A Comprehensive and Scalable Method for Analyzing Fine-Grained Source Code Change Patterns

Masatomo Hashimoto*, Akira Mori†, and Tomonori Izumida†
*RIKEN Advanced Institute for Computational Science, Japan
Email: m.hashimoto@riken.jp
†National Institute of Advanced Industrial Science and Technology, Japan
Emails: {a-mori,tomonori.izumida}@aist.go.jp

***Abstract*—This paper presents a comprehensive method for identifying fine-grained change patterns in the source code of large-scale software projects. Source code changes are computed by differencing abstract syntax trees of adjacent versions and transferred to a set of logical statements called a factbase. A factbase contains information for tracking and relating source code entities across versions and can be used to integrate analysis results of other tools such as call graphs and control flows. Users can obtain a list of change pattern instances by querying the factbase. Experiments conducted on the Linux-2.6 kernel, which involve more than 4 billions of facts, are reported to demonstrate capability of the method.**

I. Introduction

Understanding source code changes and their effects on software systems is an important and challenging problem especially for a large-scale project, where the system is developed by a group of people and has a long history. Considering the rigor and the volume of the information needed for accomplishing the task, it is imperative to automate computational steps such as change identification and classification, entity tracking, fact extraction and integration, factbase management, and query processing. Although there have been many studies, proposed methods, and tools on the issues [1], [2], [3], [4], previous approaches either focused only on specific aspects of the problems, lacked precision needed for detailed analysis, assumed particular programming systems and languages, or lacked interoperability over standard technologies. This forces developers and maintainers to customize new methods and tools to collect and integrate data across versions for comprehending the past and the present developments.

To remedy the situation, we propose in this paper a comprehensive method for analyzing fine-grained code changes in a way that is scalable, interoperable and transferable. The method takes a "query-based" approach and allows users to work with query sentences rather than source files by expressing "facts" about source code changes with vocabularies defined by a common set of "ontologies" and storing them in a database of facts called a "factbase".

The factbase contains change histories at the level of abstract syntax trees (ASTs) computed by comparing ASTs of adjacent versions [5]. They are used for integrating facts given by other tools [6], [7]. The factbase takes queries that specify graph patterns of changes and finds matching instances. The retrieved instances contain references to original source code texts and can be examined using various user friendly tools such as source code viewers. For interoperability, standard web technologies such as the RDF[1] (Resource Description Framework) data model, the OWL ontology language[2] and the SPARQL query language[3] are employed.

To illustrate the capability of the method, we explain experiments of fine-grained change pattern identification conducted for the Linux-2.6 kernel source code. The analysis involves more than 4 billions of RDF triples that describe facts about AST changes, control flows, and call relations from release 2.6.18 through 2.6.39. We first show the results of refactoring identification originally reported by Kawrykow and Robillard for Java systems [4], where cosmetic changes such as renaming and introduction of local variables are identified for further substantial analysis. Then we report identification of bug fixing patterns by taking an example from a program matching and transformation tool called Coccinelle [8]. We finally describe analysis of so-called "BKL (Big Kernel Lock) pushdown", a community wide effort to erase the obsolete locking mechanism from the entire kernel tree. Since the pushdown pattern must be specified with the locking context in terms of control flows and call relations, we employ a C language parser from Coccinelle for generating statement level control flows and a compiler tool called ncc[4] for generating function call graphs.

The results show that our method can detect change instances both precisely and efficiently. It is also suggested that the method is useful for analyzing changes in a broader perspective since queries devised for specific patterns can be easily adapted for other patterns and even for different programming languages. The method allows users to concentrate on analysis of facts rather than generation of facts, which may open the door to "query engineering" for software analysis.

The rest of the paper is organized as follows. Section II reviews technical backgrounds such as AST differencing, factbase construction and factbase queries. Experiments of source code change pattern identification are detailed in Section III.

[1] http://www.w3.org/RDF/
[2] http://www.w3.org/OWL/
[3] http://www.w3c.org/TR/sparql11-query/
[4] http://students.ceid.upatras.gr/~sxanth/ncc/

978-1-4799-8469-5/15 © 2015 IEEE

SANER 2015, Montréal, Canada

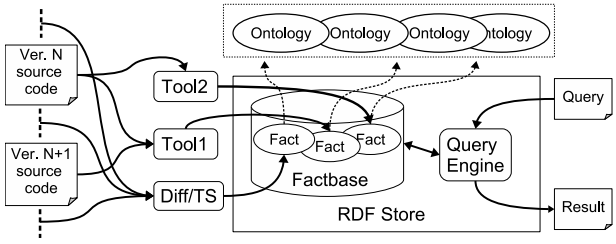

Fig. 1. Query Based Change Pattern Identification

After related work is explained in Section IV, Section V concludes with discussions.

II. Overview of The Method

In this section, we review technical backgrounds for the proposed method. Figure 1 illustrates the concept of query based change pattern detection. The method takes the following steps:

1) Common concepts and vocabularies for expressing facts about source code changes are defined as ontologies. The RDF data model and the OWL language are used for this.
2) For each adjacent version pair of the target system, change histories are computed by way of AST comparison [5] and recorded in the factbase of an RDF store. The tools is called Diff/TS.
3) Per version analysis by other tools is performed. Additional descriptions in the ontologies and modifications to the tools may be necessary. The results are imported to the factbase.
4) A user prepares a change pattern query using the SPARQL language. The query is executed on the RDF store and the user get a list of instances, which can be examined by various diff viewers.

In the following, we explain important items in detail.

A. Computing Source Code Changes

Diff/TS is a change analysis tool developed in our past project [5]. It is based on tree differencing on ASTs and identifies fine-grained changes between revisions of software systems written in C, C++, Java, Python, Verilog and Fortran. Diff/TS regards a revision of a software system as a directory tree where leaves correspond to ASTs derived from source files. It first pairs corresponding source files by comparing directory trees and parses each file pair to obtain an AST pair (t_1, t_2). It then calculates a sequence of *edit operations*, consisting of deletion, insertion, relabeling, and move of AST nodes that transforms t_1 into t_2. The *cost* of an edit sequence is estimated by giving a cost value for each edit operation and calculating the sum for the operations contained in the sequence.

Since an AST is naturally modeled by a labeled ordered tree, we can use an optimal tree differencing algorithm to compute edit sequences with minimum costs [9]. However, optimal algorithms are quasi-quadratic at best in time complexity and quadratic in space complexity [10], and inefficient for large trees that contain tens of thousands of nodes such as ASTs derived from thousands of source lines of code.

Diff/TS approximates an optimal algorithm by employing tactics such as tree decomposition, subtree hash encoding, and heuristic post-processing mechanisms. As a result, the edit sequences computed by Diff/TS may not be optimal in terms of costs, but nonetheless its precision, in terms of source code changes, has been verified by a series of advanced software change analyses [5], [6], [7]. The tool now scales to large code bases such as the whole Linux-2.6∼3.x kernel trees.

Diff/TS not only calculates the differences but also identifies the parts that are unchanged, which offers a powerful way of tracking source code entities across versions even when they are renamed and moved. In technical terms, the tool generates AST node mappings between given pairs of software revisions, where mapped pairs represent unchanged, relabeled or moved nodes. This mapping capability play an important role for integrating per version facts such as call relations and control flows, which we will see in Section III.

B. Representing Facts

Our goal is to develop a method which is independent of specific programming tools, platforms, models and languages to facilitate collaborative efforts in software change analysis. To achieve this, we construct accessible databases about source code changes using the Semantic Web technologies[5], which aim at describing, publishing, and understanding relationships between things on the Web. We will explain basic ideas of factbase construction in the following.

A fact about source code is described as a triple of *subject*, *predicate* (also called *property*), and *object* following the RDF data model. Both subjects and objects may be source code entities such as files, functions/methods, and statements. Predicates denote binary relations between source code entities or between source code entities and their attributes. In the latter case, the object may be a *literal*. For example, let v be a variable. Then $(v, name, \texttt{"x"})$ represents a fact that v has a name `x`.

At this point, we must decide how we represent source code entities such as a variable v in the above example. For this purpose, we had decided to use textual regions in source files for representing source code entities [7]. Since any tools and users should be able to point code regions of their interests, textual regions should serve as universal and independent ways of sharing and exchanging information about source code.

Since the Semantic Web technologies advocate using IRIs[6] (Internationalized Resource Identifier) to identify things on the web, we associate an IRI with a source code entity by first concatenating and then encoding the following items:

- an ID of the source file containing the entity,
- the start position of the entity in the file, and
- the end position of the entity in the file.

We can employ a hash value of the file for an ID, and triples of a line number, a column number and an offset for positions

[5]http://www.w3.org/standards/semanticweb/
[6]http://www.ietf.org/rfc/rfc3987.txt

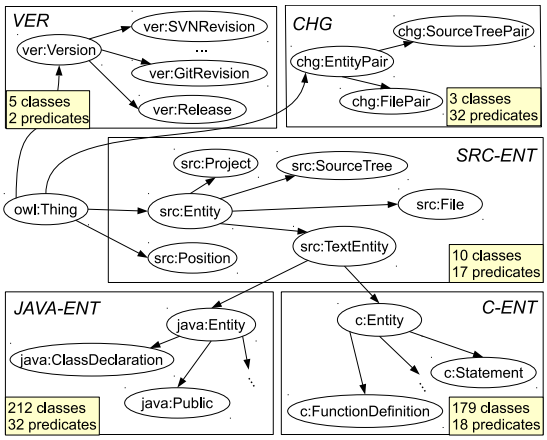

Fig. 2. Class Hierarchy of Source Code Ontologies (Excerpts)

in the file. For example, we can use an IRI
http://example.com/fact/MD5_ ⟨*hash value*⟩-51_0_2021_51_5_2026
to represent an entity e located between column 0 to 5 at line 51 (or similarly between offset 2021 to 2026) in a source file that has an MD5 hash value as encoded in the IRI[7]. We can express, for instance, a fact that a Java entity e is a public modifier by a triple $(e, \text{rdf:type}, \text{java:Public})$, in which e refers to a region in a Java source file containing a text "public". Note that the predicate rdf:type and the object java:Public both are IRIs abbreviated by using namespace prefixes.

Kinds of source code entities and predicates such as "variable", and "name" above are specified in *ontologies*. The predicate rdf:type is a built-in that relates an instance with its class. Ontologies define concepts and relationships used for expressing facts. We have defined the following ontologies needed for fine-grained change analysis on source code:

- a minimum core ontology *SRC-ENT* for specifying source code entities independent of programming languages and tools,
- an ontology *VER* for representing concepts about versions in source code management systems,
- ontologies for describing syntactic entities of programming languages, which are defined currently for C (*C-ENT*) and Java (*JAVA-ENT*), and
- an ontology *CHG* for expressing facts related to AST changes.

Figure 2 shows the hierarchy of conceptual classes of these ontologies. The numbers of classes (including subclasses) and predicates (including sub-properties) defined in each ontology are shown in shaded boxes. Ontologies are defined using the OWL ontology language, where each class is defined as a subclass of owl:Thing. We disambiguate names of conceptual classes by prefixing namespaces to names such as owl:Thing and src:Entity. We assume in this paper namespaces for ontologies shown in Figure 2, XML Schema Datatypes, RDF, RDF Schema, and OWL.

Ontologies for specific programming languages, such as *JAVA-ENT* and *C-ENT*, define subclasses of src:TextEntity. They

[7]We actually use SHA1 instead of MD5 to further avoid collisions.

TABLE I. PREDICATES FOR ONTOLOGIES (EXCERPTS)

	Predicate	Subject class	Object class
SRC-ENT	src:parent	src:Entity	src:Entity
	src:children	src:Entity	rdf:List
	src:containedIn	src:Entity	src:Entity
	⊢ src:inProject	src:Entity	src:Project
	⊢ src:inFile	src:TextEntity	src:File
	src:location	src:File	rdfs:Literal
C-ENT	c:inDeclaration	c:Entity	c:Declaration
	c:inFunction	c:Entity	c:FunctionDefinition
	c:conditionOf	c:Expression	c:IfStatement
	c:rhs	c:Assign	c:Expression
	c:name	c:Entity	rdfs:Literal
CHG	chg:deletedOrPruned	src:Entity	src:Entity
	⊢ chg:deletedFrom	src:Entity	src:Entity
	⊢ chg:prunedFrom	src:Entity	src:Entity
	chg:insertedOrGrafted	src:Entity	src:Entity
	⊢ chg:insertedInto	src:Entity	src:Entity
	⊢ chg:graftedOnto	src:Entity	src:Entity
	chg:movedTo	src:Entity	src:Entity
	chg:mappedTo	src:Entity	src:Entity
	⊢ chg:mappedEqTo	src:Entity	src:Entity
	⊢ chg:mappedNeqTo	src:Entity	src:Entity

are derived from the syntax of the language or more specifically from the syntactic categories used by parsers generating ASTs. We developed our own Java parser and hence *JAVA-ENT* reflects the official Java Language Specification[8]. For C programs, we employ a C parser developed for the Coccinelle project[9] which aims at program matching and transformation for unpreprocessed C code [8]. Note that an instance of src:TextEntity will be identified by an IRI explained above.

OWL admits two types of predicates: *object properties*, which are the relations between instances of conceptual classes, and *datatype properties*, which are the relations between instances and RDF literals or possibly XML schema datatypes[10]. Accordingly, a predicate is defined in OWL as a sub-property of either owl:ObjectProperty or owl:DatatypeProperty. For both types of predicate, the domain (the class of its subject) and the range (the class of its object) are specified.

Table I (top) lists predicates excerpted from *SRC-ENT*. The upper predicates are object properties and the lower ones are datatype properties. Indentation with ⊢ indicates that the predicate is a sub-property of the upper one. For instance, a predicate src:inFile is a sub-property of src:containedIn and may describe a textual entity contained in a local file, whose path name is specified with a predicate src:location. Note that predicates src:parent and src:children are used to specify the parent and the children of a node in ASTs.

Since all experiments reported in this paper target the Linux kernel source code, we only show some of the predicates defined in *C-ENT* in Table I (middle). The meanings should be clear. The main predicates for the versioning ontology *VER* include ver:version that relates a source code entity with its version and ver:next that relates adjacent versions. Predicates for AST changes defined in *CHG* are excerpted in Table I (bottom). The meanings of the predicates shown in the table are explained below in terms of AST changes.

- A fact $(n, \text{chg:mappedTo}, n')$ means that a node n has a corresponding node n' in another AST. In a concrete

[8]http://docs.oracle.com/javase/specs/
[9]http://coccinelle.lip6.fr
[10]http://www.w3.org/TR/xmlschema-2/

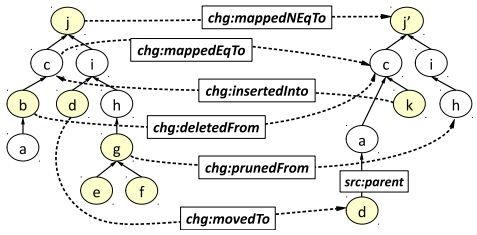

Fig. 3. A Fact Graph for AST Changes

term, n is either unchanged, relabeled or moved to be n'. When n is relabeled, it implies $(n, \text{chg:mappedNeqTo}, n')$, otherwise $(n, \text{chg:mappedEqTo}, n')$.

- A fact $(n, \text{chg:deletedFrom}, n')$ means that a node n is deleted from its parent which corresponds to a node n' in another AST. The predicate chg:prunedFrom is used instead of chg:deletedFrom when a whole subtree rooted at n is deleted.
- A fact $(n, \text{chg:insertedInto}, n')$ means that a node n is inserted to be a child of a node which node n' in another AST corresponds to. The predicate chg:graftedOnto is used instead of chg:insertedInto when a subtree rooted at n is inserted.
- A fact $(n, \text{chg:movedTo}, n')$ means that a node n is moved to be n' in another AST.

Definition of the above predicates reflects our experience in fact representation and factbase query using them. For instance, predicates chg:deletedFrom and chg:insertedInto imply correspondence of parent nodes.

According to the RDF data model, a set of facts form a directed graph called a *fact graph*, where each triple is represented by a (sub)graph $s \xrightarrow{p} o$. Figure 3 shows a fact graph illustrating intuitive meanings of AST change predicates explained above. In the graph, solid arrows represent src:parent and the arrows chg:mappedEqTo for a, h, and i are omitted for brevity.

C. Managing Factbase

We can start filling an RDF factbase with a set of facts when we have ontologies and tools to collect instances and generate facts among them. There are a number of software systems for managing factbases with ontologies. For the current research, we have chosen an open source edition of an RDF store called Virtuoso[11] as our factbase management system. An RDF store is a database system specialized for storing and managing the RDF data.

We assume that tools export analysis results to files. Therefore, users need to modify tools and/or create data exporters so that the results are loaded automatically to the RDF store according to ontologies. Although time and effort of the modification is not negligible, we regard it acceptable because it is a one-time effort and the generated factbase can be reused and shared for many different purposes.

In the following, we explain some important issues arising in querying the factbase.

[11] http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/

1) Querying Factbase: Once the factbase is filled with facts about changes, we can start querying change patterns. We use a standard query language for RDF data called SPARQL for specifying queries. SPARQL has a syntax similar to that of SQL and allows users to specify graph patterns of facts with variables. For example, the following query[12]

```
SELECT DISTINCT ?func ?fname WHERE {
  ?func a c:FunctionDefinition ;
        c:name ?fname .
}
```

instructs the RDF store to find fact graphs matching the pattern

$$\text{c:FunctionDefinition} \xleftarrow{\text{rdf:Type}} \text{?func} \xrightarrow{\text{c:name}} \text{?fname}$$

and report values for specified variables.

In a SPARQL query, a graph pattern is specified with a set of triples of the form "*subject predicate object*". placed in the **WHERE** clause. Triples having the same subject may be grouped so that the subject is omitted on and after the second one.

In a graph pattern, identifiers prefixed by "?" denote query variables. When an RDF store receives the above query, it searches for fact graphs that match the specified pattern treating query variables ?func and ?fname as wildcards to be instantiated with the matched graphs. Note that "**a**" is an abbreviation of rdf:type and the **DISTINCT** modifier inhibits the same solution from appearing multiple times. It is not difficult to see that the query lists distinct function names in the factbase.

2) Guarding Facts: An idea of representing a source code entity with its source text region and using a hash value of the source file for IRI encoding is simple and effective since the encoding is unique as long as no hash collision occurs. However, there is a subtle problem when an identical file exists for different versions or even different software system.

Suppose that an identical source file f exists in both versions v and v'. Since f is an instance of src:File, it is identified by an IRI which encodes the file hash value both in v and v'. As a result, the following fact graph is formed.

$$v \xleftarrow{\text{ver:version}} f \xrightarrow{\text{ver:version}} v'$$

While the graph alone does no harm, a loss of information occurs when f has different attributes in v and v' such as fully qualified file names (FQFNs). If f has FQFNs n and n' in v and v', respectively, a query for enumerating these file names in all versions such as

```
SELECT DISTINCT ?file ?version ?fname WHERE {
  ?file a src:File ;
        ver:version ?version ;
        src:location ?fname .
}
```

will return a list including (f, v', n) even when there is no file having name n in version v'.

To avoid the loss of information, we need to associate each fact $(f, \text{src:location}, n)$ with the version v in which f has a name

[12] In this paper, the PREFIX clauses of a SPARQL query are omitted for brevity. They define mapping from prefix names to namespace IRIs.

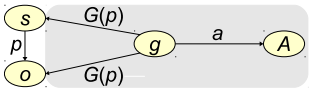

Fig. 4. Guard of a Fact

n. For this purpose, we use an RDF *blank node* to virtually create a triple having another triple as a subject. An RDF blank node is a unique anonymous resource that is neither a IRI nor a literal. It is typically used for grouping data by placing an edge from it to each group member.

Figure 4 illustrates a virtual fact $((s, p, o), a, A)$, called a *guard* of (s, p, o), where g denotes a blank node, p a predicate to be guarded such as src:location, a a distinctive property such as ver:version, A an instance of a, and $G(p)$ a predicate uniquely determined by p. We can retrieve A from (s, p, o) and a by combining simple queries.

In general, we consider guarding a predicate when it involves a class whose instance is uniquely identified by a file hash value and when it is not closed within the file boundary. For the experiments reported in this paper, we guard predicates src:location and ncc:mayCall with versions. The latter will be explained Section III-C. Note that we could use RDF reification to express statements about a fact. However, it requires more triples than guarding and is not suitable for large-scale analysis.

3) Enriching Factbase: As explained in Section II-A, Diff/TS compares ASTs of the corresponding file pairs in the directory tree. Although this saves unnecessary comparisons on remote file pairs, there may be a case where we lose track of source code entities across files and hence fail to identify changes across files. This typically arises when a class or a function is moved from one file to another or when a file is entirely removed or added.

In the former case, we can compare a removed entity and an added entity and conclude that they are the moved entity if they have the same name and/or the similarity of the bodies is above pre-defined threshold. Although it is an error prone method, it works well in many cases.

In the latter case, Diff/TS would not know whether an entity contained in the removed or added file has been removed or added, respectively. However, Diff/TS does know that the file has been removed or added in the directory tree and we would know that entities contained in the files have been removed or added, too. We can use this information to recover changes in the same way as in the former case. In both cases, we store the extra facts for future queries so that they are processed faster. We also generate facts about the class hierarchy in object-oriented languages, removal/addition of entities in the removed/added classes and functions, and types of declared variables, all of which are inferred from AST facts provided by Diff/TS.

4) Inferring Facts: RDF stores offer built-in mechanism for deriving facts from the hierarchy of ontology classes. For example, we can assume a non-existent fact $(e, \text{chg:mappedTo}, e')$ when a fact $(e, \text{chg:mappedEqTo}, e')$ exists in the factbase since chg:mappedEqTo is a sub-property of chg:mappedTo as explained in Section II-B. The inference capability of RDF stores is

```
ret = usb_control_msg(dev->udev, usb_sndctrlpipe(dev->udev, 0), req,
                      USB_DIR_OUT | USB_TYPE_VENDOR | USB_RECIP_DEVICE,
                      0x0000, reg, dev->urb_buf, len, HZ);
```

```
int pipe = usb_sndctrlpipe(dev->udev, 0);

ret = usb_control_msg(dev->udev, pipe, req,
                      USB_DIR_OUT | USB_TYPE_VENDOR | USB_RECIP_DEVICE,
                      0x0000, reg, dev->urb_buf, len, HZ);
```

Fig. 5. Local Variable Extraction in Linux 2.6 Kernel

```
SELECT DISTINCT ?decl_ ?x_ ?f_ WHERE {
 ?decl_ a c:InitDeclarator.
 ?decl_ chg:insertedInto ?e.
 ?decl_ c:initializer ?rhs_.
 ?x_ chg:insertedOrGrafted ?f.
 ?x_ src:parent ?f_
 ?x_ c:declaredBy ?decl_.
 ?a src:parent ?f.
 ?a chg:movedTo ?rhs_.
 ?f a ?cat. ?f_ a ?cat.
}
```

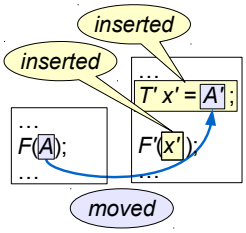

Fig. 6. A SPARQL Query for Local Variable Extraction

powerful and is one of the main reasons we have chosen the Semantic Web technologies for implementing the proposed method.

III. Experiments

In this section, we explain experiments of change pattern identification for the Linux kernel release versions from 2.6.18 through 2.6.39 taken from a local clone of the mainline Git repository[13]. All experiments described in the rest of the paper were done on a workstation with 8-core Intel Xeon processor (3.0 GHz) with 64GB RAM. Since each version has several millions of source lines of code, the number of generated facts is large. It took about 16 hours for Diff/TS to compute AST differences for 21 adjacent version pairs. More than 4 billion facts were generated and it took about 98 hours for Virtuoso to load them. In addition to facts about AST changes reported by Diff/TS, facts about control flows and call relations are used for the second and the third experiments, respectively. Details of additional facts will be explained in the corresponding sections.

A. Identifying Non-Essential Changes

We first show how we can identify cosmetic change patterns of a program by querying the factbase. We use as an example the **local variable extraction** pattern explained by Kawrykow and Robillard for Java programs [4]. The pattern introduces temporary variables for storing intermediate results to simplify expressions. Figure 5 shows such an example found in the Linux 2.6 kernel.

The pattern can be specified by the SPARQL query shown in Figure 6 with a diagram illustrating the idea. The query sentence specifies a situation where an entity ?a in the original version is moved to the initializer position of a new variable declaration ?decl_ inserted in the later version, and the declared variable ?x_ of ?decl_ is placed under ?f_, which is of the same syntactic category as ?f. Although the last clause does not guarantee that ?x_ substitutes for ?a, the query sufficiently expresses the concept of local variable extraction. Note that

[13]https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/

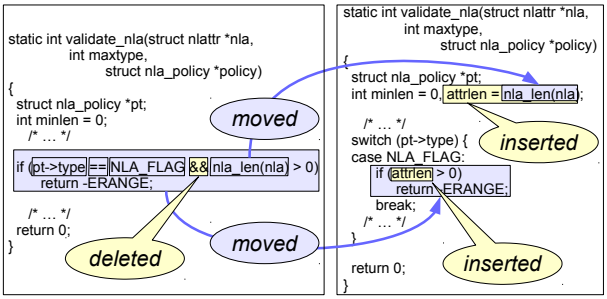

Fig. 7. A Complex Non-Essential Change Found in Linux 2.6 Kernel

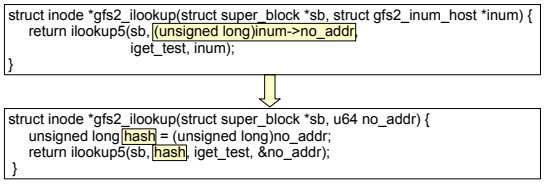

Fig. 8. A Subtle Change in Linux 2.6 Kernel

we can not use chg:graftedOnto for the first chg:insertedInto in the query since chg:graftedOnto specifies addition of a whole subtree.

The query is processed by the RDF store (Virtuoso) populated by more than 4 billion facts in about 12 seconds to generate 365 matched instances. We have manually inspected each one of them and judged 331 correct. The precision rate is about 91%. We only raised a "correct" flag when it is obvious that A is substituted by x' as in Figure 6. However, it is not easy since the difference between two adjacent versions does not always reflect changes occurring in between. Figure 7 shows such a case that is judged correct, where the context of original refactoring goes through a series of complex changes. Thanks to Diff/TS's code tracking abilities, we could still recognize a refactoring instance for this case. By contrast, Figure 8 shows a case that was not judged correct. We may suppose that a function argument was modified after a local variable hash had been introduced. However, all changes occurred in one commit and we could not judge if this was correct or not. There were 21 instances like this in 34 cases that were not judged correct. The remaining 13 were false positives caused by Diff/TS. Since tree differencing is a failure free computational process, it may generate incorrect edit sequences that relate remote entities when the gap between adjacent versions is too large.

The experiment shows that the proposed method can identify fine-grained change patterns not just precisely but also efficiently for a large-scale software system such as the Linux kernel. We have also queried other non-essential changes such as **local variable rename** and **rename induced modification** [4]. The numbers of reported instances are about 30,000 and 7,900 respectively. Due to large numbers, we were unable to check precision rates for these cases. Since the accuracy of AST differencing directly affects these precisions, we plan to evaluate the performance of Diff/TS in a separate experiment.

```
@@
type T; identifier i,fld;
expression E; statement S;
@@
- T i = E->fld;
+ T i;
  ... when != E
      when != i
  if (E == NULL) S
+ i = E->fld;
```

```
-u16 *mateid = mate->id;
+u16 *mateid;
 u8 mask = hwif->ultra_mask;

 if (mate == NULL)
   goto out;
+mateid = mate->id;
```

Fig. 9. A Simple Semantic Patch Example

B. Identifying Bug Fix Patterns

In this section, we present how we can grasp bug fixing changes by querying the factbase. We use an example of semantic patches offered by the Coccinelle tool [8]. Semantic patches are intended to fix the same programming errors at once in different contexts. A semantic patch is written in a tailored language called SmPL and the patch interpreter performs program matching and transformation by following statement level control flows generated by the parser.

Figure 9 shows (in the left box) a simple semantic patch for moving record pointer access from initializing declarations to the subsequent lines of NULL checks[14]. The lines between "`@@`" declare meta variables and the following part specifies defect code patterns and how to modify them. An example of the patch application in a diff format is shown in the right box. The meaning of the patch should be clear except for the part concerning the dots "`...`" and the `when` clauses. Intuitively, the dots are matched by any sequence of code and "`when !=`" indicates patterns that should not occur in the matching code.

Since we use the same C parser module as the Coccinelle tool, which can provide statement level control flows, we only need to introduce a new predicate c:successor whose domain and range are both c:Statement, and modify the parser so that it can export facts about control flows using the predicate. About 60 million CFG facts were added to the factbase at this point.

To identify change patterns implied by the semantic patch, we use a SPARQL query partially shown in Figure 10. The graph pattern in lines 1 to 15 will detect a NULL check ?if for an expression having a name ?ename in a function ?fdef. The pattern in lines 16 to 24 finds an access to a record pointer of a struct having the same name ?ename in an initializing declaration ?decl in the same function ?fdef. Note that we use *SPARQL Property Paths* p/q where $(s, p/q, o)$ means (s, p, s') and (s', q, o) for some s'. The predicate chg:correspondsTo at line 25 indicates that ?decl_ is the next version of ?decl, where chg:correspondsTo is a super property of mapping predicates such as chg:movedTo and chg:mappedTo.

The query sentences in lines 27 to 36 ensure that statements ?decl and ?if are in the same control flow and that the accessed struct and the declared variable do not appear in a statement between them as specified in the original semantic patch. The **FILTER** clause limits solutions to the ones satisfying the condition and the keyword **UNION** specifies alternative matches in the solutions. Although property path expressions $p+$ for specifying transitive closure of p are used in the figure, we actually used an optional function of Virtuoso to exclude loop

[14] http://coccinelle.lip6.fr/impact_linux.php

```
1  ?e a c:Expression ;
2     src:parent ?cond ;
3     c:name ?ename .
4  ?cond c:conditionOf ?if .
5  ?if a c:IfStatement ;
6     c:inFunction ?fdef .
7  { ?cond a c:Eq.
8    ?null a c:Ident ;
9          src:parent ?cond ;
10         c:name "NULL" .
11   FILTER (?ename != "NULL")
12 } UNION {
13   ?cond a c:Negation;
14         src:children (?e) .
15 }
16 ?ini a c:RecordPtrAccess ;
17    c:memberName ?fld ;
18    c:record/c:name ?ename ;
19    src:parent+ ?decl .
20 ?dtor a c:Declarator ;
21       c:name ?vname ;
22       src:parent+ ?decl .
23 ?decl a c:Declaration ;
24       c:inFunction ?fdef ;
25       chg:correspondsTo ?decl_ ;
26       c:successor+ ?if .
27 FILTER NOT EXISTS {
28   { ?x a c:Expression ;
29     c:name ?ename .
30   } UNION {
31     ?x a c:Ident ;
32       c:name ?vname . }
33   ?x c:inStatement ?stmt .
34   ?decl c:successor+ ?stmt .
35   ?stmt c:successor+ ?if .
36 }
37 FILTER NOT EXISTS {
38   ?ini_ a c:RecordPtrAccess ;
39         src:parent+ ?decl_ .
40 }
41 ?decl_ a c:Declaration ;
42       c:inFunction ?fdef_ .
43 ?if_ a c:IfStatement ;
44    c:inFunction ?fdef_ .
45    c:successor+ ?astmt_ .
46 ?if chg:correspondsTo ?if_ .
47 ?astmt_ a c:ExpressionStatement .
48 ?a_ a c:Assign ;
49       c:inStatement ?astmt_ ;
50       c:lhs/c:name ?vname ;
51       c:rhs/c:memberName ?fld;
52       c:record/c:name ?ename .
```

Fig. 10. A Query for Identifying Semantic Patching (Excerpts)

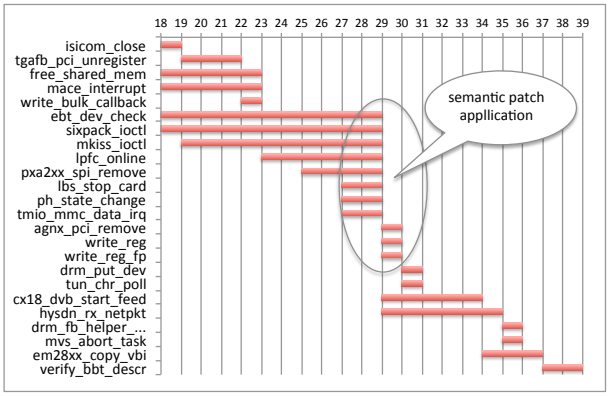

Fig. 11. Tracking Bug Fix Patterns

paths efficiently for the experiment. The **FILTER** clause in lines 37 to 40 indicates that there is no access to ?fld of ?e in ?decl_. Finally, the pattern in lines 43 to 52 ensures that the deferred assignment of the record access to a variable named ?vname follows the next version of ?if.

Virtuoso took about 90 minutes to report 24 change instances over versions from 2.6.18 through 2.6.39. Figure 11 shows traces of the defects up to the point of modification identified by the query shown in Figure 10. Each bar represents a lifetime of the defective code. The vertical axis represents defect containing functions and the horizontal axis represents minor version numbers. Note that in this particular example, no function contains multiple defects and a function determines the defective code.

It has been reported that the modification made by a semantic patch in Figure 9 was adopted in the official kernel source[14]. In fact, change instances found by the query in Figure 10 contain all such cases except for the ones that were introduced and fixed between release versions. They are marked by a circle in the figure.

It appears that the same defect was introduced several times before and after the semantic patch application, which may have been fixed in a traditional way. By slightly modifying the query, we found that 27 instances of the same defect remained uncorrected until 2.6.39, all of which were introduced after 2.6.33. We also found false positives, that is, the code segments that match the specified pattern but do not need to be modified at all. These were caused by imprecise control flows computed by the parser and a simplistic code pattern specification in the semantic patch.

Now we understand that maintaining a generic patch set is a difficult task. The same defect keeps getting introduced over and over and the code pattern that works fine with the current version may not work or even be wrong for the later versions. We believe that our query based approach helps to evolve a patch set according to source code changes. For this, we may have to develop a tool to translate code excerpts into SPARQL sentences automatically as queries often get too verbose.

C. BKL Pushdown in Linux Kernel 2.6

In this section, we explain our attempt at comprehending so-called BKL pushdown, which is a community wide effort to purge an obsolete locking mechanism from the entire Linux kernel tree.

The Big Kernel Lock (BKL) was introduced to make Linux run on symmetric multiprocessor systems by allowing only one processor to run kernel code at any given time. It had many non-traditional features as a global spinlock, which made easier the transition from kernel version 2.0 to 2.2. However, it became a major overhead and various light-weight locks were added to the kernel to replace the BKL. The BKL pushdown is the final maneuver toward the purge, which was started before the release of the version 2.6.27. It aimed to push the acquisition of the BKL down to the lower-level driver code so that BKL calls could be audited independently. Eventually, such pushed down calls would be deleted or replaced by calls to more efficient local locks.

The BKL pushdown is characterized by three steps of pairwise insertion and deletion of lock/unlock calls explained below. To abstract descriptions from the BKL, we assume a function pair, tic() and tac().

- **PUSHDOWN**. For each pair of tic/tac calls in upstream kernel areas, the same pair of calls is inserted in (many different) functions, in downstream subsystem directories, that are reachable from a function, say foo(), called between the original calls to tic() and tac().
- **DELETE**. The original calls to tic() and tac() are deleted upstream when it is safe to do so.
- **PURGE**. The inserted pairs of tic/tac calls are deleted or replaced by calls to other lock functions, say tic'() and tac'().

Figure 12 illustrates the situation. It is only assumed that the PUSHDOWN step precedes the steps DELETE and PURGE.

Our aim here is to demonstrate how we can detect such global refactoring efforts without knowing technical details of the event. First of all, we need facts about function call relations since the term "reachable" above clearly refers to

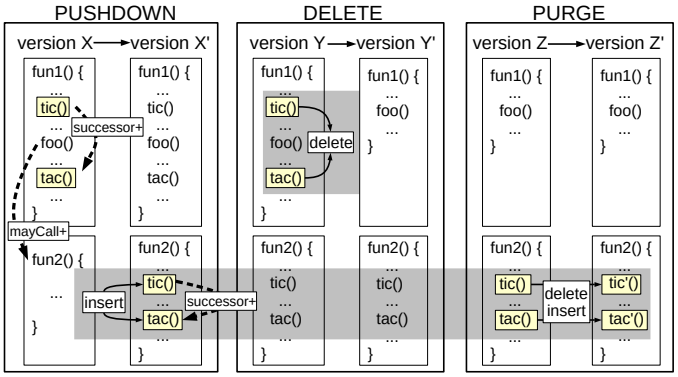

Fig. 12. BKL Pushdown Steps

it. However, creating precise call graphs for the entire Linux kernel code is demanding due to abundant use of function pointers and struct instances having function pointer members passed as function arguments.

After surveying available tools, we chose a compiler tool called ncc[4]. Although the tool does not appear well maintained, it worked fine after we fixed several serious bugs such as null pointer dereferences and made modifications for GCC extensions. Ncc works on ASTs created by its own C parser to resolve static values of function pointers. It relies on pre-defined value propagation patterns and has a simple but effective symbolic evaluator. It is well tested against early versions of Linux 2.6 and works remarkably well for the later versions including 3.x.

Since ncc relies on GCC for preprocessing C source files, we had to actually build the whole kernel for each version. For this, we used virtual machines to built the kernel in the original old environments and created call graphs in full length.[15] To represent facts of call relations, we introduced a new predicate ncc:mayCall whose domain and range are both c:FunctionDefinition. Since call graphs generated by ncc refer to a function by its name, line range, and file location, we had to match the function name, the line range, and the file location with those of a c:FunctionDefinition entity identified by Diff/TS for integrating facts. Overall, we added approximately 13 millions of call graph facts in the factbase.

We start analysis by creating what we call a *code continuum* [6] for each potential callee function. A code continuum of a named entity e such as a function definition is a list of pairs (v, n) such that e has name n in version v. A code continuum is constructed by tracing mapping of source code entities between adjacent versions created by Diff/TS as explained in Section II-A. Also in the process, we check if entity references such as function calls are renamed consistently to create accurate continuum as much as possible. This is necessary for tracing renamed functions.

Firstly, we identify groups of functions, calls to which are inserted simultaneously in the same functions in subsystem directories by querying the factbase. We chose drivers and sound sub-directories for this. At the same time, we use the git-blame command to identify commits of insertion by specifying line ranges of inserted function calls. From each group, we extract pairs of functions that are in the same control flows of c:successor and that will be deleted or replaced in the later versions (the PURGE step). These function pairs comprise pushdown candidates such as tic() and tac() in Figure 12. Let us call them the PUSHDOWN pairs. We use SPARQL query for extracting function pairs and the git-log command to check whether the pairs are deleted/replaced in later commits. A commit of the PURGE step is identified for each PUSHDOWN pair at this point. The reason for using Git commands is not only to identify commits of changes but also to explore hybrid analysis involving both factbases and repositories. The identified commits are used to evaluate the correctness of reported instances.

Secondly, we compute the DELETE pairs of functions, calls to which are connected by c:successor and deleted simultaneously in the same functions in directories other than drivers and sound, by querying the factbase. We also use the git-log command to locate commits of the DELETE step for evaluation purpose.

Finally, we compare the PUSHDOWN pairs and the DELETE pairs to determine overall candidates for the BKL function pairs. We do this by, for each pair of functions contained in both PUSHDOWN and DELETE, checking if their call site functions, such as fun1() and fun2() in Figure 12, are reachable by function calls. Note that we have approximated the PUSHDOWN step as we only check reachability between functions instead of reachability from inner calls, such as a call to foo() in Figure 12. This is because ncc does not provide any information about call sites, which is difficult to recover when function pointers are present. It implies a risk of overestimating candidate pairs.

This part is done by scripting SPARQL queries for ncc:mayCall+ reachability over common elements of the PUSHDOWN pairs and the DELETE pairs. We use Python for this by requiring the following conditions:

- The PUSHDOWN step precedes the DELETE step.
- The call sites are reachable within or equal to 10 hops.
- For each candidate pair, the number of files in which the calls to the pairs are inserted (in the PUSHDOWN step) is larger than the number of files in which the calls are deleted (in the DELETE step).

The second condition is for efficiency since reachability checks take much time. The third condition reflects an intuition that the number of calls increases as a global lock gets pushed down to a driver directory.

We executed the overall procedure against the entire factbase. It took 19 hours, including 10 hours of git-log execution for detecting commits of deletion, to compute a list of overall candidate pairs of functions shown in the upper part of Table II. Note that the "Matches" columun contains the numbers of times the call site function for the candidate in the DELETE pairs (such as fun1()) reaches the call site function for the same candidate in the PUSHDOWN pairs (such as fun2()).

The BKL pair of lock_kernel and unlock_kernel is by far the strongest candidate for the refactoring pattern illustrated in

[15] Although releases from 2.6.12 through 2.6.39 are available from the repository, we dropped the earliest 6 releases due to the lack of time for preparing very old environments.

TABLE II. PUSHDOWN CANDIDATES

Candidate Pair	Matches	Del Files	Ins Files
lock_kernel – unlock_kernel (BKL)	3,646	266	323
spin_unlock_irq – spin_lock_irq	34	19	28
spin_unlock_irqrestore – spin_lock_irqsave	9	28	44
mutex_lock_interruptible – mutex_unlock	5	13	23
down – up	5	11	22
unlikely – dev_kfree_skb_any	3	3	10
clear_bit – test_bit	2	3	12
spin_lock_irq – spin_unlock_irq	2	3	9
kfree – mutex_unlock	1	8	14
spin_lock – spin_unlock	NA	501	82
mutex_lock – mutex_unlock	NA	294	184
spin_lock_irqsave – spin_unlock_irqrestore	NA	164	135
rcu_read_lock – rcu_read_unlock	NA	138	2

TABLE III. PRECISION OF BKL PUSHDOWN ANALYSIS

	PUSHDOWN	DELETE	PURGE
Commit #	113	94	87
Prec. Rate	94.69%	93.62%	89.53%

Figure 12. Other pairs are weak not only in numbers but also in correspondence. For instance, the pair of spin_unlock_irq and spin_lock_irq appears in the opposite order, which is caused by approximation we made with the reachability in the PUSHDOWN step. The lower part of Table II shows the pairs that were dropped because of the file number condition.

Let us concentrate on the BKL pair and check if the points of insertion and deletion we found during the course of the above analysis are correct. We do this by examining the corresponding commit logs. We assume that the analysis is "correct" if the commit log contains either word "BKL" or "pushdown" and the associated diff file has lines indicating that BKL functions are added or deleted by the first character of a line ("+" or "-"). Let us call these the BKL pushdown commits.

There are 263 commits identified for the three steps of PUSHDOWN, DELETE and PURGE. Table III shows precision rates for each. We verified that false positive are caused by modifications for adjusting BKL calls rather than for pushing them down such as bug fixes and refactorings. By inspecting commit logs, we confirmed that there are 583 BKL pushdown commits in the repository, which means that the overall recall rates is rather disappointing 45.11%. However, manual inspection revealed that there are 150 commits performing pushdown in other directories than we assumed such as net, arch, kernel, fs and block, and there are 22 commits apparently related to BKL pushdown but only remotely, such as bug fixes and preparations for BKL pushdown. This means that the recall rate could improve to 63.99%.

We examined remaining 148 false negatives. Main causes include: 1) insertion or deletion of cycle_kernel_lock (36 times) instead of BKL pairs, 2) failure of call graph generation by ncc, 3) merged branches that are unreachable from the relevant release versions, and 4) deleted files that contained BKL calls. Note that cycle_kernel_lock only calls lock_kernel and unlock_kernel for serialization purposes. As for the last, we did not enable factbase enrichment in Section II-C for function calls to keep the moderate number of facts.

We list additional information here. The average time difference between the PUSHDOWN step and the DELETE steps is 527 days. The function pair that replaced the BKL pair most is mutex_lock/mutex_unlock followed by tty_lock/tty_unlock and spin_lock_irqsave/spin_unlock_irqrestore. The average hop count between call site functions of the common PUSHDOWN and DELETE pairs is 7.65. The code continuum we constructed is utilized for detecting renaming of lock_kernel or _lock_kernel.

IV. RELATED WORK

We list below fundamental studies related to the proposed method. We omit non-essential change identification [4] as it is explained in Section III.

There are studies for extracting facts about source code in mathematical form to process queries based on logical frameworks. Holt proposed a system called Grok using binary relational calculus for manipulating graph models about software architectures [11]. Hajiyev and others proposed a system called CodeQuest based on Datalog for processing recursive queries about software properties [12]. Both systems concentrate on properties within the same versions of software and do not help fine-grained change analysis as we do. Since the Semantic Web technologies are based on description logic, our method implicitly enjoys benefits of formal systems.

Xing and Stroulia explored factbase approaches to refactoring reconstruction [1]. They use a tool called JDevAn to collect evolution facts from Java projects by matching entities such as packages, classes, interfaces, fields and blocks based on similarity in names and structures. A tool called UMLDiff which infers difference between facts of a given pair of revisions is used to detect refactoring patterns.

Prete and others report a refactoring reconstruction tool called Ref-Finder [2]. Ref-Finder starts analysis by extracting facts from given revisions using logical predicates to infer difference between revisions in the form of logical assertions and applies predefined template logic rules to identify refactorings.

In both cases of JDevAn and Ref-Finder, information below the level of statements are lost during the process of fact extraction and hence changes concerning local variables and program-controls can not be easily detected. Their methods are also based on similarities and thresholds and prone to false positives caused by renaming. Our method differs by directly differencing ASTs rather than summarized facts.

ChangeDistiller [3] classifies Java source code changes according to predefined taxonomy such as function parameter addition and variable type changes, for better understanding of evolving natures of software. It uses a semi-optimal tree differencing algorithm on ASTs up to the level of program statements and switches to textual similarity for comparing statements for the sake of efficiency. Our approach differs by employing a full-scale tree differencing tool Diff/TS [5], based on an optimal algorithm, and by accepting ASTs of other programming languages than Java such as C, C++, Python, Verilog and Fortran.

There exist researches that define software ontologies for the purpose of data exchange among tools [13], [14], [15]. Our method focuses on ontologies for source code changes at the low level of ASTs, which can be incorporated into high level ontologies proposed before.

There are many attempts at creating and sharing ontologies for describing software projects as well as software

engineering projects. DOAP (Description of a Project)[16] is a project to create an RDF Schema and an XML vocabulary to describe open source software projects. It aims to facilitate sharing information about software projects. Welty presents an early attempt at aiding maintainer's low-level discovery efforts through ontologies for representing source code information based on AST [16]. Our work can be seen as a modern continuation of his work with advanced capabilities of change analysis.

V. Conclusion

We have proposed a comprehensive method for analyzing fine-grained source code changes by way of factbase queries. Enhanced capabilities of our AST differencing tool helps us to relate facts given by other analysis tools across versions and to retrieve instances of detailed change patterns. The use of standard Semantic Web technologies and the definition of ontologies for AST level source code changes helped us to simplify the process of fact integration and query processing.

To illustrate the capability of the method, we have presented experiments of fine-grained change pattern identification conducted for the Linux-2.6 kernel source code. The analysis involves more than 4 billions of RDF triples about AST changes, control flows, and call relations from release 2.6.18 through 2.6.39. The result of non-essential change identification [4] and context sensitive code patching [8] showed that the proposed method identifies fine-grained change patterns in a precise and efficient manner. We have described analysis of the so-called BKL pushdown, a community wide effort to erase the obsolete locking mechanism from the entire kernel tree. We have specified the pushdown pattern using facts about control flows and call relations to successfully identify change instances.

The proposed method allows users to concentrate on analysis of facts rather than generation of facts. As long as users understand the concept of changes and the ontologies, they can work entirely with queries to accomplish complex analysis as in the examples. We hope this opens the door to "query engineering" for software analysis, where queries are shared and reused across boundaries of software projects and even programming languages. For example, the query for local variable extraction in Figure 6 can be easily adapted to Java programs.

Since the factbase used in the paper occupies moderate 260 gigabytes, the proposed method may benefit a large number of people by sharing factbases. We are planning on a project that makes factbases and queries available for promoting collaborative work in software change analysis. We hope that the difficulties in using the SPARQL language will be mitigated in the process of collaboration.

We list some of the limitations of the method here. First, queries relying on facts about call relations and control flows may produce imprecise results since the factbase contains only static information of the program. Second, ontology development is a slow process as we note that changes to core ontologies will force a large amount of rework. Third, using SPARQL is not easy. It is not yet popular in software engineering even though it is a well-established standard Web technology. We need to create more incentives to learn SPARQL for collaborative work in software analysis. Fourth, not all changes are recorded in the factbase as we only work on release versions of Linux. We plan to look into Git to obtain commit level facts including verbal information such as commit logs.

[16] https://github.com/edumbill/doap/wiki

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 26280025.

References

[1] Z. Xing and E. Stroulia, "Refactoring detection based on UMLDiff change-facts queries," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*. IEEE Computer Society, 2006, pp. 263–274.

[2] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM'10)*, 2010, pp. 1–10.

[3] B. Fluri, M. Würsch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, November 2007.

[4] 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.

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

[6] ——, "A method for analyzing code homology in genealogy of evolving software," in *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE2010)*, ser. Lecture Notes in Computer Science, vol. 6013, Mar 2010, pp. 91–106.

[7] ——, "Enhancing history-based concern mining with fine-grained change analysis," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR2012)*, Mar 2012, pp. 75–84.

[8] Y. Padioleau, J. L. Lawall, and G. Muller, "Understanding collateral evolution in Linux device drivers," in *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. ACM, 2006, pp. 59–71.

[9] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM J. Comput.*, vol. 18, no. 6, pp. 1245–1262, 1989.

[10] P. Bille, "A survey on tree edit distance and related problems," *Theor. Comput. Sci.*, vol. 337, no. 1-3, pp. 217–239, Jun. 2005.

[11] R. C. Holt, "Structural manipulations of software architecture using Tarski relational algebra," in *WCRE*, 1998, pp. 210–219.

[12] E. Hajiyev, M. Verbaere, and O. D. Moor, "CodeQuest: Scalable source code queries with Datalog," in *In ECOOP Proceedings*. Springer, 2006, pp. 2–27.

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

[14] Y. Zhang, R. Witte, J. Rilling, and V. Haarslev, "Ontology-based program comprehension tool supporting website architectural evolution," in *Proceedings of the Eighth IEEE International Symposium on Web Site Evolution (WSE'06)*. IEEE Computer Society, 2006, pp. 41–49.

[15] M. Würsch, G. Ghezzi, M. Hert, G. Reif, and H. Gall, "SEON: A pyramid of ontologies for software evolution and its applications," *Computing*, vol. 94, no. 11, pp. 857–885, 2012.

[16] C. Welty, "Augmenting abstract syntax trees for program understanding," in *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*, Nov 1997, pp. 126–133.