT analyzes the source code history to statistically cluster functions, variables, types, and macros that have been changed together intentionally in each modified function. Such clusters are further composed to form composite seeds according to the statistical measure of how closely related program entities are, namely, *mutual information*. For example, suppose that we have versions of small pieces of C code shown in Fig. 8 which is taken from the original paper of COMMIT. In the figure, bold text corresponds to the addition or removal of program entities between two contiguous versions. COMMIT will generate the seed graph depicted in Fig. 9, where edges are placed between every pair of program entities to which call or references have been co-added or co-removed. Three boxes in the figure represents clusters. Since `lock`, `enqueue`, `unlock`, and `queue` are co-added in function `client` in Version 2, they are placed in the same cluster. Note that `lock2` and `unlock2` are co-removed, and then `start_lock3`, `lock_data_queue`, `lock_data_queue`, and `end_lock3` are co-added in function `front_end` (renamed to be `front_end2`) between Version 2 and Version 3. Thus they are related to form Cluster B. Simultaneous co-addition of `lock_data_queue` and `lock_data_queue` in function `back_end` results in an independent cluster (Cluster C).

It should be also noted that COMMIT creates redundant edges represented by dashed lines in Fig. 9. These edges are placed because COMMIT only roughly relates a bunch of `lock2` and `unlock2` to another bunch of `start_lock3`, `lock_data_queue`, `lock_data_queue`, and `end_lock3`, although `lock2` has been renamed to `start_lock3` and `unlock2` `end_lock3`.

In the subsequent sections, we will describe how COMMIT is enhanced by fine-grained change analysis and by factbase facilities.

IV. Implementation

We have implemented COMMIT using Diff/TS and factbase facilities. An overview of the implementation, which we call COMMIT/D, is illustrated in Fig. 10. COMMIT/D is composed of the following modules: Diff/TS, Retriever, StatisticalFilter, and Sorter.

Diff/TS Diff/TS computes changes between all contiguous versions/revisions for a series of versions/revisions. Computed changes are encoded into facts, and then loaded into a factbase.

Version 1

```
void client(void) {
  start_log("sending...");
  /* do something */
  end_log("done.");
}

void front_end(void) {
  start_log("receiving...");
  /* do some thing */
  end_log("done.");
}

void back_end(void) {
  start_log("storing...");
  /* do something */
  start_log("done.");
}
```

Version 2

```
extern queue_t queue;

void client(void) {
  start_log("sending...");
  /* do something */
  lock(&queue);
  enqueue(&queue,/*...*/);
  unlock(&queue);
  end_log("done.");
}

void front_end(void) {
  start_log("receiving...");
  lock2(&queue);
  /* do some thing */
  unlock2(&queue);
  end_log("done.");
}

void back_end(void) {
  start_log("storing...");
  /* do something */
  start_log("done.");
}
```

Version 3

```
extern queue_t queue;

void client(void) {
  start_log("sending...");
  /* do something */
  lock(&queue);
  enqueue(&queue,/*...*/);
  unlock(&queue);
  end_log("done.");
}

void front_end2(void) {
  start_log("receiving...");
  start_lock3(&queue);
  lock_data_queue();
  /* do something */
  unlock_data_queue();
  end_lock3(&queue);
  end_log("done.");
}

void back_end(void) {
  start_log("storing...");
  lock_data_queue();
  /* do something */
  unlock_data_queue();
  start_log("done.");
}
```

Fig. 8. Versions of Small C Code (due to Adams and others [1])

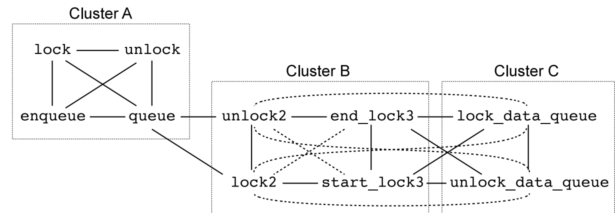

Fig. 9. A Seed Graph Generated by COMMIT

Currently, Jena TDB[5] RDF storage system is used for our factbases.

Retriever Retriever constructs initial seed graphs by retrieving co-addition/co-removal of program entities such as function/macro calls, types, and global variables from a factbase. SPARQL queries for co-addition/co-removal in C and Java programs, which we call QueryC and QueryJ, are shown in Fig. 11.

SPARQL is a query language for searching graph patterns in RDF factbases. Roughly speaking, SPARQL is an extension of SQL with graph patterns described by a set of triples with variables. For example, QueryC contains a graph pattern in the WHERE clause, where an identifier prefixed by "?" denotes a variable. It should also be noted that predicates and categories are encoded into Uniform Resource Identifiers (URIs) based on the RDF. Although typical SPARQL queries contain a

[5]http://openjena.org/TDB/

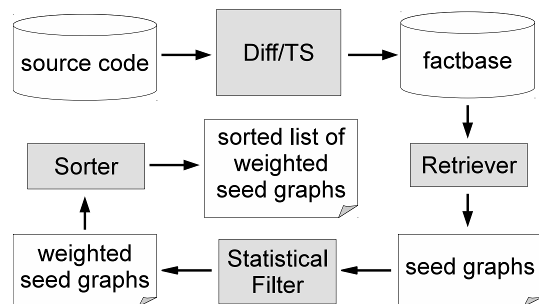

Fig. 10. COMMIT/D System

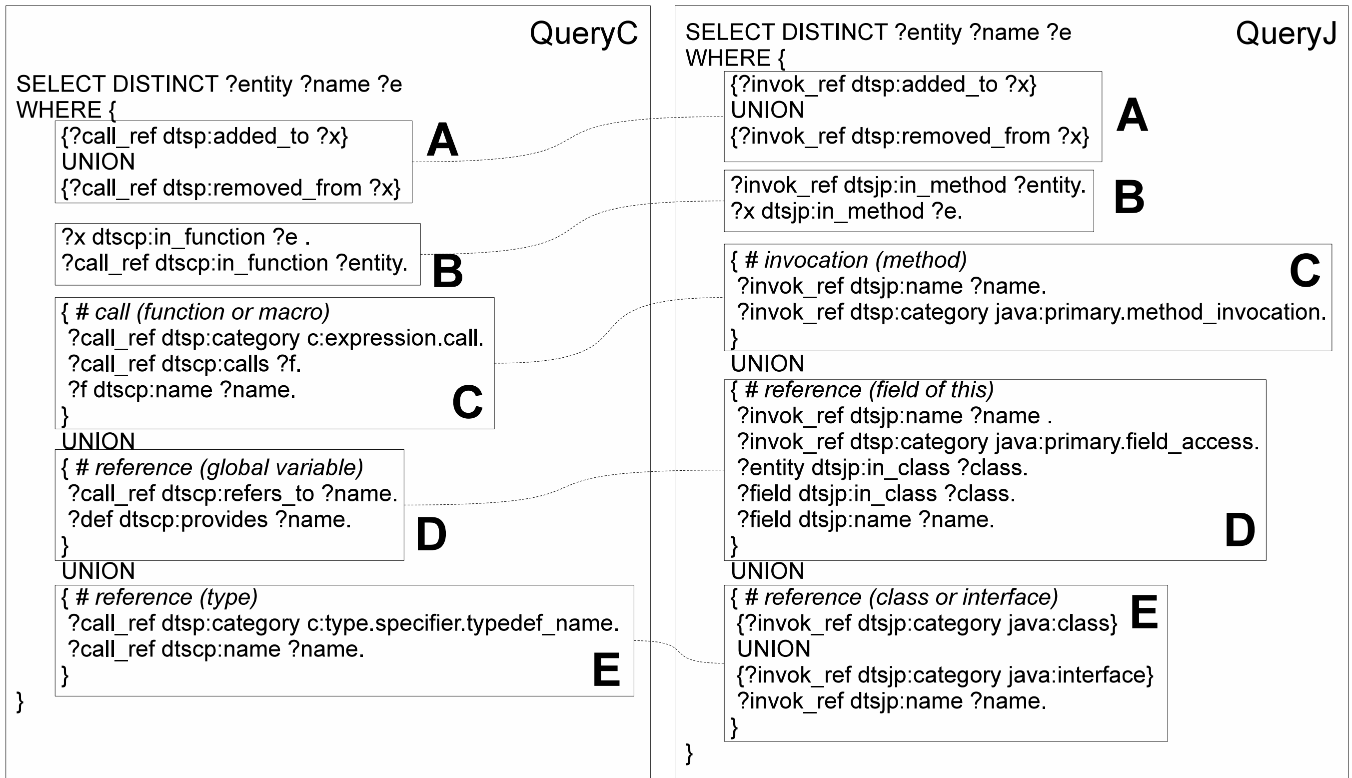

Fig. 11. Queries for Co-Additions/Co-Removals in C and Java Programs

lengthy mapping from prefixed names to URIs in the head, such mapping is omitted in QueryC and QueryJ. We only show below the meanings of the prefixes in the queries:

- dtsp: for predicates from Diff/TS,
- dtscp: for predicates from Diff/TS related to C programs,
- c: for syntactic categories of C programs,
- dtsjp: for predicates from Diff/TS related to Java programs, and
- java: for syntactic categories of Java programs.

We briefly explain QueryC. Regions from "#" to the end of line are interpreted as comments. The variables ?entity, ?e, and ?name denote the functions f that contain co-added/co-removed function calls or references denoted by ?call_ref, the functions in the previous or the next version/revision that correspond to f, and the names of the called functions, respectively. The graph pattern of QueryC consists of boxed sections A to E:

- A describes a pattern of changes,
- B describes relationships between ?call_ref, ?entity, and ?e,
- C describes a pattern of function calls,
- D describes a pattern of references of global variables, and
- E describes a pattern of references of types.

Note that QueryJ also consists of similar components to those of QueryC. We give explanations of the C related predicates that is present in QueryC. Let R be the set of code regions and S the set of strings.

s dtscp:in_function o $(s \in R, o \in R)$
$\Longrightarrow$ s is located in function definition o.
s dtscp:calls o $(s \in R, o \in R)$
$\Longrightarrow$ Function call s calls function o.
s dtscp:name o $(s \in R, o \in S)$
$\Longrightarrow$ The name of s is o.
s dtscp:refers_to o $(s \in R, o \in S)$
$\Longrightarrow$ s refers to global (external) name o.
s dtscp:provides o $(s \in R, o \in S)$
$\Longrightarrow$ s defines global (external) name o.

In the graph pattern, keyword UNION is used to state multiple alternative graph patterns. Exactly one of the alternatives must be matched by any query solution. If both branches of the UNION match, two solutions are generated. Co-addition/co-removal of entities are stated by using UNION patterns. For example, A contains the following UNION pattern.

{?call_ref dtsp:added_to ?x} (1)
UNION
{?call_ref dtsp:removed_from ?x} (2)

Then (1) or (2) must be matched. If both match, solutions like

?call_ref	?x	
$\langle fid_0, (4,4), (7,28) \rangle$ ⋮	$\langle fid_1, (3,0), (8,0) \rangle$ ⋮	solutions for (1)
$\langle fid_2, (12,2), (19,20) \rangle$ ⋮	$\langle fid_3, (10,0), (21,0) \rangle$ ⋮	solutions for (2)

are generated, where fid_0, fid_1, fid_2, and fid_3 are file identifiers.

By issuing QueryC, we obtain solutions (r, r', n) for (?entity, ?e, ?name), where $r, r' \in R$ and $n \in S$. We create a partial map M from $R \times R$ to the sets of S as follows: for each solution (r, r', n), if $N = M(r, r')$ exists, then add $(r, r') \mapsto N \cup \{n\}$ to M, or else add $(r, r') \mapsto \{n\}$ to M. Then the domain of M denotes a set of sets of co-added/co-removed entities.

Finally, based on the detected co-added/co-removed sets of entities, seed graphs are generated. A seed graph is an undirected graph with edges between every pair of program entities whose calls or references have been co-added or co-removed.

StatisticalFilter StatisticalFilter calculates *mutual information* for each pair of co-added/co-removed entities as in the original paper of COMMIT. Mutual information is a statistical measure of how closely related two entities are. In order to calculate mutual information, we first calculate frequency of entity pair (e_0, e_1), denoted by $\mathrm{freq}(e_0, e_1)$ as follows:

- Let $c = 0$.
- For each map entry $\{(r, r') \mapsto N\} \in M$, if $N \subseteq \{e_0, e_1\}$, then increment c.
- The value of c is the frequency of (e_0, e_1).

Similarly, we calculate frequency of entity e, denoted by $\mathrm{freq}(e)$ as follows:

- Let $c = 0$.
- For each map entry $\{(r, r') \mapsto N\} \in M$, if $e \in N$, then increment c.
- The value of c is the frequency of e.

Then we enumerate changed entities (functions) by another SPARQL query:

```
SELECT DISTINCT ?entity WHERE {
  ?entity dtsp:category c:definition.
  ?entity dtsp:modified ?x.
}
```

We let the number of changed entities be C. Finally, mutual information for e_0 and e_1, denoted by $I(e_0; e_1)$, is calculated as follows:

$$I(e_0; e_1) = \log_2 \left(\frac{p(e_0, e_1)}{p(e_0) \times p(e_1)} \right),$$

where

$$p(x) = \frac{\mathrm{freq}(x)}{C}, \quad p(x, y) = \frac{\mathrm{freq}(x, y)}{C}.$$

Once the mutual information between any two co-added/co-removed entities is calculated, we weight each edge in the seed graph with it.

Sorter Sorter orders the seeds first on the *dimension*, which denotes the number of program entities contained in a seed, then on the *scattering value*, which denotes the number of unique calling functions over which the concern is scattered. In practice, this means that cross-cutting concern seeds are ranked much higher than modular concerns.

We regard it remarkable that a highly complex analysis method such as COMMIT can be exposed to users in a conceptually clear way. It will be much easier for third-party users to explore further enhancement of the technique, for instance. In fact, we have successfully applied COMMIT to Java-projects, which was not even possible by the original researchers. In this sense, we can say that the method has overcome language barriers in software evolution analysis. The difference between QueryC and QueryJ directly reflects the difference between C and Java, that is, the difference between functions/methods, global variables/fields and types/classes. There may be other correspondence between these two languages to consider, of course.

V. Experiment

In order to show the effectiveness of our technique, we conducted an experiment on several open source software projects. We investigate, in particular, the impact of fine-grained rename detection on concern seeds. For that purpose, we use an implementation of the original COMMIT called COMMIT/D$^-$. It is derived from COMMIT/D by omitting rename detection. More precisely, we add the following lines at the bottom of the statements for the change pattern (A) in QueryC and the similar lines (with ?invok_ref in place of ?call_ref) for QueryJ in COMMIT/D:

```
UNION
{?call_ref dtsp:renamed ?x}
UNION
{?x dtsp:renamed ?call_ref}.
```

Adding these lines takes the same effect as regarding a rename operation as a combination of delete and insert operations. We apply COMMIT/D$^-$ and COMMIT/D to the sets of versions/revisions, and then compare the mining results. To assess the precision of rename detection of Diff/TS, we identify instances of rename operations reported by Diff/TS that affect the mining result of COMMIT/D, and then inspect the rename instances manually.

For the experiment, a PC with a pair of quad-core Intel Xeon CPU (2.93GHz) with 32GB RAM running under Mac OS X 10.6.8 was used. We retrieved versions and/or (SVN) revisions of the following projects:

- Apache Ant[6]: a build system for Java applications,
- JHotDraw[7]: a GUI framework for drawing editors,
- PostgreSQL[8]: a database management system, and
- Lighttpd[9]: a web server.

Apache Ant (Ant) and JHotDraw (JHD) are written in Java and the others are written in C except that PostgreSQL (PGSQL) contains JDBC driver code written in Java. For each project, the numbers of source files, SLOCs computed by SLOCCount[10], and examined versions/revisions with their numbers are shown in Table I, where SLOC and the number of source files are counted for the latest version/revision. A set of release versions and a set of revisions are both taken for Lighttpd (LHTV/LHTR).

[6]http://ant.apache.org/
[7]http://jhotdraw.org/
[8]http://www.postgresql.com/
[9]http://www.lighttpd.net/
[10]http://www.dwheeler.com/sloccount/

TABLE I
TARGET SOFTWARE PROJECTS AND THE FACTBASES

	Ant	JHD	PGSQL	LHTV	LHTR
SLOC	128 133	29 005	239 358	38 315	←
# src files	1 191	491	1 065	132	←
# vers/revs	45	294	21	24	2 150
ver/rev from	v1.1	r18	v6.1	v1.4.2	r648
ver/rev to	v1.8.2	r311	v7.1.3	v1.4.29	r2797
# facts (M)	26.09	4.31	36.94	11.04	19.71
FB size (GB)	3.8	0.8	5.3	1.7	2.9
time required (h)	1.4	0.8	2.0	0.5	9.4

TABLE II
REPORTED SEEDS AND AFFECTING RENAMES

		Ant	JHD	PGSQL	LHTV	LHTR
# seeds	CD^-	348	70	496	38	59
	CD	273	49	458	38	48
sum D	CD^-	2 090	197	1 868	123	199
	CD	1 551	142	1 649	117	167
decrease (%)		25.8	27.9	11.7	4.9	16.1
av. D	CD^-	6.0	2.8	3.8	3.2	3.4
	CD	5.7	2.9	3.6	3.1	3.5
av. SV	CD^-	2.8	1.0	1.0	1.0	1.0
	CD	2.4	1.0	1.0	1.0	1.0
time (s)	CD^-	59	7	103	8	11
	CD	46	6	65	6	8
# affecting renames		739	46	382	15	51
# valid renames		531	33	243	8	28
precision (%)		71.9	71.7	63.6	53.3	54.9

To build the factbases for the projects, we ran Diff/TS on all contiguous versions/revisions of each projects. Table I shows the numbers of facts (in megafacts) stored in the factbases, the file system usage (in gigabytes) of the factbases, and the time required (in hours) to build the factbases. Then we applied COMMIT/D^- and COMMIT/D on the factbases with the following mutual information thresholds: 10.3 for Apache Ant, 11.3 for JHotDraw, 13.6 for PostgreSQL, 10.1 for Lighttpd (versions), and 10.9 for Lighttpd (revisions). These values were determined to minimize the coefficient of variation for the dimensions of the reported seeds by using a simple exhaustive search technique as seen in the COMMIT paper.

Table II shows characteristics of concern seeds reported by COMMIT/D^- and COMMIT/D for the target projects. In the table (and also in the remaining tables), the following abbreviations are used: D for dimensions, SV for scattering values, CD^- for COMMIT/D^-, and CD for COMMIT/D. It should be noted that the sum of dimensions decreases for each project. This implies that rename detection reduces the amount of effort needed for inspecting concern seeds when renamed entities does not contribute to generating them. On the other hand, the number of seeds and the average dimension do not always decrease. By comparing in detail concern seeds generated by COMMIT/D^- with those generated by COMMIT/D, we noticed that seeds may appear, disappear, merge, split, and shrink in unpredictable manners when we ignore renamed entities in calculating seeds. For example, in the case of JHotDraw, 12 seeds appear, 33 seeds disappear, and 8 seeds reduce in dimension. As a result, the average dimension increases while the number of seeds decreases.

TABLE III
IMPACT OF VALID RENAMES AND RENAMES REPORTED BY COMMIT/D

	Ant	JHD	PGSQL	LHTV	LHTR
# seeds	278	55	478	40	52
sum D (valid)	1 685	161	1 748	122	181
sum D (CD)	1 551	142	1 649	117	167
false positives	134	19	99	5	14
(%)	8.0	11.8	5.7	4.1	7.7

We also measured the precision of the rename detection. First we identify the rename instances affecting seed generation, and then manually check the 1233 instances by using DiffViewer. The result is shown in Table II. We found that most of the false positives are due to the lack of semantic information such as control/data flows, call relations, and variable scopes. Since Diff/TS is a syntactic differencing tool, there is a limitation in detecting those subtle renames. Suppose that, for instance, a function definition F_0 is removed from one version, and another function definition F_1 is added at the same location as F_0 in the next version. Syntactic tools normally start by matching F_0 with F_1 even when F_0 has nothing to do with F_1. We are currently working to incorporate semantic information into rename detection. In Table III, the impact of the renames, which we judged valid by manual inspection, on the concern seed dimensions is compared with that of the renames which COMMIT/D reported. Note that the percentage of false positives in seed dimensions is much smaller than that of irrelevant (false) rename instances. Considering the decrease in sums of dimensions, it is fair to say that rename detection has a positive effect in the overall concern mining task.

The result of the experiment contains several interesting examples in which rename detection contributed to reducing concern seed members. For example, in version 1.4.11 of `mod_cml_lua.c` of Lighttpd, a code fragment shown in Fig. 12 (left) is changed to the other in version 1.4.12 (right). By manual inspection, we found that a call to `lua_open` is renamed to a call to `luaL_newstate`, and calls to `luaopen_{base,table,string,math,io}` are merged into a call to `luaL_openlibs`. COMMIT/D^- reported a concern seed that contains all call sites above, while COMMIT/D reported a seed that only contains `luaopen_{table,string,math,io}`, which means that redundant seed members has been removed. At the moment, COMMIT/D can not suggest the merge. However, we should be able to remove all the above call sites when semantic analyzers such as a call graph generator are integrated in our factbase. More powerful analysis may be performed to detect the merge by the same way as the factbase queries explained in the paper.

VI. RELATED WORK

We only mention researches that have direct connections with our method.

Concern mining or aspect recommendation has been a popular research topic in recent years. Static mining and

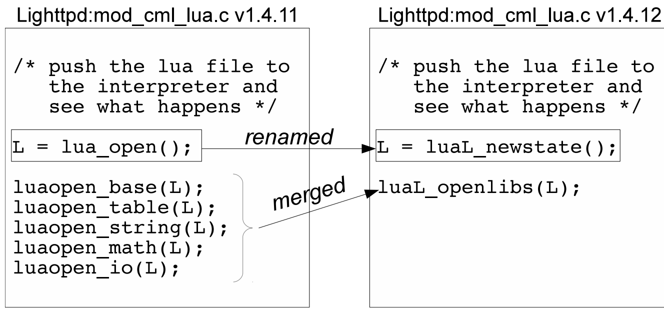

Fig. 12. Changed Code Fragments of Lighttpd

history-based mining are two major techniques based on source code analysis. The static technique analyzes source code of a version of software to extract seeds of concerns. A Fan-in value, which is the number of unique callers of each method/function, was first introduced by Marin and others [4] and further generalized by Zhang and others [5] to propose Clustering-Based Fan-in Analysis (CBFA). CBFA tries to diminish effects of common utility functions that are called too frequently, relies on textual similarity in names for generating larger seeds by clustering methods/functions, and ranks seeds by the sum of Fan-in values of method/functions in the seed. Although Fan-in values are useful for quickly assessing contributions of a method/function to concerns, it does not provide enough information on how we classify potential concerns. Nguyen and others focused on interactions of methods/functions, that is, how they call others and how they are called by others, to group and rank seeds by similarity of interaction [6]. However, as the technique becomes more complex, it becomes more difficult for others to use, maintain or improve. For example, Nguyen and others use an AST-based clone detection tool for measuring similarity. The process is, in general, highly heuristic and specific to target projects and programming languages.

Our approach, on the other hand, separates efforts of analysis from the task of fact generation so that analysis is done by way of queries to the factbase. For example, if we needed Fan-in values, we would let static analysis tools generate facts of call graphs and query callers. Even indirect callers can be easily listed. If we needed to measure interaction similarity, we would use facts of parse trees as well as those of tree differencing obtained by specifying appropriate pair of subtrees. Or more directly, we could let code clone detectors report facts of clone regions. However, when we integrate facts given by different tools, we have to perform region alignment mentioned in Section II. Its technical details will be published elsewhere.

The history-based mining technique was first adopted by Breu and others [7], who proposed History-based Aspect Mining (HAM). HAM clusters methods/functions that add or remove a call to the same method/function, and groups together methods/functions that are called by the same cluster as concern seeds. COMMIT [1] focuses on co-addition/removal of calls to function and macros, references to global variables and types in each added, modified, or removed function. Co-addition/removal are valuated by *mutual information* to generate seed graphs, in which pairs of functions connected with edges having higher values represent how they coincide in dependency changes. Ironically, these two techniques expose language barriers in source code analysis. HAM is built for Java-projects while COMMIT for C-projects. It is difficult to compare or combine these two tools. On the other hand, as demonstrated in Section V, our approach allows users to apply similar query patterns to different programming languages, such as Java and C. Our factbase is built around common concepts concerning changes such as removal, addition and renames, and is amenable to different languages. This will help developers/maintainers to exchange and share analysis ideas in the form of queries. Academic researches will benefit from the method since reproduction and comparison of results becomes easier if we share facts generated by own tools. We only have to publish analysis results in factbase without needing to make tools publicly available.

Now we turn to software analysis platforms based on factbase queries. One common approach to integrating the results of different tools is to assume a common model of software. OASIS [8] is a system for integrating information and services among reverse engineering tools based on a domain ontology and tool adapters. The ontology includes common concepts about program structures such as “systems”, “modules”, “subprograms” and “variables” and relations among them such as “containment” and “use”. Each tool maintains its own factbase and tool integration is done by on-demand factbase filtering and service brokerage.

FAMIX [9] is a model developed in the ESPRIT FAMOOS project for representing object-oriented software systems up to the granularity of program entities such as classes, methods and attributes in a language independent manner [9]. Moose [10] is an environment for reverse-/re-engineering complex software systems that was initially built around the FAMIX model. Moose maintains a factbase according to the (meta-)model by which tools exchange information.

Evolizer [11], [12] is a platform for software evolution analysis built within Eclipse[11]. Evolizer provides a set of meta-models for representing software project data and corresponding tools for obtaining data from software repositories including bug-tracking systems. Evolizer uses ChangeDistiller for classifying changes between revisions. Currently, CVS/SVN and the bug-tracking system Bugzilla are available as data source repositories.

Kenyon [13] is a system designed to facilitate software evolution research by providing a common set of solutions to common logistical problems. It accesses software repositories and stores extracted facts into a relational database with extensible fact extractors. Kenyon is unique in providing an origin analysis tool called Beagle for tracking source code entities across versions considering merge/split of entities such as functions, classes and files [14]. This feature is missing

[11] http://www.eclipse.org/

in Diff/TS. Kenyon does not seem to have a formally define model of software.

SE-Advisor [15] attempts to make software engineering activities less technical and more knowledge-centered by integrating tools and resources using ontology-based reasoning in semantic-web technologies. A formal ontology covering both software artifacts and software evolution processes is defined using the OWL-DL language[12]. Factbases are populated with data collected from software repositories according to the RDF data model, for which users can make SPARQL queries to gather information.

SOFAS [16] is a distributed and collaborative platform to enable inter-operation of various software analyses using web-service technologies. Analyses are offered as web services and can be combined via the software analysis broker to achieve more complex tasks. Software analysis ontologies are defined for data exchange between services which include an issue tracking ontology, a version control ontology and a source code ontology based on FAMIX. The ontologies and the factbase are constructed in the same ways as SE-Advisor.

Among these tools, OASIS and SOFAS assume distributed factbases, owned and accessed (via brokers) by individual tools, while others assume a centralized factbase directly accessed by each tool. Since we advocate collective efforts in software evolution analysis in the same sense as Linked Data [17], our method should not depend on a centralize factbase tied to specific analysis platform. Since we wish to accommodate a wide range of source code analysis tools, our method should not be bound by models of programs specific to certain programming languages. However, meta-models of software evolution should be used for incorporating facts of other software artifacts than source code in our method. We believe that the approach takes off early once we find a way to integrate queries for distributed factbases.The web-service based approach of SOFAS might be relevant in this regard.

iSPARQL [18] is an extension of the SPARQL query language that allows RDF factbase queries for similarities. Users can specify similarity measures such as Levenstein distance and similarity conditions in the extended queries. iSPARQL was successfully applied for software evolution analysis tasks such as code smell detection and metrics calculation by preparing ontologies similar to that of SOFAS and similarity measures suitable for software entities such as Levenstein distance, tree edit distance and graph similarity [19]. The studies of SE-Advisor and iSPARQL share the idea of "query based analysis of software evolution" with our research. Our approach differs in accepting facts in arbitrary granularity of code since it uses textual regions for representing source code entities.

VII. Conclusion

Summary

We have presented a method for recording source code changes as facts over textual regions according to the RDF (Resource Description Framework) data model to allow analysis to be performed in terms of factbase queries. The method advocates collective efforts in software evolution analysis in the same spirit as Linked Data [17] with emphasis on "query based analysis".

We have explained application of the method to history-based concern mining by reinforcing the techniques with a fine-grained change analysis based on tree differencing on abstract syntax trees. To demonstrate the capability of the method, we have reported on an experiment that emulates the state-of-the-art concern mining technique called COMMIT [1] using our own change analysis tool called Diff/TS [2]. A comparative case study on several open source projects written in C and Java shows that our technique improves the results and overcomes the language barrier in the analysis. It is remarkable that similar SPARQL query patterns are used to find cross-cutting concerns in C-projects and Java-projects according to the mining technique in use.

We envision the world where tool developers publish their analysis results and curious users try elaborated queries possibly with help from others providing custom queries. This will revolutionize the process of software analysis because practically anyone in the world can make significant contributions without needing complex hardware or software. All this can occur over the Internet using standard web technologies such as RDF and SPARQL. It will also benefit academic researches since it becomes much easier to reproduce and test experimental results.

Threats to Validity

As explained in Section III, intervals of change analysis affect the results of mining since identification of co-addition and co-removal of method/function calls is crucial in generating seeds of concerns. In section V, we analyzed change between releases and SVN change sets, which may be too coarse or fine for accurately extracting concerns. We may have to compare results with intermediate time intervals. We may also have to consider code ownership when we process release versions. This can be done by adding facts of ownership and modifying query patterns. However, the main purpose of the paper is to demonstrate effects of incorporating fine-grained change analysis into COMMIT, which is not severely affected by time intervals and code ownership.

We have observed in Section V that treating renamed entities as the same old ones prevents generating irrelevant seeds. However, there are cases where renames occur when concerns become explicit, such as a standard `printf()` function replaced by a newly defined `log()` function. In our approach, `log()` is identified with `printf()`, which makes the introduction of a logging concern difficult to detect since `printf()` appears in many places in the source code. On the other hand, `COMMIT` treats the rename as deletion of `printf()` and insertion of `log()`, which may lead to generation of useful seeds involving `log()`. A concrete example of this case is found for function `log_error_write()` of Lighttpd. Unfortunately, `log_error_write()` appears in too many places and failed to reach the top ten seeds we

[12]http://www.w3.org/owl/

calculated. We can rescue this particular case by examining insertion of the definition of `log()`, which is achieved by adding one `UNION` clause in the `SPARQL` query shown in Fig. 11.

The COMMIT paper does not describe their underlying techniques in full detail, especially its ability to track renamed entities across versions. From the context, we assumed that COMMIT tries to identify renamed entities by examining entities that have been removed and then added depending on the context. From our experience, this approach is very unstable and prone to false positives. While the results of the experiments shown in Section V support our observation, there is a chance that we are mistaken.

Future Plans

Future work includes the following items:

- We plan to make analysis data available to public over the web by hosting a SPARQL query server connected to our factbase. Again, this will revolutionize the process of software analysis because practically anyone in the world can make contributions without needing expensive hardware or software. A fine-grained change analysis tool such as Diff/TS plays a central role in organizing and comparing analysis results offered by various tools on each version.
- We are working on the Linux-2.6 kernel source code. Diff/TS is already capable of analyzing the whole versions, however, we have not yet tamed a source code analysis tool for C/C++ except code clone detectors. We are working on ROSE[13] and Clang[14] at the moment, both of which are able to generate call graphs and CFGs. It is an interesting challenge to host analysis data of the Linux as it will take much more disk space and computation power than Ant. Let us mention that Linux is approximately 70 times larger than Ant in SLOC per version.
- We plan to reformulate advanced fine-grained analyses reported in recent papers in a query-based style. We are working on code smells/churn detection [19], static/ dynamic call graph comparison between revisions [20], time-dependence analysis of code changes [21], non-essential change analysis [3], and clone tracking [22].

References

[1] B. Adams, Z. M. Jiang, and A. E. Hassan, "Identifying crosscutting concerns using historical code changes," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 305–314.

[2] M. Hashimoto and A. Mori, "Diff/TS: A tool for fine-grained structural change analysis," in *WCRE '08: Proceedings of the 15th Working Conference on Reverse Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 279–288.

[3] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering*, May 2011, pp. 351–360.

[4] M. Marin, A. V. Deursen, and L. Moonen, "Identifying crosscutting concerns using fan-in analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, pp. 3:1–3:37, December 2007.

[5] D. Zhang, Y. Guo, and X. Chen, "Automated aspect recommendation through clustering-based fan-in analysis," in *Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, 2008, pp. 278–287.

[6] T. T. Nguyen, H. V. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Aspect recommendation for evolving software," in *Proceeding of the 33rd International Conference on Software Engineering (ICSE '11)*. Waikiki, Honolulu, HI, USA: ACM, May 2011, pp. 361–370.

[7] S. Breu and T. Zimmermann, "Mining aspects from version history," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 221–230.

[8] D. Jin and J. R. Cordy, "Ontology-based software analysis and reengineering tool integration: the OASIS service-sharing methodology," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 613–616.

[9] S. Tichelaar, S. Ducasse, and S. Demeyer, "FAMIX and XMI," in *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 296–298.

[10] O. Nierstrasz, S. Ducasse, and T. Gǐrba, "The story of Moose: an agile reengineering environment," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 1–10.

[11] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, pp. 26–33, January/February 2009.

[12] 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*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 23–32.

[13] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating software evolution research with kenyon," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2005, pp. 177–186.

[14] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, 2005.

[15] J. Rilling, R. Witte, P. Schuegerl, and P. Charland, "Beyond information silos – an omnipresent approach to software evolution," *International Journal of Semantic Computing (IJSC)*, vol. 2, no. 4, pp. 431–468, December 2008, special Issue on Ambient Semantic Computing.

[16] G. Ghezzi and H. C. Gall, "Sofas : A lightweight architecture for software analysis as a service," in *Working IEEE/IFIP Conference on Software Architecture (WICSA 2011), 20-24 June 2011, Boulder, Colorado, USA*. IEEE Computer Society, 2011, p. to appear.

[17] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data - the story so far," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 5, no. 3, pp. 1–23, 2009.

[18] Kiefer, A. Bernstein, and M. Stocker, "The fundamentals of iSPARQL: a virtual triple approach for Similarity-Based semantic web tasks," *Lecture Notes In Computer Science*, vol. 4825, pp. 295–309, 2007.

[19] C. Kiefer, A. Bernstein, and J. Tappolet, "Analyzing software with isparql," in *Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2007)*. Springer, June 2007.

[20] R. Holmes and D. Notkin, "Identifying program, test, and environmental changes that affect behaviour," in *Proceeding of the 33rd international conference on Software engineering*. New York, NY, USA: ACM, 2011, pp. 371–380.

[21] O. Alam, B. Adams, and A. E. Hassan, "A study of the time dependence of code changes," in *Proc. of the 16th Working Conference on Reverse Engineering, WCRE 2009*, October 2009, pp. 21–30.

[22] E. Duala-Ekoko and M. P. Robillard, "Clone region descriptors: Representing and tracking duplication in source code," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 1, pp. 1–31, 2010.

[13]http://www.rosecompiler.org/

[14]http://clang.llvm.org/