Specifying and Detecting Meaningful Changes in Programs

Yijun Yu*, Thein Than Tun* and Bashar Nuseibeh* †
* The Open University, Milton Keynes, UK
† Lero, Irish Software Engineering Research Centre, Limerick, Ireland

Abstract—**Software developers are often interested in particular changes in programs that are relevant to their current tasks: not all changes to evolving software are equally important. However, most existing `diff` tools notify developers of more changes than they wish to see. In this paper, we propose an automated technique to specify and detect only those changes in programs deemed meaningful – or relevant – to developers. Using four elementary extensions to programming language grammars (Ignore, Order, Prefer and Keep), developers can specify, with limited effort, what types of changes are relevant to their programming tasks. The algorithms for generating the normalisation and clone removal transformations distill non-trivial and relevant differences automatically. We evaluate our tool on a benchmark of programs to demonstrate its improved precision compared with other `diff` techniques.**

I. Introduction

"Nothing endures but change."

Heraclitus (c.535 BC—475 BC)

This philosophy is largely true in most software development projects. However, not all changes are equally relevant to developers engaged in different development tasks. For example, changing the indentations of program statements does not necessarily alter the meaning or semantics expressed by a C/Java program. Nonetheless it could lead to false alarms in many revision control systems, which typically use text-based difference comparison algorithms (e.g., the `diff` utility in Unix [1]). Although an indentation is not meaningful to the execution semantics of C/Java programs, it can be very important in other programming languages such as Python. Furthermore, for those who are concerend with pretty-prints of C/Java programs, indentation is still important. Another example is API evolution: thanks to the widely adopted principles of information hiding and modular design [2], users of object-oriented programming libraries are encouraged to separate the API from its implementation. Some developers may wish to identify only those changes made the API [3], whilst others may want to find out changes in the API implementation only.

A change considered *meaningful* for one purpose may be irrelevant for another. For a given purpose, how can one specify the types of changes that are relevant to a specific development task? How can such a specification be used for automatic detection? Most change detection tools are either good at reporting *all* changes through general purpose diff algorithms on programs represented as line-separated text [1], [4] and structured models [5]; or good at finding out certain or all changes that are *specific* to one particular language such as UML class diagrams [6], dependency graphs [7], or Verilog [8]. Few techniques aim to provide a generic solution that can also be customised to the specific language and the specific needs of the developers.

In this paper, we propose a new way to specify meaningful changes as the composition of elementary changes that are defined on a *normalising* transformation of the source programs. Two programs that are not different due to orderings, abstractions, and preferences will be normalised into the same program, valid in the source language's syntax. We show that such normalisations can be specified as simple annotations on the original "production rules" [9], while specific needs of further normalisation can be accommodated by user-defined functions. Each type of elementary normalisation corresponds to an elementary kind of term in the production rules of the source language. Once such annotations are specified, a fully automated meta-transformation can turn them into a composed transformation that operates directly on the source programs, separating meaningful changes from the irrelevant ones.

The meta-transformation is written as a generic modification to the meta-grammar[1] of the TXL transformation system [10]. Therefore, it is applicable to any source language specifiable by TXL, which currently supports several general-purpose programming languages (C/Java/C#/Python) as well as several modeling languages (e.g., XML, XMI, GXL). To evaluate our *Meaningful Change Tool* (hereafter `mct`), we show how few changes are required to be added to the grammars for a few typical programming tasks, such as detecting changes to the APIs. Also we show how we applied `mct` to detect these meaningful changes in CVS repositories of two medium-sized software projects and one small hardware project.

The remainder of the paper is organised as follows: Section II introduces a small running example to illustrate the problem and the requirements for specifying and detecting meaningful changes. Using this example, Section III explains the approach we adopt to bootstrap the normalisation transformations needed in the implementation of the tool. Section IV presents the results of a number of experiments in using the tool, and comparing the performance with existing `diff` tools. Section V compares the conceptual differences in the design of existing approaches, and indicates some limitations of our approach. Section VI concludes the findings.

[1]The grammar of the TXL language is expressed in TXL too.

II. A Motivating Example

The essence of meaningful change can be illustrated using a simple Java program in Listing 1. After some trivial changes, it is still the same program shown in Listing 2. Unix `diff` utility [1] reports these changes as 1 deletion and 1 modification of a big chuck in Listing 3. Applying a more advanced algorithm `ldiff` [4] to this example, line-based changes are reported as 2 insertions, 1 deletion and 2 modifications. Each of the 5 changes is at most two lines for programmers to check. Note that we have applied both diff algorithm to ignore the whitespaces.

Listing 1. `cat -n HelloWorld.java`

```java
public class HelloWorld
{
 static private String hello = "Hello";
 private static String world = "world";
 static public void main(String args[]) {
  System.out.println(hello + ", " + world + "!");
 }
}
```

Listing 2. `cat -n HelloWorld-2.java`

```java
public class HelloWorld
{
 private static String world = "world";

 static private String hello = "Hello";
 public  static void main(String args[]) {
  System.out.println  (hello + ", "
     + world + "!");
 }
}
```

Listing 3. `diff -w HelloWorld.java HelloWorld-2.java`

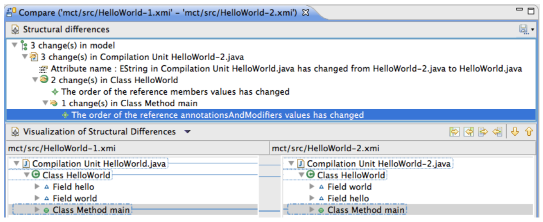
```
3d2
<  static private String hello = "Hello";
5,6c4,8
<  static public void main(String args[]) {
<   System.out.println(hello + ", " + world + "!");
---
>
>  static private String hello = "Hello";
>  public  static void main(String args[]) {
>   System.out.println  (hello + ", "
>      + world + "!");
```

Listing 4. `ldiff.pl -w -o diff HelloWorld.java HelloWorld-2.java`

```
3,3d2
<  static private String hello = "Hello";
4a4,5
>
>  static private String hello = "Hello";
5,5c6,6
<  static public void main(String args[]) {
---
>  public  static void main(String args[]) {
6,6c7,7
<   System.out.println(hello + ", " + world + "!");
---
>   System.out.println  (hello + ", "
6a8,8
>      + world + "!");
```

Applying a structured diff algorithm [2] to the EMF models corresponding to the abstract syntax structure of the two example Java programs[5]), 3 changes are reported: one concerns the renamed compilation units, one concerns the class

[2]EMFCompare, www.eclipse.org/emf/compare


Compare ('mct/src/HelloWorld-1.xmi' – 'mct/src/HelloWorld-2.xmi')

Structural differences

```
3 change(s) in model
  3 change(s) in Compilation Unit HelloWorld-2.java
    Attribute name : EString in Compilation Unit HelloWorld.java has changed from HelloWorld-2.java to HelloWorld.java
    2 change(s) in Class HelloWorld
      The order of the reference members values has changed
      1 change(s) in Class Method main
        The order of the reference annotationsAndModifiers values has changed
```

Visualization of Structural Differences

mct/src/HelloWorld-1.xmi	mct/src/HelloWorld-2.xmi
Compilation Unit HelloWorld.java	Compilation Unit HelloWorld-2.java
Class HelloWorld	Class HelloWorld
Field hello	Field world
Field world	Field hello
Class Method main	Class Method main

Fig. 1. The differences found by EMFCompare on the two EMF models

`HelloWorld` for "*the order of the reference members*", and the final one concerns the method "`main`" for "*the order of reference annotationsAndModifiers values*".

In fact, none of the changes identified in this example is meaningful if the programmer only wants to see non-trivial changes: just as adding a newline or some whitespaces would not change the syntax of the program, nor would swapping the keywords `public` and `static` in the declaration of the `main` method make any semantic differences.

To find a meaningful change between two versions of a program, our proposed solution includes two major steps in Figure 2.

- *Step 1. Specification*: a developer specifies a number of annotations to the given grammar of the programs, represented by the dotted arrows;
- *Step 2. Detection*: the tool generates a refined grammar and two sets of transformations (normalisations and clone-removals) from the specification in Step 1 and applies these transformations to a pair of two source programs, reporting the meaningful changes to the developer.

In a typical workflow the specification step for annotating the given grammar is done manually by the developer, whilst the transformation step for generating the refined grammar and transforming the rules to detect meaningful changes from the programs in the CVS repository is done automatically by the `mct` system.

III. Specifying Meaningful Changes

Before specifying our tool, we first define a few requirements for detecting *meaningful changes* through normalisation transformations.

Definition 1: **Normalise into equivalence classes**. A program P is said to be *meaningfully equivalent* to program P' if and only if $(P' = P) \vee (N(P') = N(P))$ where $N(P)$ is the normalisation transformation of P. In other words, P' introduces no meaningful changes to P. Typically $N(P)$ is a many-to-one transformation.

As discussed earlier, the exact meaning for 'meaningful equivalent' in the Definition 1 is intentionally left open for user to define by the normalisation function, because it depends on the purpose of the analysis and the relevance to the tasks. The definition provides the general criteria for determining whether a transformation is a suitable normalisation once it is clear what is meaningfully equivalent to the users: any trivial or irrelevant changes should be normalised to the same value. After the normalisation transformation is defined, the

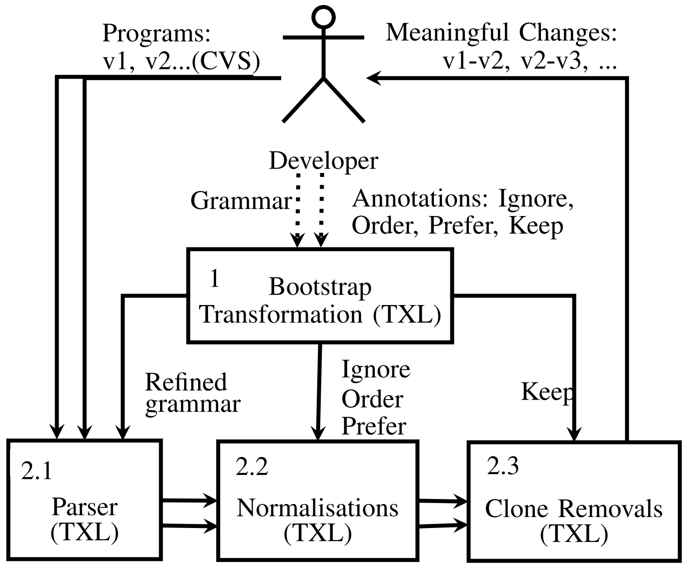


Fig. 2. Specifying and detecting meaningful changes, an overview

detection of the meaningful changes becomes comparing the two normalised programs.

In principle, however, there can be infinite possible normalisations for defining an equivalent class. For example, adding any number of whitespaces can be regarded as normalisation transformations as opposed to removing the whitespaces. Therefore, such a normalisation transformation needs to be *terminable* by restricting the size of the targets.

Definition 2: **Terminable normalisation.** A normalisation N is *terminable* if $N(P)$ is smaller than P in size.
The *identity* transformation which preserves everything in the source is a normalisation, however it is trivial that all changes are meaningful if the identity is used as normalisation. According to Definition 2, we only consider the transformations that indeed make the output smaller than the input.

In the remainder of the paper, we focus on generating normalisation transformations based on elementary composable normalisations that can be derived from the language grammars. Since the language grammar is made of production rules, one can start with the basic types of structural terms on each production rule, mandatory, optional, repeat and alternatives. For the optional/repeat terms, if one modifies it into the mandatory terms the transformation can violate the Definition 2. Similarly modifying optional to repeat could also introduce non-terminations. Therefore this leave us with three basic types of normalisations.

Definition 3: **Elementary structural normalisations.** Let a production rule be $N \leftarrow (\dots T_i? \dots T_j * \dots)| \dots$ where A is a non-terminal, and T_i is the i-th term (which could be either terminal or non-terminal), and optionally the rule could contain more than one alternative sequence patterns. Every optional (denoted by '?') term can be *ignored* if with or without the value makes no difference to the developer; elements of the repeated (denoted by '*') term can be *ordered* if the ordering of the values are not important to the developer; and the whole element that matches with one mandatory alternative ('|') can be *preferred* to a simpler values if a difference between the alternative rules are not significant to the developers.

The three elementory structural normalisations preserve the validty of the normalised program in terms of the source programming language.

Property 1: **Syntax validity.** The normalised program by the three elementary transformations are valid programs in the *source* programming language.

The validity of the normalised programs is not a concern when the purpose of checking meaningful changes is not to obtain a compilable program. In such cases, one could choose to redefine the target grammar to be incompatible to the source language. On the other hand, if there is a simple alternative in the source language's production rule, such as semicolon, then it is useful to apply the Prefer rule to switch to a simpler alternative while preserving the source grammar in the transformed targets.

Also it is found that the elementary normalisations can be further customized. The unconditional ignored rule for an optional term '?' can be associated with a conditional check, for example in the API extraction example, one would remove the declaration if and only if it does not have a 'public' or 'protected' modifiers. Similarly, the ordered rule for repeating terms '*' can be customized to ascending or descending orders, and the ordering criteria can be associated with certain key' substructures.

The following property guarantees that a functional composition of the three basic normalisation transformations still satisfy the requirements of terminable normalisation.

Property 2: **Composibility of terminable normalisations** If two basic normalisations N_1 and N_2 satisfy the terminable normalisation requirements in Definitions 1 and 2, then the functional composition $(N_1 \oplus N_2)(P) = N_2(N_1(P))$ also satisfies the terminable normalisation requirements.

After each program revision is normalised, the next task is to detect the non-trivial changes. One extreme of the methods is to apply an existing `diff` algorithm, which in fact may or may not detect the exact differences. Another extreme of the methods is to apply clone detection [11]. The benefit of applying clone detection is that one could take advantage of knowning the meaningful structures. Therefore, if the ordering is not imporant and as long as the normalised entities are the same, the clone detector could find them. On the other hand, if the two entities are similar but not exactly the same, a meaningful context of the difference can be shown. Not all language constructs should be considered as clones. E.g., the index variables of a for-loop are apparently not the best candidate to tell meaningful changes. It makes little sense to keep the for-loop structure while removing all the index variables. To be able to specify which part of the language construct needs to be considered as clones to removal, another kind of annotation to the non-terminals are introduced as the following rule.

Property 3: **Elementary structural clone removal.** Any optional or repeat term in the grammar can be marked as possible clones such that removing them do not change the validity of the source programming language. Any mandatory term in the grammar can be marked as possible clones if required, while the target programming language may violate the syntax rules of the source programming language.

Table I summarises the elementary annotations to the meta-grammar for the normalisation/clone removals. Although they are only complete for syntax validity preserving changes, with the help of user-defined functions, we found they are sufficient for relevant change detection tasks such as API evolution.

A. *Running Example*

To illustrate the features of our method, here we use the example of the Java 5 grammar provided by the TXL site[3], with 970 lines of code. Listing 5 selects a few production rules, with line numbers of the original Java5 grammar. In the conventions of TXL meta-grammar, a non-terminal is enbraced by square brackets, and a production rule is defined by the 'define...end define' blocks. TXL is a functional programming language in which a transformation is defined by either a non-recursive `function` or a recursive `rule` [10].

Although it is tempting to show fewer lines of "pseudo code" to illustrate the algorithms, in this paper, we choose to show some of the exact declarative rules such that it is sufficient to reproduce the work. Furthermore, we simplified away some lines to economise the space. Those omitted rules can be found at http://sead1.open.ac.uk/mct.

Listing 5. `cat -n java.grm`

```
151 define program
152     [package_declaration]
153 end define
158 define package_declaration
159     [opt package_header]
160     [repeat import_declaration]
161     [repeat type_declaration]
162 end define
197 define type_declaration
198         [class_declaration]      [NL][NL]
199     |   [interface_declaration]  [NL][NL]
200     |   [enum_declaration]   [NL]
201 end define
254 define modifier
255         'abstract
256     |   'final
257     |   'public
258     |   'protected
259     |   'private
260     |   'static
261     |   'transient
262     |   'volatile
263     |   'native
264     |   'synchronized
265     |   'strictfp
266     |   [annotation]
267 end define
285 define class_or_interface_body
286     '{                                          [NL][IN]
287         [repeat class_body_declaration]         [EX]
288     '} [opt ';]                                 [NL][NL]
289 end define
377 define method_declaration
378     [NL] [repeat modifier] [opt generic_parameter]
                [type_specifier] [method_declarator]
                [opt throws] [method_body]
379 end define
407 define method_body
408         [block]         [NL][NL]
409     |  [opt annotation_default] '; [NL][NL]
410 end define
486 define block
487     '{               [NL][IN]
488         [repeat declaration_or_statement]     [EX]
489     '}
490 end define
```

Lines 151-153 define a `program` as a single instance of package declaration; Lines 158-162 define each package declaration to have an optional description of the package header, zero to many import declaration(s), before zero to many type declaration(s). Lines 197-201 define a type declaration as either one of three alternatives for, namely a class, an interface or an enum type. In these lines, [NL], [IN] or [EX] are predefined indentation tokens that will be inserted by the unparser to pretty print the transformed code, respectively for new line, increasing and decreasing indention levels. Therefore the output has two lines per import declaration. Lines 377-379 define a method declaration as zero to many modifiers (defined in Lines 254-267), plus an optional parameter, a type specifier, a method declarator, optional throws exception declarations and the mandatory method body. It is also notable that the method body is defined in Lines 407-410, which refers the block, defined in Lines 486-490, as a curly brace encloded array of zero to many declarations or statements.

It is possible to redefine the grammar of Java5 in TXL in many ways without necessarily changing the validity of a Java 5 program. For example, by replacing the two [NL] [NL] to a single [NL], one can already remove all the empty lines following the import statements. In the remainder of the section, we explain how normalisation is done by using these grammar rules as the input.

Listing 6. `cat -n java.annotated.grm`

```
1 include "java.grm"
2 redefine package_declaration
3     [opt package_header kept]
4     [repeat import_declaration ignored]
5     [repeat type_declaration kept ordered]
6 end define
7 define class_or_interface_body
8     '{ [NL][IN]
9        [repeat class_body_declaration kept ordered ignored
              when Private] [EX]
10    '} [opt ';] [NL][NL]
11 end define
12 redefine method_declaration
13    [NL] [repeat modifier ordered by Descending] [opt
          generic_parameter] [type_specifier] [
          method_declarator] [opt throws] [method_body
          preferred with ';]
14 end define
15 % ...
16 function Private A  [class_body_declaration]
17     match  [class_or_interface_body] B  [
          class_or_interface_body]
18     construct M  [modifier *] _  [^ A]
19     construct PublicModifiers [modifier*] 'public '
          protected
20     where not M [contains each PublicModifiers]
21 end function
22 rule Descending B [modifier]
23  match [modifier] A [modifier]
24  construct SA [stringlit] _ [quote A]
25  construct SB [stringlit] _ [quote B]
26  where SA [< SB]
27 end rule
```

[3]http://txl.ca

TABLE I
BASIC ANNOTATIONS TO THE TERMS IN A TXL GRAMMAR

Transformation	**Application scope**	**TXL annotations**	**Example**
Ignore	Repeat/List (∗), Optional (|)	[. . . ignored when F]	[repeat member_declaration *ignored when Private*]
Order	Repeat/List (∗)	[. . . ordered by F]	[member_declaration *ordered by Ascending*]
Prefer	Alternative (|)	[. . . preferred with C]	[method_body *preferred with ';*]
Keep	Any non-terminal term	[. . . kept]	[class_body_declaration *kept*]

Comparing the annotated Java 5 grammar as shown in Listing 6 with the original in Listing 5, it is clear that one only needs to "redefine" existing production rules while leaving other rules intact by simply including them (e.g., Line 1).

The redefined rules in Table I are used or composed in some of the term extensions. Here we explain the rationale behind these extensions. The annotations "kept" are appended to the terms such as "package_header" (Line 3), "type_declaration" (Line 5), "class_body_declaration" (Line 9). These instruct a clone detector to compare these three types of entities for possible clones. Although the technique is similar to a general-purpose clone detection, the purpose of the cross-program clone-removal is exactly the opposite: those non-clones are the differences to be detected.

However, the ordering of elements or appearance of ignoreable details can get in the way of meaingful change detections. Therefore several normalisation transformations are required to be applied before the change detection step.

Since one does not care whether a modifier is before another one or not (e.g., 'private static' is the same as 'static private'), the ordering of the elements in the array of `repeat modifier` (Line 378) is unimportant to the Java semantics. However, the default behaviour of TXL parser preserves the ordering of the modifiers in the parsing tree as they occur in the source program. To specify the "Order" normalisation, one only needs to insert the ordered at the end of the `[repeat modifier]` term.

Furthermore, if one would like to normalise the elements by the descending order, a user-defined rule `Descending` (Lines 22-27) is added in Listing 6. This is just to illustrate how easy it is to customize the comparison function, in case one would like to define a different key or ordering for the structure to be normalised. For the sake of identifying meaningful changes in this particular case, ordering these members ascendingly is the same as ordering them descendingly.

The Ignore annotation (ignored), on the other hand, will replace the optional or repeated terms by empty. The terms `import_declaration` at Line 4, `class_body_declaration` at Line 9, are examples. In particular, the Ignore annotation to `class_body_declaration` is conditional, it uses a user-defined function from Lines 16-21 to check when the term has not used the `public` or `protected` modifiers. As a result, it will achieve the effect of extracting API methods from all members.

Without specifying such user-defined functions, the default behaviour of Ignore extension would simply ignore the term, just as what `import_declaration`. Because such terms are unconditionally ignored, therefore it is unnecessary to compose it with the Keep annotation as other sibling do. As a result, this will ignore the import statements in the API regardless, so any difference in such statements will not be considered as meaningful.

Finally, the Prefer annotation (preferred) are appended to the terms that have more than one alternative expansions. The user is free to choose a sequence of literals to make a constant instance of the terms in any one of its alternative production rules. For example, when the `method_body` at Line 13 is annotated by `preferred ';`, which will lead to a transformation to turn any block into a semicolon.

B. Brief discussion about the implementation

The meaningful change detection tool `mct` is implemented completely as a TXL program. The first part of the implementation is an extension to the TXL's metagrammar `txl.grm`. Listing 7 shows the extension to the existing `typeSpec` rule and the addition of four annotation rules `orderedBy`, `ignoredWhen`, `preferred` and `kept`.

Listing 7. `cat -n norm.grm`

```
include "txl.grm"
% The extension of the Txl grammar
keys
  ... 'kept 'ordered 'by 'ignored 'when 'preferred 'with
end keys
define typeSpec
  ... [opt kept] [opt orderedBy] [opt ignoredWhen] [opt
      preferredWith]
end define
define kept 'kept end define
define orderedBy 'ordered [opt byFunction] end define
define byFunction 'by [id] end define
define ignoredWhen 'ignored [opt whenFunction] end define
define whenFunction 'when [id] end define
define preferred 'preferred 'with [literal+] end define
```

The second part of the implementation is a specification of the normalisation transformations. For brevity, we only show a simplified Listing 8 for the Order extension. It generates the rules for eliminating `ordered` annotations so it is recognizable by TXL at runtime, and for producing the rules for ordering the parsed terms.

A TXL program can be understood top-down from the back. Lines 44-64 specify how to generate the transformation rules `Rules` on the fly by checking every `redefineStatement` in the TXL grammar such as those in Listings 6. For each occurrence of `[repeat X ordered by F]`, the transformation in Lines 9-31 is invoked to generate a rule such as those instantiated in Lines 22-27. These rules have unique names constructed from the names of the `redefineStatement` and the term `X`. By the end of the main transformation, the rule in Lines 2-8 are applied to eliminate the Order annotations from the extended grammar.

Listing 8. `cat -n norm.Txl`

```
1 include "norm.grm"
2 rule typeSpec_eliminateOrderedBy
3  replace * [typeSpec] T [typeSpec]
4  deconstruct T M [opt typeModifier] I [typeid] R [opt typeRepeater] O [orderedBy]
5  deconstruct O 'ordered B [opt byField]
6  construct T1 [typeSpec] M I R
7  by T1
8 end rule
9 function typeSpec_repeat_byField DS [redefineStatement] T [typeSpec]
10  import Rules [statement*]
11  import RuleIDs [id*]
12  replace [statement*] _ [statement*]
13  deconstruct DS 'redefine TID [typeid] TYPE [literalOrType*] REST [barLiteralsAndTypes*] 'end 'define
14  deconstruct T 'repeat I [typeid] R [opt typeRepeater] O [opt orderedBy]
15  deconstruct O 'ordered B [opt byField]
16  deconstruct B 'by F [id]
17  construct StrID [id] _ [quote TID]
18  deconstruct I TypeID [id]
19  construct ID [id] 'normalise_list
20  construct ruleID [id] ID [_ StrID] [_ TypeID]
21  construct S [statement*]
22   'rule ruleID
23    'replace '[ 'repeat I ']
24      'N1 '[ I '] 'N2 '[ I '] 'Rest '[ 'repeat I ']
25    'where 'N1 '[ F 'N2 ']
26    'by 'N2 'N1 'Rest
27   'end 'rule
28  export Rules Rules [. S]
29  export RuleIDs RuleIDs [. ruleID]
30  by S
31 end function
32 function DS_replace DS [redefineStatement]
33  replace [statement*] S0 [statement*]
34  construct T [typeSpec*] _ [^ DS]
35  construct S2 [statement*] _ [typeSpec_repeat_byField DS each T]
36  construct S [statement*] S0 [. S1] [. S2] [. S3]
37  by S
38 end function
39 function id_to_type ID [id]
40  replace [literalOrExpression*] L [literalOrExpression*]
41  construct T [literalOrExpression*] '[ ID ']
42  by L [. T]
43 end function
44 function main
45  replace [program] P [program]
46  export Rules [statement*] _
47  export RuleIDs [id*] _
48  construct DS [defineStatement*] _ [^ P]
49  construct S [statement*] _ [DS_replace each DS]
50  import Rules
51  import RuleIDs
52  deconstruct P S0 [statement*]
53  construct ID [id*] RuleIDs [print]
54  construct PL [literalOrExpression*] 'Prg
55  construct PL2 [literalOrExpression*] _ [id_to_type each RuleIDs]
56  construct L [literalOrExpression*] _ [. PL] [. PL2]
57  construct REPLACE [replacement] L
58  construct MAIN [statement]
59    'function 'main 'replace '[ 'program ']
60        'Prg '[ 'program '] 'by REPLACE
61    'end 'function
62  construct P1 [program] S0 [. Rules] [. MAIN ]
63  by P1 [typeSpec_eliminateOrderedBy]
64 end function
```

C. Generated normalisation transformation

The above generic implementation is done on the meta-grammar of TXL. When it is applied to a concrete TXL grammar, such as the one specified in Listing 6, a concrete normalisation transformation is produced in the original syntax of TXL, as shown in Listing 9. Lines 1-9 are the same as the original rules in the Listing 5 because of the elimination rule. Lines 10-17 are generated from the `[repeat method_declaration]` of the `orderedBy` annotations from Listing 6, using the user-defined comparison function `Descending`, which was listed in Lines 29-34 in Listing 6.

Listing 9. `cat -n java.Txl`

```
1  include "java.grm"
2  redefine class_or_interface_body
3          '{ [NL] [IN]
4               [repeat class_body_declaration]  [EX] [NL]
5          '}  [opt  ';]  [NL] [NL]
6  end define
7  redefine method_declaration
8     [NL] [repeat modifier]  [opt generic_parameter]  [type_specifier]  [method_declarator]  [opt throws]  [method_body]
9  end define
10 rule normalise_list_method_declaration_modifier
11     replace  [repeat modifier]
12         N1  [modifier]  N2  [modifier]  Rest  [repeat modifier]
13     where
14         N1  [Descending  N2]
15     by
16         N2  N1  Rest
17 end rule
18 function main
19     replace  [program]
20         Prg  [ program ]
21     by
22         Prg  [ normalise_list_method_declaration_modifier ]
23 end function
```

The above extension of Java5 grammar added 8 annotations, and 1 user-defined string comparison rule for sorting the nodes in inverse alphabatical order and 1 user-defined function for selecting non-API members to be removed.

D. The normalised programs and relevant changes

We have implemented the processor of all the four types of elementary annotations using TXL, which generates a few transformation rules per annotated term. Applying the composed normalisation transformation to the two Java programs in Listings 1 and 2 separately, the same result is obtained, as shown in in Listing 10. Both `hello` and `world` members are removed because they are not public nor protected members of the class. The `main` method has the modifiers ordered ascendingly as `public static`, whilst its method body is replaced by the preferred simplification semicolon alternative. To display the differences of two compared programs, a generated transformation is applied to remove all inter-program cloned instances of the annotated terms. This transformation only applies to those *"kept"* annotated terms, because one usually would not remove duplication of low-level term such as identifiers. As a result, there is no longer anything left, leaving the output empty as shown in Listing 11.

Listing 10. `mct HelloWorld.java`

```
public class HelloWorld {

   public static void main (String args []);
}
```

Listing 11. `mct HelloWorld.java -diff HelloWorld-2.java`

TABLE II
SIZE OF THE FULLY EXTENDED GRAMMARS AND TIME TO GENERATE THE NORMALISATION RULES

Grammar	Description	LOC	Terms	Rules	Time
txl.grm	TXL meta-grammar	408	58	1	
mct.Txl	Implementation	+673	+7	+28	
java5.Txl (L)	Java 5 grammar	976	168	1	
java.norm	annotation	+132	+21	+1	
java.Txl (M)	result	+729	+5	+43	0.07s
v.Txl (L)	Verilog grammar	233	37	1	
verilog2.norm	annotation	+21	+4	0	
verilog2.Txl (M)	result	+215	+5	+7	0.06s

IV. EXPERIMENTAL RESULTS

To evaluate the `mct` tool, we experimented on a benchmark of three evolving programs, namely `gmf`, `jhotdraw` in Java and `Uart16650` in Verilog. These programs are in the public domain: `JHotDraw` is a GUI framework for technical and structured Graphics, which was studied for the API evolution and refactoring opportunities [3]; `GMF` is a model-driven code generator for Eclipse Graph Editors, which was studied for the evolution of the model/code co-evolution [12]; `OpenCores Uart16650` is a specification of FIFO queue for hardware, which was used for the study of Verilog Diff [8].

To apply to these programs, we first defined the changes meaningful to API evolution based on their corresponding programming languages. Table II lists the size of the meta-grammar (txl.grm), Java 5 grammar (java5.Txl) and the Verilog grammars (v.Txl). The '+' sign before the numbers is used for counting those incremental grammars that include the original one. The `mct` tool is implemented as `mct.Txl`, which consists of 28 additional rules that transform the 7 extended terms. The Java API normalisation tool is implemented by redefining terms using 22 (3 Keep, 14 Order, 3 Ignore and 2 Prefer) annotations, and 1 user-defined rule in additional to the original grammar. As a result, 47 new transformation rules are generated, which also defines 6 new refined terms. The Verilog annotations include 1 Keep, 1 Order and 1 Ignore, which generates 7 additional transformation rules. On a server running RedHat Enterprise Linux 5.0, with a 3.16 GHz Intel Xeon X5464 CPU, 6MB cache, 24GB memory, the automated generation of these normalisation rules takes no more than 0.07 seconds. The programming language grammars in TXL (L) and the generated normalisation transformation rules in TXL (M) were then used in the remaining experiments.

We accessed the history of `gmf` and `jhotdraw` by analysing all commits from their public CVS repositories; whilst we were using the same set of selected revisions of `Uart16650` provided by Dulay et al [8]. Let X be 'J' stand for the original code, 'L' stand for the commentless and 'M' stand for the normalised code. The Java parsers adopted from TXL site already remove all the comments and white spaces in the programs. Thus it is perhaps better to compare the normalised results (M) with the unparsed ones (L) rather than with the original code (J).

Table III lists the size metrics of these programs. The metric 'LOC(X)' is the number of accumulated lines of code of all the revisions; 'Δ LOC(X)' is the number of accumulated lines of changes detected by the `diff`/`ldiff` utilities respectively.

TABLE III
CHANGE OF SOURCE PROGRAMS WITH COMMENTS/WHITESPACES (J), W/O WHITESPACES/COMMENTS (L) OR AFTER NORMALISATIONS (M)

Program	File	LOC(J)	LOC(L)	LOC(M)
$\Delta =$ `diff`	Commit	Δ LOC(J)	Δ LOC(L)	Δ LOC(M)
`uart16650`	12	51,601	28,805	417
`(mct)`	8	19	19	19
`(diff)`	128	1,864	879	29
`(ldiff)`	71	1, 552	694	29
`jhotdraw`	1,012	316,248	212,145	41,614
`(mct)`	724	2,506	2,506	2,506
`(diff)`	1,582	29,088	21,479	3,472
`(ldiff)`	1,264	28,882	28,699	3,284
`gmf`	5,263	8,944,841	4,288,931	453,181
`(mct)`	4,810	31,797	31,797	31,797
`(diff)`	17,522	437,860	274,835	44,516
`(ldiff)`	14,046	440,160	278,190	37,838

TABLE IV
COMPARING FILE-LEVEL CHANGES

Program	Pairs	`diff`	`txl+diff`	`mct`
Uart16650	71	62(-12.7%)	53(-25.4%)	8(-88.7%)
jHotDraw	1,302	1,264(-2.9%)	1,107(-15.0%)	724(-44.4%)
GMF	14,046	11,196(-20.3%)	9,767(-30.5%)	4,810(-65.8%)

All the size metrics show that `UArt16650` << `JHotDraw` << `GMF`, roughly by a magnitude of 10. The accumulated lines of code in the GMF CVS repository has close to 9 million lines of code. Taking out whitespaces/comments does help reduce the size to nearly a half, indicating that the three open-source programs were all well-commented. The performance of `ldiff` in terms of detecting file-level changes is the same as that of `diff`. However, `ldiff` generally reduces the amount of information presented to developers when changes did occur.

The absolute size of the normalised code LOC(M) is almost 10 times smaller than LOC(L), and the change Δ LOC(M) is also much smaller than the counterparts. As a result, in all the three examples, fewer file-level changes are found by `diff`/`ldiff`. Table IV highlights the ratio of reduction of the file-level changes after `diff`, `txl+diff` and `mct` are

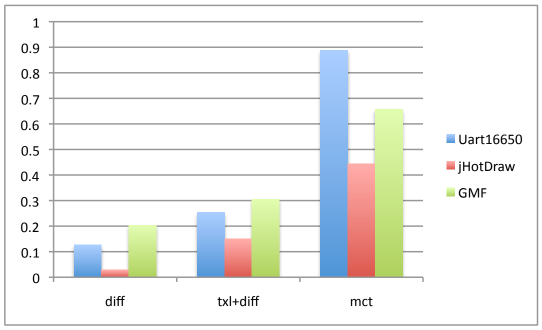


Fig. 3. Contrast the effectiveness of change detection at the file level as the percentage of reduced changed files

TABLE V
ABSOLUTE TIME PERFORMANCE IN SECONDS

Program Commits	Tool	T(J)	txl T(L)	mct T(M)
uart16650 128	codegen	0.0	1.7	2.8
	diff	0.2	0.2	
	ldiff	18.5	18.0	
jhotdraw 1,582	codegen	0.0	16.5	120.4
	diff	18.7	24.1	
	ldiff	633.5	866.7	
gmf 17,522	codegen	0.0	845.2	23,954.3
	diff	105.5	193.5	
	ldiff	42,740.0	18,525.0	

TABLE VI
SCALABILITY: TIME RELATIVE TO THE INPUT SIZE (SEC/MLOC)

LOC(J)	txl +ldiff	ldiff	mct	txl +diff	diff
51,601	391.5	358.5	54.3	36.8	3.9
316,248	2055.3	2003.2	380.7	128.4	59.1
8,944,841	4872.7	4778.2	2678.0	116.1	11.8

applied, where mct has the most effective result. The ratios are compared in Figure 3.

The time it took for the experiment is shown in Table V, in the units of second. "T(X)" is the time it took to generate the programs, and to measure the differences using diff or ldiff between pairs of consequent revisions. For the original (X=J) program, no computation is needed from generate the program for the diff/ldiff measurements. The generation of the pretty printed (X=L) is done by the default TXL parser, and generation of the normalised (X=M) programs is done by the mct generated transformations.

Amongst the three evolving programs, diff is the fastest, on average only as little as (0.2 + 18.5 + 105.7) /(128+1582+17522)=124.4/19232 $<$ 0.007 seconds per revision pair; whilst using txl then ldiff is the slowest, on average 44255.4/19232=2.30 seconds per revision. On average, the normalisations and clone removals in one step mct takes about 24077.5/19232=1.25 seconds per revision. Finally, Table VI lists the time performance relative to the input size, in seconds per million lines of code. The columns are ordered by the last row, in the order from the slowest on the leftmost to the fastest on the rightmost. We only compare the

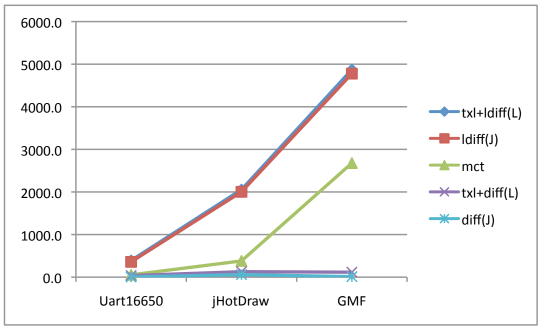


Fig. 4. Contrast the scalability of the experimented command line tools

scalability of five configurations for detecting changes, using a combination of 4 tools, diff, ldiff, txl and mct. Figure 4 plots the data in this table.

A. Threats to validity

Although some of our experiments were large scale, we should not over-generalise our findings. The GMF project is model-driven, and an unknown portion of CVS commits are attribute to the code generations, thus are less relevant to the API-evolution compared to those manual changes in the JHotDraw project. In these experiments, however, we did not separate generated code from the CVS repository. The Verilog normalisation applies fewer annotated transformations than that of the Java API, which could explain why the average time/MLOC performance of Verilog is better than JHotdraw. The performance of ldiff in terms of file level change detection is not better than that of diff, however, the quality of diff-chunks in ldiff may be finer than that of diff. Therefore we included experiments of both to indicate how mct compares with the scalable ldiff tool. To avoid any skewing of the running environment, we took the timing measurement 5 times and the minimal elapsed time. However, a different hardware configuration may lead to slight different measurements in the figures. Finally, we did not compare the interactive supervised diff results as some of the related work does because such supervisions require a large amount of time on the code base, and it also requires truthful interpretation from the developers which are intrusive to the development projects. As a trade-offs, our command line tool mct could substitute the underlying diff algorithms for the interactive use cases.

V. RELATED WORK

General problem. Brunet et at [13] define the challenges of model mangements in terms of *merge, match, diff, split* and *slice* operations and the properties that should be perserved by those operations. Their operations and properties are independent of models and modelling languages. Our *normalisation* steps are similar to *slice* operations because they are a series of transformations performed from model to model based on slicing criteria. And the *clone removal* step is similar to the *diff* operation that transforms two models to a set of changes. Model matching and merging [14] are also related to our work in the sense that they handle how to merge the detected changes. Our emphasis is on ensuring the transformed programs (or models) remain valid according to the original language syntax so that it is easier for to compare the differences to the original programs.

Structural diff. Xing and Stroulia [6] propose an approach to recover UML design models (such as class diagrams) from Java code, then compare their structural changes. The approach is specific to UML, which uses similarity metrics for names and structures to determine the types of changes made. Schmidt and Gloetzner [15] describe an approach that integrates the SiDiff tool to detect semantic changes on diagram structures. It then visualises the changes on UML diagrams and SimuLink models. These diagrams, in principle, can be specified in domain specific languages such as UMLasSketch[4]. Thus it is possible to apply our tool to detect changes on the textual models. The main difference with our work is that our approach allows the tool user to specify selective changes to be detected.

Apiwattanapong et al [16] present a graph-based algorithm for differencing object-oriented programs. Since their approach and the tool `JDiff` is geared towards Java, there is explicit support of Java-specific features, such as the exception hierarchies. Their tool is therefore not language-independent as ours.

Beyer et al [17] present an efficient relational calculator `crocopat` that can perform `diff` calculation on two sets of tuples very efficiently, and thus has been widely used in *visualization* reverse engineered facts such as call graphs or inheritance/aggregation relationships. However, `crocopat` treats all differences in sets rather than ordered lists. Therefore it is not suitable to check the differences among ordered structures such as statements or parameter lists.

Fluri et al [18] describe an approach to find differences on abstract syntax trees. The criteria for measuring the size of differences is the length of minimal number of editing operations, such as *insert, delete, move, update*. They also define several refactoring-based operations, such as condition expression change, method renaming, parameter delete, ordering change, type change, statement insertion, parent changes, etc. Most of their basic editing operations can be expressed using a composition of the four basic annotations rules used in this paper. On the other hand, the refactoring changes on semantic behaviour preserving require more advanced transformations that depends much on the specific Java semantics. Our design trade-offs that specificity with the generalisable principle so that a substantially different programming language such as Verilog can also be supported. In principle, refactoring type of transformations can be supported by specifying those changes as user-defined equivalence functions.

Semantic diff. There are several differencing tool working at the semantic level which may be complementary to us. Jackson and Ladd [7] use dependency between input and output variables of a procedure as a way to dectect certain changes. The depenedncy is represented as a graph and any difference in two graphs is taken as a change to the semantics of the procedure. There are, of course, changes that affect other kinds of semantics but not the dependency graph, such as the changes in constants. Working at the level of program grammar, our tool presents all the choices for the user to decide whether a change in constant should be ignored or not.

Kawaguchi et al [19] present a static semantic diff tool called `SymDiff`, which uses the notion of partial/conditional equivalence where two versions of a program is equivalent for a subset of inputs. The tool can infer certain conditions of equivalence, and therefore behavioural differences can be lazily computed. Our tool does not work at the level of program semantics.

Duley et al [8] present `VDiff` for differencing non-sequential, "position independent" Verilog programs. Their algorithm extracts the abstract syntax trees of the programs, and matches the subtrees in the ASTs whilst traversing them top-down. Furthermore, Boolean expressions are checked using a SAT solver and the results of differencing are presented as Verilog-specific change types. In this work, we used their datasets to demonstrate that our work can be applied to this language too. Although we do not classify the changes into 25 types as they did, we can also classify the changes according to annotated terms.

Applications. Wenzel et al [20] present a view that the evolution history of models can be traced into modeling elements and visualise the change metrics. When large amount of data are being processed, again it is important to be able to extract the relevant information to compare with each other. Our work has shown that it is not only possible to trace the history of evolution, but also possible to specify such views based on the relevance to programming tasks. On the other hand, adding some visualisation capability to our command-line tools should be possible, especially since we have obtained the exact structures in the comparisons.

Dagenais and Robillard [21] present a promising way to make use of change detection for recommendation systems, especially for the framework evolution. The example normalisation of this paper demonstrate that it is possible to extract API's of a framework from the evolving programs efficiently. Learning from these results, when more precise, it is our belief that the recommendation results can be improved.

Loh and Kim [22] present the `LSDiff` tool integrated with the Eclipse IDE to automatically identifies structural changes as logic rules. The idea is to integrate the change detection to the IDE such that developers can interact and instruct the classification of change types, and such logic rules can then be integrated with the change detection to produce results closer to developers interpretations. To a large extent, such change types are the relevant changes one aimed at specifying. Instead of specifying them as logic rules, in our work we specify them on top of the production rules with an emphasis that it is also important to preserve the validity of the relevant changes to the source programming languages.

Godfrey and Zou [23] introduce the notion of "origin analysis" for detecting structural changes made to program entities, in order to find out reasons for merging and splitting of code. With more precise changes identified on the structure, it is possible to increase the chance of identifying such origins.

Refactoring transformations improve program structures without introducing behavioural changes (see Mens and Tourwé [24]). The four basic annotations may already express simple forms of refactorings, which "normalises" the structures in order to remove irrelevant changes when evolving programs are compared. To detect an advanced refactoring, especially those happened to the APIs (see Dig et al[3]), a user-defined function working on a term of appropriate scope (e.g., class_declaration, or method_body), depending on the

[4]http://martinfowler. com/bliki/UmlAsSketch.html

nature of the refactoring.

Implementation. Although our implementation is based on TXL [10], the implementation of the concepts could be replaced using other generic transformation systems/ language engineering tools such as GrammarWare [25]. Currently our implementation of the clone removal is limited to exact clones. When parameterised clones are concerned [11], one should be able to ignore more structural changes.

VI. Conclusions

In this paper, we presented a declarative approach to specify changes in programs that are relevant to different programming tasks. In particular, we demonstrated its use in extracting the changes at the API levels for evolving Java programs and hardware specifications. We showed that four elementary annotations can account for all light-weight syntactical normalisations by default, and can be customized for more advanced semantic changes through user-defined functions. Our generic specification approach is declarative and already works on several programming and domain-specific languages. Our automated tool was implemented on top of a generic transformation system, TXL, and the results of detecting relevant differences in the three programs are evaluated and compared with other `diff` utilities such as line-based `diff`, showing more precise detection and acceptable scalability and time performance. The tool and results can be downloaded from http://sead1.open.ac.uk/mct.

Acknowledgement

The work is in part supported by the EU FP7 Security Engineering of Lifelong Evolvable Systems (SecureChange) project, the Microsoft Software Engineering Innovative Foundation (SEIF 2011) award, and the SFI CSET2 project at Lero. We thank our colleague Dr. Charles B. Haley for useful discussions.

References

[1] D. MacKenzie, P. Eggert, and R. Stallman, *Comparing and Merging Files with GNU diff and patch*. Network Theory, Ltd, December 2002.

[2] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, pp. 1053–1058, December 1972. [Online]. Available: http://doi.acm.org/10.1145/361598.361623

[3] D. Dig and R. E. Johnson, "How do APIs evolve? a story of refactoring," *Journal of Software Maintenance*, vol. 18, no. 2, pp. 83–107, 2006.

[4] G. Canfora, L. Cerulo, and M. Di Penta, "Tracking your changes: A language-independent approach," *IEEE Softw.*, vol. 26, pp. 50–57, January 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1495795.1495956

[5] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, "Closing the gap between modelling and Java," in *SLE*, ser. Lecture Notes in Computer Science, M. van den Brand, D. Gasevic, and J. Gray, Eds., vol. 5969. Springer, 2009, pp. 374–383.

[6] Z. Xing and E. Stroulia, "UMLDiff: an algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 54–65. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101919

[7] D. Jackson and D. A. Ladd, "Semantic diff: A tool for summarizing the effects of modifications," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 243–252. [Online]. Available: http://portal.acm.org/citation.cfm?id=645543.655704

[8] A. Duley, C. Spandikow, and M. Kim, "A program differencing algorithm for Verilog HDL," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 477–486. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859093

[9] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[10] J. Cordy, "The TXL source transformation language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[11] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *ICPC*, R. L. Krikhaar, R. Lämmel, and C. Verhoef, Eds. IEEE Computer Society, 2008, pp. 172–181.

[12] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth, "Language evolution in practice: The history of gmf," in *SLE*, ser. Lecture Notes in Computer Science, M. van den Brand, D. Gasevic, and J. Gray, Eds., vol. 5969. Springer, 2009, pp. 3–22.

[13] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh, "A manifesto for model merging," in *Proceedings of the 2006 international workshop on Global integrated model management*, ser. GaMMa '06. New York, NY, USA: ACM, 2006, pp. 5–12. [Online]. Available: http://doi.acm.org/10.1145/1138304.1138307

[14] S. Nejati, M. Sabetzadeh, M. Chechik, S. Uchitel, and P. Zave, "Towards compositional synthesis of evolving systems," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 285–296. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453143

[15] M. Schmidt and T. Gloetzner, "Constructing difference tools for models using the SiDiff framework," in *Companion of the 30th international conference on Software engineering*, ser. ICSE Companion '08. New York, NY, USA: ACM, 2008, pp. 947–948. [Online]. Available: http://doi.acm.org/10.1145/1370175.1370201

[16] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *Proceedings of the 19th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 2–13. [Online]. Available: http://portal.acm.org/citation.cfm?id=1025115.1025202

[17] D. Beyer, A. Noack, and C. Lewerentz, "Efficient relational calculation for software analysis," *IEEE Trans. Software Eng.*, vol. 31, no. 2, pp. 137–149, 2005.

[18] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, pp. 725–743, 2007.

[19] M. Kawaguchi, S. K. Lahiri, and H. Rebelo, "Conditional equivalence," Microsoft, Tech. Rep. MSR-TR-2010-119, October 2010.

[20] S. Wenzel and U. Kelter, "Analyzing model evolution," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 831–834. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368214

[21] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 481–490. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368154

[22] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, may 2009, pp. 309 –319.

[23] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Softw. Eng.*, vol. 31, pp. 166–181, February 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1048724.1048920

[24] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 126–139, February 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=972215.972286

[25] P. Klint, R. Lämmel, and C. Verhoef, "Toward an engineering discipline for grammarware," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, pp. 331–380, July 2005. [Online]. Available: http://doi.acm.org/10.1145/1072997.1073000

[26] M. van den Brand, D. Gasevic, and J. Gray, Eds., *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 5969. Springer, 2010.