Improving Change Descriptions with Change Contexts

Chris Parnin
vector@cc.gatech.edu
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Carsten Görg
goerg@cc.gatech.edu
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

ABSTRACT

Software archives are one of the best sources available to researchers for understanding the software development process. However, much detective work is still necessary in order to unravel the software development story. During this process, researchers must isolate changes and follow their trails over time. In support of this analysis, several research tools have provided different representations for connecting the many changes extracted from software archives. Most of these tools are based on textual analysis of source code and use line-based differencing between software versions. This approach limits the ability to process changes structurally resulting in less concise and comparable items. Adoption of structure-based approaches have been hampered by complex implementations and overly verbose change descriptions. We present a technique for expressing changes that is fine-grained but preserves some structural aspects. The structural information itself may not have changed, but instead provides a context for interpreting the change. This in turn, enables more relevant and concise descriptions in terms of software types and programming activities. We apply our technique to common challenges that researchers face, and then we discuss and compare our results with other techniques.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*

General Terms

Management, Measurement

Keywords

bytecode analysis, semantic diff, change pairs

1. INTRODUCTION

Researchers and software metrics enthusiasts turn to repositories for insights into software products and their developers. Software changes across revisions offer particularly insightful clues about bug fixes, feature and concept changes, task difficulty, and latent relationships among components.

Determining what is meant by *change* is an ongoing problem in software engineering. Many software processes require documenting changes including maintenance logs, bug tracking, peer review checklists, feature change lists, and software change requests. Human expression of change is usually described using a mixture of feature and code structure vocabulary. Although the description is concise, by nature, it is neither consistent nor complete.

Software processes, repository tools, and software engineers would benefit from a rich and consistent vocabulary for expressing change. Unfortunately, the most readily available automated unit of change, the method or function that the change is located in, has several limitations. First, a change to a method is not a fine-grained description. The change may be to fix an off-by-one error in an array index. In this case, the change is more strongly associated with the array and with array indexing than it is with the location of the statement. Second, changes are not always localizable. Some changes impact several modules or methods. For example, internationalizing string values in a program involves changes to the many methods processing strings. When the impact is large, as in this case, describing changes in terms of all method locations is overly verbose. In general, we want to be able to characterize change in terms of associated features.

In this paper, we propose a technique for capturing the context of a change. This context can be used to better summarize fine-grained changes, query changes in association with features and topics, and consolidate a change across many locations (*e.g.* a renamed method).

The technique works on fully annotated bytecode to take advantage of type information that is harder to obtain with textual representations. For each method body, we decompile the bytecode instructions into a source code representation and perform a line-based difference across versions. For each statement that has changed, we emit a set of symbol pairs and find which pairs have been deleted or added between successive revisions. The symbols are annotated with the type information and fully qualified source code location. With this set of *change pairs*, we can establish a relationship between a change and its context. Furthermore, we can consolidate the analysis to allow the development of tools supporting automatic summarization and tagging of software revisions, measurement of feature and topic activity, and detection of restructuring.

MSR'08, May 10-11, 2008, Leipzig, Germany.

The contributions of this research are to show how to

1. perform bytecode differencing
2. extract a minimal change along with its type-rich context for later analysis
3. use the context to perform several analyses without reexamining the program structure.

In the next section we discuss the related work, and in Section 3 we introduce the role vocabulary has in describing changes. In Section 4 we describe our motivation for bytecode analysis, and in Section 5 detail how we extract the change and its context. In Section 6, we present technical details on performing bytecode differencing, and in Section 7 we describe a case study in which we use our technique for measuring feature activity and the restructuring of code. Finally, we conclude with a discussion of future work.

2. RELATED WORK

Software researchers have not always agreed on how to describe changes made to software, nor what definition or granularity to use for describing the location of a change. Change and location have been primarily defined textually, dating back to work with `UNIX` **diff** [8, 16]. Many techniques since then have attempted to cope with the line-based idiosyncrasies and the necessity of reconstructing structural-level modifications in order to recover higher-level changes. For instance, coping with the physical swapping of two methods can be troublesome for traditional diff. However, using text similarity metrics [2], these type of problems can be alleviated.

The task of recovering structural-level modifications has received considerable attention from researchers attempting to detect moves, renaming, and other refactorings. Several methods work only at a signature-based level [19, 20, 12]. They primarily rely on finding some correspondence in method signatures. Others do try to account for text similarity [5] and binary matching [18] to relate regions or basic blocks between two versions. Overall these techniques fail to cope with more drastic changes such as splits and merges of methods.

In origin analysis [23, 13], several factors are taken into account to detect movement, renames, merging, and splitting of methods. The key insight of this work is the use of caller and callee patterns to identify methods among versions. Origin analysis matches the structural role of the method instead of matching the corresponding names or relying on textual similarity of the body. Although most operations are defined at the method level, the technique can be applied to work on a part of a method's body. Techniques such as origin analysis and refactoring reconstruction work at the structural level. They provide a good sense of how much activity has occurred in code movement; however, they do not assist in understanding fine-grain changes.

Other techniques extract fine-grain change representations. With semantic diff analysis [10], data flow pairs in a method body are computed for each version. Afterward, the set of pairs are compared to find any difference in the pairs. Any new pairs are displayed to inform the programmer of dependence relations. The technique has interesting insight, but is limited in several regards. Control flow is not accounted for, and no further aggregation of the relations is explored. Some context for a change is captured in terms of the dependency pairs, but it is limited in its ability to express to the nature of the change.

Another approach to fine-grain changes compares sets of *tokens* between two versions [22]. A token is composed of a token type and a value. In each version, the set of tokens corresponding to code and method bodies are extracted. Using this technique, tokens frequencies can be used to determine patterns in token types and token values. In this technique, the context of the change is limited to the token type (*e.g.* call or operator) and the method location of the token.

Researchers wishing to achieve more soundness in analysis have explored graph-based differencing [6]. Several approaches later refined this technique by simplifying the graph representation [14, 1] and using tree-differencing [7, 4]. However, the scope of the graph techniques differs in focusing on data and control dependence. The results in tree-differencing is sound but precision tends to be poor.

An unfortunate problem with text-based approaches is that they do not offer an easy way of getting type information without the burden of parsing and type-checking. Our technique avoids this problem and others associated with text-based difference by using bytecode analysis.

Like semantic diff, we compute pairs; however, semantic diff only computes the data-dependence among local variables. Instead, we compute dependencies between any symbolic values in the language. For example, we capture the relationship between a method call and a variable that stores the method's return value. This relationship does not follow from the abstract syntax tree (AST) of the program, but rather the pushing and popping behavior of the bytecode instruction stack. This allows us to capture differences in the local information flow dependence among symbolic elements. Furthermore, because the symbols can appear in contexts other than just a single site, we develop algorithms for consolidating this information in order to take advantage of its broader scope.

3. CHANGE CONTEXT MODEL

Software companies need to summarize developer's work sessions on a particular task. The summarization must consistently categorize changes in a manner that enables comparison with other projects, and with the project itself as it matures. The data also should be accessible to engineers for feedback.

There are several problems with addressing this need. Even though a manager specifies a task for programmers to work on, the actual tasks performed can differ greatly. Unanticipated design flaws, requests from teammates to get something else working, and task switching, make it difficult to get a sense of what actually occurs.

Revision history offers an interesting potential – changes and times are recorded in detail; however, the challenge becomes that of reconstructing the work efforts without the interpretation context of the original programmer's intent. Without intent, we must focus on how to enrich the vocabulary used to categorize change.

In this paper, we use several terms that we would like to define explicitly:

token: An atomic element of syntax.

feature: An atomic domain element implemented by one or more program elements.

change: The difference between two versions of a program expressed in terms of tokens.

revision: A collection of changes simultaneously committed to a source code repository. Changes may occur in multiple files.

location: A fully qualified path in the program, (*e.g.* a method's location is its package and class.)

symbol: A fully qualified name composed of a location, name, and type (*e.g.* a variable's location is the package, class, and method signature; its name is the variable name; its type is the variable type.)

The focus of much revision history research has been on *where* change occurs. But the vocabulary used to describe change should be much more expressive. We would like to understand change in terms of what features, domain topics, programming tasks, design patterns, and architecture were involved. This allows better traceability between the vocabulary of a software developers and managers and the analysis the researchers perform.

Consider the vocabulary found in a language's API. That set of terms is relatively fixed. It only changes with new releases of the language. However, the language API use is extensive throughout software development. The language API is also very distinctive. The API was designed for describing specific programming tasks such as file handling, network connections, string processing, and math operations.

In addition to language API calls, programmers also express changes with other mechanisms. The developer may refer to their own vocabulary from methods, and classes in the program. Developers can also share vocabulary. They can use, exchange, and introduce new vocabulary in the form of design patterns or frameworks.

However, if a developer makes changes by replacing an integer value with a different integer value, then that is not the most revealing change description. Sometimes we want to understand the context of the change. If we know that the integer change happened in the context of a math call, we can infer that some calculation was not desirable and was altered. The context of a change improves our interpretation of software changes, in the absence of the original change intent.

The goal of this paper is to both identify which entities describe change, and situate change with entities interacting with that change. We believe future analysis will use this information for measuring activity of features, topics, program plans, design patterns, and domain concepts.

We propose a preliminary model for capturing change context:

location: Where the change occurs in the program.

values: The types of the symbols associated with the change.

sources: Where values are produced.

destinations: Where values are consumed.

For example, the addition of the following line of code results in the change activity below:

```
+ db.Store( "name", nameTextBox.Text );
```

location:	Login.cs
values:	String
sources:	"name", TextBox.Text
destinations:	DataTable.Store

Future analysis can use this result to label this change as a *data-entry* activity. The analysis makes the inference by recognizing the symbol TextBox.Text as a UI component (the symbol is from the Forms namespace of the language API), and the symbol DataTable.Store as a persistence mechanism (the type is from the Data namespace of the language API). Finally, the act of sending a String from UI component to a database is classified as data-entry activity.

4. BYTECODE ANALYSIS

Many comparison techniques use lightweight methods that avoid structural constructions, leaving the burden to the user to reconstruct the aspects of structure they need. The challenge is to design a lightweight technique [15] that performs within the desired level of efficacy. What often arises is that performance or viability of queries suffer from structural information omitted by the technique. However, if too much structural information is provided, then the interaction with that data becomes complex and unwieldy. Some balance must be achieved between providing a change description that is both concise and appropriately annotated.

Consider if some statistics needs to be gathered on the prevalence of off-by-one errors in software development. One possible approach searches software change data for when values decrease by a literal `- 1`. Using a token-based approach [22], one could recall a significant portion of instances of these errors by simply noting the introduction of the two tokens `-`, `1`. However, many irrelevant queries would have to be sifted through to get the appropriate data.

Using a line-based unit of change, one could go further. Now we can attempt to limit the cases of finding the tokens `- 1` to be only used within the context of an array index or looping statements. But how do we do this? Without type information, we must instead rely on syntax clues, such as if the difference appears within an array's boundaries. Issues with operator overloading exists; however, even bigger problems are ahead. Syntax clues become less helpful as more code uses collection objects instead of arrays. Suppose now a framework designer wanted to extend the study to measure error rate differences between a method parameter design choice of using a list `range` method that accepts the parameters `(int start_index, int end_index)` instead of `(int start_index, int count)`. Such a query, now requires knowing the class type and method call. Unfortunately, such a query cannot be written leaving no choice but to rely on matching the method name with a lightweight tokenizer. A large study would be hampered by having to filter out all the instances of mistaken identity with other `range` functions. Again the burden of recovering the structure is placed on the query user.

The goal is a lightweight technique that arrives at the smallest unit of change with the least context necessary to understand that change. A token-based technique presents the smallest unit of change, but it gives no context. A line-based technique gives a change with a context, but the context has no structure. Structured-based techniques can provide types and structure, but at a cost: the changes are expressed in terms of the structural changes, which are often not minimal because they do not have the strict order-

Revision	Author	Change	Relation	Context	Location	Intent
43	Bob	3	Parameter	Server.timeout(int)	init()	"Increase timeout"
237	Amy	- 1	Index	array[length]	visitNodes()	"Fix array-out-of-bounds error"
311	Steve	CalcScale()	Assignment	graphPane.xAxisScale	OnResize()	"Refactor to use CalcScale()"

Table 1: Example changes in context.

ing and flatness imposed by source code lines. Furthermore, structured-based techniques are not lightweight or robust. Parsing and type-checking is notoriously difficult, especially with advanced object-oriented features, code-generation techniques, and in-house and third party tools used in the build process.

About the only tool that is privileged with both the build environment of the software product and the proper language know-how is the compiler or integrated development environment (IDE). One approach is to use the IDE's interface and parsing functionality to ease tool development. While possible, this is not ideal. The approach is not entirely lightweight, and it restricts the ability to process the test subject that requires certain licenses to commercial products and tools outside of its original build environment. However, compilers and IDEs do provide another interface to their services: the generated bytecode.

Using bytecode-based differences offer several advantages:

1. Bytecode has access to type information from the compiler.
2. Analysis can be performed at both the class and method level [11], as well as the instruction level.
3. The artifacts are smaller and self-contained.
4. Bytecode is executable: dynamic analysis, testing, performance analysis can be more readily performed.
5. Fine-grained transactions can be collected. Collecting intermediate builds from software developers can offer insights into software development and without having to instrument the IDE for change operations [17].

There are several challenges as well:

1. Some operations hidden from the programming language are exposed at the bytecode level.
2. Source code statements can be decompiled from bytecode, but they will not be exact.
3. Compilation may re-order basic blocks and may apply strength reduction optimizations on statements.
4. Without debug symbols, local variables do not have names and may change with addition of new variables.

However, these challenges can be addressed to improve how we understand change in software.

5. EXTRACTING CONTEXT PAIRS

When examining the differences between software versions, we can only understand what has changed by looking at what stays the same across both versions; that is, the change must be contrasted to some anchoring context. For example, if a programmer makes a change to the value of a parameter of a method call, then the parameter is the *subject* of the change. But what does the change fasten to? In this example, the change can be said to be anchored onto the method call, the method call is anchored onto the calling method body, and so on.

Our goal is to provide additional context to any change. When we identify a change, we find which symbols also interact with that changed symbol. This context can later be used for analysis. In this view, one symbol is a producer of a value, and the other is a consumer of the value. If a symbol produces a value, then the goal of the symbol pair analysis is then to locate the symbol that consumes the value.

The interaction of stable elements with changing elements offers insight into the concepts programmers are concerned with during a programming session. Furthermore, they provide landmarks for understanding and situating change. Capturing these interactions requires examining element connections to find new connections that signify the presence of key phenomenon.

In Table 1, we provide some examples of how a change is associated with its context. The location refers to the method location where the change occurred. The relation between the change and context describes how values are propagated between the change and context. A revision of software may have many changes associated with it; this is only one such change. Intent describes the original programmer's reason for performing the change. This may or may not be reflected in the revision log.

5.1 Symbols

Recall that a symbol is defined by its location, name, and a type. The location is a fully qualified path in the program, whereas a type is either a system type or user-defined type. For example, in the following code snippet the resulting symbols are described in Table 2.

```
SetName(e,fullname)
e.Name = fullname.SubString( fullname.LastIndexOf(".") + 1);
```

Code Element	Symbol Kind	Location
fullname	Parameter	Class.SetName
SubString	Method	String
LastIndexOf	Method	String
"."	Literal	
Name	Field	Class

Table 2: Resulting symbols of code elements.

5.2 Pairs

Consider the following source code difference:

```
- sum( list );
+ sum( values() );
```

In the above code example, the receiver of the value produced by `values` is `sum`. This was formerly produced by the symbol `list`.

We model the interaction of two symbols with what we call a *change pair*. In a change pair, the left-hand symbol produces a value that is consumed by the right hand symbol. These are the change pairs of the two versions.

```
- list -> sum
+ values -> sum
```

Changes can be related to a symbol through a fixed set of relationships. The category of relationships and category of symbols discussed in this paper are described in Table 3.

Symbols	Relationships
Type	Assignment
Method	Parameter
Field	Index
Parameter	Condition
Variable	Name qualification
Literal	Return

Table 3: Supported symbols and relationships.

Suppose in another version of the software a change is made to only **sum** a subset of the values in the list.

```
- sum( values() );
+ sum( values().subset(x) );
```

The change pairs are as follows:

```
- values -> sum
+ x -> subset
+ values -> subset
+ subset -> sum
```

In the new version, `values` no longer is associated directly with **sum**. Further, three new pairs are introduced. Comparing the two sets of change pairs, we find that some symbols existed before, whereas others are newly introduced. In general, we refer to a symbol that previously existed in the method body as an *anchor* or anchored symbol.

In the previous code example, if `x` did not previously exist, then it implies that the method body had to add a parameter, or an additional query had to be added to retrieve the value. The consequence is that several ripple changes must be created to flow new information through the program. If `x` was already present in the method, then we know the scope of the change would be less severe.

6. BYTECODE DIFFERENCING

In this section we describe how to extract a set of symbol pairs from a method body. We do this by recovering the code statements from the bytecode and then performing a line-based difference between the code statements. From the differences in code statements, we identify which statements have been affected. For each affected statement, we emit each symbol with its related symbol. We then compare the differences between the symbol pairs.

By restricting the analysis to capturing only a set of pairs, we avoid problems caused by block reordering and statement movement while still achieving our goal of finding the change context. Consider the following code example:

```
- int x = A( 1 + B(y) );
+ int x = A( 1 + B(y) + y );
```

In the code example, the expression "`+ y`" is added. We wish to associate that change with the symbol `A`.

If we were processing source code, then we would use the AST to guide our process. But because we analyze bytecode in order to to ease access to type and property information, we must use another method to recover a similar representation. Furthermore, we need a mechanism to know which symbol is associated with an instruction. Both goals can be readily achieved with the assistance of bytecode tools such as `Mono.Cecil` [3] and `JABA` [9].

First, we construct a flow graph of the method body instructions. This creates a graph of basic blocks with each block containing a list of bytecode instructions. Segmenting the instructions into basic blocks is necessary in order to properly recover the statements. The next challenge is then to create an AST-like representation for the statements. We use an abstraction called a *pop tree* over the bytecode instructions to recover a simplified model of the AST.

6.1 Pop Trees

Both Java and C# bytecode are executed on a stack-based virtual machine. Because of this, if the stack state is not accounted for, then it is difficult to tell the difference between a sequence of method calls from a complex chain of nested method calls.

The instructions for the above code example

```
int x = A( 1 + B(y) + y );
```

would appear as following:

Program 1 CIL assembly for addition example.

```
this  IL_003: ldarg.0
y     IL_004: ldloc.1
this  IL_005: ldarg.0
y     IL_006: ldloc.1
      IL_007: call   instance int Exp.Form::B(int)
      IL_00c: add
1     IL_00d: ldc.i4.1
      IL_00e: add
      IL_00f: call   instance int Exp.Form::A(int)
x     IL_014: stloc.0
```

A pop tree corresponds to a single statement in the original source code. A pop tree is constructed by simulating the instruction stream to push and pop various values onto a stack. For example, an `add` instruction pops two values off of the stack and pushes the sum onto the stack, whereas a `load` instruction pushes one value onto the stack. An instruction *owns* another instruction if it pops its value off the stack. The resulting tree of instructions ends with the top node consuming the value of all the other child instructions. If desired, a post-order transversal would closely correspond to the original AST. The resulting pop tree for the code example is shown in Figure 1.

6.2 Emitting Symbol Pairs

After constructing the flow graph and pop trees for each block, the pairs are ready to be emitted. To emit a pair, the pop tree is visited in post-order. If the node is associated with a symbol, then the tree is searched up for the first symbol to consume the child symbol. Only calls, loads, and stores can be associated with a symbol. Consider the following code statement:

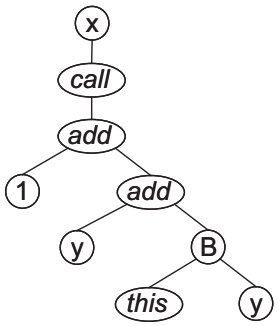

Figure 1: Pop tree for addition example.

```
int x = array[ array.Length - 1 ];
```

Figure 2 shows the resulting pop tree.

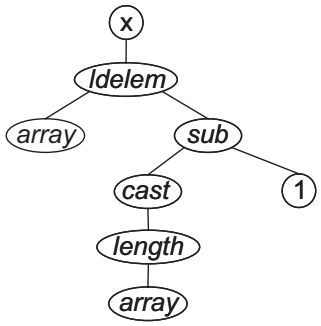

Figure 2: Pop tree for array example.

The following pairs would be emitted:

```
array  -> x
array  -> length
length -> array
1      -> array
```

6.3 Context Paths

Further variations are possible by choosing different policies in how to select the ancestor. For example, the top-most ancestor can always be selected. Second, symbols do not have to be so restrictive, they could include any bytecode instruction desired; whereas, in this paper, we focused on higher-order operands.

Recall, we emit pairs as a set. This was primarily done to avoid problems caused by block reordering and statement movement. If finer-grain analysis was desired, then approaches used in binary matching [18] may be incorporated. This allows differences to be scoped at the basic block or statement level. However, it is not clear if the noise from errors is worth the extra precision.

In contrast to performing differences scoped at the basic block or statement level, another approach is to extract a fully specified context path from each version. This path could contain all the symbols that are ancestors of the symbol, as well as the symbols of any predicates that dominate the basic block that the statement resides in. This captures the control flow relationship between basic blocks without enforcing a specific matching policy.

In our implementation, we find that pairs offer sufficient context for our analysis. However, this technique can easily be extended to collect the appropriate level of context as needed by a particular analysis technique.

7. CASE STUDY

We have developed a tool `CILDiff` which operates on C# bytecode and reports annotated byte differences. These differences are queried by our tool `CILQuery` .

We have performed a case study to evaluate the efficacy of our approach in supporting several classes of queries. In particular, we examine the following two topics:

1. Detect code movement and restructuring

2. Categorize change according to participating types

There are two key categories in which the analysis of work efforts in software development can be assisted. The first category reduces the effects of code movements and renamings. In general we would like to characterize the type of coding activity to assist in understanding the change history. For example, process management would be improved by tracking when refactoring efforts took place and seeing what impact they had on the overall project. We will discuss this category in Section 7.1.

The second category abstracts over changes and the associated features in a matter that developers and managers can make use of. We will discuss this category in Section 7.2.

We examine two systems written in C#: `Cecil` [3] is a library for inspecting C# bytecode. We study 579 revisions of it which date back to 2005. `Cecil` has 280 classes, 2806 methods, and 216 unique system calls. `ZedGraph` [21] is a library for charting in Windows applications. We study 29 revisions of it which date back to 2003. `ZedGraph` has 100 classes, 1656 methods, and 340 unique system calls.

7.1 Movement and Restructuring

Eventually, a developer realizes the need to refactor code. The transformation of code results in several key properties. First, the names and locations of many artifacts will no longer be the same. Second, the refactoring event acts as a beacon. The refactored revision marks when errors might have been introduced and provides evidence for when and how often refactoring efforts have been made.

7.1.1 An Example

In revision #77129 of `Cecil`, the log reads "one refactoring a day can get you a long way." We have good reason to believe refactoring did occur in this revision, but what semantic description can we provide to better measure this refactoring?

We use this opportunity to illustrate how different techniques would operate on this example and contrast the results with our analysis.

At the largest granularity, we know two files have changed:

```
Mono.Cecil.Binary/ImageReader.cs
Mono.Cecil.Metadata/MetadataReader.cs
```

Applying line differences we find 99 lines were deleted and 59 lines were added. Further, with lightweight parsing, we can associate those additions and deletions to method locations.

```
+ private RVA ReadRVA ();
+ private DataDirectory ReadDataDirectory ();
* public void VisitImportAddressTable( ... )
* ...
```

Two methods were added and nine methods were modified. To understand what changed in the methods, we apply a token-based difference.

```
- Mono, Cecil, Binary
- new, RVA(), DataDirectory()
+ ReadRVA(), ReadDictionary()
```

The token approach eliminated some of the differences caused by code formatting occurring across several lines. In fact, the file `MetadataReader.cs` never semantically changed: it only contained superficial formatting changes. The tokens: `Mono`, `Cecil`, `Binary` occurred from the removal an unnecessary namespace qualification.

But more importantly, we now understand the modification of the nine methods includes the addition of two methods and the removal of constructor calls.

Applying our technique, we derive a set of change pairs for the methods. First, we examine change pairs that contain anchored symbols. This means the pair is not completely new or completely removed from the method body. The pairs are stylized to display type information, symbol kinds, and the relationship between the symbols.

```
- DataDirectoriesHeader.Reserved = new DataDirectory()
+ DataDirectoriesHeader.Reserved = ReadDictionary())
- NTSpecificFieldsHeader.BaseOfCode = new RVA()
+ NTSpecificFieldsHeader.BaseOfCode = ReadRVA()
```

We can confirm our suspicions that method call additions replaced the constructor calls. A subtle difference still exists in the source code:

```
header.HeaderSize = m_binaryReader.ReadUInt32();
```

The token-based difference did not notice the removal of several `ReadUInt32` calls. This function is still being used; however, it is not longer used as an argument to the removed constructors. This is observed by deleted change pairs that are no longer anchored.

```
- ReadUInt32() -> new DataDirectory()
- ReadUInt32() -> new RVA()
```

Finally, a single change can be cataloged in the following manner.

location:	ImageReader.VisitCLIHeader(CLIHeader)
value:	RVA
source:	ImageReader.ReadRVA()
destination:	CLIHeader.Flags

Further analysis could summarize this information to generalize that the header's properties previously were assigned from a newly constructed DataDictionary object and now are assigned from a private helper method.

7.1.2 Detection Rules

The challenge of detecting refactoring occurs when signatures and content of methods no longer match. However, an insight from origin analysis [23], focusing on the caller and callee signatures, can be applied in a bottom-up manner.

A very simple and effective rule can be used to detect a general refactoring substitution:

Conservative Generalized Substitution.

```
before(X <-> anchor), after(S <-> anchor) :
where locations(X) = locations(S) and
      X is deleted, S is added.
```

The advantage of this rule is that its application can be detected from just the set of change pairs produced by the bytecode without examining the call hierarchy of the program. This reduces the analysis burden.

Change Signature.

```
before(X <-> anchor and not parameter -> S),
after(S <-> anchor and parameter -> S) :
where references(X) = references(S) and
      X is deleted method, S is added method.
```

A change signature is marked when a new method is called with a parameter value but a method referenced in the previous version did not have the same parameter flow to it.

Move Method.

```
before(X <-> anchor),
after(S <-> anchor) :
where references(X) = references(S) and
      X is deleted method, S is added method and
      X.class != S.class
```

A move is marked when two symbols are referenced in the same manner but belong to a different class.

Extract Method.

```
before(P)
after(P -> S) :
      pairs(S) in before(pairs(P)) and
      pairs(S) not in after(pairs(P))
```

An extract method is marked when a change pair now exists in the new version, and the pairs of the new method were in the previous version but not in the current version.

7.1.3 Results

We applied these techniques to evaluate the effectiveness of locating refactorings.

In `Cecil`, several revisions of refactoring have occurred. For example, there was one case in which seven method locations had been reported as changed and two methods as added. Instead, we can report the following as two extract methods with seven reference updates. This allows analysis to focus on non-reference update changes and marks when a refactoring has happened in the development history.

Overall, only seven revisions were labeled in the change log as refactoring. However, we found 55 revisions with evidence of refactoring shown in Table 4. Of the seven revisions, we were able to identify six of the revisions as refactoring. In the other revision, a group of methods where moved together into another namespace. Because they moved together, but also referenced each other, they were not properly found. This result is consistent with origin analysis which also has similar problems.

Refactoring Type	# Revisions	Total Number
Move	3	5
Rename	10	15
Parameter	13	32
Extraction	29	97

Table 4: Refactorings in `Cecil`.

7.2 Measuring Features

In this part of the case study, we want to understand how our model of change context assists in studying features in software. We do not automatically identify features or the relationships to the program elements. Instead, we label certain program methods, classes or system calls that are strongly associated with a feature. For example, we associate graphic calls for measuring strings with a *font* feature. In another example, we use types beginning with the "System.IO" to be associated with the *IO* feature.

To measure feature activity, we count whenever a program element belonging to a feature appears in a change context. We look in the types, values, sources, and destinations of our context model for this evidence. We discuss four questions in particular:

1. How does the number of the change symbols compare to the number of change locations?
2. Can we measure feature activity over time?
3. Can we study when features were worked on or used by other features?
4. Can we detect when one feature is abstracted over by another feature?

7.2.1 Number of Change Symbols

Sometimes a change may be in only one method but involve many symbols. Other times a change is made across many method locations but only concerns a few symbols. We can measure the ratio between the number of method locations and the number of symbols to better understand the change task.

Figure 3 shows the trend of the number of method change locations against the number of symbols. In this example we restrict the symbol kinds associated with changes to be methods, properties, types, system calls and system types. In the figure, we see that sometimes the number of symbols can be larger than the number of locations. In particular, during these types of sessions, we notice a larger ratio of system calls.

Other times, the change locations can be quite large but the number of symbols is much smaller. During revisions 307 to 345, `Cecil` underwent one of the most extensive periods of activity in its revision history. Five of those revisions had an extremely high number of method modifications. Interestingly, none of these five revisions added new methods. This eliminates the possibility of renaming of methods or move and extract method refactorings because they would require new methods to have been added. Instead, some activity occurred that touched at the definition of much of the program. In Table 5, the change locations can be seen in comparison to the symbols composed of properties, methods, types, and system calls. Figure 4 shows a similar trend in `ZedGraph`'s revision history.

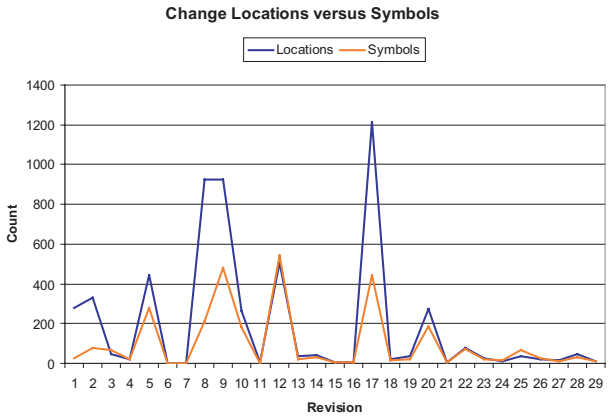

Figure 3: Several large code events occurred where many parts of the system were touched. But only a relatively small amount of symbols was used to describe those changes in `Cecil`.

Revision	# Change Locations	Symbols
307	2480	558
312	2480	344
330	2529	570
344	2543	347
345	2543	573

Table 5: The number of change locations in comparison to the number of symbols in `Cecil`.

7.2.2 Feature Activity

A large program has many features. Understanding which features are actively used in development gives perspective on which symbols are needed to program current tasks.

We identified several prominent features and their association with user-defined types in `ZedGraph`. Figure 5 shows the activity of these features over time.

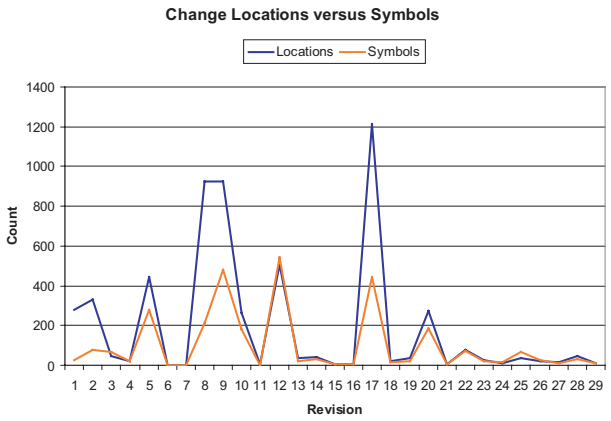

Figure 4: In general, the number of change locations is larger than the number of symbols associated with the change for `ZedGraph`.

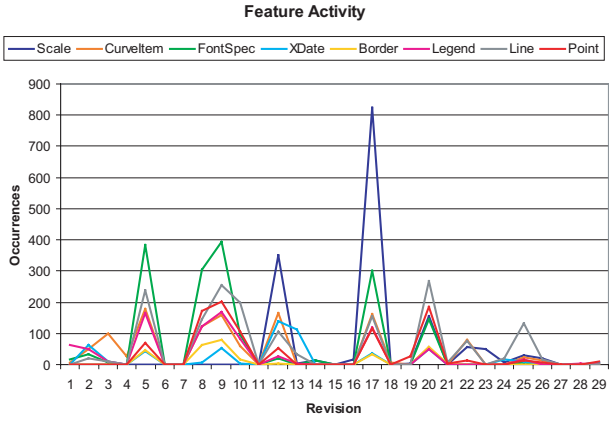

Figure 5: The activity of various features over time.

7.2.3 *Feature Stability*

When examining the activity of a feature, we would like to understand how stable it is. There are several benefits of measuring stability. First, stability gives a baseline to compare a feature against error rates. This baseline can be used to judge the severity of the errors based on if the feature is considered to be stable or not. Second, the stability measure provides programmers feedback about how stable a concept is when choosing a program abstraction. For example, a programmer might choose to delay working on a certain feature if the programmer could see that another teammate had not yet stabilized the feature.

There are two criteria we would like to use in measuring stability: periods of development and periods of use. Overall, we would like to see the amount of development effort on the feature decrease over time. Seeing periods of feature use gives confidence that the feature has been integrated with the system and is less likely to be changed.

In Figure 6, we compare the period when the Scale feature was being implemented in `ZedGraph` versus when the feature was used by other components in the system. From the graph, we can conclude that the feature had fairly stabilized.

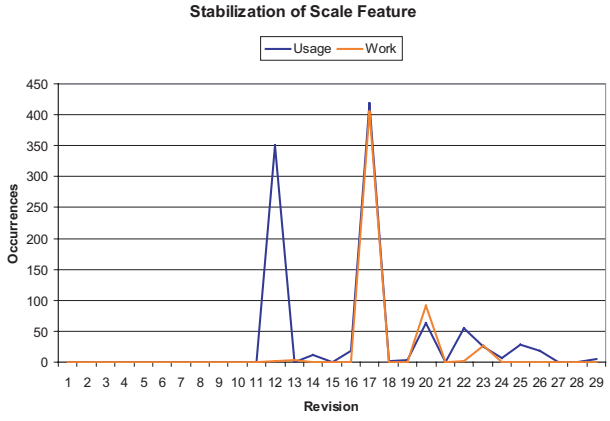

Figure 6: The scale feature was used for several revisions before becoming limited, several revisions of work followed, until the feature again stabilized and use resumed.

7.2.4 *Feature Abstraction*

The initial vocabulary that developers start with is the language API and programming constructs. As a program starts to share a common concern, a developer creates an abstraction to describe those concerns. During the development of the abstraction, the programmer is actively using the language API to implement the abstraction. If successful, the developer should have created appropriate abstractions over the system calls and the frequency of those calls should decrease.

In Figure 7, the `ZedGraph` developer created a `FontSpec` class to handle issues with fonts and font rendering. Initially, the developer uses the library API to assist in the development (*e.g.* `System.Drawing.Graphics.MeasureString`, and `System.Drawing.Font`). But eventually, the development of the `FontSpec` class no longer has to rely on using the system calls. Instead, the usage of the `FontSpec` class increases, and a successful abstraction is created. If there were problems with incompleteness of the abstraction, then we might notice it if more system calls are introduced later on in development, but we do not see any in this case.

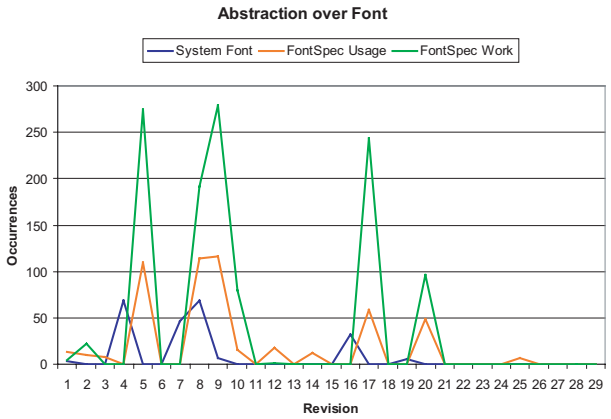

Figure 7: The class ZedGraph.FontSpec assisted with handling fonts. System calls related to measuring strings and fonts, are at first used, but use eventually drops as the user-defined class is successful in abstracting over the system class.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented techniques for improving the richness of how we describe software changes. Approaches deriving type information from source code text generally face technical limitations that hinder industrial adoption. We circumvent this problem by deriving type information in a robust and lightweight manner from statements reconstructed from bytecode. We identify which bytecode statements have changed between revisions. For each statement that has changed, we emit a set of symbol pairs and find which pairs have been deleted and added between the revisions. The symbols are annotated with the type information and fully qualified source code location. This information provides enough context to perform a range of interesting analyses without needing to reconstruct the source code.

To demonstrate our approach, we performed a case study on two software systems to show how our tools `CILDiff` and

`CILQuery` find and use change pairs. In our case study, we show how instead of just focusing on the method locations where changes occur, we can also extract the symbols that developers use. By examining the symbols, we have a better sense of the features and topics that the developers were focusing on. We show how to measure feature activity, feature stability, and features abstracting over other features.

In this work, we are striving to simplify the process of working with revision histories in a way that facilitates adoption in every day software practice. For future work, we anticipate that further analysis can take advantage of the rich type information available with our tools. In particular, examining how change efforts relate in terms of object collaboration and architectural patterns will offer powerful ways of understanding and incorporating feedback into the software development process.

9. ACKNOWLEDGMENTS

Spencer Rugaber and our anonymous reviewers gave helpful comments on earlier revisions of this paper.

10. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. A Differencing Algorithm for Object-Oriented Programs. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[2] G. Canfora, L. Cerulo, and M. D. Penta. Identifying Changed Source Code Lines from Version Repositories. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 14, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Cecil Mono, 2008. Bytecode inspector. `http://www.mono-project.com/Cecil`.

[4] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, 2007.

[5] C. Görg and P. Weißgerber. Error Detection by Refactoring Reconstruction. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[6] S. Horwitz. Identifying the Semantic and Textual Differences Between Two Versions of a Program. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 1990. ACM.

[7] J. J. Hunt and W. F. Tichy. Extensible Language-Aware Merging. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 511–520, Washington, DC, USA, 2002. IEEE Computer Society.

[8] J. W. Hunt and M. D. McIlroy. An Algorithm for Differential File Comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[9] ARISTOTLE RESEARCH GROUP, 2003. JABA: Java Architecture for Bytecode Analysis. `http://www.cc.gatech.edu/aristotle/Tools/jaba.html`.

[10] D. Jackson and D. A. Ladd. Semantic Diff: A Tool for Summarizing the Effects of Modifications. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society.

[11] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 194–202, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[12] M. Kim, D. Notkin, and D. Grossman. Automatic Inference of Structural Changes for Matching across Program Versions. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[13] S. Kim, K. Pan, and J. E. James Whitehead. When Functions Change Their Names: Automatic Detection of Origin Relationships. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[14] W. Laski, J.; Szermer. Identification of Program Modifications and its Applications in Software Maintenance. *Software Maintenance, 1992. Proceerdings., Conference on*, pages 282–290, 9-12 Nov 1992.

[15] G. C. Murphy and D. Notkin. Lightweight Lexical Source Model Extraction. *ACM Trans. Softw. Eng. Methodol.*, 5(3):262–292, 1996.

[16] E. W. Myers. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, 1(2):251–266, 1986.

[17] R. Robbes. Mining a Change-Based Software Repository. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 15, Washington, DC, USA, 2007. IEEE Computer Society.

[18] Z. Wang, K. Pierce, and S. McFarling. BMAT — A Binary Matching Tool for Stale Profile Propagation. *Instruction-Level Parallelism*, 2:1–20, 2000.

[19] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-Oriented. Design Differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA, 2005. ACM.

[20] Z. Xing and E. Stroulia. Refactoring Detection based on UMLDiff Change-Facts Queries. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 263–274, Washington, DC, USA, 2006. IEEE Computer Society.

[21] ZedGraph , 2008. ZedGraph Wiki. `http://zedgraph.org/`.

[22] T. Zimmermann. Fine-grained processing of CVS archives with APFEL. In *eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 16–20, New York, NY, USA, 2006. ACM.

[23] L. Zou and M. W. Godfrey. Detecting Merging and Splitting using Origin Analysis. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 146, Washington, DC, USA, 2003. IEEE Computer Society.