Fine-grained Processing of CVS Archives with APFEL

Thomas Zimmermann
Department of Computer Science
Saarland University, Saarbrücken, Germany
tz@acm.org

ABSTRACT
In this paper, we present the APFEL plug-in that collects fine-grained changes from version archives in a database. APFEL is built upon the Eclipse infrastructure for CVS and Java. In order to describe changes, APFEL uses tokens such as method calls, exceptions, and variable usages. We demonstrate the usefulness of APFEL's database with several case studies.

Categories and Subject Descriptors
D.2.6 **[Software Engineering]:** Programming Environments – *Integrated environments*; D.2.7 **[Software Engineering]:** Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering*, *Version control*; D.2.9 **[Software Engineering]:** Management – *Software configuration management.*

General Terms
Management, Measurement, Experimentation.

Keywords
Changes, Tokens, Abstract Syntax Trees, Eclipse, CVS

1. INTRODUCTION
Nowadays, software development produces a huge amount of information: changes to source code are recorded in version archives, bugs are reported to problem databases, and development is discussed in mailing lists and newsgroups. Recently, a new research area called *mining software repositories* has emerged. It showed that historical data is a valuable asset when it comes to understanding change tasks [6], guiding programmers [23, 26], and identifying logical coupling [10] of huge software systems.

The Eclipse project was involved in this research from its first day. Mainly for two reasons: (1) The development process of Eclipse is well documented and organized; Eclipse quickly became one of the most popular evaluation subjects. (2) The Eclipse platform offered functionality that is required for mining software repositories, such as CVS access, a Java parser, and an easy way to demonstrate results to the user.

OOPSLA Eclipse Technology Exchange October 22 -23, 2006, Portland, Oregon, USA

Hipikat [6] is a good example of a tool that leverages software repositories in Eclipse. Hipikat connects all kinds of software artifacts (documentation, bug reports, changes) and allows searching for related artifacts. Another tool is eROSE [26] that mines CVS archives to create recommendations of the form "Programmers who changed function `f()` also changed `g()`."

Hipikat and eROSE both stop at the artifact or element level. However, more fine-grained changes—such the addition of method calls—contain also valuable information. Livshits and Zimmermann mined the addition of method calls for usage patterns [17] and Breu and Zimmermann identified cross-cutting concerns in history [4].

In this paper, we present the APFEL plug-in for Eclipse[1]. APFEL processes CVS archives and stores fine-grained changes such as the addition of method calls in a database. It greatly benefits from the Eclipse platform by using the CVS plug-in to access CVS archives and by parsing Java files with the JDT parser.

After giving an overview of related work (Section 2), we describe in general how APFEL computes fine-grained changes using tokens (Section 3). Next we shed more light on the concept of tokens (Section 4), before we discuss how individual changes are computed (Section 5). We then present several case studies to demonstrate the usefulness of APFEL (Section 6). We conclude the paper with ideas for future work (Section 7).

2. RELATED WORK
Most work on preprocessing version archives covers problems specific to CVS such as mirroring CVS archives [7, 25], reconstructing transactions [11, 25], reducing noise and finding out the locations (methods) that changed [9, 25]. The Kenyon tool combines these techniques in one framework; it is frequently used for software evolution research [1].

Previous research in the area of mining software repositories investigated the *location* of a change—such as files [2], classes [3, 7], or methods [24]—and *properties* of changes—such as number of lines changed, developers, or whether a change is a fix [18].

Recently, the focus shifted from locations to changes themselves: Kim et al. identified signature change patterns in version histories [15], Williams and Hollingsworth [22] and Livshits and Zimmermann [17] mined usage patterns from co-added method calls, and Breu and Zimmermann identified cross-cutting concerns [4]. Fluri and Gall classified fine-grained changes [8]. Finding out what was changed is an instance of the program element matching problem that has been surveyed by Kim and Notkin [13].

[1] APFEL is German for *apple* and short for "A Preprocessing Framework for Eclipse (and CVS)"

Comparing abstract syntax trees is one approach to compute fine-grained changes, however it is complicated and time-consuming [19, 20]. In contrast, APFEL represents the syntactic content of elements (e.g., methods) with tokensets, thus neglecting the order within an element. This token-based approach is motivated by the research of Li and Zhou [16] who inferred implicit programming rules based on method call and variable type tokens, however not on changes, but on a single snapshot of a program. They identified several violations of these rules that turned out to be defects.

As mentioned in the introduction, two plug-ins for Eclipse that leverage software repositories are Hipikat [6] and eROSE [26]. Both could benefit from fine-grained changes as computed by APFEL: Hipikat could link artifacts to syntactic elements such as method calls and eROSE could make additional recommendations how to change a code location rather than just providing the location itself.

3. FINE-GRAINED CHANGES

The APFEL plug-in investigates fine-grained changes at the level of *tokens*. A token represents some syntactic content of an element. Table 1 shows that APFEL distinguishes between different kinds of tokens: For methods, it captures method calls, variable usages, and exception handling; for classes, it captures inheritance relations; for compilation units, it captures imported classes.

Using tokens, it is straightforward to compute fine-grained changes between two revisions r_1 and r_2 (see Figure 1). First, we represent each element of revision r_1 as a multiset of tokens; we do the same for the elements of revision r_2. Finally, we compare the multisets of matching elements. As a result we get differences such as in method `b()` one call to method `foo()` was deleted and one call to method `bar()` was inserted. Other possible changes that we can detect are "two usages of `String` variables were deleted" and "one throw statement for `EmptyStackException` was added".

4. Tokenizing Source Code

In this section, we will describe tokens more in detail. In APFEL, every token consists of a type, name, context, and instance.

Type. The type of a token describes, what kind of syntactic content it captures. Examples are method calls, variable usages, and keywords.

Name. The name of a token contains the syntactic content, e.g., for a method call token, the name of the method that is called.

Context. Some tokens are connected to syntactic elements. For instance, in `obj.foo()` the method `foo()` is called on the object `obj`. Another example are JavaDoc comments: they are typically linked to the succeeding Java element.

Instance. For method calls, APFEL captures the number of arguments as the token instance.

In order to collect tokens, APFEL traverses abstract syntax trees and creates for every compilation unit, class, and method a separate tokenset. These sets are organized hierarchically: for instance, the tokenset of a class is the union of its methods' tokensets plus additional class-specific tokens.

Section 9 in the appendix contains a full list of tokens supported by APFEL. We use the following syntax for tokens:

Table 1. Different kinds of tokens.

Token type	For what?	What is captured?
Modifier	Modifier	`public, private, final, …`
Call	method call	method name and signature
Name	variable usage	variable name
Type	variable usage	variable type
Throws	method declaration	thrown exception
Throw	throw statement	thrown exception
Catch	catch expression	caught exception
Keyword	Keywords	`if, for, while, …`
Extends	type declaration	extended type
Implements	type declaration	implemented interface
Import	import statement	imported class/package

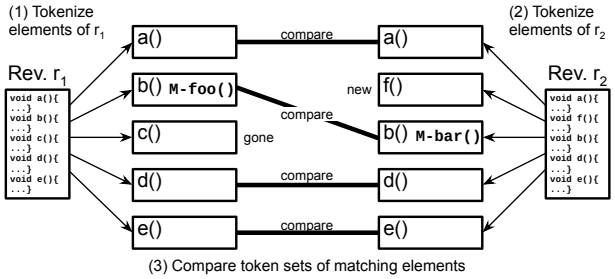

Figure 1. Comparing two revisions r_1 and r_2 of a file.

context.**type–name** instance

As an example take the token "Plugin.+.**M–findMember**(1)": the context is "Plugin.+", the type is method call "M", the name is "findMember" and the instance is "(1)".

Since APFEL works only on syntactic information, it cannot resolve the signatures for the methods that are called. As an approximation, APFEL uses the number of arguments, e.g., `findMember(1)` for `findMember(fullPath)`.

For the context `plug.getWorkspace().getRoot()` of the method call `findMember(fullPath)`, APFEL resolves the type of `plug` (if possible), and summarizes the two method calls `getWorkspace()` and `getRoot()` by a plus character, resulting in the APFEL context `Plugin.+`. This helps to identify the class on which the method is called; in the presence of the plus character the class is unknown.

APFEL distinguishes between intermediate calls and the final call of a sequence. In the above example, `getWorkspace()` and `getRoot()` are intermediate calls (F-tokens), and `findMember(1)` is the final call (M-token).

5. Comparing Tokensets

When comparing tokensets for an element, we distinguish between different *types of changes*:

Modification of an element (CHG). The element exists in both revisions with the tokensets T_{old} and T_{new}. We compute the added tokens with $T_{new}–T_{old}$ (stored with positive counts in the database) and the deleted tokens $T_{old}–T_{new}$ (stored with negative counts).

Addition of an element (ADD). The element exists only in the newer revision. All tokens T_{new} are inserted into the database (with positive counts).

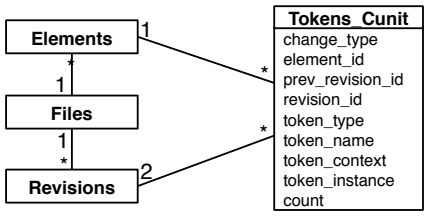

Figure 2. Database schema on compilation unit level.

Deletion of an element (DEL). The element exists only in the old revision. All tokens T_{old} are inserted into the database (with positive counts).

5.1 Database Schema

APFEL stores all token changes into a database. For every level one table is created. Figure 2 shows a simplified database schema for compilation units (_Cunit). For every token change in *Tokens_Cunit*, we store the *change_type* (CHG, ADD, DEL) of the surrounding element. The *Element* is referenced with *element_id* and the old and new *Revision* with *prev_revision_id* and *revision_id* respectively. The token itself is described as in Section 4 with *token_type*, *token_name*, *token_context and token_instance*. The field named *count* shows how often the token was added *(count>0)* or deleted *(count<0)*.

5.2 Limitations

In this section we discuss limitations of APFEL, in particular of the lightweight parsing approach:

No renaming. When an element is renamed, this is recognized as two changes: a deletion of the old element and an addition of the new element. Origin analysis can recognize such renaming [12, 14, 21]; we plan to build our own origin analysis based on the similarity of tokensets T_{old} and T_{new}.

Canceling changes. Since APFEL neglects the order within methods, APFEL may miss changes because they are canceling themselves. An example is swapping two lines within a method. However, moving code from one method to another is canceling on class level, but not on method level.

Method signatures. In order to keep the processing of version archives lightweight, APFEL parses only one source file at a time. Considering snapshots—a version of the whole program that compiles—is too expensive. As a result, APFEL cannot resolve the signature of called methods and approximates it instead with the number of arguments. In every second case, the approximated signature directly identifies the original method; however, for some methods such as `dispose` or `visit` there are over 2,000 possible candidates. The overall precision is 68.4% for Eclipse, the number of arguments accounts for 4.4% points. We obtained similar precision values for other open source projects.

6. CASE STUDIES

In this section we present three small case studies. All of them can be realized with one SQL query on the APFEL database. For Eclipse, the database contains in total 25,848,371 token changes on method level, 11,670,183 on class level, and 12,038,328 on compilation unit (file) level. The total size of the database is approximately 3.6GB for 5 years development history, enclosing 97,996 transactions and 423,991 checkins. Parsing of pre-fetched Java files takes about 12 hours and pre-fetching several days.

6.1 Crosscutting Concerns

A crosscutting concern is functionality that does not align with the given modularization of a program, thus, ending up scattered across the program. If such functionality exists, it must have been added in the history. We can use the APFEL database to identify crosscutting/scattered changes with a simple SQL query.

```
SELECT token_name, COUNT(DISTINCT element_id)
FROM cvs_tokens_method NATURAL JOIN cvs_revisions
WHERE token_type='M' AND change_type='CHG'
GROUP BY transaction_id, token_name
ORDER BY COUNT(DISTINCT element_id) DESC;
```

We create groups for every added/deleted method call (M-token) within a transaction. Then we count the number of distinct elements that contain this token. The more elements the more crosscutting a change is. Here are the first five rows returned by the query.

```
  token_name   | count
---------------+-------
 getString     |  1462
 lock          |  1284
 unlock        |  1284
 error         |   996
 isValidWidget |   988
```

And indeed we found crosscutting concerns. The methods `lock` and `unlock` handle locking for 1284 code locations. All calls were inserted within one transaction "76595 (new lock)". The calls to `isValidWidget` and `error` check whether a widget is disposed. Breu et al. used the observation that *cross-cutting emerges over time* to identify aspect candidates with concept analysis [4, 5].

6.2 Pairs of Variable Names

Version archives have been used to identify usage patterns of methods that describe which methods should be called together [17, 22]. With the APFEL database, we check whether such patterns also exist for variable names: we identify pairs of names that are frequently inserted (used) together. If such pairs exist, they could improve the recommendation of variable names.

Here are the variables that are most frequently inserted together in Eclipse (we ignored variable names with a single character):

```
 variable_a  |   variable_b    | count
-------------+-----------------+-------
 height      | width           |   720
 bCodeStream | classFileOffset |   457
 end         | start           |   431
 DEBUG       | position        |   254
 length      | offset          |   194
 buffer      | length          |   168
```

The first row means that `height` and `width` have been inserted together into 720 methods. In the second row, names `bCodeStream` and `classFileOffset` are not related by name; however, they are frequently used together in source code:

```
classFileOffset + 2 >= bCodeStream.length
```

In total we identified 3,367 pairs with a minimum count of 10.

6.3 Renaming of Variables

With the refactoring support of state-of-the-art IDEs, developers frequently rename entities, especially variables. With the APFEL database we can find evidence for this hypothesis. We search for changes where one variable name was deleted and another one inserted with the same number of occurrences. Additionally, we consider only changes that exclusively touched variable names.

On method level we get the following results (sorted descending by number of references *refs* to the variable):

old_name	new_name	refs
trueTb	valueIfTrueType	41
falseTb	valueIfFalseType	36
key	accelerator	28
keyBindingDefinition	keySequenceBindingDefinition	24
tab	item	20
endAngle	arcAngle	19
bundledata	bundleData	18
contentAssistant	fContentAssistant	18

In total, we identified 543 changes that renamed variables with at least five references.

7. CONCLUSIONS

In this paper we presented the APFEL plug-in. We showed how to compute fine-grained changed with tokensets. Additionally, we presented case studies to demonstrate the usefulness of APFEL.

We will continue to improve APFEL. Currently, we are working on the following topics.

More tokens. Since extending APFEL with new tokens is not difficult, we are planning to support control-flow changes, such as changes in `switch` cases or the conditions of `if` statements. Right now they are collected, but not marked explicitly.

Incremental processing. Every time APFEL processes a CVS repository it recreates the database. In order to save computation time and network traffic, we a working on an incremental version.

Headless operation. The current APFEL version requires the Eclipse IDE to be running. We are working on a headless version; this way APFEL can be integrated into the CVS commit process.

For more information on APFEL logon to

http://www.st.cs.uni-sb.de/softevo/

Acknowledgements. Thomas Zimmermann is funded by the DFG Graduiertenkolleg “Leistungsgarantien für Rechnersysteme”. Thanks to Valentin Dallmeier, Melih Demir, Thomas LaToza, Daniel Schreck, and Markus Thiele for dogfooding APFEL.

8. REFERENCES

[1] Bevan, J., E. James Whitehead, J., Kim, S. and Godfrey, M., Facilitating software evolution research with Kenyon. In *European Software Engineering Conference/International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Lisbon, Portugal, 2005.

[2] Bevan, J. and Whitehead, J., Identification of Software Instabilities. In *Proc. 10th Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, 2003, 134-143.

[3] Bieman, J.M., Andrews, A.A. and Yang, H.J., Understanding Change-proneness in OO Software through Visualization. In *Proc. 11th International Workshop on Program Comprehension*, Portland, Oregon, 2003, 44-53.

[4] Breu, S. and Zimmermann, T., Mining Aspects from Version History. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, Tokyo, Japan, 2006.

[5] Breu, S., Zimmermann, T. and Lindig, C., Mining eclipse for cross-cutting concerns. In *Proceedings of the 2006 international workshop on Mining software repositories*, Shanghai, China, 2006.

[6] Cubranic, D., Murphy, G.C., Singer, J. and Booth, K.S. Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering*, *31*(6). 446-465.

[7] Fischer, M., Pinzger, M. and Gall, H., Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, 2003.

[8] Fluri, B. and Gall, H.C., Classifying Change Types for Qualifying Change Couplings. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC)*, Athens, Greece, 2006, 35-45.

[9] Fluri, B., Gall, H.C. and Pinzger, M., Fine-Grained Analysis of Change Couplings. In *IEEE 5th International Workshop on Source Code Analysis and Manipulation (SCAM)*, Budapest, Hungary, 2005, 66-74.

[10] Gall, H., Jazayeri, M. and Krajewski, J., CVS Release History Data for Detecting Logical Couplings. In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, Helsinki, Finland, 2003, 13-23.

[11] German, D., Mining CVS repositories, the softChange experience. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, Edinburgh, Scotland, UK, 2004, 17-21.

[12] Godfrey, M.W. and Zou, L. Using Origin Analysis to Detect Merging and Splitting of Source Code Entities. *IEEE Trans. Software Engineering*, *31*(2). 166-181.

[13] Kim, M. and Notkin, D., Program element matching for multi-version program analyses. In *Proceedings of the 2006 international workshop on Mining software repositories*, Shanghai, China, 2006.

[14] Kim, S., Pan, K. and Jr., E.J.W., When Functions Change Their Names: Automatic Detection of Origin Relationships. In *12th Working Conference on Reverse Engineering (WCRE 2005)*, Pittsburgh, PA, USA, 2005, 143-152.

[15] Kim, S., Whitehead, E.J. and Bevan, J., Analysis of signature change patterns. In *Proceedings of the 2005 international workshop on Mining software repositories*, St. Louis, Missouri, 2005.

[16] Li, Z. and Zhou, Y., PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *European Software Engineering Conference/International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Lisbon, Portugal, 2005.

[17] Livshits, V.B. and Zimmermann, T., DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *European Software Engineering Conference/International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Lisbon, Portugal, 2005.

[18] Mockus, A. and Weiss, D.M. Predicting risk of software changes. *Bell Labs Technical Journal*, *5*(2). 169-180.

[19] Neamtiu, I., Foster, J.S. and Hicks, M., Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, St. Louis, Missouri, 2005.

[20] Sager, T., Bernstein, A., Pinzger, M. and Kiefer, C., Detecting similar Java classes using tree algorithms. In *Proceedings of the 2006 international workshop on Mining software repositories*, Shanghai, China, 2006.

[21] Weißgerber, P. and Diehl, S., Identifying Refactorings from Source-Code Changes. In *International Conference on Automated Software Engineering (ASE 2006)*, Tokyo, Japan, 2006.

[22] Williams, C.C. and Hollingsworth, J.K., Recovering system specific rules from software repositories. In *Proceedings of the 2005 international workshop on Mining software repositories*, St. Louis, Missouri, 2005.

[23] Ying, A.T.T., Murphy, G.C., Ng, R. and Chu-Carroll, M.C. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering*, *30*(9). 574-586.

[24] Zimmermann, T., Diehl, S. and Zeller, A., How History Justifies System Architecture (or not). In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, Helsinki, Finland, 2003, 73-83.

[25] Zimmermann, T. and Weißgerber, P., Preprocessing CVS Data for Fine-Grained Analysis. In *Proc. Intl. Workshop on Mining Software Repositories (MSR)*, Edinburgh, Scotland, 2004.

[26] Zimmermann, T., Weißgerber, P., Diehl, S. and Zeller, A. Mining Version Histories to Guide Software Changes. *IEEE Trans. Software Engineering*, *31*(6). 429-445.

9. APPENDIX: OVERVIEW OF TOKENS WITH EXAMPLES

		Example	
	Syntax	**Source Code**	**Tokens**
Modularization			
Package declaration Import statement	**Q–***package-name* **I–***qualified-name*	`package org.eclipse.compare;` `import java.util.ResourceBundle;`	**Q–org.eclipse.compare** **I–java.utilResourceBundle**
Inheritance			

Extension
Implementation
	E–*class-name*
	C–*interface-name*
		`class ResourceNode`
		`  extends BufferedContent`
		`  implements ITypedElement`
			E–BufferedContent
			C–ITypedElement

Method calls			

Last method call in a sequence
Intermediate method calls
	M–*method-name*
	F–*method-name*
		`Plugin plug = ...`
		`member = plug.getWorkspace()`
		`  .getRoot().findMember(fullPath);`
		`return getName().hashCode();`
			Plugin.**F–getWorkspace**(0)
			Plugin.+.**F–getRoot**(0)
			Plugin.+.**M–findMember**(1)
			F–getName(0)
			+. **M–hashCode**(0)

Variables			
Variable name Variable type	**V–***variable-name* **T–***variable-type*	`HistoryItem item = new ...;` `System.out.println(line);`	**T–HistoryItem** **V–item** **T–String** **V–line**
Exceptions			

Throws
Throw
Catch
	X–*exception-name*
	R–*exception-name*
	H–*exception-name*
		`void init() throws InitException`
		`throw new RuntimeException();`
		`} catch (IOException e) {`
			X–InitException
			R–RuntimeException
			H–IOException

Comments			

Line comment
Block comment
Doc comment
	L–*comment-text*
	B–*comment-text*
	J–*comment-text*
		`// Need to refresh`
		`/* Logs given exception */`
		`/** outline page */`
		`CompareOutlinePage fPage;`
			L–// Need to refresh
			B–/* Logs given exception */
			fPage.**J–/** outline page */**

Various tokens			
Literal	**Y–***literal*	`String s = "modified";` `int i = 42;`	**Y– "modified"** **Y–42**
Operator	**O–***operator-name*	**Examples of operator names:** `^ ~ < << <= == > >= >> >>> | || - -- ! != ?: / () * & && % + ++`	
AST	**A–***node-type*	**Examples of AST node types** (for complete list, see class ASTNode in Eclipse): ANONYMOUS_CLASS_DECLARATION, ARRAY_ACCESS, ARRAY_CREATION, ARRAY_INITIALIZER, ARRAY_TYPE, …	
Keyword	**K–***keyword*	**Examples of keywords:** break, case, catch, class, continue, default, do, else, false, finally, for, if, import, instanceof, interface, new, noop, null, package, return, super, switch, synchronized, this, throw, true, try, while	
Modifier	**P–***modifier*	**Examples of modifiers:** abstract, final, native, private, protected, public, static, synchronized, transient, volatile	